CS 164, Spring 2010 CS 164: Test #2 (revised) Solutions P. N. Hilfinger

1. [4 points] Give short answers to the questions below.

a. Java allows multiple inheritance only in the sense that a class may implement any number of interfaces, which don't contain method bodies or instance variables, but may only extend one class. C++ allows classes to extend (in the Java sense) any number of classes. In lecture, I showed that this complicates the implementation of instance method calls in C++ relative to Java. Suppose that Java used a restriction on multiple inheritance halfway between C++ and current Java: interfaces may have method bodies (not just abstract methods), but not instance variables. For a definition such as

```
class A extends B implements I_1, I_2, ... { }
```

if we inherit a method body (that is, a non-abstract method) from both `B` and one or more of the $\texttt{I}_k$, we use the one from `B`. If we inherit a method body from both $\texttt{I}_j$ and $\texttt{I}_k$ where $j < k$, we use the one from $\texttt{I}_j$. How (if at all) would this change simplify the implementation of multiple inheritance for Java relative to C++?

> **Answer:** You don't need the various tricks for manipulating the receiver pointer (`this`) in order to get at instance variables. In fact, basically, you can use Java's representation unchanged, except that now, some items in the virtual table will come from interfaces.

b. In standard Python, every object instance has a dictionary that maps names of attributes (the '`a`' in '`x.a`') to their values (which may be methods as well as ordinary object references). But Java, on the other hand, has fixed tables, generated at compile time, containing method pointers and does not keep any kind of look-up structure around for routine uses of instance methods and instance variables. Illustrate why Python requires its use of dictionaries (that is, supply an example that shows why the Java implementation strategy cannot work and briefly explain why).

> **Answer:** Consider
>
> ```
> class A(object):
>     def f(self):
>         BODY1
> x = A()
> y = A()
> x.f = lambda self, x: x + self.a
> x.a = 3
> y.b = 4
> ```
>
> At this point, x has a new implementation of `f` and a new instance variable `a`, but `y`, although of the same type, does not. This sort of variabilty in methods and instance variables won't work when objects of a type share a single virtual table, and all have the same instance variables.

2. [3 points] For a statically typed dialect of Python, give the type of the function `iterate` defined below in ML notation (using `'a`, `'b`, etc. for type variables, $A \rightarrow B$ for function types, $A \star B$ for a tuple whose elements have types A and B, etc.). For a function that takes no argument and returns a value of type T, write the type as $() \rightarrow T$. *Show your reasoning* (which need *not* involve showing the application of the unification algorithm). Be sure to define the type of *iterate* itself plus any non-free type variables you introduce in the process.

```
def iterate(f, x):
  y = f(x)
  def g():
    return iterate(f, y)
  return tuple(y, g)
```

Answer:

```
iterate : ('a -> 'a) -> 'a -> 'b,
          'b = 'a * (() -> 'b)
```

3. [1 point] Which of the following terms least belongs?

anapest, antispast, dactyl, foot, iamb, paeon, pyrrhus, spondee, synecdoche

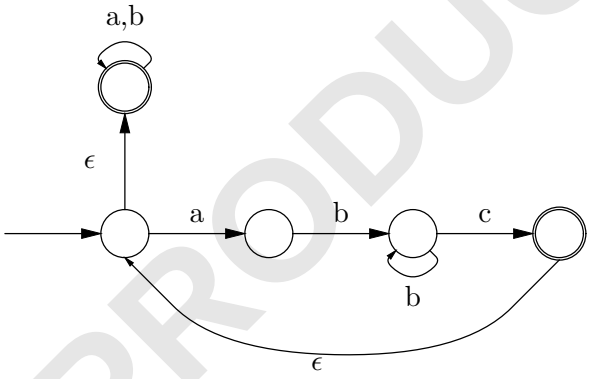
Answer: A *synecdoche* is a kind of metaphorical figure of speech, while the other terms can all related to metrical feet (as in poetry).

4. [2 points] Give the simplest NDFA you can that recognizes the same language as the regular expression

```
(ab+c)*(a|b)*
```

Do not use more than 6 states (5 actually suffice). In particular, *don't* use the result of applying the general regular expression to NDFA construction.

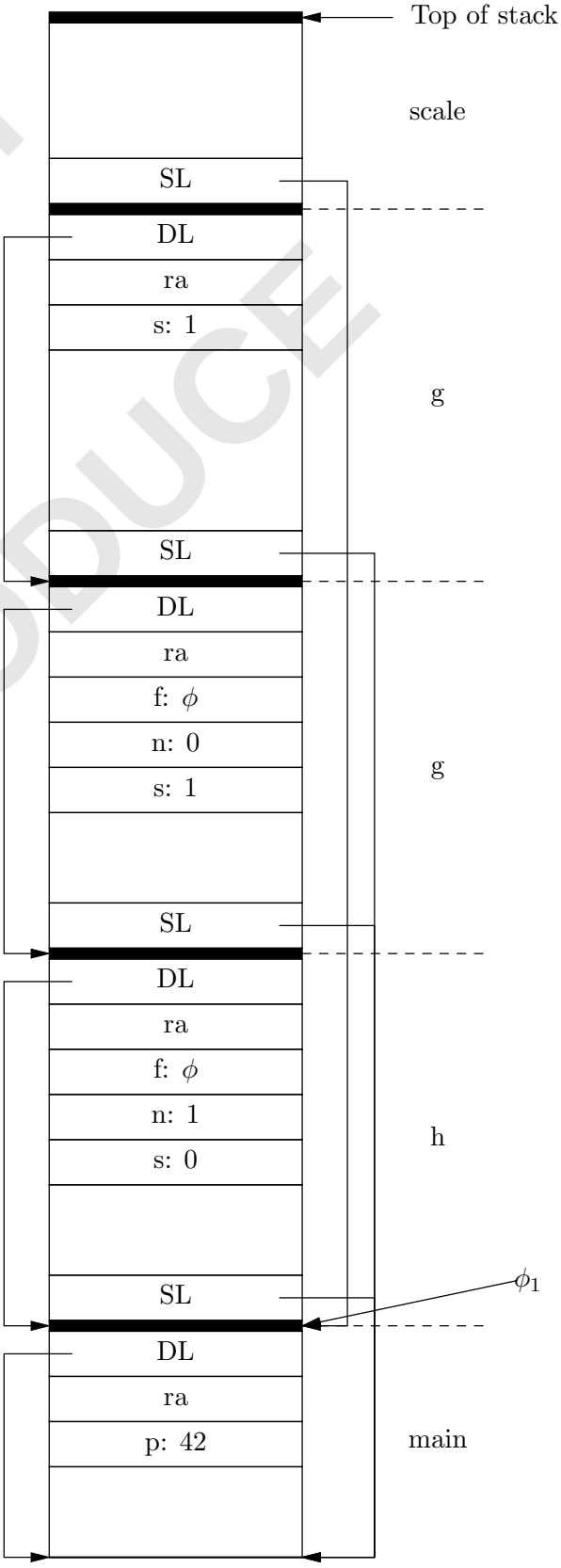
Answer:



5. [4 points]
Consider the following toy program:

```
def g(f, n, s):
    if n <= 0:
        return f(s)
    else:
        return g(f, n-1, s+n)
def h(p):
    def scale(x):
        return x * p   # Stop here
    print g(scale, 1, 0)
h(42)
```

Assuming that we restrict Python (as we will in Project 3) so that functional values that survive beyond the scope in which they are defined are invalid (so that you may call the value of `lambda x: x+y`, or pass it as a parameter, but may not return it from a function and then call it), complete the diagram of the contents of the stack at the point where execution reaches the spot marked "Stop here." In particular, show where dynamic (DL) and static (SL) links point, and values and locations of actual parameters on the stack (don't forget the value of `f`). We're assuming an ia32 architecture.



Answer: Here, ϕ points to a pair of variables in memory, one containing the code address for `scale` and the other containing the frame pointer ϕ_1.

The important static link is ϕ_1; it is less important to get the other static links right, since these are outer functions. There are no local variables other than parameters. The parameters for a function appear under the frame pointer for that function—that is, on top of the frame for the calling function.

6. [3 points] For each of the following Python programs (in our Project #2 subset), determine the *earliest* phase in which processing could detect an error. If possible, indicate a location in the program text. There may be more than one error in each program. Treat each as a full program, without surrounding statements.

- Mark the location L if it will first fail in lexical analysis.
- Mark the location P if it will first fail in parsing.
- Mark the location S if it will first fail in static analysis.
- Mark the location R if it will first fail at run-time.
- Mark the program N if there is no error.

a.
```
if x > 3:
   y = x
     x += 1              # P (unexpected INDENT)
```

b.
```
def f(x):
    print x * y, x + y   # S (y undefined)

f(12)
```

c.
```
if x > 3:
     y = x
  x += 1                 # L (inconsistent indentation)
```

d.
```
class A(object):
    pass

def g(x):
    print x * y, x + y   # R (y has inappropriate type)

y = A()
g(12)
```

Continues on next page

e.
```
class A(object):
    x = 3
    def f(self, y):
        self.x = y

class B(A):
    def g(self, y):
        self.x = self.x + y

class C(B):
    def h(self):
        self.g(self.y)      # S (y undefined in type C).

x::C = C()
C.h()
```

f.
```
def f(x):
    global y
    y = x                   # R (x has dynamic type Int at runtime)

y::String = ""
f(12)
```

7. [4 points] For each of the type rules below (for a statically typed language), indicate whether it allows illegal programs, disallows legal programs, or neither, assuming standard interpretations of the program constructs. Predicates typeof, defn, and subtype are as described in the lecture notes: typeof(V, T, E) means the static type of expression V is T in environment E (a list of the form [def(V_1, T_1),...,def(V_n, T_n)] for variables V_i and types T_i); defn(I, T, E) means that identifier I has static type T in environment E, and subtype(T, T') means that T is a subtype of T' (possibly equal). If a rule is wrong, explain why by giving an example of an excluded legal program or a permitted illegal program.

a. Assume that `list(`T`)` is intended to refer to a Python-style list (as in `[1,2,3]`), but with statically typed elements, and that `setitem(X,I,E1)` is the AST for '`X[I]=E1`'. Assume here that assignment statements have the types of their left operands.

```
typeof(setitem(X,I,E1), T, Env) :-
   typeof(X, list(T), Env), typeof(E1, T1, Env), subtype(T1,T).
```

Answer: Various problems here. For a Python list, the index must be an integer, but there is no restriction on I here—not even that the expression be type correct. Thus, it allows

```
L["foo"] = 3
```

If you interpreted this as an extension of our version of Python, then you could also complain that it forbids assignment from variables of static type 'any', which we allow.

b. Here, `suite([`$S_1, \ldots, S_n$`])` is intended to represent a suite of statements as in our Python dialect, with the same semantics, and `assign(X,T,E)` stands for the assignment `X::T=E` (again according to our rules).

```
typeof(suite([]),void,_).
typeof(suite([assign(X,T,E) | Rest]), void, Env) :-
   typeof(E, T1, Env), typeof(suite(Rest), void, [def(X,T) | Env]),
   subtype(T1, T).
```

rules for suites that start with other kinds of statements

Answer: The definition of X does not apply to the part of the suite prior to the definition, allowing both kinds of error:

```
x = "Foo"          # Should not be allowed
x::Int = 3
```

and

```
x::String = "Foo"
def f():
    x = 3          # Should be allowed
    x::Int = 6
```

Also, you can again complain about assignments from type 'any'.