Fall, 2003 Name: _____

Com S 541 — Programming Languages 1

Test on OO Type Systems, Aspect-Oriented Languages, and AspectJ

This test has 10 questions and pages numbered 1 through 12.

Special Directions for this Test

This test is open book and notes.
If you need more space, use the back of a page. Note when you do that on the front.
This exam is timed. We will not grade your exam if you try to take more than the time allowed. Therefore, before you begin, please take a moment to look over the entire exam so that you can budget your time.
Clarity is important; if your diagrams or programs are sloppy and hard to read, you will lose points.
Correct syntax also makes a difference for programming questions.
When you write AspectJ code on this test, you may use anything in the Java standard libraries, or in the packages `org.aspectj.lang` or `org.aspectj.lang.reflect` without writing it in your test. You are encouraged to define classes, interfaces, fields, and methods not specifically asked for if they are useful to your programming; if they are not in the standard libraries or the packages named above, please write them into your test.

Base Program

In the first several problems we will use the following code as a base program. The base program contains an interface, `Term`, and three classes that implement it: `Variable`, `Application`, and `Lambda`.

```
package exam;
import java.util.Set;
/** Terms in the lambda calculus. */
public interface Term {
    /** Reduce one step. */
    //@ ensures \result != null;
    Term reduce1Step();

    /** Return the set of all Strings that name
     * free variables referenced in this term. */
    //@ ensures \result != null;
    Set freeVars();

    /** Substitute arg for all free occurrences of var. */
    //@ requires var != null && arg != null;
    //@ ensures \result != null;
    Term substituteFor(String var, Term arg);
}
```

```
package exam;
import java.util.*;
/** Variable reference terms. */
public class Variable implements Term {
    /** The name of this Variable */
    protected final /*@ non_null @*/ String name;

    /** Initialize this Variable to have the given name. */
    //@ requires name != null;
    public Variable(String name) {
        this.name = name;
    }

    public Term reduce1Step() {
        return this;
    }

    public Set freeVars() {
        Set s = new HashSet();
        s.add(this.name);
        return s;
    }

    public Term substituteFor(String var, Term arg) {
        if (name.equals(var)) {
            return arg;
        } else {
            return this;
        }
    }
    public String toString() { return name; }

    public boolean equals(Object o) {
        return (o instanceof Variable) && name.equals( ((Variable)o).name );
    }

    public int hashCode() { return name.hashCode(); }
}
```

```
package exam;
import java.util.*;
/** Application terms. */
public class Application implements Term {
    /** The operator and operand expressions. */
    protected final Term[] exps;  //@ private invariant \nonnullelements(exps);

    /** Initialize this to be an application of e0 to e1. */
    //@ requires e0 != null && e1 != null;
    public Application(Term e0, Term e1) {
        exps = new Term[] { e0, e1 };
    }

    public Term reduce1Step() {
        Term oper = exps[0].reduce1Step();
        if (oper != exps[0]) {
            return new Application(oper, exps[1]);
        } else if (oper instanceof Lambda) {
            return ((Lambda) oper).apply(exps[1]);
        } else {
            return this;
        }
    }

    public Set freeVars() {
        Set s = new HashSet();
        for (int i = 0; i < 2; i++) {
            s.addAll(exps[i].freeVars());
        }
        return s;
    }

    public Term substituteFor(String var, Term arg) {
        Term substituted[] = new Term[2];
        for (int i = 0; i < 2; i++) {
            substituted[i] = exps[i].substituteFor(var, arg);
        }
        return new Application(substituted[0], substituted[1]);
    }

    public String toString() { return "(" + exps[0].toString() + " " + exps[1].toString() + ")"; }

    public boolean equals(Object o) {
        return (o instanceof Application) && exps[0].equals(((Application) o).exps[0])
            && exps[1].equals(((Application) o).exps[1]);
    }

    public int hashCode() { return exps[0].hashCode() + exps[1].hashCode(); }
}
```

```
package exam;
import java.util.*;
/** Lambda (i.e., function) terms. */
public class Lambda implements Term {
    /** The bound variable. */
    protected final /*@ non_null @*/ String var;
    /** The body. */
    protected final /*@ non_null @*/ Term body;

    /** Initialize this to have variable v and body b. */
    //@ requires v != null && b != null;
    public Lambda(String v, Term b) {
        var = v;
        body = b;
    }

    public Term reduce1Step() {
        return this;
    }

    public Set freeVars() {
        Set s = new HashSet();
        s.addAll(body.freeVars());
        s.remove(var);
        return s;
    }

    public Term apply(Term arg) {
        return body.substituteFor(var, arg);
    }

    public Term substituteFor(String var2, Term arg) {
        if (var2.equals(var)) {
            return this;
        } else if (arg.freeVars().contains(var)) {
            throw new SubstitutionException();
        } else {
            Term newBody = body.substituteFor(var2, arg);
            return new Lambda(var, newBody);
        }
    }

    /** Exception class used in substituteFor */
    public class SubstitutionException extends RuntimeException { }

    public String toString() { return "(\\ " + var + " -> " + body.toString() + ")"; }

    public boolean equals(Object o) {
        return (o instanceof Lambda) && var.equals(((Lambda) o).var)
            && body.equals(((Lambda) o).body);
    }

    public int hashCode() { return var.hashCode() + body.hashCode(); }
}
```

1. (15 points) In AspectJ, without changing `Term` or its subtypes (see the previous pages), write an aspect `TraceBeta` that prints information about each β-reduction step that results from a call to `Application`'s `reduce1Step` method. This information must be printed to `System.out`, in a line of the form "`Reducing` e `-->` e'", where e is the application being reduced, and e' is the resulting term. (Hint: Note that all terms support a `toString()` method that can produce the required output. The type of `System.out` supports both a `print` and a `println` method; the `print` method does not put a new line on the output, while the `println` method does.)

   For example, with the required aspect, the following program

```
package exam;
public class ExerciseBeta {
    public static void main(String[] args) {
        Term t =
            new Application(
                new Lambda("x", new Variable("x")),
                new Variable("y"));
        Term s = t.reduce1Step();
        System.out.println(s.toString());
        t = new Application(
                new Application(
                    new Lambda(
                        "x",
                        new Application(new Variable("x"), new Variable("x"))),
                    new Lambda("z", new Variable("z"))),
                new Variable("q"));
        s = t.reduce1Step();
        s = s.reduce1Step();
        s = s.reduce1Step();
        System.out.println(s.toString());
    }
}
```

   would produce the following output.

```
Reducing ((\ x -> x) y) --> y
y
Reducing ((\ x -> (x x)) (\ z -> z)) --> ((\ z -> z) (\ z -> z))
Reducing ((\ z -> z) (\ z -> z)) --> (\ z -> z)
Reducing ((\ z -> z) q) --> q
q
```

   Your task is to write an aspect that would produce output like the above in general.

   There is space for your answer on the next page.

2. (10 points) Answer the question in whichever one of the following two paragraphs applies to you.

   If you solved the previous problem using `around` advice, can you also solve it using only `before` and `after returning` advice? If so do that; if not, describe why it can't be done.

   If you solved the previous problem using only `before` and `after returning` advice, can you also solve it using only `around` advice? If so do that; if not, describe why it can't be done.

3. (15 points) Without changing the code for `Term` or its subtypes, write an aspect, `BigStep` that introduces a method `evaluate()` into the interface `Term` and its three subtypes, `Variable`, `Application`, and `Lambda`. Like the method `reduce1Step()`, the `evaluate` method takes no arguments and returns a `Term`. However, unlike `reduce1Step`, the result returned should be in β-normal form; that is, if the result of t`.evaluate()` is n, then n`.reduce1Step()` should return a term that `equals` n. (Hint: don't forget that this is a problem about introduction, not about advice.)

   A solution should pass the tests in the JUnit test class `BigStepTest` shown below. In the JUnit test, `assertEquals` compares its two arguments using the `equals` method.

```
package exam;
import junit.framework.TestCase;
public class BigStepTest extends TestCase {
    public BigStepTest(String name) { super(name); }
    public static void main(String[] args) {
        junit.textui.TestRunner.run(BigStepTest.class);
    }
    public void testEvaluate() {
        assertEquals(new Variable("x"), new Variable("x").evaluate());
        assertEquals(
            new Application(new Variable("x"), new Variable("x")),
            new Application(new Variable("x"), new Variable("x")).evaluate());
        assertEquals(
            new Lambda("x", new Variable("x")), new Lambda("x", new Variable("x")).evaluate());
        assertEquals(
            new Variable("z"),
            new Application(new Lambda("x", new Variable("x")),
                            new Variable("z")).evaluate());
        assertEquals(
            new Variable("z"),
            new Application(
                new Application(
                    new Lambda("x", new Variable("x")),
                    new Lambda("x", new Variable("x"))),
                new Variable("z"))
                .evaluate());
        assertEquals(
            new Variable("z"),
            new Application(
                new Application(
                    new Lambda("x", new Variable("x")),
                    new Application(
                        new Lambda("x", new Variable("x")),
                        new Lambda("x", new Variable("x")))),
                new Application(
                    new Application(
                        new Lambda("x", new Variable("x")),
                        new Lambda("x", new Variable("x"))),
                    new Variable("z")))
                .evaluate());
    }
}
```

   There is space for your answer on the next page.

4. (5 points) Suppose that you wanted to modify the behavior of the `reduce1Step()` method for `Lambda` terms so that, instead of simply returning the original term, it checked to see if the η rule of the λ-calculus applied, and did an η-reduction if so. If you wanted to do this without using introductions and without changing the code of `Lambda`, then what kind of advice would you need to use to do that? (Don't write the advice, just say what kind of advice you would need and briefly explain why you must use that kind of advice as opposed to other kinds of advice. For example, `after throwing` is one kind of advice, and `before` is another kind of advice.)

5. (10 points) Does it make sense to use both the `target` and `this` pointcuts in conjunction with an `execution` pointcut? That is, does a pointcut of the form

   ```
   execution(* f()) && target(x) && this(y)
   ```

   make sense? Briefly explain your answer.

6. (10 points) Can one use AspectJ to enforce the following programming rule?

   Code in package `b` may not call code in package `a`?

   If so, show how this would be done in AspectJ; if not, then explain why this is impossible.

7. (5 points) What pointcuts in AspectJ can "bind" formals declared in advice or a pointcut declaration?

8. Suppose AspectJ were extended with a primitive pointcut `arrayfetch`, that takes a field pattern as an argument. For example, the pointcut

   ```
   arrayfetch(Term[] Application.exps)
   ```

   describes the joinpoints in the class `Application` which occur when the array `exps`, of type `Term[]`, has elements fetched from it. To simplify the discussion below, let us suppose that, in general, this pointcut has the form `arrayfetch(`JT`[] `TP`.`$IP`)`, where JT is a Java type (the type of the array's elements), TP is a type pattern that describes the types where the field is to be found, and IP is an identifier pattern that describes the names of the array fields that are being observed.

   (a) (5 points) If a joinpoint of the form `arrayfetch(`JT`[] `TP`.`$IP`)` is used in `around` advice, what type constraints should there be on the return type of a `proceed()` expression in the body of the `around` advice?

   (b) (5 points) If a joinpoint of the form `arrayfetch(`JT`[] `TP`.`$IP`)` is used in `after returning` advice, what type constraints should there be on the return type of a formal parameter that is declared for the return value in such advice?

9. (10 points) Give an example of "obliviousness" in Smalltalk.

10. (10 points) If you were to give a type system to Smalltalk, would you use a structural or a by-name type system? Explain the advantages of your choice.