FALL 2004
COMPUTER SCIENCES DEPARTMENT
UNIVERSITY OF WISCONSIN—MADISON
PH. D. QUALIFYING EXAMINATION

Programming Languages and Compilers
Qualifying Examination

Monday, September 20, 2004

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 4 of 5 questions.

POLICY ON MISPRINTS AND AMBIGUITIES:

The Exam Committee tries 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 be such that the problem is nontrivial.

Question 1.

This question concerns how to determine a certain property for each of the nonterminals of a context-free grammar:

> Given a context-free grammar, for each nonterminal N, what is the length of the shortest string that can be derived from N? (Our convention will be that if it is not possible to derive any string from N, then the answer for N should be ∞.)

One way to do this is by solving a set of equations over variables whose values are taken from $Nat \cup \{ \infty \}$. The equations are defined as follows:

(i) For each terminal or nonterminal n of the grammar, let there be a variable x_n; let there also be a variable x_ε.

(ii) Let there be an equation $x_\varepsilon = 0$.

(iii) For each terminal symbol t, let there be an equation $x_t = 1$.

(iv) Let "min" be an infix operator that computes the minimum of two numbers; e.g., $4 \min 1 = 1$. min is commutative and associative; e.g., $4 \min 1 = 1 \min 4 = 1$, and $(4 \min 1) \min 2 = 4 \min (1 \min 2) = 4 \min 1 \min 2 = 1$. For each nonterminal A with productions of the form

$A \rightarrow a_1 \cdots a_i$
$A \rightarrow b_1 \cdots b_j$
$\cdots$
$A \rightarrow h_1 \cdots h_p$

let there be an equation

$$x_A = (x_{a_1} + x_{a_2} + \cdots + x_{a_i}) \min (x_{b_1} + x_{b_2} + \cdots + x_{b_j}) \min \cdots \min (x_{h_1} + x_{h_2} + \cdots + x_{h_p}).$$

For example, the grammar

$A \rightarrow t$
$A \rightarrow A \ \ t$

gives rise to the following set of equations:

$x_\varepsilon = 0$
$x_t = 1$
$x_A = (x_t) \min (x_A + x_t)$

These equations happen to have a unique solution: $x_\varepsilon = 0$; $x_t = 1$; $x_A = 1$.

As a second example, the grammar

$A \rightarrow \varepsilon$
$A \rightarrow A \ \ t$

gives rise to the following set of equations:

$x_\varepsilon = 0$
$x_t = 1$
$x_A = (x_\varepsilon) \min (x_A + x_t)$

These equations also have a unique solution: $x_\varepsilon = 0$; $x_t = 1$; $x_A = 0$.

(Question continued on the next page.)

Part (a)

Not all sets of equations that arise from the above definition have a unique solution. Give a grammar (and its set of equations) such that the grammar's equations have more than one solution. List at least two of the solutions to the equations that you give.

(Note: For your grammar to be an acceptable answer, it must meet the following conditions: (i) all nonterminals must be derivable from the grammar's root symbol; (ii) for every nonterminal, there must be some derivation of either ε or some terminal string.)

Part (b)

Recall that if you have a complete partial order D with a least element $\perp$, and F is a continuous function from D to D, a recursive equation of the form $X = F(X)$ has a least solution $\bar{X}$ (where "least" is defined with respect to the ordering relation $\sqsubseteq$ of D) that can be obtained as

(*) $$\bar{X} = \bigsqcup_{i=0}^{\infty} F^i(\perp).$$

Given the set of equations that arise from a grammar G, as defined above, *define* an appropriate partial order D_G and a function F such that the *least* fixed-point of F is the desired solution; that is, the goal is that in the least fixed-point solution the value for variable x_N, where N is a nonterminal, is the length of the shortest string that can be derived from N (and $x_N = \infty$ if it is not possible to derive any string from N). (Be sure to say what the elements of D_G are, and what the ordering is on elements of D_G since the notion of "least" is defined with respect to this ordering.)

(Note: You are not required to prove that your function F is continuous in order to receive full credit.)

Question 2.

Information flow can be a security issue. For example, a program may have both high- and low-security inputs and outputs. There is a security leak if the value of a low-security output can be affected by the value of a high-security input.

Consider a simple programming language with assignment statements, if-then-else statements, while loops, initial reads (i.e., all read statements must come at the beginning of the program), and final writes (i.e., all writes must come at the end of the program), where a read or write only reads or writes the value of a single variable. Assume that the language has no procedures or procedure calls, no non-structured control flow, no pointers, no arrays or structures, and no global variables.

Here is an example program in this language:

```
begin program
   int w, x, y, z;
   read(w);
   read(y);
   read(z);
   w = 0;
   if (y > 10) {
      x = z + w;
   } else {
      x = z*2;
   }
   write(w);
   write(x);
end program
```

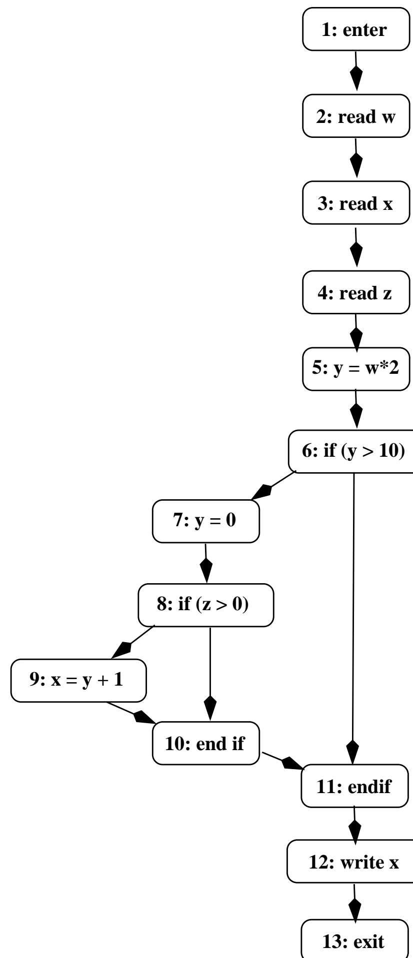
In this program, the output value of x can be affected by the input values of both y and z, but not the input value of w. The output value of w cannot be affected by the input value of any variable.

(Question continued on the next page.)

Part (a)

Now consider the program whose code and CFG are shown below (CFG nodes are numbered for use in the answer to this question).

```
begin program
   int w, x, y, z;
   read(w);
   read(x);
   read(z);
   y = w * 2;
   if (y > 10) {
      y = 0;
      if (z > 0) {
         x = y + 1;
      }
   }
   write(x);
end program
```



What input values can affect the output value of x?

Consider defining a dataflow-analysis problem whose solution can be used to determine, for each "write(v)" the set of variables on whose input values the output value depends. Define the problem for the example program given above by specifying the underlying lattice (of dataflow facts) including the meet operator, and by defining a set of equations, two for each node n, of the form:

n.before = ...
n.after = ...

where n.before is the dataflow fact that holds before node n executes, and n.after is the dataflow fact that holds immediately after n executes. Note that, unlike the usual equations for a dataflow-analysis problem, some n.afters will depend on the before facts of nodes other than node n.

(Question continued on the next page.)

Part (b)

Now assume that the language can include gotos, and consider the following program:

```
 1: begin program
 2:    int w, x, y, z;
 3:    read(w);
 4:    read(y);
 5:    read(z);
 6:    w = 0;
 7:    if (y > 10) {
 8:       x = z + w;
 9:       goto L;
       }
10:    w = 2;
11:    x = z*2;
12: L:
13:    write(w);
14:    write(x);
15: end program
```

On which input values might the output values of x and w depend?

What are the dataflow equations for the four assignment statements now?

In general, how does allowing gotos affect the way a node's dataflow equations are defined?

Question 3.

A function that looks up a value, x, in a list L is given below.

```
define lookup(x, L)
   cases (L) of
      nil:     false
      x::tail: true
      y::tail: lookup(x, tail)
```

Part (a)

Write function look1 that, given value x and list L, returns a list L' such that:

if x is not in L, then $L' = L$
else L' is L with x moved to the front of the list

Your code must be purely functional (e.g., no assignments).

Part (b)

What is the worst-case running time for look1 (in terms of the length of L)? (Note: The efficiency of your function will not affect your grade.)

Part(c)

Prove that for all x and all L

not lookup(x, L) => look1(x, L) = L

Question 4.

This question involves the design and implementation of exception handlers.

Part (a)

Outline an implementation of the throw/catch construct found in Java. That is, detail the machine-level steps needed to transfer control from the point of a throw to the handler responsible for the object that is thrown.

Part (b)

Java uses a *termination* semantics for exceptions: after a throw, execution continues in the context of the exception handler. Some languages use a *resumption* semantics for exceptions: After a thrown object is handled, execution is resumed with the statement that normally follows the throw (the throw acts like a "backwards call" to a handler, with a return after the handler completes). Explain how you would change your answer to Part (a) to support a resumption semantics for the throw/catch construct of Java.

Question 5.

Part (a)

Languages like C that do not guarantee array-bounds checking and that allow pointer arithmetic can lead to programs that are vulnerable to certain kinds of malicious attacks. Explain how a malicious user can exploit a buffer-overrun vulnerability in a program.

Part (b)

Consider the program shown below:

```
 1: main(int argc, char* argv[]){
 2:    char header[2048], buf[1024];
 3:    int counter;
 4:    FILE *fp;
 5:    ...
 6:    fgets(header, 2048, fp);
 7:    copy_buffer(header);
 8:    fgets (buf, 1024, fp);
 9:    copy_buffer(buf);
10: }
11:
12: void copy_buffer(char *buffer){
13:    char copy[20];
14:    strcpy(copy, buffer);
15: }
```

How could a malicious user exploit the buffer overrun in this program, to execute the system call `system(``exec /bin/sh'')`? Assume that the malicious user controls the contents of the file that `fp` points to.

Part (c)

One technique to address the buffer-overrun vulnerability is to make the stack *non executable*. Explain why this addresses the buffer-overrun vulnerability.

Part (d)

Another technique for addressing the buffer-overrun vulnerability is to keep an auxiliary stack. At a function call site, the return address is pushed on the auxiliary stack. Before returning to that call site, the program verifies that the top of the auxiliary stack matches the actual return address. Give details about this technique in the context of the program discussed in Part (b). Explain why this technique addresses the buffer-overrun problem.