ECE 468 — Midterm 2

November 4, 2010

Name: _____

Purdue email: _____

Please sign the following:
I affirm that the answers given on this test are mine and mine alone. I did not receive help from any person or material (other than those explicitly allowed).

X _____

Part	Points	Score
1	30	
2	20	
3	15	
4	35	
Total	100	

Part 1: Semantic actions and functions (30 pts)

1) Explain when pass-by-value will produce the same results as pass-by-reference (2 pts):

No partial credit.
Note: a few people brought up passing arrays. The key with arrays is that they are *always* passed by reference in C/C++, not that pass-by-reference and pass-by-value do the same thing. You could design a language that could pass arrays by value (i.e., by copying them into the arguments), in which case pass-by-value would differ from pass-by-reference.

2) a) Give one reason that caller-saves is better than callee-saves (2 points). b) Give one reason that callee-saves is better than caller-saves (2 points).

No partial credit

3) Draw the AST that would be generated for the expression `x := y + z * w` and for each AST node, show the data object that would be generated. For each data object, show the code that would be generated (if any), the temporary the result will be stored in, and whether the temporary is an R-value or an L-value. (14 pts)

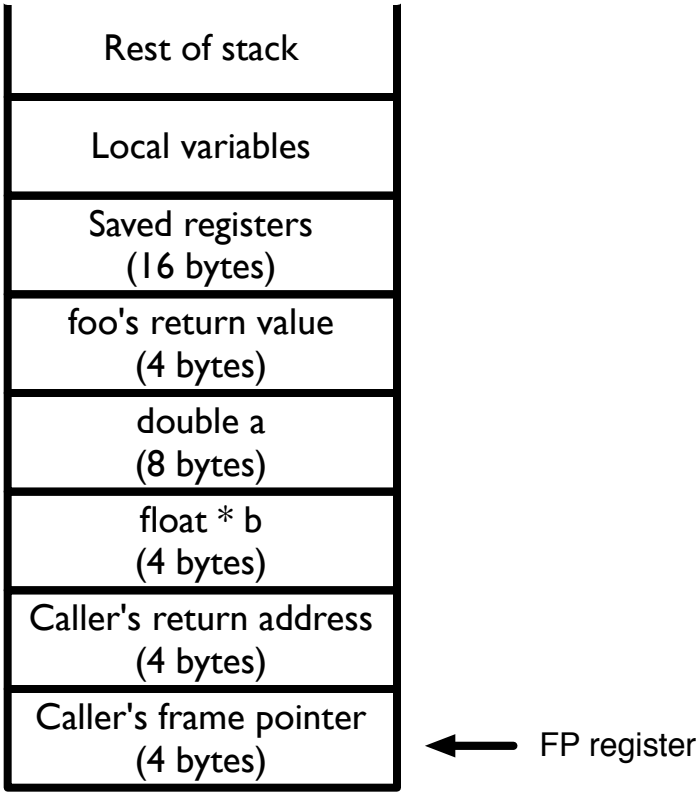
7 AST nodes, 1 point per node, 1 point per piece of code. Don't forget to add in the AST nodes for "x :="!

Note that a lot of people loaded from variables immediately upon seeing them (as opposed to passing them up as L-values). If you do this, *the temporary you load into is an R-value, not an L-value*. I took off one point per instance of this.

-2 pts if your AST didn't observe the correct order of operations

Many people swapped the operands of the '+' operator, building an AST for "z * w + y". While this works for '+', this is not actually correct—consider what would have happened if the '+' were a '–'. I took off one point for this.

3) Here is a partial stack of a method being executed (the stack grows down). Show the stack after calling `int foo(double a, float* b)`. Show the frame pointer, and note how much space each part of the stack occupies (32-bit ints and pointers, 64-bit doubles). Assume that we are using a *caller-saves* convention, and that the machine has 4 registers (plus FP & SP registers). Assume that there are no nested scopes or local variables in foo. (10 pts)



-2 points per missing box.
-1 point for incorrect sizes.
-1 point for box in wrong place.
-2 points for missing FP.
-2 points for registers in the callee-saves vs. caller-saves position.
(up to a maximum of -10 points)

Note: the Tiny calling convention puts the registers after the return value and arguments, so I accepted that ordering of the stack as well.

Part 2: Code generation (20 points)

For the next problems, consider a for-loop:

```
for (<init_stmt>; <test_exp>; <incr_stmt>) {<body>}
```

Now consider a new statement, akin to a continue statement:

```
skip_one;
```

This statement *skips one iteration* of a for-loop. Rather than evaluating the incr_stmt once, like continue, it evaluates it *twice*. If either the skipped iteration or the next one would have broken out of the loop (because test_exp evaluates to false), then the loop should terminate. Otherwise, the loop will continue, having *skipped one iteration*. Think of skip_one as a continue statement that skips past the end of the current iteration and the next one, as well.

1) Using a format like we did in the class slides, show what code you would generate for a for-loop in order to support the skip_one statement (16 points):

Missing test_expr after incr. of skip_one: -2
Running two incr_stmts on every iteration (not just when calling skip_one): -4
Using a flag to implement skip, instead of control flow: -2
Other sorts of non-working code: -10

2) Using your code for part 1 as a guide, show what code you would generate for a skip_one statement in the body of the for-loop (2 points):

No partial credit

3) Is there an optimization that could make the code generated for a for-loop more compact? If so, describe it (2 points):

This problem seemed to be confusing for a lot of people. Everyone got 2 extra points as a result (even if you got it right to begin with).

Part 3: Common subexpression elimination (15 pts)

For the next questions, consider the following piece of code:

```
1: A = B + C;
2: B = A * Q;
3: P = B + C;
4: A = A * Q;
5: E = B + D;
6: C = A * P;
```

1) Assume there is no aliasing between variables. For each statement, list which expressions are "available" *after* the statement executes (6 pts)

1 point per entry.

2) What does the code look like after performing CSE (when eliminating a redundant expression, replace it with the variable that holds the calculated value of the expression) (5 pts)

-1 point per incorrect statement, up to a maximum of 5 points

3) There are two variables which, if they were aliased, would make one additional expression redundant. Which are they (4 pts)?

No partial credit.

Part 4: Register allocation (35 pts)

For the next problems, consider the following code:

```
1:  LD A, T1
2:  LD B, T2
3:  LD C, T3
4:  T4 = T1 * T2
5:  T5 = T1 + T3
6:  LD D, T6
7:  T7 = T2 + T5;
8:  T8 = T6 + T7;
9:  T9 = T1 + T7;
10: ST T9, A
```

1) Show which temporaries are live *after* each instruction (10 pts)

1 point per entry

2) Assume we have a machine with 3 registers available (not including spill registers). What temporaries get assigned to registers if we perform *top-down* register allocation. If there is a tie in the algorithm, choose the lowest numbered temporary. (5 pts)

No partial credit.

Many people counted the number of instructions during which a temporary was *live*, not the number of instructions in which a temporary was *used.*

3) Perform bottom-up register allocation on this piece of code. At each instruction, show which temporary is assigned to which register *after* the instruction is executed (if a register is freed, mark it as such even if it still holds a value). When a register needs to be spilled, pick the one whose value is next used the farthest away. If there is a tie, pick the lowest numbered register. If multiple registers are free when allocating registers, choose the lowest numbered one. Indicate where loads and stores due to spills happen (20 pts)

0.5 points per box (must be consistent with answers to #1)