CSE 401 Final Exam

March 19, 2009

Name ________Sample Solution________

The exam is closed book, closed notes, closed electronics, closed neighbors, open mind, ...

Please wait to turn the page until everyone has their exam and you have been told to begin.

If you have questions during the exam, raise your hand and someone will come to you.

Legibility is a plus as is showing your work. We can't read your mind, but we'll try to make sense of what you write.

1	/ 10
2	/ 8
3	/ 8
4	/ 10
5	/ 8
6	/ 9
7	/ 8
8	/ 15
9	/ 12
10	/ 12
Total	/ 100

Question 1. (10 points) The original version 1.0 of Java used an interpreter in the Java VM to execute programs. Later versions of Java added a "just-in-time" compiler, which translated the Java intermediate code to native machine code (x86, powerpc, or others) on the fly, and executed the translated code instead of interpreting the intermediate code directly.

(a) Give one plausible reason why Java 1.0 only included an interpreter, and not a just-in-time compiler. Why was this a reasonable engineering decision?

There were at least two good reasons:

- **Faster to implement. No need to supply a full compiler back-end.**
- **Portable. Java was easier to move to different architectures because only the interpreter and core parts of the runtime environment needed to be re-implemented on the new machine.**

(b) Give one plausible reason why later versions of Java include a just-in-time compiler in addition to the interpreter. Why was this decision made, particularly since the interpreter already existed and already executed programs correctly?

Execution speed. Java programs run by the interpreter are significantly slower than programs compiled to native code, or equivalent C/C++ programs translated by a conventional compiler.

Question 2. (8 points) In class we looked at the *visitor pattern*, which was a way of organizing code for the type checker and other AST operations. This involved adding an `accept` method with a visitor object parameter to each AST node type, and each visitor object had a collection of `visit` methods, one for each type of AST node.

Although this is a somewhat complicated scheme, it is widely used in compilers and other tools implemented in object-oriented languages. Why is this a useful way to organize the code, and what problem is it trying to solve?

It solves a modularity problem. If operations like type-checking are spread across all of the individual AST nodes, then adding or modifying an operation requires changing multiple AST classes. The visitor pattern allows us to group all of the methods for something like type-checking into a single module.

Question 3. (8 points) The MiniJava project compiler includes separate symbol tables for different scopes, such as a global symbol table for class information and separate symbol tables for individual methods. The compiler makes several passes over the AST to fill in these tables.

Is it really necessary to make several passes over the AST to collect this information? Is there some technical reason that this couldn't be done in a single pass?

Yes. Java allows programs to use classes and methods before their declarations have appeared in the program. We need multiple passes to collect declaration information before we can check uses of things to be sure they are correct.

Question 4. (10 points) It is certainly possible to generate code for a target machine by making a pass over the AST and generating code directly. This is actually done in some simple compilers. But in most production compilers the AST is lowered to create a lower-level IL program, and that is used to generate target code, as in MiniJava.

Give **two** different reasons why generating a lower-level IL version of the program is a useful thing to do in a compiler (not just in MiniJava).

Reason 1:

The IL code is usually closer to the target machine and it is easier to generate code from it compared to generating code directly from the AST.

Reason 2:

Many optimizations are easier or more effective on IL code.

Question 5. (8 points) The lower-level IL code in MiniJava and in most compilers creates and uses an unbounded number of temporary variables (`t1, t2, t3, ...`) for intermediate values, instead of using the registers of the target machine in the IL code. Why? Why not just use the target machine registers directly? (There are at least a few reasons – you only need to give one.)

There are at least a couple of good reasons:

- **It is easier to generate and optimize IL code if we have as many temporaries as we need.**
- **It avoids tying the IL code and parts of the compiler that process it (including most of the optimizer) to the peculiarities of a particular machine register set or architecture.**

Question 6. (9 points) x86 hacking, part I. Suppose x is an integer variable located at offset +12 from the stack frame base register ebp (i.e., 12(%ebp), or [ebp+12] in intel notation). Write three different x86 instruction sequences that implement the assignment x=x+1 (i.e., three different ways to increment the value of x and store the incremented value back in memory). You can use any registers you wish, and you can use either intel or gnu assembler syntax (but don't mix the two assembler syntaxes together). The sequences must use different instructions or have instructions in a different order – it is not ok to just repeat the same exact instructions but using different registers. It is also not ok to add useless instructions that do nothing just to make the code different.

Code for x=x+1, given that x is located at ebp+12:

There are many possibilities. Here are several (using Intel syntax):

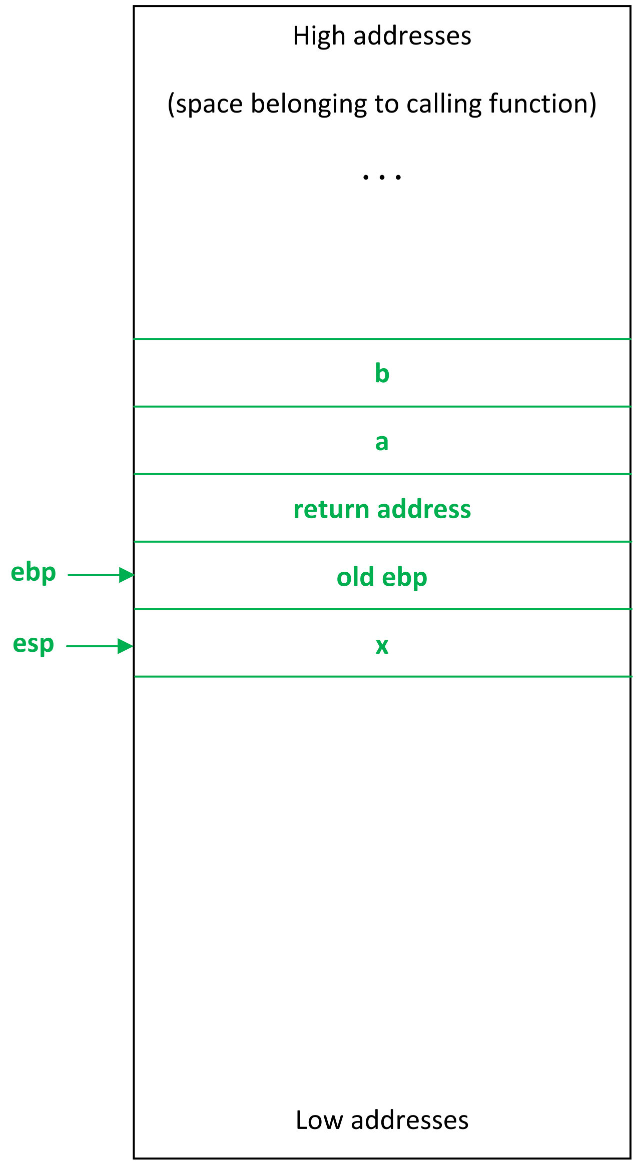
```
inc   [ebp+12]

mov   eax,[ebp+12]
inc   eax
mov   [ebp+12],eax

mov   eax,[ebp+12]
add   eax,1
mov   [ebp+12],eax

mov   eax,1
add   eax,[ebp+12]
mov   [ebp+12],eax

mov   eax,1
mov   ebx,[ebp+12]
add   eax,ebx
mov   [ebp+12],eax

mov   eax,[ebp+12]
lea   eax,[eax+1]
mov   [ebp+12],eax
```

One that is not possible is add [ebp+12],1. The trouble is that an x86 instruction can only contain a single immediate constant or displacement. So it can contain either the offset 12 or the value 1, but not both.

Question 7. (8 points) x86 hacking, part II. Consider the following C function.

```
int foo(int a, int b) {
  int x;
  // draw frame at this point in execution
  x = a+b;
  return x;
}
```

In the space below draw a picture of the stack frame for function `foo` right before execution of the assignment statement in the body of the function. Your picture should show where the parameters and variables are located, as well as any additional items that are part of the stack frame for `foo`, such as the return address. You should also draw labeled arrows showing where in the stack frame the registers `ebp` (frame pointer) and `esp` (stack pointer) point to.



Question 8. (15 points) Having successfully added arrays to MiniJava, we'd now like to add Java's for-each statement to our compiler. In full Java, the for-each statement iterates through an array or a collection. For this question we only want to implement it for arrays.

The syntax of a for-each statement is

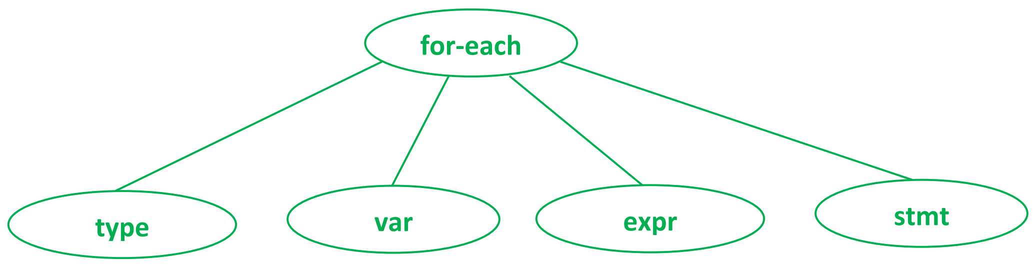
```
for ( type  loop-var : array-expr ) statement
```

The meaning is to execute the statement repeatedly with *loop-var* assigned to successive values from the *array-expr* beginning with element 0. The *loop-var* is declared in the for statement and it has the given *type*, which must match the type of the array elements.

For example, if v is an array of integers, then the following statements would add up the elements of the array v and store the result in `sum` and the number of array elements in `count`:

```
sum = 0; count = 0;
for (int x : v) { sum = sum+x; count = count+1; }
```

(a) Draw the AST node that you would need to add to the compiler to support this new for-each statement.



(b) What changes would need to be made to the semantic analysis part of the compiler to handle the new for-each statement?

- **Verify that the expression has an array type**
- **Verify that the array element type matches (or is assignment-compatible with) the variable type**
- **Add the loop variable to an appropriate symbol table with the declared type**

(continued next page)

Question 8. (cont.) For-each syntax repeated for reference.

```
for ( type  loop-var : array-expr ) statement
```

(c) What IL code would be generated for a for-each statement? To answer this question, you should use the IL code as shown in lecture (e.g., `t0=t1+t2;  iffalse t17 goto L;`), not the specific IL class instances used in the MiniJava project code (`ILExprWhatever`).

A copy of the slides summarizing the MiniJava's IL code is attached as the last page of this exam for reference. You will not be graded on whether your code conforms strictly to the IL syntax, but it should be basically the same operations, and at the same semantic level.

There were, of course, various ways to solve the problem, and we also were flexible on the notation as long as it captured operations at the IL level. Here is one solution.

```
t0 = array-expr -> length;    # loop bound

t1 = 0;                       # loop counter

label test;

t2 = t1 < t0;

iffalse t2 goto done;

loop-var = array-expr -> items[t1];

<code for statement goes here>

t1 = t1 + 1;

goto test;

label done;
```

A couple of solutions also included a test to check that array-expr was not null, which should be done in a full implementation. But we didn't mark off if that was not included.

Question 9. (12 points) For this question we'd like to perform local constant propagation and folding, and dead assignment elimination on the IL code generated for the following statements:

```
x = 10;
i = 3;
a[i] = x;
```

Assume that x, i, and the array a all contain 4-byte integer variables.

In the table below, the first column shows the original IL code generated for these statements.

(a) Fill in the second column with the statements from the first column after local constant propagation and folding (compile-time arithmetic) have been performed. You should assume that constant propagation tracks the contents of variables stored in the stack frame as well as temporaries like t2.

(b) In the third column, check the box "delete" if the statement would be eliminated by dead assignment elimination after performing the optimizations in part (a). You should assume that the variable x and the array a are *live* on exit from the block of code, and the variable i is *not live* on exit.

Original code	After constant propagation and folding	Deleted by dead code elimination? (mark X if so)
t1 = 10	**t1 = 10**	**X**
*(fp + xoffset) = t1	***(fp + xoffset) = 10**	
t2 = 3	**t2 = 3**	**X**
*(fp + ioffset) = t2	***(fp + ioffset) = 3**	**X**
t3 = *(fp + xoffset)	**t3 = 10**	**X**
t4 = *(fp + ioffset)	**t4 = 3**	**X**
t5 = t4 * 4	**t5 = 12**	**X**
t6 = t5 + fp	**t6 = 12 + fp**	
*(t6 + aoffset) = t3	***(t6 + aoffset) = 10**	

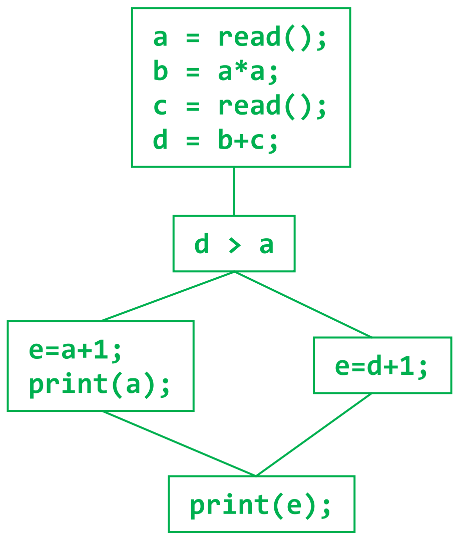
Reminder: x and the contents of array a are live on exit; i is not live on exit.

A few people noted that if the compiler were a bit cleverer, it could recognize that aoffset+12 could be computed at compile time, and then the assignment to t6 would be dead and could be eliminated.

Question 10. (12 points) A little coloring. Considering the following code fragment:

```
a = read();
b = a*a;
c = read();
d = b+c;
if (d > a) {
  e = d+1;
} else {
  e = a+1;
  print(a);
}
print(e);
```

(a) Draw the control flow graph for the code, keeping the diagram to the left side of the paper.



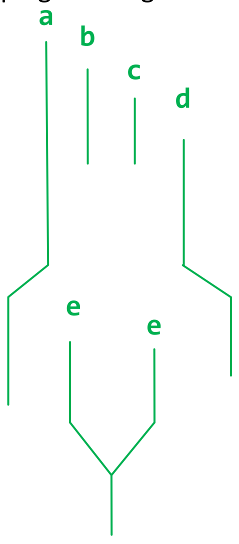
```
a = read();
b = a*a;
c = read();
d = b+c;
```

```
d > a
```

```
e=a+1;
print(a);
```

```
e=d+1;
```

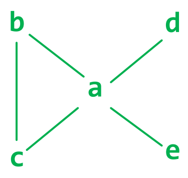
```
print(e);
```



(b) To the right of the control flow graph, neatly show the live ranges of the variables.

(c) Below, draw the interference graph for the variables. Use the left side of the paper.

Notes: b and c do not interfere with d since they are dead after b+c is computed. d does not interfere with e for a similar reason. Three registers are needed, but there are several possible assignments.



r1: a;	**r2: b, d, e;**	**r3: c**	**or**
r1: a;	**r2: b;**	**r3: c, d, e**	**or**
r1: a;	**r2: b, d;**	**r3: c, e**	**or**
r1: a;	**r2: b, e;**	**r3: c, d**	

(d) To the right of the interference graph, indicate which groups of variables can occupy the same register, based on the information in the interference graph. You do not need to go through the steps of the graph coloring algorithm explicitly, although it may be helpful as a guide to assigning registers. If there is more than one possible answer that uses the minimum number of registers, any of them will be fine.