SPRING 1994
COMPUTER SCIENCES DEPARTMENT
UNIVERSITY OF WISCONSIN—MADISON
PH. D. QUALIFYING EXAMINATION

Programming Languages
Depth Examination

Wednesday, February 9, 1994
3:00 – 7:00 PM

GENERAL INSTRUCTIONS:

1. Answer each question in a separate book.
2. Indicate on the cover of *each* book the area of the exam, your code number, and the question answered in that book. On *one* of your books list the numbers of *all* the questions answered. *Do not write your name on any answer book.*
3. Return all answer books in the folder provided. Additional answer books are available if needed.

SPECIFIC INSTRUCTIONS:

Answer *five* of the *six* questions.

If you are unable to provide a complete solution to a question, you can earn partial credit by explaining techniques for solving such problems, relating these problems to known similar problems, *etc*, so please give your best answer for the required number of questions.

For questions that require writing code, any readable pseudo code is acceptable, and you may use operations on standard data structures (*e.g.*, stacks) without writing code for the operations.

POLICY ON MISPRINTS AND AMBIGUITIES:

The Exam Committee tires to proofread the exam as carefully as possible. Nevertheless, the exam sometimes contains misprints and ambiguities. If you are convinced a problem has been stated incorrectly, mention this to the proctor. If necessary, the proctor can contact a representative of the area to resolve problems during the *first hour* of the exam. In any case, you should indicate your interpretation of the problem in your written answer. Your interpretation should not make the problem trivial.

Question 1

A widely-used compiler optimization involves moving *loop-invariant expressions* from the body of a loop to the loop's preheader.

Part A.

Outline how we could determine if an expression written in C is loop-invariant.

Part B.

Potential optimizations are judged by their *safety* and *profitability*. Under what circumstance can we be sure moving a loop-invariant expression is safe? Under what circumstance can we be sure moving a loop-invariant expression is profitable?

Part C.

Assume we assign each loop-invariant expression a cost value that measures its cost of evaluation (in instructions or micro-seconds). Assume further that we have n registers available in a loop to hold invariants. Explain how to use cost values and loop structure to decide which invariants to hold in registers. Under what circumstances is it profitable to move a loop-invariant that can't be held in a register?

Question 2

Consider a small programming language that contains the following type of statements:

variable declarations
if-then-else
while-do
assignment

Further assume that the language has only scalar integer variables and has the usual complement of arithmetic operations.

Part A.

Write the dataflow equations for the reaching-definitions problem in this language.

Part B.

Suppose we add a goto-statement and labels to this language. Write dataflow equations for reaching-definitions in this extended language.

Part C.

Suppose we introduce pointer variables (which can point only to integer variables). Explain how this new feature complicates the reaching definition problem and how you might solve the problem.

Question 3

Memoizing is an optimization that is frequently used in functional languages.

Part A.

Explain what memoizing is and give an example of its utility.

Part B.

Are all C functions suitable for memoizing? If not, explain how to select functions that might be memoized.

Part C.

Even is a function isn't suitable for memoizing, some execution paths through it may be. Explain how we could determine, at run-time, that a particular execution could be memoized.

Question 4

This is a question about inheritance in object-oriented languages. Consider an example:

```
object animal is
  name : string;
  age : integer;

  procedure print ();
  function int age();
  function bool is_in_zoo ();
end;

object dog inherit from animal is
  function bool is_in_zoo ();
end;
```

where dog is an object that inherits fields and methods from animal objects and redefines a method (is_in_zoo).

With a static view of inheritance (similar to C++'s non-virtual functions), if the print function in animals invokes the is_in_zoo function, then it always invokes the function for animals, even if the animal is a dog.

Part A.

Why is this a good idea? Why is it a bad idea?

With a more dynamic view of inheritance (such as the one in Smalltalk, or C++s virtual functions), invoking the print function in animals on a dog would invoke the is_in_zoo function for dogs.

Part B.

Why is this a good idea? Why is it a bad idea?

Part C.

Describe how you would implement this dynamic form of inheritance at run-time (as is done in Smalltalk).

Part D.

Describe how you would implement this dynamic form of inheritance at compile-time (as is done in C++).

Question 5

Debuggers such as GDB allow a variable to be *watched.* That is, whenever the variable gets a new value, the user is informed. Debuggers typically operate on already-compiled programs (a.out files), typically supplemented with extra debugging information (symbolic names, addresses and sizes for variables and subprograms, etc).

A watch of a variable v ought not to slow execution of parts of the program not involving v (though statements that do manipulate v likely will be slowed).

Assume we have the assembly language file from which an a.out file was created. This assembly language file was created by a non-optimizing C compiler.

Part A.

Assume we wish to watch some global scalar variable, g. How can we transform the program to efficiently watch for changes to g?

Part B.

Now assume we wish to watch some local scalar variable, l. How can we transform the program to efficiently watch for changes to l?

Part C.

Now assume we wish to watch some global scalar array, a. How can we transform the program to efficiently watch for changes to a?

Part D.

How must your answers to parts A to C be changed if an optimizing C compiler is used. That is, what optimizations would complicate or invalidate the watching schemes you have proposed?

Question 6

Consider the problem of scheduling instructions for machines with delayed branches. Assume that conditional and unconditional branches have 2 delay slots after them, so each branch executes the next two instructions before the control-transfer occurs. (For simplicity, assume that neither of these two instructions can transfer control.) For example, given

```
beq r4, r5, foo
add r1, r2, r3
lw r1, 5(r7)
```

the branch executes the add and load (lw) instruction before jumping to the instruction labeled foo.

The compiler initially generates the instructions under the assumption that branches have no delay slots and then passes the instructions to the scheduler, which attempts to fill the delay slots. If the instruction scheduler cannot find instructions to put in this delay slot, it must fill it with nop's.

For example, the compiler may have generated

```
add r1, r2, r3
lw r1, 5(r7)
beq r4, r5, foo
```

initially, but the scheduler produced the code above.

Part A.

Write an algorithm to schedule instructions generated by a compiler. Assume that you are given the unscheduled sequence of instructions produced by the compiler and have constructed a control-flow graph from it.

Part B.

Suppose the computer had an *annulled delayed branch* that "squashes" (does not execute) the instructions in the delay slot if the branch is *not* taken. For example, given

```
beqa r4, r5, foo
add r1, r2, r3
lw r1, 5(r7)
li r2, 9
```

where beqa is an annulled branch if equal, the add and lw are executed prior to the branch to foo if r4 is equal to r5. If they are not equal, execution continues with the li instruction.

How would this change your scheduling algorithm to handle annulled delayed branches?