Fall 1993

COMPUTER SCIENCES DEPARTMENT

UNIVERSITY OF WISCONSIN–MADISON

PH. D. QUALIFYING EXAMINATION

Programming Languages Depth Examination

September 20, 1993

GENERAL INSTRUCTIONS:

Answer each question in a separate book.

Indicate on the cover of each book: the area of the exam, your code number, and the question answered in that book. On one books, list the numbers of all questions answered.

Do **not** write your name on any answer book.

Return all answer books in the folder provided. Additional answer books are available if needed.

SPECIFIC INSTRUCTIONS:

Answer all 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 a problem to known similar problems, etc.; so please give your best answer for all required questions.

For questions that require writing code, any readable pseudo code is acceptable. You may use operations on standard data structures (e.g., stacks) without providing 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, inform 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.

1. In almost all programming languages, a function call may appear as an *r-value* but not as an *l-value*. Thus:

```
a = f(x);
```

is allowed, but

```
f(x) = a;
```

usually is not permitted.

(a) For references to variables and components of data structures, a compiler normally computes an address (an *l-value*). If an *r-value* is required, the *l-value* is simply dereferenced. Explain the problems that would ensue if we generalize functions to compute and return *l-values* rather than *r-value*s.

(b) An alternative to the approach in part (a) is to implement two versions of a function, one for value contexts and the other for use in *l-value* contexts. Illustrate this approach by outlining the two functions that you would use to define `sqrt(x)`. Again, what are the limitations of this, two-definition approach?

(c) To make the two-definition approach of part (b) work, we need to automatically (at compile-time) deduce whether a function call is in an *l-value* context or an *r-value* context (so the correct function can be called). Outline how this analysis could be done in Pascal or C. What would you do when a function is passed as a *var* parameter?

2. C allows a programmer to designate certain variables as "register" variables, which should reside in registers rather than memory locations.

(a) Assume that while compiling a procedure `P`, a compiler sees N local variables marked as register variables. If N registers are available, the programmer's directive is easy to follow. If N registers are not available, explain how a compiler could decide if all N local variables could be kept in registers within `P`.

(b) Normally when a value resides in a register, it must be saved and restored across procedure calls. Thus if N is large, calls within `P` may become expensive. Outline the analyses that a compiler might perform on `P` and the routines that it calls avoid unnecessary saving or restoring of registers.

3. Many older architectures provide two rather than three operand instructions in which one of the operands is overwritten with the result. Assume an architecture of this design, with the following instructions to load, store, and compute values:

```
Load Ri, Adr    # Load word at address Adr into register Ri
Store Ri, Adr   # Store word in register Ri at address Adr
 Op Ri, Rj      # Compute Ri Op Rj and store result in Rj
```

A problem with this type of architecture is that an operand that a compiler would like to preserve is sometimes unavoidably overwritten. A general solution is to provide three operand instructions, but sometimes an instruction format does not have enough bits to accommodate the third operand.

Another alternative is "pseudo" three address instructions that put the result of an operation in a third register, which is determined by applying a simple rule to the specified operand registers. A simple rule is:

```
Op  Ri,Rj        # Compute Ri Op Rj and store result in Rj+1
```

That is, the result of the operation is put into the register immediately following `Rj`, `Rj+1`. If `Rj+1` doesn't exist, the result is put into Rj.

This innovation is attractive since the operand register need not be destroyed. However, it is not clear that code as efficient as previously possible can always be obtained. Assume that we are translating expressions involving binary operators and distinct integer variable operands (i.e., no common subexpressions). For the original, two-operand architecture we know how to find the optimal translation of any such expression using the Sethi-Ullman algorithm.

For our proposed new architecture, the question is whether translations as efficient as those possible under the old architecture can always be realized. Let E be any expression involving only binary operators and distinct integer variable operands. If E can be optimally translated into I instructions using R registers in our old architecture, can E always be translated using no more than I instructions and R registers in our new architecture? Consider two cases:

(a) The operators can be commutative or non-commutative.

(b) The operators are all commutative.

4. Consider adding assertions to the C language. An assertion is a precondition consisting of a predicate and a statement *S* (which can be a block containing many statements). The predicate is tested **before** each statement in *S*. If the predicated evaluates to false, the program terminates with an error message. To be more concrete, consider a new statement with the syntax:

```
<stmt> -> assert ( <expression> ) <stmt>
```

For example:

```
assert (x > 4) x = f(g(1));

assert (y != 0 && z > y) {
 x = f (x);
 y = 2 * x;
 z = 2 * y;
}

for (i = 0; i < 100; i++)
 assert (g(i))
     a[i] = 2*i;
```

(a) Describe how a compiler could implement assertions in a straight-forward manner.

(b) This approach could be improved by only re-evaluating parts of the assertion's predicate between statements. Provide an algorithm to choose the portion of the predicate that must be evaluated between statements.

(c) Another use for assertions is to aid a compiler in producing better code. Describe three assertions that a programmer could add to a program to aid the compiler's optimizer and explain how the compiler would use the information in these assertions.

5. Consider the problem of adding prefetch operations to a program. A prefetch is similar to a load instruction. It produces an address, which is given to the memory system. If the addressed location is in the cache, nothing happens. However, if the addressed location is not in the cache, the memory system brings the location into the cache, just as if the location had been referenced by a load instruction. The difference between a prefetch and a load is that a prefetch always completes in a single cycle, even if the data is not in the cache, and a prefetch does not bring data into a processor register.

To make this discussion concrete, assume the following parameters. A prefetch and load instruction each take a single operand (the address) and normally complete in a single cycle. If the location reference by a load is in the cache, the value is loaded into a register at the end of the following cycle. If the location is not in the cache, the processor stalls for 20 cycles while it is loaded from main memory into the cache.

(a) Describe how a compiler could insert prefetch instructions into a program to hide some of the memory latency. For simplicity, concentrate on loops in programs that manipulate dense arrays. Be sure to discuss how the compiler decides where to place a prefetch instruction and what code must be produced for each prefetch.

(b) What additional costs are introduced by these prefetch operations? How can the compiler minimize these costs? Give a formula that the compiler can use to decide if the cost of a prefetch outweighs its benefits.

6. (NOTE: this page defining dataflow analysis frameworks was copied from the Spring '92 exam.)

Dataflow analysis can be performed on a flowgraph representation of a program. A flowgraph includes two special nodes: **Entry** and **Exit**; all other nodes represent the statements and predicates of the program. The edges of the flowgraph represent the program's flow of control. For every node n, there is a path from the **Entry** node to n, and there is a path from n to the **Exit** node.

The goal of dataflow analysis is to associate two values, n_{in} and n_{out}, with every node n in a program's flowgraph. The value n_{in} represents information that is guaranteed to be valid just before the execution of node n and the value n_{out} represents information that is guaranteed to be valid just after the execution of node n.

Dataflow analysis problems can be defined using a **dataflow analysis framework**, which consists of the following three parts:

(1) A lattice L with meet operation $\wedge$, least element *bottom* (**bot)**, greatest element *top* (**top**), and partial order **<**. (Lattice elements are the values that are to be associated with flowgraph nodes.)

(2) A set of monotonic functions F of type $L \rightarrow L$. (Every flowgraph node n other than **Entry** and **Exit will have** an associated function f_n.)

(3) A value v, an element of the set {**bot**, **top**}. For forward problems, v represents the information that is true before the program starts executing, and v is associated with $\textbf{Entry}_{\text{out}}$. For backward problems, v represents the information that is true after the program finishes executing, and v is associated with $\textbf{Exit}_{\text{in}}$.

Given: (1) a framework as described above for a particular dataflow analysis problem, (2) a flowgraph G, and (3) a mapping from the set of functions F to the nodes of G, the solution to the dataflow problem can be found by finding the greatest fixed point of the following system of equations:

FOR FORWARD PROBLEMS	FOR BACKWARD PROBLEMS
for all nodes n other than **Entry**: $n_{in} = \wedge_{p \in predecessor(n)} p_{out}$	for all nodes n other than **Exit**: $n_{in} = f_n(n_{out})$
$n_{out} = f_n(n_{in})$	$n_{out} = \wedge_{s \in successor(n)} s_{in}$

For example, the reaching-definitions problem (a forward problem) can be defined using the following framework:

(1) Elements of lattice L are sets of pairs (x, n), where x is a variable and n is a flowgraph node that represents a statement that defines variable x (e.g., an assignment or read statement). The set n_{in} associated with node n is the set of definitions that reach node the beginning of node n. The meet operation for L is set union; the top element is the empty

set; the bottom element is the cross product of the set of all variables and the set of all nodes.

(2) The function associated with a node n that defines variable x is of the form:

λ *s.* $((s - \{\textit{all pairs with first element } x\}) \cup \{(x, n)\})$

The function associated with a node that does not define any variable is the identity function.

(3) The value **Entry**$_{\text{out}}$ is the empty set.

(a) Consider a language that includes assignment statements, **if-then-else** statements, **while** loops and two kinds of procedure calls: **normal** procedure calls and **indirect** procedure calls (defined below). In this language, variables are either of type **integer** or of type **procedure**. If a program includes a procedure **foo**, and variables x and y, both of type procedure, then all of the following are legal statements:

```
x := foo       /* assign literal to variable of type procedure */
y := x         /* assign one procedure variable to another */
call foo       /* a normal procedure call */
call using x   /* an indirect procedure call */
```

Assume procedures have no parameters and there are no global variables. Therefore, a call to a procedure cannot change the value of any non-local variable.

Consider the problem of determining, for an individual procedure written in the language discussed above, which procedures might be called at each indirect call statement (i.e. for every statement of the form "call using x," what are the values that variable x might have at that point in the program?). Is this a **forward** or a **backward** problem? Give a dataflow analysis framework as defined above for the problem.

(b) It has been shown that if the functions specified as part (2) of a dataflow analysis framework are all **distributive**, then the greatest fixed-point solution will also be the meet-over-all-paths solution to the problem. (A function f is distributive iff for all:

$l_1, l_2 \in L, f(l_1 \wedge l_2) = f(l_1) \wedge f(l_2)$.)

For every function that you defined in Part A, either show that the function is distributive, or give an example showing that it is not distributive.