Qualifying Exam in Programming Languages and Compilers

University of Wisconsin
Fall 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

Part A.

In either Pascal or C (your choice), explain the run-time steps needed to call a subprogram and return from it. Illustrate each of these steps using the following subprogram and call:

```
int max3(int a, int b, int c) {
      int temp = (a<b?b:a);
      return (temp<c?c:temp);
}

val = max3(i,j,10);
```

Part B.

Since subprogram calls are very common, translating them efficiently is often critical. For each of the steps you mentioned in part A, explain how the step can be optimized or even removed entirely. When is such optimization or removal appropriate?

Question 2

Consider the problem of generating code for switch statements, defined as follows:

<switch-stmt> → "switch" <exp> <cases> <otherwise> "end-switch"

<cases> → <case>
→ <case> <cases>

<case> → "case" <int-lits> ":" <stmt> "end-case"

<int-lits> → INTEGER-LITERAL
→ INTEGER-LITERAL "," <int-lits>

<otherwise> → "otherwise" ":" <stmt> "end-case"

That is, a switch statement includes an integer expression and a list of cases, each labeled with one or more integers, and ending with a case labeled "otherwise" (the integer labels of the cases must be disjoint). At runtime, the integer expression is evaluated. If its value matches one of the labels, the corresponding case is executed; if its value does not match any of the labels, the "otherwise" case is executed.

Part A.

Describe three ways in which a compiler could generate code for a switch statement. For each alternative, write (in some reasonably high-level pseudo language) the code that a compiler would produce for this switch statement:

switch (x)
  case 101: S_1
    end-case

  case 1100,2: S_2
    end-case

  otherwise: S_3
    end-case
end-switch

Part B.

Describe how a compiler might choose among your implementation techniques so as to minimize the average number of instructions that would be executed for each switch statement. (That is, describe how and why the compiler might choose to use one technique for one switch statement and another technique for another switch statement).

Question 3

Part A.

Give an example Prolog program (a set of facts, a set of rules, and a query) so that both of the following hold:

(1) Evaluation of the query using Prolog's depth-first approach to satisfying a goal leads to an infinite loop.

(2) Evaluation of the query would *not* lead to an infinite loop if a breadth-first approach were used instead.

Part B.

For the example of Part A, show enough of the proof tree that would be constructed using the depth-first approach to demonstrate the infinite loop. Show the entire proof tree that would be constructed using the breadth-first approach.

Part C.

What are the disadvantages of a breadth-first approach (compared to a depth-first approach)?

Question 4

Almost all programming languages provide the notion of functions and function application. Languages vary greatly, however, in the way functions may be manipulated.

Part A.

Pascal allows functions to be passed as parameters to other subprograms. Thus we might have:

```
procedure filter(function f(i:integer):integer;var t:text);
  var     i:integer;
  begin
          while not eof(t) do begin
                  read(t,i);
                  write(f(i));
          end
  end;
```

What information must be provided as the “value” of a function when it is passed as an actual parameter?

Part B.

In C a pointer to a function may be stored in a variable and later called. Thus (in ANSI C) we might have

```
int   plus1(int i) {
      return (i+1);
}

int   (*fct)(int);
fct = plus1;
i = (*fct)(10);
```

However Pascal does not allow functions to be stored in variables. Why does Pascal impose this restriction while C does not? Suppose we wish to extend Pascal to allow functions to be stored in variables. What changes would be needed to the run-time environment to support this change?

Question 5

This question is about heap-allocated storage in a language like C. Suppose that the storage allocator caches storage blocks in an array "chunks", where chunks[i] is either NULL or a pointer to a linked list of blocks that have i bytes of free storage. Further suppose that each storage block has a "hidden" size field; that is, if p points to a (free or allocated) storage block, the size can be retrieved by the expression "`((int *)p)[-1]`". Storage blocks are chained together using the first few bytes of each storage block to hold a pointer to the next block, with NULL being used to indicate the end of the list. That is, if p points to a free storage block, the expression "`*(BLOCK_PTR *)p`" yields the next free block of the list, or NULL if p points to the last block of the list.

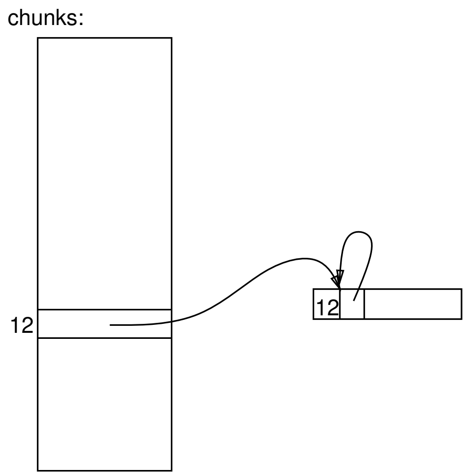
Here is ANSI C code for a part of the storage allocator:

```
BLOCK_PTR chunks[MAX_CHUNKS+1];

BLOCK_PTR allocate(int m)           /* it is assumed that: 4 ≤ m ≤ MAX_CHUNKS */
{
    BLOCK_PTR p;

    if (chunks[m] != NULL) {
        p = chunks[m];              /* Retrieve first storage block of the list */
        chunks[m] = *(BLOCK_PTR *)p;/* Chunks[m] now points to the second block
                                     * of the list, or is NULL if the list
                                     * was of length 1 */
        return(p);
    }
    else { /* No cached storage block of this size */
        . . .
    }
}

void free(BLOCK_PTR p)
{
    int m = ((int *)p)[-1];         /* Retrieve the "hidden" block size */

    *(BLOCK_PTR *)p = chunks[m];    /* attach the returned block to the
                                     * front of the appropriate list */
    chunks[m] = p;
    return;
}
```

(**Question continued on the next page.**)

Part A.

Suppose your code, which includes calls to *allocate* and *free*, does not work correctly. When you debug your code, you find the following anomaly in the storage allocator's structures:



Explain how this could have occurred.

Part B.

Describe how to modify the storage allocator so that the error you identified as being the source of the problem in Part A would be caught at run-time as soon as it was about to corrupt the storage allocator's structures.

Qualifying Exam in Programming Languages and Compilers

University of Wisconsin
Fall 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

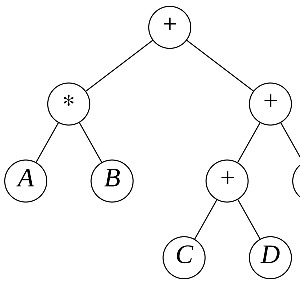
Assume we have a computer with the following instructions (reg denotes any register; mem denotes any memory location):

Ld	reg,mem	Load reg with value at mem
St	reg,mem	Store contents of reg at mem
Add	reg1,reg2	reg1 := reg1 + reg2
Add	reg,mem	reg := reg + value at mem
Mult	reg1,reg2	reg1 := reg1 * reg2
Mult	reg,mem	reg := reg * value at mem

Part A.

Let T be any binary tree with internal nodes of + and * and leaves that are distinct identifiers. Give an algorithm that computes the fewest registers needed to evaluate T without generating any store instructions (*i.e.*, without "spilling" any registers).

Illustrate your algorithm using the following tree:



Part B.

Assume the execution times for the 6 instructions listed above are 4, 4, 1, 6, 7, and 15 respectively. Let T be a tree as per part A. Given R registers (where R is sufficient to avoid spilling), give an algorithm that optimally translates T (in terms of minimum execution time). Illustrate your algorithm using the tree of part A with R=2.

Question 7

Consider the following scheme for "evaluating" trees:

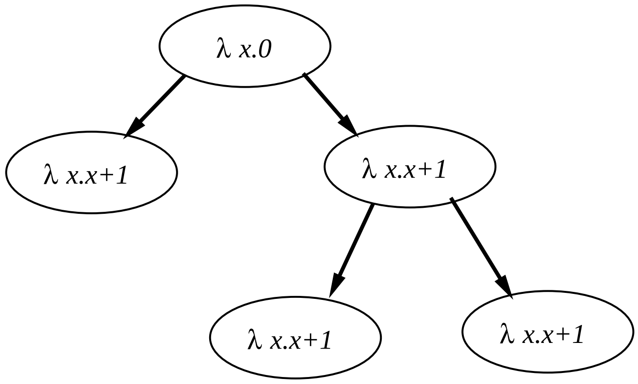
(i) Every node of a tree is labeled with a one-argument function.

(ii) The node functions are evaluated in pre-order according to their position in the tree, with the result of the previous evaluation used as the argument to the next function evaluation.

(iii) There is a distinguished constant that is used as the argument to the function at the root.

The "value of the tree" is the value produced after the function associated with the rightmost leaf is evaluated.

For example, to count the number of edges in a tree, we could use a tree-evaluation scheme in which the root is labeled with $\lambda x.0$, all other nodes are labeled with $\lambda x.x+1$, and an arbitrary value is used for the distinguished constant.



Part A.

Present a tree-evaluation scheme to determine the leftmost occurrence of a tree-node of type "foo". (Assume that the function associated with a node has access to the the node's "name" via the special symbol "self" and to the node's type via the expression "type(self)".)

Part B.

Suppose it is possible for trees to undergo modifications, but we wish to minimize the amount of work performed to recompute the "value of the tree" after such changes. In particular, suppose that the only kind of modification permitted is the replacement of a single subtree with some other subtree. Give an algorithm for the re-evaluation process that minimizes the amount of work performed to recompute the "value of the tree". Assume that the input to the algorithm is the original tree, a pointer to the node at the root of the subtree to be replaced, and the new subtree. In your answer, be sure to explain what additional information, if any, would need to be stored in the tree in order to permit your algorithm to work (and how that information itself would be correctly updated in preparation for the next tree modification).

Question 8

This question concerns how to determine a property of the nonterminals of a context-free grammar. The problem is: given a context-free grammar, which nonterminals of the grammar can derive the empty string? (In this question we will denote the empty string by “ε”.)

One way to do this is by solving a set of equations, which are defined as follows:

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

(ii) Let there be an equation $x_\varepsilon = \textbf{true}$.

(iii) For each terminal symbol t, let there be an equation $x_t = \textbf{false}$.

(iv) For each nonterminal A with productions of the form

$$\begin{array}{l} A \to a_1 \cdots a_i \\ A \to b_1 \cdots b_j \\ \cdots \\ A \to h_1 \cdots h_p \end{array}$$

let there be an equation

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

For example, the grammar

$$\begin{array}{l} A \to t \\ A \to A \ \ t \end{array}$$

gives rise to the following set of equations:

$$\begin{array}{l} x_\varepsilon = \textbf{true} \\ x_t = \textbf{false} \\ x_A = (x_t) \vee (x_A \wedge x_t) \end{array}$$

These equations happen to have a unique solution: $x_\varepsilon = \textbf{true}$; $x_t = \textbf{false}$; $x_A = \textbf{false}$.

The grammar

$$\begin{array}{l} A \to \varepsilon \\ A \to A \ \ t \end{array}$$

gives rise to the following set of equations:

$$\begin{array}{l} x_\varepsilon = \textbf{true} \\ x_t = \textbf{false} \\ x_A = (x_\varepsilon) \vee (x_A \wedge x_t) \end{array}$$

These equations also have a unique solution: $x_\varepsilon = \textbf{true}$; $x_t = \textbf{false}$; $x_A = \textbf{true}$.

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 all the solutions to the equations 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.)

(**Question continued on the next page.**)

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 $\overline{X}$ (where "least" is defined with respect to the ordering relation $\sqsubseteq$ of D) that can be obtained as

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

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 only variables with value **true** correspond exactly to the nonterminals of the grammar that can derive ε. (Be sure to say explicitly what the ordering is on elements of D_G since the notion of "least" is defined with respect to this ordering.)

(Hint: Recall that if a partial order D is finite, then (i) every monotonic function from D to D is continuous and (ii) D is a *complete* partial order.)

Part C.

Recall that (if D has no infinite ascending chains) equation (*) has operational significance—it corresponds to an iterative process for finding $\overline{X}$. For your example grammar from Part A, trace to completion the iterative process that would arise from your answer to Part B.

Question 9

Consider the simple imperative language defined below.

$$\begin{array}{ll} program & \rightarrow cmd \\ cmd & \rightarrow \text{Id} := exp \;\mid\; \text{if } cond \text{ then } cmd \text{ else } cmd \text{ fi} \;\mid\; cmd \;;\; cmd \end{array}$$

That is, a program is a command, and a command is an assignment, an if-then-else, or a command followed by another command. (The definitions of expressions and conditions are not needed for this question.)

The usual way to define the meaning of programs in the language defined above is as a function from an initial state to a final state (where a state is a map from identifiers to values). A denotational semantics that defines program meanings in this way is given below (only the meaning (valuation) functions **P** and **C**, for programs and commands, respectively, are given; the meaning functions **B** for boolean expressions, and **E** for arithmetic expressions are the usual ones). The function "update" used in the definition of the meaning of the assignment command 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.

$\mathbf{P}[\![\ C\ ]\!] = \lambda\sigma.\ \mathbf{C}\ [\![\ C\ ]\!]\ (\sigma)$
$\mathbf{C}[\![\ Id := exp\ ]\!] = \lambda\sigma.\ \text{update}(\sigma, Id, \mathbf{E}[\![\ exp\ ]\!]\ (\sigma))$
$\mathbf{C}[\![\ \text{if } cond \text{ then } C_1 \text{ else } C_2 \text{ fi}]\!] = \lambda\sigma.\ \text{if } \mathbf{B}[\![\ cond\ ]\!]\ (\sigma) \text{ then } \mathbf{C}[\![\ C_1\ ]\!]\ (\sigma) \text{ else } \mathbf{C}[\![\ C_2\ ]\!]\ (\sigma)$
$\mathbf{C}[\![\ C_1\ ;\ C_2\ ]\!] = \lambda\sigma.\ \mathbf{C}[\![\ C_2\ ]\!]\ (\mathbf{C}[\![\ C_1\ ]\!]\ (\sigma))$

Now consider extending the language by adding *read* and *write* commands as shown below:

$$cmd \rightarrow \text{read(Id)} \;\mid\; \text{write(Id)}$$

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

$$(\text{initial-state} \times \text{input-int-list}) \rightarrow \text{output-int-list}$$

The idea is that the program is given an initial state and a list of integers (the input), and produces another list of integers (the output). Executing a command of the form "read(x)" removes the leftmost value from the input list and assigns the value to x. Executing a command of the form "write(x)" appends the current value of x to the right end of the output list. Assume that the input list of integers is sufficiently long (*i.e.*, do not worry about handling "end-of-file" errors). HINT: define meaning functions **P** and **C** to have the following types:

P: (state $\times$ int-list) $\rightarrow$ int-list
C: (state $\times$ int-list $\times$ int-list) $\rightarrow$ (state $\times$ int-list $\times$ int-list)

Complete the definitions given on the next page. You may use the standard functions *car*, *cdr*, and *append* (function *append* takes two parameters: an integer j and a list L, and attaches j to the right end of L). Be sure to provide or at least explain any other auxiliary functions that you use in your definitions.

(**Question continued on the next page.**)

$\mathbf{P}[\![\; C \;]\!] = \lambda < \sigma, \; in > .$ ____

$\mathbf{C}[\![\; Id := exp \;]\!] = \lambda < \sigma, \; in, \; out > .$ ____

$\mathbf{C}[\![\; \text{if } cond \text{ then } C_1 \text{ else } C_2 \text{ fi} \;]\!] = \lambda < \sigma, \; in, \; out > .$ ____

$\mathbf{C}[\![\; C_1 \; ; C_2 \;]\!] = \lambda < \sigma, \; in, \; out > .$ ____

$\mathbf{C}[\![\; read \, (\, Id \,) \;]\!] = \lambda < \sigma, \; in, \; out > .$ ____

$\mathbf{C}[\![\; write \, (\, Id \,) \;]\!] = \lambda < \sigma, \; in, \; out > .$ ____