SPRING 1993
COMPUTER SCIENCES DEPARTMENT
UNIVERSITY OF WISCONSIN—MADISON
PH. D. QUALIFYING EXAMINATION

Programming Languages
Depth Examination, Part I

Tuesday, February 9, 1993
3:00 – 5:00 PM
113 Psychology

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 *four* of the *five* questions 1-5. At the end of two hours, your answers will be collected and you will be given 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.

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 be such that the problem is nontrivial.

Question 1

Part A.

What is the difference between *true polymorphism* and *overloading* (sometimes called *ad-hoc polymorphism*)?

Part B.

Choose 3 of the following languages:

Pascal, ML, Ada, C++, Scheme

and discuss what support they provide for true polymorphism and/or overloading, both built-in and user-defined.

Question 2

Most programming languages allow programmers to define new types or subprograms. Some even allow existing operators (like + or ==) to be redefined or overloaded. Few programming languages allow brand new operators to be defined.

Assume we allow brand new operators to be defined using declarations of the following form:

NewOp <token> level = <n>

<token> is a blank-terminated sequence of non-alphanumeric characters not corresponding to any predefined operator (for example, !! or += would be legal values for <token>). <n> is an integer in the range 1 to 5; this value defines the operator's precedence level, with 1 = lowest and 5 = highest. Every user-defined operator is binary and left-associative. Every predefined binary operator is also left-associative and has a fixed, preset precedence level in the range 1 to 5.

For the purposes of this question, we'll focus on only the scanner and parser of the compiler. How must these components be designed so that user-defined operators (and of course all other constructs) are correctly scanned and parsed?

Question 3

Is is always nice to have informative error messages when a program is terminated due to some run-time error. Consider an error message and call traceback of the form:

> Error at line *nnn* in procedure *ppp*.
> Called from line *mmm* in procedure *qqq*.
> .
> .
> .
> Called from line *lll* in Main.

Part A.

What changes must be made to the usual run-time data structures and generated code to allow line numbers and procedure names to be recoverable when termination occurs?

Part B.

Since we expect that run-time errors will be rare, it is preferable that we add as little run-time overhead as possible in preparing for run-time error messages that likely will never occur. That is, we are willing to do a great deal of work at compile-time, or at run-time *after* an error is detected, but we wish to do little or no extra work at run-time prior to error detection.

Give another solution in which as much of the overhead of determining line numbers and procedure names as possible takes place either at compile-time, or at run-time after an error is detected.

Question 4

Consider a Pascal-like language in which all variables are of type *integer* (and are not explicitly declared), and that includes as statements only assignment, read, write, if-then-else, and while-loop. The following productions define the language's syntax (ID is an identifier denoting a variable, LIT is an integer literal, and OP is a predefined operator).

Program	→	Stmt
Stmt	→	ID := Exp
Stmt	→	read (ID)
Stmt	→	write (Exp)
Stmt	→	Stmt ; Stmt
Stmt	→	if (Exp) then begin Stmt end else begin Stmt end
Stmt	→	while (Exp) do begin Stmt end
Exp	→	ID
Exp	→	LIT
Exp	→	Exp OP Exp

Let S be a statement (as defined above). Define MayUse(S) to be the set of variables whose value S may use. Define MayDef(S) to be the set of variables whose value S may set.

Part A.

Give a syntax-directed (structure driven) translation scheme using the grammar given above so that each Stmt nonterminal has two associated values, Stmt.MayUse and Stmt.MayDef, which are the MayUse and MayDef sets defined above. For productions that include multiple occurrences of the same nonterminal, use subscripts to identify each occurrence. For example, the translation rules for the production

Stmt → Stmt ; Stmt

will be of the following form.

$Stmt_1$.MayUse = < some function of $Stmt_2$.MayUse and $Stmt_3$.MayUse >
$Stmt_1$.MayDef = < some function of $Stmt_2$.MayDef and $Stmt_3$.MayDef >

Part B.

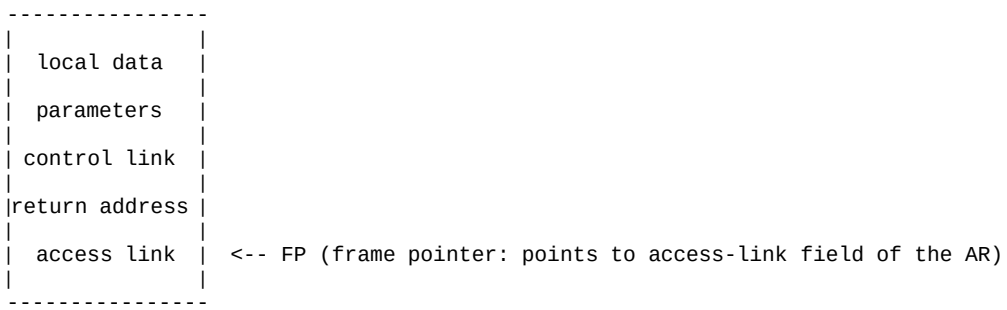
Recall that S1;S2 means execute statement S1 then execute statement S2. Based on the values MayUse(S1), MayUse(S2), MayDef(S1), and MayDef(S2), when must S1;S2 and S2;S1 have exactly the same semantics?

Question 5

This question concerns runtime access to non-local variables in Pascal. The second and third parts of the question also deal with subprograms as parameters. (The terms "control link" and "access link" used in this question are synonyms for "dynamic link" and "static link," respectively).

Part A.

Assume that activation records are organized as shown below.



Further assume that the target of an access link is another activation record's access-link field. Explain how a subprogram accesses both local and non-local data.

Part B.

Now consider what happens when a subprogram is passed as a parameter, as in the example program given below. Procedure *Usesx* should access the *x* in the main program, not the *x* in procedure *PassesP*. How is this accomplished?

```
program main;
var x: integer;
    procedure Usesx;
    begin
      write(x)
    end;
    procedure PassesP;
    var x: integer;
        procedure CallsP(procedure P);
        begin
          P;
        end;
    begin /* PassesP */
      x := 0;
      CallsP(Usesx);
    end;
begin /* main */
  x := 50;
  PassesP;
end.
```

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

Part C.

Suppose that we want to change the definition of Pascal so that when a subprogram is passed as a parameter, and that subprogram uses a non-local variable, the variable used is the one that is in the most closely enclosing scope of the subprogram that *passed* the parameter (for example, in the program above, we want *Usesx* to access the *x* declared in *PassesP*).

A suggestion is made that the only change to the compiler that is needed is to change the code that is generated for calls that pass subprogram parameters; the change would be to pass as the *access link* part of the parameter a pointer to the calling procedure's activation record rather than the access link that would normally be passed.

Although this suggestion will work in the example program given above; it will not always work. Why not?