

CMSC330 Spring 2009 Practice Problems 6 Solutions

1. Programming languages

- a. Describe the difference between ad-hoc and parametric polymorphism.
Ad hoc polymorphism applies to code supporting a finite range of types whose combinations must be specified, parametric polymorphism applies to code written without mention to type that can transparently support an arbitrary number of types.
- b. Describe the difference between starvation and deadlock.
Deadlocked threads are halted waiting for each other's locks, whereas starving threads are waiting for locks from other (non-starving) threads.
- c. Describe how functional programming may be used to simulate OOP.
An object may be simulated as a tuple, where each element of the tuple is a closures representing a method for the object.
- d. Describe the difference between HTML and XML.
HTML tags are predefined and presentation-oriented, whereas XML tags are user defined and are intended for describing data and metadata.
- e. Describe the difference between query languages and programming languages.
Query languages are designed to make requests to a database or information system, whereas programming languages are designed to express computations that can be performed by a machine.

2. Polymorphism

Consider the following Java classes:

```
class A { public void a() { ... } }  
class B extends A { public void b() { ... } }  
class C extends B { public void c() { ... } }
```

Explain why the following code is or is not legal

- a. `int count(Set<A> s) { ... } ... count(new TreeSet<A>());`
Legal. Actual parameter type (Set<A>) matches formal parameter type (Set<A>)
- b. `int count(Set<A> s) { ... } ... count(new TreeSet());`
Illegal. Actual parameter type (Set) is not a subclass of formal parameter type (Set<A>), even though B is a subclass of A.
- c. `int count(Set s) { ... } ... count(new TreeSet<A>());`
Legal. Type erasure will cause formal parameter type (TreeSet<A>) to become TreeSet, which matches actual parameter type (Set).
- d. `int count(Set<?> s) { ... } ... count(new TreeSet<A>());`
Legal. Actual parameter type (Set<A>) matches formal parameter type (Set<?>), since ? matches A.
- e. `int count(Set<? extends A> s) { ... } ... count(new TreeSet());`
Legal. Actual parameter type (Set) matches formal parameter type (Set<? extends A>), since “? extends A” can match A and its subclasses B & C (classes that extend A, including A)

- f. `int count(Set<? extends B> s) { ... } ... count(new TreeSet<A>());`
Illegal. Actual parameter type (`Set<A>`) does not match formal parameter type (`Set<? extends B>`), since “`? extends B`” can match only `B` and its subclass `C` (classes that extend `B`, including `B`)
- g. `int count(Set<? extends B> s) { for (A x : s) x.a(); ... }`
Legal. The actual parameter type (`Set<? extends B>`) indicates `s` contains elements of class `B` or its subclasses. So any element of `s` may be treated as an object of class `B` or its subclasses (e.g., `C`). The for loop treats elements of `s` as objects of class `A`, which is a superclass of `B`, and thus is legal (can use subclass in place of superclass).
- h. `int count(Set<? extends B> s) { for (C x : s) x.c(); ... }`
Illegal. The actual parameter type (`Set<? extends B>`) indicates `s` contains elements of class `B` or its subclasses. So any element of `s` may be treated as an object of class `B` or its subclasses (e.g., `C`). The for loop treats elements of `s` as objects of class `C`, and is illegal since elements of `s` may be objects of class `B` (cannot use superclass in place of subclass).
- i. `int count(Set<? super B> s) { for (A x : s) x.a(); ... }`
Illegal. The actual parameter type (`Set<? super B>`) indicates `s` contains elements of class `B` or its superclasses. So any element of `s` may be treated as an object of class `B` or its superclasses (e.g., `A`, `Object`). The for loop treats elements of `s` as objects of class `A`, and is illegal since elements of `s` may be objects of class `Object` (cannot use superclass in place of subclass).
- j. `int count(Set<? super B> s) { for (C x : s) x.c(); ... }`
Illegal. The actual parameter type (`Set<? super B>`) indicates `s` contains elements of class `B` or its superclasses. So any element of `s` may be treated as an object of class `B` or its superclasses (e.g., `A`, `Object`). The for loop treats elements of `s` as objects of class `C`, which is not included and thus illegal.

3. Multithreading

- a. Using Java Conditions, you must implement a synchronization construct called MyBarrier. A MyBarrier object is created with a certain value n. When a thread calls the method enter(), it enters the barrier and blocks until a total of n threads have entered the barrier. When the nth threads enters the barrier, all the threads waiting at the barrier wake up and unblock, and the nth thread continues without blocking. When a thread calls the method reset(), the barrier is reset so that it starts fresh in counting up to n (i.e., n more threads must enter the MyBarrier).

```
public class MyBarrier {
    int num;           // shared read-only data
    int current = 0;   // shared modifiable data
    Lock lock = new ReentrantLock();
    Condition ready = lock.newCondition();

    public MyBarrier (int n) {
        num = n;
    }

    public void enter() throws InterruptedException {
        lock.lock();           // prevent data race on current
        current++;            // incr # of threads at barrier

        if (current == num) { // enough threads at barrier
            ready.signalAll(); // wake up other threads
        }                     // continue execution
        else {
            while (current < num) { // wait for more threads to enter
                ready.await();      // sleep until enough threads enter
            }                       // use while ( ) in case reset( ) called
        }
        lock.unlock();
    }

    public void reset() {
        lock.lock();           // prevent data race on current
        current = 0;
        lock.unlock();
    }
}
```

- b. Implement MyBarrier using Ruby monitors.

```
require "monitor.rb"

class MyBarrier
  def initialize n
    @num = n
    @current = 0
    @myLock = Monitor.new
    @myCondition = @myLock.new_cond
  end

  def enter
    @myLock.synchronize {
      @current = @current + 1
      if @current == @num then
        @myCondition.broadcast
      else
        @myCondition.wait_while { @current < @num }
      end
    }
  end

  def reset
    @myLock.synchronize {
      @current = 0
    }
  end
end
```

- c. Write a Ruby program that creates a barrier for 2 threads, then creates 2 threads that each print out “hello”, enters the barrier, then prints out “goodbye”.

```
bar = MyBarrier.new 2

t1 = Thread.new {
  puts "hello"
  bar.enter
  puts "goodbye"
}

t2 = Thread.new {
  puts "hello"
  bar.enter
  puts "goodbye"
}
```

4. Lambda calculus

Make all parentheses explicit in the following λ -expressions

- a. $\lambda x.xz \lambda y.x y \rightarrow (\lambda x.((x z) (\lambda y.(x y))))$
 b. $(\lambda x.xz) \lambda y.w \lambda w.wyzx \rightarrow ((\lambda x.(x z)) (\lambda y.(w (\lambda w.(((w y) z) x))))))$
 c. $\lambda x.xy \lambda x.yx \rightarrow (\lambda x.((x y) (\lambda x.(y x))))$

Find all free (unbound) variables in the following λ -expressions

- d. $\lambda x.x z \lambda y.x y \rightarrow (\lambda x.((x \underline{z}) (\lambda y.(x y))))$
 e. $(\lambda x. x z) \lambda y. w \lambda w. w y z x \rightarrow ((\lambda x.(x \underline{z})) (\lambda y.(\underline{w} (\lambda w.(((w y) \underline{z}) \underline{x}))))))$
 f. $\lambda x. x y \lambda x. y x \rightarrow (\lambda x.((x \underline{y}) (\lambda x.(\underline{y} x))))$

Apply β -reduction to the following λ -expressions as much as possible

- g. $(\lambda z.z) (\lambda y.y y) (\lambda x.x a) \rightarrow (\lambda z.z) (\lambda y.y y) (\lambda x.x a) \rightarrow (\lambda y.y y) (\lambda x.x a) \rightarrow (\lambda x.x a) (\lambda x.x a) \rightarrow (\lambda x.x a) a \rightarrow a a$
 // β -reduction = body[sym/replacement]
 // $z[z/(\lambda y.y y)]$ replace z with $\lambda y.y y$
 // $y y[y/(\lambda x.x a)]$ replace y with $\lambda x.x a$
 // $x a[x/(\lambda x.x a)]$ replace x with $\lambda x.x a$
 // $x a[x/a]$ replace x with a
- h. $(\lambda z.z) (\lambda z.z z) (\lambda z.z y) \rightarrow (\lambda z.z) (\lambda z.z z) (\lambda z.z y) \rightarrow (\lambda z.z z) (\lambda z.z y) \rightarrow (\lambda z.z y) (\lambda z.z y) \rightarrow (\lambda z.z y) y \rightarrow y y$
 // β -reduction: replace z with $\lambda z.z z$
 // β -reduction: replace z with $\lambda z.z y$
 // β -reduction: replace z with $\lambda z.z y$
 // β -reduction: replace z with y
- i. $(\lambda x.\lambda y.x y y) (\lambda a.a) b \rightarrow (\lambda x.\lambda y.x y y) (\lambda a.a) b \rightarrow (\lambda y.(\lambda a.a) y y) b \rightarrow (\lambda a.a) b b \rightarrow b b$
 // β -reduction: replace x with $\lambda a.a$
 // β -reduction: replace y with b
 // β -reduction: replace a with b
- j. $(\lambda x.\lambda y.x y y) (\lambda y.y) y \rightarrow (\lambda x.\lambda y.x y y) (\lambda y.y) y \rightarrow (\lambda x.\lambda a.x a a) (\lambda y.y) y \rightarrow (\lambda a.(\lambda y.y) a a) y \rightarrow (\lambda y.y) y y \rightarrow y y$
 // α -conversion: rename y to a
 // β -reduction: replacing x with $\lambda y.y$
 // β -reduction: replacing a with y
 // β -reduction: replacing y with y
- k. $(\lambda x.x x) (\lambda y.y x) z \rightarrow (\lambda x.x x) (\lambda y.y x) z \rightarrow (\lambda y.y x) (\lambda y.y x) z \rightarrow (\lambda y.y x) x z \rightarrow x x z$
 // β -reduction: replacing x with $\lambda y.y x$
 // β -reduction: replacing y with $\lambda y.y x$
 // β -reduction: replacing y with x
- l. $(\lambda x. (\lambda y. (x y)) y) z \rightarrow (\lambda x. (\lambda y. (x y)) y) z \rightarrow (\lambda x. (\lambda a. (x a)) y) z \rightarrow (\lambda a. (z a)) y \rightarrow z y$
 // α -conversion: rename y to a
 // β -reduction: replacing x with z
 // β -reduction: replacing a with y
- m. $((\lambda x.x x) (\lambda y.y)) (\lambda y.y) \rightarrow ((\lambda x.x x) (\lambda y.y)) (\lambda y.y) \rightarrow ((\lambda y.y) (\lambda y.y)) (\lambda y.y) \rightarrow (\lambda y.y) (\lambda y.y) \rightarrow \lambda y.y$
 // β -reduction: replacing x with $\lambda y.y$
 // β -reduction: replacing y with $\lambda y.y$
 // β -reduction: replacing y with $\lambda y.y$

- n. $(((\lambda x. \lambda y. (x y))(\lambda y. y)) w)$
 $(((\lambda x. \lambda y. (x y))(\lambda y. y)) w) \rightarrow$ // α -conversion: rename y to a
 $(((\lambda x. \lambda a. (x a))(\lambda y. y)) w) \rightarrow$ // β -reduction: replacing x with $\lambda y. y$
 $((\lambda a. ((\lambda y. y) a)) w) \rightarrow$ // β -reduction: replacing a with w
 $(\lambda y. y) w \rightarrow$ // β -reduction: replacing y with $\lambda y. y$
 w

Show that the following expression has multiple reduction sequences

- o. $(\lambda x. y) ((\lambda y. y y) (\lambda x. x x x))$
// β -reduction: replace x in $\lambda x. y$ with $((\lambda y. y y) (\lambda x. x x x))$
// (no x in body, so just discard argument and replace $(\lambda x. y) \langle \dots \rangle$ with y)
 $(\lambda x. y) ((\lambda y. y y) (\lambda x. x x x)) \rightarrow y$
OR
// β -reduction: replace y in $\lambda y. y y$ with $\lambda x. x x x$
 $(\lambda x. y) ((\lambda y. y y) (\lambda x. x x x)) \rightarrow (\lambda x. y) ((\lambda x. x x x) (\lambda x. x x x) (\lambda x. x x x))$

Can repeat β -reduction for x as many times as we wish!

5. Lambda calculus encodings

Prove the following using the appropriate λ -calculus encodings

- a. $\text{not} (\text{not true}) = \text{true}$

Given:

$\text{not} = \lambda x. ((x \text{ false}) \text{ true})$

$\text{true} = \lambda x. \lambda y. x$

$\text{false} = \lambda x. \lambda y. y$

Proof:

$\text{not} (\text{not true})$

$= \lambda x. ((x \text{ false}) \text{ true}) (\text{not true})$

$= ((\text{not true}) \text{ false}) \text{ true}$

$= ((\lambda x. ((x \text{ false}) \text{ true}) \text{ true}) \text{ false}) \text{ true}$

$= (((\text{true} \text{ false}) \text{ true}) \text{ false}) \text{ true}$

$= (((\lambda x. \lambda y. x) \text{ false}) \text{ true}) \text{ false}) \text{ true}$

$= (((\lambda y. \text{false}) \text{ true}) \text{ false}) \text{ true}$

$= ((\text{false}) \text{ false}) \text{ true}$

$= ((\lambda x. \lambda y. y) \text{ false}) \text{ true}$

$= (\lambda y. y) \text{ true}$

$= \text{true}$

// replacing 1st not w/ encoding

// β -reduction: $x \rightarrow \text{not true}$

// replacing not w/ encoding

// β -reduction: $x \rightarrow \text{true}$

// replace true w/ encoding

// β -reduction: 1st $x \rightarrow \text{false}$

// β -reduction: $y \rightarrow \text{true}$

// replace false w/ encoding

// β -reduction: $x \rightarrow \text{false}$

// β -reduction: $y \rightarrow \text{true}$

// $\text{not} (\text{not true}) = \text{true}$

- b. $\text{or false true} = \text{true}$

Given:

$\text{or} = \lambda x. \lambda y. ((x \text{ true}) y)$

$\text{true} = \lambda x. \lambda y. x$

$\text{false} = \lambda x. \lambda y. y$

Proof:

or false true

// replacing or w/ encoding

$= \lambda x. \lambda y. ((x \text{ true}) y) \text{ false true}$
 $= \lambda y. ((\text{false true}) y) \text{ true}$
 $= (\text{false true}) \text{ true}$
 $= ((\lambda x. \lambda y. y) \text{ false}) \text{ true}$
 $= (\lambda y. y) \text{ true}$
 $= \text{true}$

// β -reduction: $x \rightarrow \text{false}$
// β -reduction: $y \rightarrow \text{true}$
// replace 1st false w/ encoding
// β -reduction: $x \rightarrow \text{false}$
// β -reduction: $y \rightarrow \text{true}$
// or false true = true

c. if false then x else y = y

Given:

if a then b else c = a b c
 true = $\lambda x. \lambda y. x$
 false = $\lambda x. \lambda y. y$

Proof:

if false then x else y
 $= \text{false } x \text{ } y$
 $= (\lambda x. \lambda y. y) x \text{ } y$
 $= (\lambda y. y) y$
 $= y$

// replacing if... w/ encoding
// replacing false w/ encoding
// β -reduction: $x \rightarrow x$
// β -reduction: $y \rightarrow y$
// if false then x else y = y

d. succ 2 = 3

Given:

$2 = \lambda f. \lambda y. f (f y)$
 $3 = \lambda f. \lambda y. f (f (f y))$
 succ = $\lambda z. \lambda f. \lambda y. f (z f y)$

Proof:

succ 2
 $= (\lambda z. \lambda f. \lambda y. f (z f y)) 2$
 $= \lambda f. \lambda y. f (2 f y)$
 $= \lambda f. \lambda y. f ((\lambda f. \lambda y. f (f y)) f y)$
 $= \lambda f. \lambda y. f ((\lambda y. f (f y)) y)$
 $= \lambda f. \lambda y. f (f (f y))$
 $= 3$

// replacing succ w/ encoding
// β -reduction: $z \rightarrow 2$
// expanding 2 w/ encoding
// β -reduction: 1st f \rightarrow f
// β -reduction: 1st y \rightarrow y
// apply encoding for 3
// succ 2 = 3

e. (* 1 3) = 3

Given:

$1 = \lambda f. \lambda y. f y$
 $3 = \lambda f. \lambda y. f (f (f y))$
 $M * N = \lambda x. (M (N x))$

Proof:

(* 1 3)
 $= \lambda x. (1 (3 x))$
 $= \lambda x. (1 (\lambda f. \lambda y. f (f (f y)) x))$
 $= \lambda x. (1 (\lambda y. x (x (x y))))$
 $= \lambda x. ((\lambda f. \lambda y. f y) (\lambda y. x (x (x y))))$
 $= \lambda x. (\lambda y. (\lambda y. x (x (x y)))) y$
 $= \lambda x. \lambda y. x (x (x y))$
 $= \lambda f. \lambda y. f (f (f y))$
 $= 3$

*// replacing * w/ encoding*
// replacing 3 w/ encoding
// β -reduction: 1st f \rightarrow x
// replacing 1 w/ encoding
// β -reduction: 1st f w/ $\lambda y. x (x (x y))$
// β -reduction: 1st y \rightarrow y
// α -conversion: replace x with f
// apply encoding for 3

f. $(+ 2 1) = 3$

Given:

$$\begin{aligned} 1 &= \lambda f. \lambda y. f y \\ 2 &= \lambda f. \lambda y. f (f y) \\ 3 &= \lambda f. \lambda y. f (f (f y)) \\ M + N &= \lambda x. \lambda y. (M x)((N x) y) \end{aligned}$$

Proof:

$$\begin{aligned} (+ 2 1) & & // \text{ replacing } + \text{ w/ encoding} \\ = \lambda x. \lambda y. (2 x)((1 x) y) & & // \text{ replacing } 2 \text{ w/ encoding} \\ = \lambda x. \lambda y. ((\lambda f. \lambda y. f (f y)) x)((1 x) y) & & // \beta\text{-reduction: } 1^{\text{st}} f \rightarrow x \\ = \lambda x. \lambda y. (\lambda y. x (x y))((1 x) y) & & // \text{ replacing } 1 \text{ w/ encoding} \\ = \lambda x. \lambda y. (\lambda y. x (x y))(((\lambda f. \lambda y. f y) x) y) & & // \beta\text{-reduction: } 1^{\text{st}} f \rightarrow x \\ = \lambda x. \lambda y. (\lambda y. x (x y))((\lambda y. x y) y) & & // \beta\text{-reduction: } 3^{\text{rd}} y \rightarrow y \\ = \lambda x. \lambda y. (\lambda y. x (x y))(x y) & & // \beta\text{-reduction: } 2^{\text{nd}} y \rightarrow x y \\ = \lambda x. \lambda y. x (x (x y)) & & // \alpha\text{-conversion: replace } x \text{ with } f \\ = \lambda f. \lambda y. f (f (f y)) & & // \text{ apply encoding for } 3 \\ = 3 & & \end{aligned}$$

g. $(Y \text{ fact}) 2 = 2$ // you do not need to expand any operators except fact & Y

Given:

$$\begin{aligned} Y &= \lambda f. (\lambda x. f (x x)) (\lambda x. f (x x)) \\ \text{fact} &= \lambda f. \lambda n. \text{if } n = 0 \text{ then } 1 \text{ else } n * (f (n-1)) \end{aligned}$$

Proof:

$$\begin{aligned} (Y \text{ fact}) 2 & & // \text{ replacing } Y \text{ w/ encoding} \\ = (\lambda f. (\lambda x. f (x x)) (\lambda x. f (x x))) \text{fact} 2 & & // \beta\text{-reduction: } 1^{\text{st}} f \rightarrow \text{fact} \\ = (\lambda x. \text{fact} (x x)) (\lambda x. \text{fact} (x x)) 2 & & // \beta\text{-reduction: } 1^{\text{st}} x \rightarrow \lambda x. \text{fact} (x x) \\ = (\text{fact} ((\lambda x. \text{fact} (x x)) (\lambda x. \text{fact} (x x)))) 2 & & \\ & // \text{ apply encoding for } (Y \text{ fact}) \\ & // ((\lambda x. \text{fact} (x x)) (\lambda x. \text{fact} (x x))) \rightarrow (Y \text{ fact}) \\ & // \text{ we know this is the encoding for } (Y \text{ fact}) \text{ from } 3^{\text{rd}} \text{ line of proof} \\ = (\text{fact} (Y \text{ fact})) 2 & & // \text{ apply encoding for fact} \\ = (\lambda f. \lambda n. \text{if } n = 0 \text{ then } 1 \text{ else } n * (f (n-1))) (Y \text{ fact}) 2 & & \\ & // \beta\text{-reduction: } 1^{\text{st}} f \rightarrow (Y \text{ fact}) \\ = (\lambda n. \text{if } n = 0 \text{ then } 1 \text{ else } n * ((Y \text{ fact}) (n-1))) 2 & & // \beta\text{-reduction: } n \rightarrow 2 \\ = \text{if } 2=0 \text{ then } 1 \text{ else } 2 * ((Y \text{ fact}) (2-1)) & & // \text{ apply if} \\ = 2 * ((Y \text{ fact}) 1) & & // \text{ showed in class } (Y \text{ fact}) 1 = 1 \\ = 2 * 1 & & // \text{ apply } * \\ = 2 & & \end{aligned}$$

6. Operational semantics

Use operational semantics to determine the values of the following OCaml codes:

a. 1

$$\frac{}{1 \rightarrow 1}$$

b. + 3 7

$$\frac{3 \rightarrow 3 \quad 7 \rightarrow 7}{+ 3 7 \rightarrow 10}$$

c. + 1 (+ 2 3)

$$\frac{1 \rightarrow 1 \quad \frac{2 \rightarrow 2 \quad 3 \rightarrow 3}{(+ 2 3) \rightarrow 5}}{+ 1 (+ 2 3) \rightarrow 6}$$

d. (fun x = 4) 5

$$\frac{\begin{array}{l} \bullet ; (\text{fun } x = 4) \rightarrow (\bullet, \lambda x.4) \quad // \text{ evaluate function to produce a closure} \\ \bullet ; 5 \rightarrow 5 \quad // \text{ evaluate the argument} \\ // \text{ evaluate body of closure, after extending} \\ // \text{ environment w/ binding for parameter} \end{array}}{(x:5, 4) \rightarrow 4}}{(\text{fun } x = 4) 5 \rightarrow 4}$$

e. (fun x = + x 6) 7

$$\frac{\begin{array}{l} \bullet ; (\text{fun } x = + x 6) \rightarrow (\bullet, \lambda x.+ x 6) \quad // \text{ evaluate function to produce a closure} \\ \bullet ; 7 \rightarrow 7 \quad // \text{ evaluate the argument} \\ // \text{ evaluate body of closure, after extending} \\ // \text{ environment w/ binding for parameter} \end{array}}{(x:7, + x 6) \rightarrow 13}}{\bullet ; (\text{fun } x = + x 6) 7 \rightarrow 13}$$

f. (fun x = (fun y = + y x)) 8 9

$$\frac{\begin{array}{l} \bullet ; (\text{fun } x = (\text{fun } y = + y x)) \rightarrow (\bullet, \lambda x.(\text{fun } y = + y x)) \quad // \text{ eval func} \\ \bullet ; 8 \rightarrow 8 \quad // \text{ eval arg} \\ // \text{ eval body} \end{array}}{x:8, (\text{fun } y = + y x) \rightarrow (x:8, \lambda y.(+ y x))}}{\bullet ; (\text{fun } x = (\text{fun } y = + y x)) 8 \rightarrow (x:8, \lambda y.(+ y x))}}{\bullet ; 9 \rightarrow 9 \quad // \text{ eval arg}}{\frac{x:8, y:9 ; (+ y x) \rightarrow 17}{\bullet ; (\text{fun } x = (\text{fun } y = + y x)) 8 9 \rightarrow 17}} \quad // \text{ eval body}$$

7. Markup languages

- a. Creating your own XML tags, write an XML document that organizes the following information: 1-hour test on Spanish Monday in Jiménez worth 15%. 1-hour test on Computers Tuesday in CSIC worth 10%. 30-minute test on Computers Friday in AVW worth 5%.

```
<testList>
  <test>
    <length>1 hour</length>
    <subject>Spanish</subject>
    <date>Monday</date>
    <location>Jiménez</location>
    <value>15 % </value>
  </test>
  <test>
    <length>1 hour</length>
    <subject>Computers</subject>
    <date>Tuesday</date>
    <location>CSIC</location>
    <value>10 % </value>
  </test>
  <test>
    <length>30 minute</length>
    <subject>Computers</subject>
    <date>Friday</date>
    <location>A VW</location>
    <value>5 % </value>
  </test>
</testList>

or

<testList>
  <test subject="Spanish">
    <length unit="hour">1</length>
    ...
  </test>
  <test subject="Computers" >
    <length unit="hour">1</length>
    ...
  </test>
  <test subject="Computers" >
    <length unit="minute">30</length>
    ...
  </test>
</testList>
```

8. Garbage collection

Consider the following Java code.

```
Object a, b, c;
public foo() {
    a = new Object();    // object 1
    b = new Object();    // object 2
    c = new Object();    // object 3
    a = b;
    b = c;
    c = a;
}
```

- a. What object(s) are garbage when foo() returns? Explain why.

Object 1 is garbage there are no longer any references to it within the program. After foo() returns, a → object 2, b → object 3, c → object 2.

- b. Describe the difference between mark-and-sweep & stop-and-copy.

Mark-and-sweep stops the program to determine what objects are still reachable. Stop-and-copy in addition will move reachable objects to new locations.