Second Midterm Exam
CS164, Fall 2009

Dec 3, 2009

- Please read all instructions (including these) carefully.
- Write your name, login, and SID.
- No electronic devices are allowed, including cell phones used merely as watches.
- Silence your cell phones and place them in your bag.
- The exam is closed book, but you may refer to two pages of handwritten notes.
- 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 8 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: ________________________

SID: ________________________

Problem	**Max points**	**Points**
1	**40**	
2	**40**	
3	**20**	
TOTAL	**100**	
Extra credit	**10**	

Question 1: Inheritance [40 points]

This question asks you to develop an object system with prototype-based inheritance. You can (but don't have to) describe the solution you used in your project. This question also asks you to go beyond what you were required to implement in the project.

Part 1: Client code [9 points]

You may find it easier to first answer Part 2, where you *implement* the object system. In this part, you will *use* the system.

Write a code fragment in your 164 language that creates a prototype named `Foo` that contains fields `x` and `y`. Make the default values of `x` and `y` be `0`.

```
def Foo = Object:new({ x = 0, y = 0 })
```

Write a code fragment that creates a prototype `Bar` that is a "subclass" of `Foo` and contains a field `z`. Make the default value of `z` be `1`.

```
def Bar = Foo:new({ z = 1 })
```

Write a code fragment that creates an instance of `Foo` and an instance of `Bar`.

```
def foo = Foo:new({})
def bar = Bar:new({})
```

Add an instance method `f(a)` to `Foo` that returns the sum of `x`, `y`, and `a`.

```
Foo.f = lambda(self, a) { self.x + self.y + a }
```

Call the method `f` on your instance of `Bar` with the argument `42`.

```
bar:f(42)
```

Part 2: Implementation [9 points]

Show the implementation, in your 164 language, of the constructs you used in Part 1 (object constructor, method call, field access, etc). You may deviate from your project but your implementation must be consistent with Part 1.

Give the code that is used to construct an object.

```
def Object = {}
Object.new = lambda(self,o) {
      o = cond(o != null, o, {})
      setmetatable(o, self)
      self.__index = self
      o
}

def o = Object:new({})
```

Give the code that constructs a prototype object.

```
def Node = Object:new({name = ""})
```

Write how you implement method calls. If sugar is involved, show the desugaring rule.

When you make a method call with a colon, the "self" argument is automatically added.

E:Id(args) --> lambda() { $t1 = E; $t2 = $t1.Id; $t2($t1, args) }()

Part 3: Adding super [13 points]

Design and implement the ability to call a method in a superclass. You must be able to call methods of the superclass both from the constructor and from an instance method. You may not modify the interpreter. It may be easier for you to start with Part 4, where you *use* the super calls. In this part, you are asked to *implement* the super calls.

```
Object.super = lambda(self) {
    if ("__index" in self) { # prototype
        getmetatable(self).__index
    } else { # object
        getmetatable(self):super()
    }
}
```

Note that we have to treat instances and prototypes differently. If you call super when in an instance (e.g. `bar`), you can't just do `getmetatable(self).__index`, since that would lead to its prototype (e.g. `Bar`). Thus our code recognizes objects (because they do not have `__index` fields) and recurses on their prototypes and simple goes to the parent of a normal prototype.

There is an extra subtlety in how we call super. Consider a call `self:super():foo()`. Because of the desugaring rule for : presented above, this would be rewritten to `self.super(self).foo(self.super(self))`. Specifically, the self parameter to the `foo` function is wrong: it is the parent and not the child. We thus have to call this function like `self:super().foo(self)`. See part 4 for an example.

Part 4: Calling super [9 points]

Add to prototype `Bar` a method `f(a)` that adds the value of the field `z` to the value of the call to the same method in the superclass of `Bar`. The method `f(a)` must be written in your 164 language. Here is this method in Java:

```
class Bar extends Foo {
  ...
  int f(int a) {
    return super.f(a) + this.z;
  }
  ...
}
```

```
Bar.f = lambda(self, a) {
    self:super().f(self, a) + self.z
}
```

Add a constructor to `Bar` that calls the constructor of its superclass and then initializes `z` to the sum of `x` and `y`. In Java, this would look like:

```
class Bar extends Foo {
  ...
  Bar() {
    super();
    this.z = this.x + this.y;
  }
  ...
}
```

```
Bar.new = lambda(self) {
    def o = self:super().new(self, {})
    o.z = o.x + o.y
    o
}
```

Question 2: Iterators and Lists [40 points]

In this question, you will build on your implementation of lists from Project 3. You will add the ability to concatenate lists.

Part 1 [6 points]

Consider the following program in the 164 language.

```
def range(min,max) {
    min = min - 1
    lambda() { if (min < max) { min = min + 1 } else { null } }
}

def v1 = [2*n for n in range(4,7)]  // this is a list comprehension
def v2 = [n/2 for n in v1]
```

What is the 164 type of the value of the expression `range(4,7)`? ___function____

What is the 164 type of the value of the expression `v2`? ___dictionary________

One can implement lists as a special kind of dictionary, eg as a dictionary with a special key "length". (In contrast, our implementation defines a function length on dictionaries that returns the highest integer key in the dictionary.) The type of such a special dictionary can be considered to be *list,* hence we accepted "list" as a correct answer, too.

Part 2 [17 points]

Implement function `concat1(x,y)` that concatenates `x` and `y`. The arguments `x` and `y` could be either lists or iterators. The result of `concat1(x,y)` must be a list. If `x` and `y` are lists, these lists cannot be modified by `concat1`; instead, the result is a new list. Assume that `append(lst,elmnt)`, `for i in e { S }`, list comprehensions and coroutines are available to you. You may not modify the interpreter. Example:

```
def lst = concat1(v1,range(0,1))    # lst has value [8, 10, 12, 14, 0,
1]
print lst[2]                        # outputs 12
```

Your code for `concat1`:

The answer:

```
def concat1(l1,l2) {
    def l1copy = [e for e in l1]
    for e in l2 { append(l1copy,e) }
    l1copy
}
```

Part 3 [17 points]

Implement function `concat2` that concatenates its arguments but avoids creating an explicit list. The arguments `x` and `y` could be either lists or iterators. Assume that `append`, `for`, list comprehensions and coroutines are available. More readable and concise solutions will score higher. You may not modify the interpreter. In the following example, `concat2` does not copy the 100,000+ list elements.

```
for i in concat2(range(3,100000), lst) {  # lst is defined in part 2
    print i
    if (i>5) { null } # return from the function and end the loop
}
```

We implement the concatenation as a coroutine-based iterator:

```
def concat2(l1,l2) {
    def co = coroutine(lambda(_) {
        for e in l1 { yield(e) }
        for e in l2 { yield(e) }
        null
    })
    lambda() { resume(co,0) }
}
```

A partial credit solution: It is possible to implement the iterator factory concat2 without coroutines. However, the iterator becomes somewhat hairy and the iterator invokes the function _getIterator_ that should ideally be hidden from the program (one could argue that only the `for` construct should call this function):

```
def concat2(l1,l2) {
    # if the argument is a list, get its iterator
    if (type(l1) == "obj") { l1 = _getIterator_(l1) }
    if (type(l2) == "obj") { l2 = _getIterator_(l2) }
    def e1 = l1()
    lambda() {
        if (e1 != null) {
            def old_e1 = e1
            e1 = l1();
            old_e1
        } else {
            l2()
        }
    }
}
```

Part 4 [10 extra credit points] ***This is a bonus question. Solve it only if you are done with the exam.***

Consider the following code.

```
def lst2 = concat2(v1,v2)
def x = lst2[1]
```

Because the loop in Part 3 iterates over the value returned by `concat2`, one can think of `lst2` as a list. Yet the evaluation of `lst2[1]` fails. Why?

The value of lst2 is a function (an iterator). The operator [] cannot be applied on a function value, only on an dictionary value.

Outline a language extension that would make `lst2[1]` evaluate to the second element of the concatenation of `v1` and `v2` without modifying the interpreter:

Key idea: We allow the programmer to extend the default behavior of []. When [] is called on a function, we assume it is an iterator. We call the iterator idx-1 times, then return iterator().

More precisely, given an expression E1[E2], assume that iter is the value of E1 and idx is the value of E2. Assume that iter is a function value. We implement E1[E2] by calling iter idx-1 times, then returning iter().

Ideally, we should make a copy of the iterator iter, so that after iter[v] it is in the same position as it was before, but this is hard to do in general, so our suggested implementation does not do it. We settle for the semantics that indexing into an iterator with iter[v] advances the state of the iterator.

Step 1: Extend the language so that a program can add a hook to the operator []. This hook is a metamethod called __get. Once this metamethod is registered, whenever v1[v2] is attempted to be evaluated on value v1 that is not a dictionary, the interpreter calls __get(v1, v2).

We can decide to register this metamethod into the metatable of the global environment. In Lua, the global environment is a dictionary stored in variable _G.

```
setmetatable(_G,{__get=funGet})  # funGet is defined below.
```

Step 2: Define the hook. Here is where we implement the key idea describe above.

```
def funGet(fun,idx) {
    # assume: fun is an iterator

    # the value of the while is the value of the last statement
    while (idx>=0) {
        idx = idx - 1
        fun() # last call to fun is what is returned from funGet
    }
}
```

Question 3: Static Typing [20 points]

Consider the following Java program, which outputs "A A B". The first "A" indicates that, according to Java semantics, the methods `B::f` and `C::f` do not override the method `A::f`. Instead, they are considered to be unrelated methods (as if they were not even named `f`).

These methods are considered not to override `A::f` because the types of their parameters differ from that of `A::f`. As a result, an instance of class `B` has two methods: `f(A x)` and `f(B x)`. The former is inherited from class `A`.

This question asks why Java's designers decided that `B::f` and `C::f` do not override `A::f`.

```
class A {
    int a;
    void f(A x) {
        System.out.println("A");
    }
}

class B extends A {
    int b;
    void f(B x) {
        System.out.println("B");
        x.b = 100;
    }
}

class C extends A {
    A c;
    void f(C x) {
        System.out.println("C");
    }
}

class Main {
    public static void main(String[] args) {
        A aa = new B();
        aa.f(new B());          // in Java, this prints "A"
        B bb = new B();
        bb.f(new A());          // in Java, this prints "A"
        bb.f(new B());          // in Java, this prints "B"
        // <your code for question 2 goes here>
    }
}
```

Part 1 [6 points]

Suppose we define the semantics such that that `B::f` and `C::f` do override `A::f`. That is, given `x.f(y)`, the method `f` to call is determined by the dynamic type of `x`. For example, if the dynamic type of `x` is `B`, then `B::f` is invoked.

What would the output of the above program be under these semantics?

```
B
B
B
```

Part 2 [14 points]

Consider again the modified semantics given in Part 1. Add code to the `main` method that, when executed, would violate the invariant that the static type system seeks to maintain.

Hint: It may help you to draw how instances of classes A, B, and C are laid out in memory.

Hint: Java does not check the types of method arguments at runtime.

Write the inserted code here.

```
B b = new B();
C c = new C();
b.f(c);
```

Which variable or object field violates the invariant?

c.c should be an instance of A but is instead whatever happens to be at memory location 100.

or

In the method B::f(B x), x is bound to a instance of class C but is of static type B.

Describe in prose how this violated invariant allows you to do something bad.

We can set c.c to an arbitrary integer value and then treat it as a pointer. This allows us to read and write arbitrary locations in memory.