UNIVERSITY OF CALIFORNIA
Department of Electrical Engineering
and Computer Sciences
Computer Science Division

CS 164 **P. N. Hilfinger**
Fall 2011

CS 164: Test #2 Solutions

1. [2 points] We'd like to add inheritance to our statically typed subset of Python from Project 2. Consider the following two classes, `A` and `B`, in which class `B` overrides method `f`, inherited from class `A`.

```
class A (object):
    def f(arg::T) :: R:
        ...

class B (A):
    def f(arg::U) :: S:
        ...
```

The following questions concern certain design choices about the legality rules for these methods. In each case, either give a brief (1 or 2 sentences) explanation why the design choice is sound. And if not, give an example of a program that demonstrates showing why it is unsound (i.e. a program in which the dynamic type of some expression is not a subtype of its static type).

a. Is it sound to allow `U` to be a subtype of `T`?

Solution: *No. Since a* `B` *is also an* `A`*, one can call* `B.f` *through a value whose static type is* `A`*. If* `U` *is a strict subtype of* `T`*, that would allow a value that is not a* `U` *to get passed to* `B.f`*.*

b. Is it sound to allow `U` to be a supertype of `T`?

Solution: *Yes. Whether one refers to a* `B` *through a variable of static type* `B` *or* `A`*, the value of* `arg` *will also be some kind of* `U`*.*

c. Is it sound to allow `S` to be a subtype of `R`?

Solution: *Yes. Any value returned by* `B.f` *would be a valid return value from a call from which a value of type* `R` *is expected.*

d. Is it sound to allow `S` to be a supertype of `R`?

Solution: *No:* `B.f` *would return a value that is not an* `R`*, so that a call to* `B.f` *through a variable whose static type is* `A` *could produce an unexpected value (not of the return type of* `A.f`*).*

2. [2 points] We discussed the *global display* as an implementation device for allowing nested functions access to local variables and parameters of enclosing functions. This strategy requires functions to save their frame pointers in the global display on entry and restore the previous value on exit. In what cases is it safe for a function to skip these two steps? Give a succinct justification for your answer.

> **Solution:** *If the function contains no nested functions, the value that would be stored in the global display need never be used. This assumes that (as is usual), the function in question uses a dedicated register (either the frame or stack pointer) as the base register for access to local variables (and parameters).*

3. [3 points] Infer the type of the following function (that is, the types of `fold`, `f`, `z`, and `xs`), using our project 2 semantics. We don't need to see the actual type equations or their solution.

```
def fold(f, z, xs):
    if xs == None:
        return z
    else:
        (y, ys) = xs
        return f(y, fold(f, z, ys))
```

> **Solution:** *Actually, this problem showed up an oversight in the Project #2 spec: the* `==` *and* `!=` *operators are not supposed to have any limitations on the types of their operands, as long as those types match. Under that assumption:*

```
fold:: (($a, $d)->$d, $d, $c)->$d
f::    ($a, $d)->$d
xs::   $c
   where $c = tuple($a, $c)
z::    $d
```

> *Here* `$a` *and* `$d` *are free type variables.*

4. [2 points] I execute a certain program in our Python dialect. During the entire lifetime of a particular local variable, that variable is also in scope.

a. What can you say about the variable and the function that defines it?

> **Solution:** *The function declaring that variable (call it f) must not call any function that is not nested within f. Furthermore, if it calls a nested function, a variable of the same name must not be declared inside that nested functio (which would hide f's local variable).*

b. If I execute the same program a second time, will the variable necessarily have this same property?

> **Solution:** *No. The property described in part (a) is dynamic: the occurrence of a call to a non-nested function could be data-dependent.*

c. Give an example of a local variable where this property *cannot* hold, regardless of any input to the program.

> **Solution:**

```
    def g(y):
        print y    # x is not in scope

    def f(z):
        x = 2*z
        g(x)
```

5. [3 points] Python (real Python in this case), like C++ and Java, is a statically scoped language. Suppose, however, that a Python programmer wanted to find a general way to achieve the *effect* of dynamic scoping. The idea is to replace a general function:

```
def f(p1, p2, ..., pn):
    statements defining local variables v1, ..., vn
```

with something like

```
def f(p1, p2, ..., pn):
    PROLOGUE
    try:
       modified statements defining local variables v1, ..., vn
    finally:
       EPILOGUE
```

Assume that our transformation knows which local variables are defined in f (and can therefore choose *PROLOGUE* and *EPILOGUE* appropriately. Assume also that, as in regular Python, it is an error to use a variable that is defined in a scope before that variable has been assigned to (so that '`x = y; y = 3`' would be erroneous as the body of a function.)

In case your your full Python is rusty, `finally` works as in Java to guarantee that *EPILOGUE* happens last, regardless of how you exit f. If you exit with a `return`, for example, it saves the return values, executes *EPILOGUE*, and then returns.

The idea is to use some global data structure to get the desired effect.

a. Suggest such a data structure. (We are not interested in gritty details of its implementation—just its behavior. Feel free to use Python types.) Indicate what to put for *PROLOGUE* and *EPILOGUE*; and indicate how to modify any variable references in the "statements..." (both to the v_i and to variables that are used but not assigned to.) Describe in enough detail that a reasonably skilled programmer could reliably carry out your design.

    Solution: *A straightforward solution is to use a dictionary; let's call it* `DYN`. *In PROLOGUE, save a local copy of the contents of* `DYN` *(not just a pointer to it), and delete all entries for $v_1, \ldots, v_n$. Replace each use of v_i in the body with* `DYN["`v_i`"]`. *In EPILOGUE, restore the contents of* `DYN` *from the local copy (you can optimize by saving only the contents, if any, of the v_i from* `DYN`.

b. Why can't this idea be applied to the Python dialect used in our projects?

    Solution: *For one thing, there's a serious typing problem: the dictionary can't handle values of different types in the same dictionary.*

6. [1 point] You travel due north for a distance of 600 nautical miles. Approximately what effect does this have on the apparent elevation of the north star?

> **Solution:** *A nautical mile along a great circle (such as a line of longitude, which one follows when traveling due north) corresponds to one minute of arc along that circle. Thus you would travel* $10°$ *north, causing the north star to climb that amount.*

7. [4 points] Consider the following two Java classes:

```
class A {
     int x;
     static int y;
     int z;
     void f() { BODY #1 }
     void g() { BODY #2 }
}

class B extends A {
      int x;
      void f() { BODY #3 }
      void h() { BODY #4 }
}
```

and the allocation:

```
A x = new B();
```

Answer the following questions on the next page.

a. Show the layout of the object pointed to by `x` and of the other runtime data structures used to effect object-oriented behavior for classes A and B in Java. (Ignore for this problem the fact that class `A` inherits from type `Object`.)

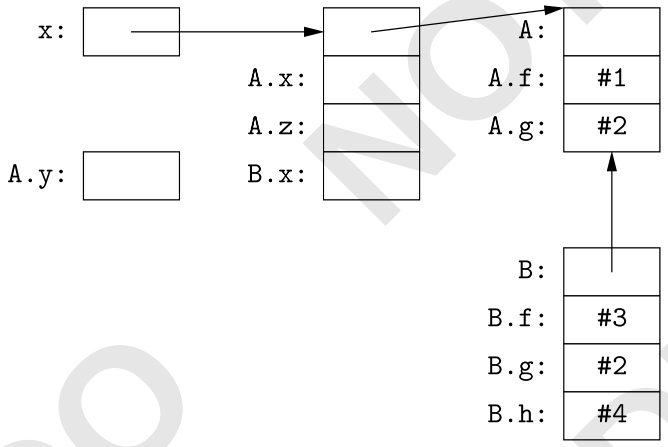
b. How does `B.f` (BODY #3) access `z`?

c. Java allows inheritance only from a single class. Show why allowing `B` to inherit *both* from `A` and some other class, `C`, causes trouble for the Java implementation. Be specific: give a concrete example of a problematic class `C` and indicate specifically what "breaks" and under what circumstances.

Answer to problem 7.

Solution:

a. *In the follow diagram,* `A` *and* `B` *refer to the runtime type descriptors ("virtual tables") for classes* `A` *and* `B`*.*



b. *Assuming pointers and the variable* `A.x` *are 4-byte quantities,* `z` *would be at offset 8 from '***this***' (which is simply an argument to the function.)*

c. *Suppose that class* `C` *defines a single instance variable* `cv`*, and suppose also that* `C` *defines a function,* `e`*, that* `B` *inherits. We can no longer simply copy the function pointer for* `e` *into* `B`*'s virtual table, as we did for* `g`*. This is because (assuming we place the representation of* `C` *after that of* `A`*)* `cv` *will be at offset 4 from* `this` `e` *is called on something created by* `new C()`*, but will be at offset 16 on something created by* `new B()`*.*

8. [4 points] We can abstract the primitive operations emitted by the compiler to access variables in a program into:

A Access (read or write) a memory value at a fixed offset from a (virtual) register into a register.

SL Load a register with the static link of a frame whose address is currently stored in a register.

SLF Load a register with the static link of a function value pointed to by a register.

SLC Pass a value that is currently stored in a register to a function that is about to be called. The function is to use the value as its static link.

CALLR Call a function whose code address is in a register.

Consider an execution of the program below:

```
def f0(g, c1, x1):
    def f1(c2):
        def f2(c3):
            def f3():
                print x1      # 1

            if c3 == 0:
                f0(f3, 1, 0)
            else:
                g()           # 2
        f2(c2)
    f1(c1)

f0(None, 0, 12)
```

a. At point #1, what sequence of primitive operations does the compiler emit to fetch x1? (It is sufficient to give the operations without specifying operands.) Give a short justification of your answer.

Solution:

```
SL   # Load f3's static link (to a frame for f2) to R1 (some register)
     #    using the current frame pointer
SL   # Load f2's static link (to a frame for f1) to R1 using the
     #    frame pointer in R1
SL   # Load f1's static link (to a frame for f0) to R1 using the
     #    frame pointer in R1
A    # Access x1 from f0's frame (whose address is in R1).
```

Continued

b. At point #2, what sequence of primitive operations does the compiler emit to call `g`? (Again, specify the operations; you need not give the operands.) Give a short justification of your answer.

Solution:

```
SL SL A     # As for part (a), above, but this time we only have to go
            #    up from f2's frame, and we fetch g.
SLF SLC     # Fetch and pass the static link stored in g.
A           # Fetch the code address from g
CALLR       # Call the function at the address just fetched.
```