P1:	of 10	P5:	of 15
P2:	of 10	P6:	of 15
P3:	of 10	P7:	of 15
P4:	of 25		

CS 411 Final
April 18th, 2005

Exam number:

Student Name:

CPSC Login Username:

READ THIS CAREFULLY BEFORE PROCEEDING
DO NOT TURN THE PAGE UNTIL WE TELL YOU TO DO SO

CAUTION -- Candidates suspected of any of the following, or similar, dishonest practices will be immediately dismissed from the examination and will be liable to further disciplinary action:

a) Making use of any books, papers or memoranda.

b) Speaking or communicating with other candidates.

c) Purposely exposing written papers to the view of other candidates. The plea of accident or forgetfulness will not be received.

Instructions

1. Each candidate should be prepared to produce his/her student-ID.

2. READ AND OBSERVE THE FOLLOWING RULES:

- No candidate will be permitted to ask questions of the invigilators except in case of supposed errors or ambiguities in exam questions.
- No candidate will be permitted to enter the examination room after the expiration of one half hour, or to leave during the first half hour of the examination.
- No candidate shall be permitted to leave during the last half hour of the examination. Please remain in your seat until your paper has been collected.
- Smoking is not permitted during examinations.

3. You have 2 hours to complete this exam.

4. **Give concise answers. Long answers will be looked down upon with disregard - you will get points deducted for fluff. Put all your answers on the exam. Do any rough work on the back of the exam pages. All rough work must be handed in.**

1) Compiler Architecture (10 points)

Draw a diagram of the standard architecture of a compiler. Label each element of the architecture with the name of 1 class in the Mini AspectJ compiler that serves to implement that element.

2) AST Structure (10 points)

This problem has two parts, which you will find on the next two pages.

In this problem you will show the abstract syntax tree (AST) and the AST class hierarchy for the pointcut sub-language of your AspectJ extension to the Mini-Java compiler. The design you show should be for the following language of pointcuts:

```
<pointcut> ::= call(<method signature>)
               execution(<method signature>)
               get(<field signature>)
               set(<field signature>)
               within(<type pattern>)
               cflow(<pointcut>)
               cflowbelow(<pointcut>)
               this(<variable name>)
               target(<variable name>)
               args(<variable name>..)
               <pointcut> && <pointcut>
               <pointcut> || <pointcut>
               ! <pointcut>
```

(A) Show the AST class hierarchy you would use (or did use in your compiler) to support the above grammar. Include enough classes in your hierarchy so that you get all the way down to strings or other primitive tokens. To save space, you can leave FieldSignature, set, cflowbelow, || and ! out of the actual drawing.

(B) Show the AST for the following pointcut expression. Use syntax similar to what toLongString returns.

```
within(Foo) && call(void Point.setX(int)) && this(o)
```

3) Code Generation (10 points)

Consider the following class. In the space at the right, show Java byte codes for the **body** of each method. You should assume that getY and setY exist, but you don't need to show the code for them.

```
class Point {

  int x, y;
  Point buddy;

  int getX() {
    return x;
  }



  void setX(int nx) {
    x = nx;
  }




  void foo() {
    buddy.setX(getX());
    buddy.setY(getY());
  }
}
```

4) Phases of Compilation (25 points)

For the following typical phases of compilation, explain what each phase does and why that is important. In each case, use a simple concrete example to help make your answer clear.

Scanning and Parsing

Name Binding

Type Checking

Resource Allocation

Code Generation

Optimization

Emitting

5) Compiling Pointcuts and Advice [15 points]

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

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 use the next page to draw a simple diagram to help show those differences and connections.

join point

pointcut

advice

join point shadow

advice implementation method

6) Using AspectJ to Write a Compiler (15 points)

At the beginning of the semester, we hoped to be able to use AspectJ in the actual MAJ compiler implementation. Unfortunately, we never got a chance to do that, because the AspectJ 5 implementation was delayed.

Nonetheless, you should have some good ideas about how using AspectJ in your compiler would have improved the architecture – made it more modular, more elegant etc.

Describe 3 specific examples of how you could use AspectJ to improve the MAJ compiler. (Either the basic one we gave you, or the one you handed in as your final project). You do not necessarily need to write code, but do be specific as to exactly what would be re-implemented using aspects. You may use any AspectJ features you like.

7) Byte Code Advice Weaving (15 points)

Sally and Ben took 411 before graduating, and now they want to use AOP pointcuts and advice in a project at their company. But their company uses its own special Java like language – called Peet's – so they cannot use AspectJ or any of its variants.

So they have decided to integrate pointcuts and advice into Peet's. They call their new language Forte. To make Forte most useful, they would like to provide *byte code weaving*. That is, they want to compile code with advice declarations in it into byte codes first, write that out to files, and then later be able to weave those files with ordinary files containing byte code for classes. This means they can separately compile files with advice and ordinary classes, and then right before loading, weave them together.

Sally and Ben realize that the first step in doing this is to simply extend the Peet's compiler to compile advice into byte codes. This involves simply parsing advice declarations with pointcuts, converting the advice to helper methods, and then storing somewhere in the file the associations that say that the helper methods should be called before/after (ignore around) the join points matched by the pointcuts. (Basically the allAdvice list for the file.) This simple extension to the Peet's compiler lets them write out byte code files with advice in them.

But now comes the harder part. They must build a weaver that weaves the byte codes and writes out new woven byte code files. Fortunately, the byte codes for Peet's look exactly like Java byte codes!

Describe the architecture of the byte code weaver Sally and Ben must build. For each element of the architecture, describe the salient points that must be handled. Your answer need not be an essay. It can just be a list of the key points, in a coherent order, perhaps supplemented with figures and small amounts of code. You will be graded on coverage of the key issues, as well as on clarity of the overall answer.