CPSC 411, Spring 2005 – Quiz 2

Name: _____

Q1:	Q4:
Q2:	Q5:
Q3:	
TOTAL:	

Problem 1 [20 points]

In the MJ compiler, name binding and type checking are done separately by the BindNames and CheckTypes visitors. But there are two notable exceptions where name binding and type checking are combined.

What are the exceptions? Explain why name binding and type checking are combined in these cases. Be concrete and specific about the issue, and provide an example that illustrates your point.

Problem 2 [20 points]

Earlier in the semester we described code generation as a single pass (visitor) over the AST. But in the past couple of weeks we learned that a real code generator is more complex.

Describe the structure of code generation and emitting, from after type checking, up through writing the code to a file. What are the passes? What are the key data structures (objects). Give a brief (1-2 sentences) description of each pass and each kind of data structure.

Problem 3 [20 points]

For this question, assume that we are talking about pointcuts and advice, as they appear in AspectJ, or the Mini AspectJ subset we are developing.

Define each of the following concepts. Be sure that your definitions make **both** the differences and the connections between the concepts clear. If you choose, you can draw a simple diagram to help show those differences and connections.

join point

pointcut

advice

join point shadow

advice implementation method

Problem 4 [20 Points]

Consider the following class. In the space at the right, show Java byte codes for the **body** of each methods.

```
class Point {

  int x, y;

  int mumble(int a, int b) {
    return 2 * a + b;
  }

  int foo(int a) {
    return bar(a);
  }

  int bar(int a) {
    ... you don't need to  ...
    ... write code for bar ...
  }

  int getX() {
    return x;
  }
}
```

Problem 5 [20 Points]

In this problem you will consider byte-code level weaving of calls to advice helper methods. Assume that:

- your weaver only has byte code to operate on (no source code),
- that join point shadow matching has already been implemented,
- that advice helper methods like those used in the Mini Java compiler have been generated
- that the other parts of the runtime architecture, including the aspectOf method have also been generated

So all you have to concern yourself with is how to modify the code at a join point shadow so that the advice helper method is called properly, and the surrounding code also continues to work properly.

In particular, we have a sequence of byte codes, in which there is a call to the m method of Foo, which takes three arguments; and we have before advice that applies at this shadow. The signature of the helper method is "A/before$1()V".

Show what your weaver would have to convert the following byte code sequence to, in order to properly call the advice helper method.

.

.

invokevirtual Foo/m(I,I,I)V

.

.

Now consider a case we have not yet implemented, where the advice takes parameter – in particular the arguments to the method m. So the source for the advice might look like:

after(int a, int b, int c): call(void Foo.m(int)) && args(a, b, c) { …}

and the advice implementation method would have a different signature A/after$2(I,I,I)V.

Show the bytecodes your weaver would have to produce to properly call this advice.

.

.

.

invokevirtual Foo/m(I,I,I)/V

.

.