Qualifying Exam in Programming Languages and Compilers

University of Wisconsin
Fall 1991

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**. Students taking the exam to satisfy the depth requirement 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.

Part I

Question 1

In translating a record (or struct) definition, a compiler normally assigns fields sequentially and contiguously, with each field assigned an offset that immediately follows the end of its predecessor's space allocation.

Many machines have *alignment restrictions* that limit how data may be addressed. For example, a 2-byte halfword may be required to have an even address, a 4-byte fullword may be required to have an address that is a multiple of 4, an 8-byte doubleword may be required to have an address that is a multiple of 8, etc.

(a) Explain how the assignment of field offsets must be changed when alignment restructions are imposed. What is the overall alignment restriction for an entire record given the alignment restrictions of its fields? Be sure to handle the case in which records appear as fields in other records.

(b) Show that the order in which fields are processed can make a difference in the overall size of a record when alignment restrictions are imposed. That is, show that permuting the fields within a record can change the overall size of the record. Outline an algorithm (or heuristic) to reorder fields that minimizes the overall size of a record. What is the running time of your routine in terms of the number of fields in the record?

Question 2

Assume we are analyzing a program written in Pascal or C (your choice), and that we are concerned with two points x and y that occur in the same procedure.

(a) Explain how we could determine whether once the flow of control reaches x it is possible to reach y. You may assume that all legs of conditionals are executable, that loops execute at least once and always terminate, that all calls return, and that no statement or expression faults and thereby terminates execution.

(b) How must your analysis be changed if x and y are in different procedures?

(c) Assume that point x represents the *definition* of some global scalar variable v (an assignment to v, a read into v, etc), and that point y represents a use of variable v. We say that a definition of v at x *reaches* a use of v at y if there is a path from x to y along which the value of v is not changed.

Outline how we can determine whether a definition of v at point x reaches a use of v at y.

Question 3

Although the programming language C was originally designed with explicit storage allocation and deallocation (*malloc* and *free*), C has recently been augmented by *conservative garbage collectors*. This type of garbage collector is a library routine that is linked into a program. When the program exhausts available storage, the collector is invoked and it searches memory to find unreachable blocks of storage (garbage).

The collector begins its search at the program's "root set." A C program's root set is all the active, non-heap storage that contains potential pointers into the heap. From the root set, the garbage collector transitively traces each potential pointer into the program's address space. Since the collector does not know the type of the value held in a memory location, it must treat every bit-pattern that is a valid data address as a potential pointer. Any memory that is unreachable by this traversal is garbage.

(a) What are the advantages of garbage collection over explicit allocation and deallocation? What are the advantages of explicit allocation and deallocation over garbage collection?

(b) What is the root set of a C program? Where would a garbage collector find this set?

(c) How could a C compiler assist the garbage collector?

Question 4

In most programming languages, the number of parameters in a procedure call must match the number of parameters in the procedure header. The goal of this question is the design of a language in which it is possible to specify (in a procedure's header) that some of the procedure's formal parameters are optional. Calls to the procedure that omit the corresponding actual parameters are legal and default values are used for those parameters in the procedure body.

(a) Design a reasonable syntax for procedure headers and procedure calls that can be used to meet the goal described above. Include a context-free grammar as part of your syntax definition. (Don't worry about providing rules for all nonterminals—*e.g.*, you can use the nonterminal *type* without further defining it—just give the rules relevant to the problem of optional parameters.) Also include at least one example procedure header and some example calls.

(b) Discuss how the language you defined in Part (a) could be implemented; that is, describe what, if anything, unusual would have to be done at compile time and/or at run time.

(c) What are the advantages and disadvantages (from a programmer's point of view) of using optional parameters?

Question 5

This question is about runtime access to non-local variables in a statically scoped language with nested procedures (*e.g.*, Pascal). Recall that in such a language, runtime access to non-local variables can be implemented either using *access links* (also known as *static links*) or using *displays*. The program shown below will be used throughout this question.

```
program main;
    procedure P1;
    begin {P1} end;

    procedure P2;
        procedure P3;
            procedure P4;
                procedure P5;
                begin {P5} call P1 end;
            begin {P4} call P5 end;
        begin {P3} call P4 end;
    begin {P2} call P3 end

    procedure P6;
      procedure P7;
      begin {P7} call P2 end
    begin {P6} call P7 end;

begin {main}
call P6;
end
```

(a) Assume that access links are used and that the contents of the access-link field in an activation record is a pointer to another activation record's access-link field. Draw the runtime stack as it would appear after the call to procedure P1 in the program shown above. (You need only show the values of the activation records' access-link fields.) Be sure to label each activation record with its procedure name.

(b) Assume that a display is used and that each activation record has a "save-display" field that is filled in as part of a procedure call (*i.e.*, the calling procedure fills in the save-display field of the called procedure with the value in the display that is about to be overwritten). Draw the runtime stack and the display as they would appear after the call to procedure P1 in the program shown above. (You need only show the values of the activation records' save-display fields.) Be sure to label each activation record with its procedure name, and also label the components of the display "[1]", "[2]", *etc*.

NOTE: Part (c) is on the next page ⇒.

(c) Suppose there are only three registers available to be used for the display (and for some reason it has been decided that no memory space is to be used for the display). Design a scheme for access to non-local variables that *combines* the use of access links and displays; *i.e.*, the (limited size) display is used to access a non-local variable whenever that is possible, and access links are used when the display cannot be used. Be sure to discuss each of the following issues:

(i) When a procedure *P* at nesting level *i* is executing, which activation records are pointed to by Display[1], Display[2], and Display[3]?

(ii) When a procedure *P* at nesting level *i* is executing, which activation records have access links, and where do they point?

(iii) When a procedure *P* at nesting level *i* accesses a variable that was declared at nesting level *j*, how is the activation record that has space for *j* found (you may want to break this up into cases depending on the value of *i* and/or *j*)?

Illustrate your scheme by drawing the runtime stack and the display as they would appear after the call to procedure P1 in the program shown above. Show whatever fields of the activation records are relevant to your scheme.

Part II

Question 6

In C a scalar variable may be given a *register* attribute, signifying that it should be allocated in a register. One obvious way to implement a register variable is simply to assign a particular register to a register variable for the entire lifetime of the variable. This approach is undesirable for (at least) two reasons. First, a register variable isn't necessarily live at all points within its scope. A register is a terrible thing to waste, especially when it contains a dead value. Second, a register variable is often assigned a value at many points within its scope. It is unnecessary that the *same* register be used at two definition points if the points don't interact.

(a) Define a program-analysis technique that determines, for each register variable, the portions of the code in which the register variable is live, and that identifies when definition points are independent (and hence can utilize different register assignments). Be sure that your approach can handle subroutine calls.

Describe how to use the results of the analysis to allocate registers to register variables. You may assume that there are an unbounded number of registers; however, the goal is to minimize the number of registers allocated to register variables at each point in the code.

Illustrate your techniques on the following C subprogram:

```
int f()
{
register int i,a,b;
scanf("%d", &a);
for (i=1;i<=100;i++) {
    b = i*i–27;
    printf("%d\n",b);
    if (i< 50)
        b = i*i+50;
    else   b = i*i–50;
    printf("%d\n",b);
  };
return (a+1);
};
```

(b) One of the difficulties in using register variables is that different machine architectures provide different numbers of registers. Overuse of register variables can force a situation in which we have more register variables than we have registers. Rather than terminate compilation, we can *cancel* a register variable; *i.e.*, we can choose to ignore the register attribute of one of the variables that was declared to be a register variable. For example, in the example subprogram shown above, register variable *a* might be chosen to be cancelled. Variable *a* is the best choice because when it is stored in memory rather than in a register, its value is only stored once and only loaded once, while the other variables would require multiple loads and stores.

Assume that we can estimate the number of times each definition and use of a register variable will be executed. Extend your register-allocation scheme of Part (a) to handle cancelling register variables when necessary (the goal is to choose which register variables to cancel so as to minimize the total number of loads and stores that are executed when the program is run).

Illustrate your technique on the example subprogram of Part (a), assuming that 2 registers are available for allocation.

Question 7

This question is about *program slicing* in the very simple language defined below.

```
<program> → <stmt>
<stmt>    → skip
           | VAR := <expr>
           | if <expr> then <stmt> else <stmt> fi
           | while <expr> do <stmt> od
           | <stmt> ; <stmt>
```

The *slice* of a program *P* is taken with respect to a set of variables *V* that is a subset of the variables in program *P*. Intuitively, the slice includes all of the components of *P* that are needed to compute the final values of the variables in *V*.

More precisely, a slice of program *P* taken with respect to variable-set *V* must have the following two properties:

(1) The slice is a program obtained from *P* by replacing zero or more statements of *P* with *skip* statements.

(2) Whenever program *P* halts: (a) the slice halts, and (b) for all variables *v* in *V*, the final value of *v* computed by the slice is the same as the final value of *v* computed by *P*.

Example: Shown below is a program and its slice with respect to the set {*i*}.

PROGRAM *P*	SLICE of *P* WITH RESPECT TO {*i*}
x := 1;	x := 1;
i := x;	i := x;
sum := 0;	skip;
while i < 11 do	while i < 11 do
sum := sum + i;	skip;
i := i + 1	i := i + 1
od;	od;
i := i – 1;	i := i – 1;
mean := sum / i	skip

Slices of program *P* can be computed using a collection of slicing functions, one function for each statement instance in *P*. The function for statement instance *S* is called $SLICE_S$. If *S* is an assignment statement, $SLICE_S$ maps a set of variables to either *S* itself or to a *skip* statement. If *S* is either an *if*-statement, a *while*-statement, or a compound statement, $SLICE_S$ maps *S* to a single *skip* statement or to a statement that is of the same kind as *S* (*i.e.*, *if*, *while*, or compound, respectively), but possibly with some of the substatements within *S* replaced by *skip*s.

The *SLICE* functions can make use of a collection of *VARS* functions (described below); similarly, *VARS* functions can make use of *SLICE* functions.

> $VARS_S(V)$: returns the set of variables whose values before the execution of statement instance *S* might influence the values of the variables in set *V* after the execution of *S*.

CONTINUED NEXT PAGE ⇒

The table given below shows how to define *SLICE* functions for the different kinds of statements in the language. (The auxiliary function REF returns the set of variables used in the given expression. The auxiliary function AllSkips returns true if its argument is a *skip* statement, or a "list" of *skip* statements, such as "*skip*; *skip*; *skip*".)

The slice of a *while* statement can depend on more than one iteration of the loop. However, the only way the entire while statement can become a *skip* is if AllSkips($SLICE_{S1}(V)$) = true. Thus, $VARS_S$ for *while*-statement S can be defined in the following way:

$VARS_S = \lambda V.$ if AllSkips($SLICE_{S1}(V)$) = true then V
else $AUX_S(V)$

AUX_S = ???

where AUX_S computes the influencing variables for 0 iterations, 1 iteration, . . .

Complete the table by giving definitions for $SLICE_S$ and $VARS_S$ when S is of the form "S_1 ; S_2" and by giving definitions for $SLICE_S$ and AUX_S when S is of the form "**while** *expr* **do** S_1". For full credit, write AUX_S in recursive form rather than as an infinite union.

Form of Statement S	$SLICE_S$ and $VARS_S$
skip	$SLICE_S = \lambda V.$ (**skip**) $VARS_S = \lambda V. (V)$
v := *expr*	$SLICE_S = \lambda V.$ if $v \in V$ then (v := *expr*) else (**skip**) $VARS_S = \lambda V.$ if $v \in V$ then $((V - \{v\}) \cup$ REF(*expr*)) else (V)
if *expr* **then** S_1 **else** S_2	$SLICE_S = \lambda V.$ if (AllSkips($SLICE_{S1}(V)$) and AllSkips($SLICE_{S2}(V)$)) then (**skip**) else (**if** *expr* **then** $SLICE_{S1}(V)$ **else** $SLICE_{S2}(V)$) $VARS_S = \lambda V.$ if (AllSkips($SLICE_{S1}(V)$) and AllSkips($SLICE_{S2}(V)$)) then (V) else ($VARS_{S1}(V) \cup VARS_{S2}(V) \cup$ REF(*expr*))
S_1 ; S_2	$SLICE_S$ = ??? $VARS_S$ = ???
while *expr* **do** S_1 **od**	$SLICE_S$ = ??? $VARS_S = \lambda V.$ if AllSkips($SLICE_{S1}(V)$) = true then V else $AUX_S(V)$ AUX_S = ???

Question 8

The language PSIL has global variables and run-time typing of values. It has the standard data types *integer*, *string*, and *array*, (but no records or pointers) as well as the special types SYM and ENV. There is a function called INTERN (of type string $\rightarrow$ SYM), that takes string values to SYMs with the property that

INTERN (S) = INTERN (S') iff $S = S'$

There are also two side-effect-free functions called BIND and LOOKUP whose types are shown below.

BIND: (ENV x SYM x α) $\rightarrow$ ENV
LOOKUP: (ENV x SYM) $\rightarrow$ α

(where α is an arbitrary type)

For an ENV E, SYMs S and S' such that $S \neq S'$, and value X, the following hold:

LOOKUP (BIND (E, S, X), S) = X
LOOKUP (BIND (E, S', X), S) = LOOKUP (E, S)
LOOKUP (NULLENV, S) = undefined

(where the constant NULLENV is an ENV)

In the following questions, you are to indicate how to use the data types and functions of the PSIL language to implement features from more conventional languages. In each case, describe a systematic procedure for translating a program that uses a conventional construct into a PSIL program.

(a) How can you use ENVs to simulate records and pointers to records?

(b) How would you simulate local, lexically-scoped variables in subroutines (as in C)?

(c) How would you simulate dynamically-scoped variables (as in Lisp 1.5)?

Question 9

This question concerns the Church-Rosser property for relations.

Definition (Church-Rosser property). A relation $\longrightarrow$ *has the Church-Rosser property* iff $W \longrightarrow X$ and $W \longrightarrow Y$ implies $\exists Z$ such that $X \longrightarrow Z$ and $Y \longrightarrow Z$.

Instead of the reduction relation on lambda-terms that was studied in CS 704, this question concerns relations that arise in several different transition systems that will be introduced in the different sub-parts of the question.

(a) Consider the following "parity-counting" system. There is an urn containing (only) green and red balls. Transitions are performed according to the following rules: Two balls are drawn at random and then one of the following actions is taken:

(1) If both balls are green, they are both discarded.
(2) If one ball is green and the other one is red, the green one is returned to the urn and the red one is discarded.
(3) If both balls are red, they are both returned to the urn.

The configuration of an urn is given by the pair <# of red balls, # of green balls>. Let $C_1 \longrightarrow C_2$ denote the following relation on urn-configurations: "An urn in configuration C_1 can be transformed to an urn in configuration C_2 by a single application of one of the transition rules."

Does relation $\longrightarrow$ have the Church-Rosser property? If so, give a proof; if not, give a counter-example.

(b) Consider the following string-transition system: Strings are made up of the symbols R and G. If two G's are adjacent, then the two symbols may be removed from the string (*i.e.*, there is a transition rule $GG \Rightarrow \varepsilon$). If an R stands immediately to the left of a G, the two symbols may be transposed (*i.e.*, there is a transition rule $RG \Rightarrow GR$).

Let $S_1 \longrightarrow S_2$ denote the following relation on these strings: "S_1 can be transformed to S_2 by a single application of one of the two transition rules."

Does relation $\longrightarrow$ have the Church-Rosser property? If so, give a proof; if not, give a counter-example.

(c) Let $S_1 \longrightarrow^* S_2$ denote the following relation on the strings from Part (b): "S_1 can be transformed to S_2 by zero or more applications of the transition rules from Part (b)."

Does relation $\longrightarrow^*$ have the Church-Rosser property? If so, give a proof; if not, give a counter-example.