Qualifying Exam in Programming Languages and Compilers

University of Wisconsin
Spring 1992

Instructions

This exam contains nine questions, divided into two parts. All students taking the exam should answer *four* of the five questions in Part I. **The answers to these questions will be collected after two hours** and Part II will be distributed to those students taking the exam to satisfy the depth requirement, who will have an additional two hours to answer *three* of the four questions in Part II.

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.

Part I

Question 1

It is very common for a program to be replaced by an "improved" version that fixes bugs or improves performance.

Certain programs, however, are designed to run continuously (*e.g.*, programs that control powerplants or monitor airport airspace). These kinds of programs will also occasionally need to be replaced by improved versions, but it may not be possible simply to stop the old version and start the new version. Rather, it may be necessary to replace program components *while* the program is executing.

Part A.

Assume we have a program, P, written in C or Pascal (your choice) and we wish to replace subroutine S. Subroutine S is non-recursive and has no local static variables. The functionality of S hasn't been changed, but its internal structure probably has been altered. How should P be compiled so that it is possible to replace subroutine S with a new version S' during the execution of some routine other than S? That is, after a certain point, S' is "installed" and all calls that would have gone to S now go to S'.

Part B.

How much more difficult is this replacement process if we must attempt to replace subroutine S with S' *while* S is executing? Under what circumstances could an executing subroutine be replaced?

Question 2

This question concerns the implementation of a programming language with the following features:

(1) The only types allowed are *integer*, *array of integer*, and *pointer to integer*.

(2) Heap storage is allocated explicitly using a call to the function *Allocate*. *Allocate* takes one parameter, a type, allocates enough storage for an object of that type, and returns the address of the newly allocated storage. When storage is allocated for an array of integers, the size of the array is part of the type passed to *Allocate*. For example, if *p* is a variable of type *pointer to integer*, storage pointed to by *p* can be allocated using either of the following calls:

```
p := Allocate(integer)                  /* allocate space for a single integer */
p := Allocate(array[10] of integer)     /* allocate space for 10 integers */
```

(3) There is no “address of” operation (like “&” in C). The only way to give a pointer a value is by calling *Allocate*, or by assigning from one pointer to another.

(4) Pointer arithmetic (increment and decrement) is allowed only for a pointer that points to an array of integers. However, pointer arithmetic is not allowed to move a pointer off either end of an array.

Describe how to implement the language so as to achieve the following:

(1) Arithmetic performed on pointers that do not point to arrays is detected as a run-time error.

(2) An increment or decrement that moves a pointer off the end of an array is detected as a run-time error.

(3) Deallocation of heap storage is done implicitly (*i.e.*, either using reference counting or using garbage collection).

In particular, describe the structure of a piece of heap-allocated storage, and discuss how pointers are implemented: what is done when a pointer is given a new value via a call to *Allocate*, via an assignment from another pointer, or via a use of increment or decrement.

Question 3

This question is about a variation on context-free grammars called an *extended grammar* and defined as follows: In an extended grammar, sequences of one or more symbols on a production's right-hand side can be enclosed in square brackets or in curly braces. Nested square brackets and/or curly braces are *not* allowed. For example, the following are legal extended grammar productions (nonterminals are lower-case words enclosed in angle brackets; terminals are upper-case words):

<proc head> → PROC ID [LPAREN <params> RPAREN] SEMICOLON
<params> → ID { COMMA ID }

Square brackets mean "zero or one instance"; curly braces mean "zero or more instances". Thus, the extended grammar given above defines the same language as the following context-free grammar.

<proc head>	→ PROC ID <opt params> SEMICOLON
<opt params>	→ ε | LPAREN <params> RPAREN
<params>	→ ID <more params>
<more params>	→ ε | COMMA ID <more params>

Part A.

Give an algorithm that converts an extended grammar to an equivalent context-free grammar. Your algorithm should be of the form

for each production *P* of the form X → α [β] γ **do** replace production *P* with the following productions: ...
for each production *P* of the form X → α { β } γ **do** replace production *P* with the following productions: ...

Part B.

Recall that, to determine whether a grammar is LL(1) it is necessary to compute FIRST sets for all productions' right-hand sides. Define how to compute the FIRST set for the right-hand side of an extended grammar production of the form

$$X \rightarrow Y_1\, Y_2 \; \cdots \; Y_n$$

where each Y_i is either a terminal, a nonterminal, or a sequence of terminals and/or nonterminals enclosed in square brackets or in curly braces. (Your definition should *not* involve converting the production to a standard context-free grammar production.)

Use your definition to compute FIRST sets for the right-hand sides of the following productions.

<block>	→ [<decls>] { <stmt> }
<decls>	→ VAR <decl> { <more decls> }
<decl>	→ ID COLON INT
<more decls>	→ [VAR] ID COLON INT
<stmt>	→ ID ASSIGN INT

Part C.

Recall that, to build the underlying finite-state machine for an LR(0) parser, it is necessary to compute the *closure* of an item of the form

$$X \rightarrow \alpha \,.\, \beta$$

where α and β are sequences of terminals and/or nonterminals. Define how to compute the closure of an item when "X → α β" is an extended grammar production. Again, your definition should not involve converting the production to a standard context-free grammar production.

Use your definition to compute the closure of the item

<block> → . [<decls>] { <stmt> }

using the grammar given above for Part B.

Question 4

Consider the following grammar fragment:

```
<prog>         →   <block>
<block>        →   BEGIN <stmts> END
<stmts>        →   <decl-stmt> | <decl-stmt> <stmts>
<decl-stmt>    →   <decl> | <stmt>
<decl>         →   DECLARE IDENTIFIER SEMICOLON
<stmt>         →   IDENTIFIER SEMICOLON | <block>
```

As usual for block-structured languages, an identifier cannot be declared more than once in the same block, and an identifier that is used in a statement must be declared either in the same block as the statement or in an enclosing block. However, an identifier does *not* have to be declared before it is used. Illegal declarations and uses are indicated in the example program shown below.

```
begin
declare x;

   begin
   declare z;
   end

   begin
   x;
   y;
   z;                       /* ERROR: z not declared in this or an enclosing block */
   declare x;
   end

z;                          /* ERROR: z not declared in this or an enclosing block */
declare y;
declare x;                  /* ERROR: x is declared twice in this block */
end
```

Part A.

What are the advantages and disadvantages of allowing declarations to follow uses as in this language?

Part B.

Add YACC actions to the grammar given above to check for multiple declarations and to report an error if they occur.

Part C.

Add YACC actions to the grammar given above to check for undeclared variables and to report an error if they occur.

Question 5

Consider adding an exception-handling mechanism to a language such as C. The mechanism consists of two new constructs: an *on-statement*, which associates an exception handler with a statement, and a *raise statement*, which causes an exception to be raised.

The syntax of these two statements is shown below:

```
on <id> in <stmt1> do <stmt2>
raise <id>
```

The name of the exception is <id>; the on-statement associates that exception with <stmt1>; and <stmt2> is the exception handler. Both <stmt1> and <stmt2> are arbitrary C statements, which may contain calls. When an on-statement is executed, it executes <stmt1> first. If that statement completes without raising an exception, the on-statement terminates. However, if exception <id> is raised during the execution of <stmt1> (including the execution of any routines called directly or indirectly by <stmt1>), control is transferred to the handler that has been most recently associated with <id>, and that is still active (that is, the handler to which control is transferred is determined *dynamically*). When the handler terminates, the on-statement also terminates and control passes to the on-statement's successor.

For example, if the input to the program shown below is two negative numbers followed by a positive number, then EX1 is raised twice (once for each negative input); the first time EX1 is raised it is handled by HandleEx1, and the second time EX1 is raised it is handled by HandleEx1Again. The message 'Thank you' is never written, but the message 'All done' is written (at the very end).

```
integer i;

GetInt()
{  write('Enter a positive integer: ');
   read(i);
   if (i<=0) raise EX1;
   write('Thank you!');  }

HandleEx1()
{  on EX1 in GetInt() do HandleEx1Again();  }

HandleEx1Again()
{  while (i <= 0) {
     write('A POSITIVE integer, please: ');
     read(i);
     }  }

main()
{  on EX1 in GetInt() do HandleEx1();
   write('All done');  }
```

Part A.

Describe the changes that would need to be made to a C compiler and runtime system to implement this mechanism.

Part B.

Suppose we want to be able to pass a value from the place where an exception is raised to its exception handler. Extend the syntax of the on-statement and the raise statement to allow a single value to be passed between the two points. When and how would type checking be done to ensure that the value passed by the raise statement was consistent with the type expected in the on-statement?

Qualifying Exam in Programming Languages and Compilers

University of Wisconsin
Spring 1992

Part II

Instructions

All students taking this exam to satisfy the depth requirement should answer *three* of the four questions in Part II.

Question 6

In this question, assume that D is a chain-complete partial order, and that g is a continuous (and hence monotonic) function in $D \rightarrow D$ that is also *inflationary*; by inflationary, we mean that for all x, $g(x) \sqsupseteq x$.

Part A.

Show that for all z, the least fixed point of g that is greater than or equal to z is

$$G_z =_{df} \bigsqcup_{i=0}^{\infty} g^i(z),$$

where $g^0(z)$ is defined to be z. (Hint, you need to show two things: (i) show that G_z is a fixed point of g; (ii) show that for all $w \sqsupseteq z$ such that w is a fixed point of g, $w \sqsupseteq G_z$.)

Part B.

What is the least fixed point of the following recursive functional equation?

$$h = \lambda x\,.\,(g(h(x)) \sqcup g(x))$$

Question 7

Part A.

It sometimes happens that the same expression, *E*, occurs in more than one place in a program. The optimization called *code hoisting* seeks to replace such multiple occurrences with a single occurrence of *E* by inserting an evaluation of *E* at a new point *p* and then removing the original occurrences of *E*. What properties must be true of point *p* to permit this optimization to be performed? If *E* could be hoisted to either point p_1 or point p_2, under what circumstances would one point be preferred over the other? What are the advantages and disadvantages of performing this optimization?

Part B.

An interesting variant of code hoisting involves moving the evaluation of an expression from the body of a subprogram to the call site(s) of the subprogram. In effect, part of the subprogram body is expanded at the point of call, while the rest of the body is evaluated during the call.

Many recursive functions are of the form

F(*N*) = **if** Pred(*N*) **then** Const **else** Expr

where Pred(*N*) is a predicate dependent only on the parameter *N*, Const is a constant-valued expression, and Expr involves a call to F. Discuss how "hoisting" the evaluation of parts of F's body to its call site(s) can be an optimization. Under what circumstances is this form of optimization useful for non-recursive calls?

Part C.

Now assume that we analyze or profile the recursive function F and find that it is most frequently called with the arguments V1, V2, ... , Vn.

Using this information, F can be rewritten into the form

F(*N*) = **case** (*N*) **of**
V1: Const1;
V2: Const2;
...
Vn: ConstN;
Otherwise : **if** Pred(*N*) **then** Const **else** Expr;
endcase;

Under what circumstances is this rewriting of F an optimization? Is it amenable to the "hoisting" optimization suggested in Part B?

Question 8

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 $\sqcap$, least element $\bot$ (bottom), greatest element $\top$ (top), and partial order $\sqsubseteq$. (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 $Entry_{out}$. For backward problems, v represents the information that is true after the program finishes executing, and v is associated with $Exit_{in}$.

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* and *Exit*: $n_{in} = \sqcap_{p \in predecessor(n)} p_{out}$ $n_{out} = f_n(n_{in})$	for all nodes n other than *Entry* and *Exit* : $n_{in} = f_n(n_{out})$ $n_{out} = \sqcap_{s \in successor(n)} s_{in}$

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:

$$\lambda\, s.\ ((s - \{\text{all pairs with first element } x\}) \cup \{(x,n)\})$$

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

(3) The value $Entry_{out}$ is the empty set.

Part A.

Choose two of the following four dataflow problems: (1) constant propagation, (2) live variables, (3) available expressions, (4) very busy expressions. For each of the problems that you choose, give a brief English description of the goal of the analysis, say whether the problem is a forward or a backward problem, and specify the corresponding dataflow framework.

Part B.

Give an algorithm that can be used to solve an arbitrary dataflow problem; that is, an algorithm that, given a flowgraph and a dataflow framework, finds the greatest fixed point of the system of equations given above.

Question 9

This question concerns two denotational semantics for a very simple imperative language in which a program is a list of one or more statements, and a statement is either an assignment or an if-then-else.

The usual way to define the meaning of programs in the language described above is as a function from an initial state to a final state (where a state is a map from identifiers to values). Such a denotational semantics is given below (only the meaning (valuation) function **M** for statements is given; the abstract syntax and semantic algebras are the usual ones, as are the meaning functions **B** for boolean expressions, and **E** for arithmetic expressions). The function "update" used in the definition of the meaning of the assignment statement takes three parameters: a state σ, an identifier x, and a value v, and returns a state that is the same as σ except that it maps x to v.

$\mathbf{M}[\![\, Id := exp \,]\!] = \lambda\sigma.\ \text{update}(\sigma, Id, \mathbf{E}[\![\, exp \,]\!]\,(\sigma))$
$\mathbf{M}[\![\, S_1 \,;\, S_2 \,]\!] = \lambda\sigma.\ \mathbf{M}[\![\, S_2 \,]\!]\,(\mathbf{M}[\![\, S_1 \,]\!]\,(\sigma))$
$\mathbf{M}[\![\, \text{if } cond \text{ then } S_1 \text{ else } S_2 \text{ fi}]\!] = \lambda\sigma.\ \text{if } \mathbf{B}[\![\, cond \,]\!]\,(\sigma) \text{ then } \mathbf{M}[\![\, S_1 \,]\!]\,(\sigma) \text{ else } \mathbf{M}[\![\, S_2 \,]\!]\,(\sigma)$

You are to write an alternative denotational semantics for this language so that the meaning of a program is a function of the form:

(initial-state $\times$ initial-map) $\rightarrow$ (final-state $\times$ final-map)

where initial-map and final-map are maps from program points to values. The intention is that initial-map will always be the map that maps all points to the value $\perp$ (bottom); final-map is to map a point p in the program to the value produced at that point when the program is executed on the given initial state. A program point is either an assignment statement or a predicate (the condition of an if-then-else). By "the value produced at a point", we mean: for an assignment statement, the value assigned to the left-hand-side variable, and for a predicate, the boolean value to which the predicate evaluates.

An example is given below (all program points have integer labels).

Program P

```
1: x := 0;
if 2: z > 0 then 3: x := x + 1 else 4: y := 10 fi;
if 5: x = z then 6: z := z + 1 else 7: z := y fi
```

Initial State	Final State	Final Map
x→0, y→0, z→0	x→0, y→10, z→1	1→0, 2→false, 3→⊥, 4→10, 5→true, 6→1, 7→⊥

Complete the definitions given below; be sure to provide or at least explain any auxiliary functions that you use in your definitions.

$\mathbf{M}[\![\, p : Id := exp \,]\!] = \lambda\sigma.\ \lambda m.$____

$\mathbf{M}[\![\, p_1 : S_1 \,;\, p_2 : S_2 \,]\!] = \lambda\sigma.\ \lambda m.$____

$\mathbf{M}[\![\, \text{if } p_1 : cond \text{ then } p_2 : S_1 \text{ else } p_3 : S_2 \text{ fi} \,]\!] = \lambda\sigma.\ \lambda m.$____