**SPRING 1996**
**COMPUTER SCIENCES DEPARTMENT**
**UNIVERSITY OF WISCONSIN—MADISON**
**PH.D. QUALIFYING EXAMINATION**

Programming Languages and Compilers
Qualifying Examination

Monday, February 5, 1996
3:00 – 7:00 PM
2104 Chamberlain

**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 5 of 6 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.**

Consider the two (recursive) function definitions given below.

```
let rec reduce1( f, L, b ) =
cases L of
  nil:    b
  x::L:   f( x, reduce1( f, L, b ) )

let rec reduce2( f, L, b ) =
cases L of
  nil:    b
  x::L:   reduce2( f, L, f( b, x ) )
```

Given a list of the form $[x_1, x_2, ..., x_n]$, and a value $b$, function *reduce1* returns the value $f(x_1, f(x_2, \cdots f(x_n, b)...))$. Given a list of the form $[x_1, x_2, ..., x_n]$, and a value $b$, function *reduce2* returns the value $f(f(...f(f(b, x_1), x_2)...))$.

Show (using structural induction) that for all $L$, for all $b$, and for all commutative, associative $f$:

reduce1( *f, L, b* ) = reduce2( *f, L, b* )

Be sure to justify every step of your proof.

**Question 2.**

Consider the simple imperative language defined below.

*program* → *cmd*
*cmd* → Id := *exp* | for Id := Int to Int do *cmd* od | *cmd* ; *cmd*
*exp* → Int | Id | *exp* + *exp*

That is, a program is a command, and a command is an assignment, a for-loop, or a command followed by another command; an expression is either an integer literal or an identifier or the sum of two expressions.

A partially defined denotational semantics for this language is given below. A *State* is a mapping from identifiers to values; initial state $\sigma_0$ is the state that maps all identifiers to zero. The meaning function **I** used to define the meaning of an integer literal simply returns the value of its argument; the function "update" used to define the meaning of the assignment command takes three parameters: a state $\sigma$, an identifier *x*, and an integer value *v*, and returns a state that is the same as $\sigma$ except that it maps *x* to *v*.

**Meaning Functions**

**P**: Command → State
**C**: Command → State → State
**E**: Expression → State → Integer

**P**[[ *C* ]] = **C** [[ *C* ]] $\sigma_0$
**C**[[ Id := *exp* ]] = $\lambda\sigma$. update($\sigma$, *Id*, **E**[[*exp*]])
**C**[[ $C_1$ ; $C_2$ ]] = $\lambda\sigma$. **C**[[ $C_2$ ]] (**C**[[ $C_1$ ]] ($\sigma$))
**E**[[ Int ]] = $\lambda\sigma$.**I**[[Int]]
**E**[[ Id ]] = $\lambda\sigma.\sigma$( Id )
**E**[[ $exp_1$ + $exp_2$ ]] = $\lambda\sigma$.(**E**[[$exp_1$]] $\sigma$) + (**E**[[$exp_2$]] $\sigma$)

Part (a)

You are to supply the definition of **C**[[ for Id := $\text{Int}_1$ to $\text{Int}_2$ do *C* od ]]. You must give a reasonable definition for a for-loop (for example, you may not make a for-loop be a no-op or give it some other trivial semantics). In writing the meaning function you may use the *fix* operator (which returns the least-fixed-point of its functional argument), as well as the usual functional constructs (*e.g.*, *let*, *if*).

Part (b)

Given the definition you wrote for part (a), what is the final value of variable *x* in the program shown below (does assigning to the loop's index variable affect the number of times the loop is executed)?

```
x := 0;
for k := 1 to 10 do
  k := 11;
  x := x + 1;
od
```

Part (c)

What is the final value of variable *x* in each of the two programs shown below? In general, what is the value of a loop's index variable immediately after the loop?

Program 1

```
for k := 1 to 10 do
  x := 0;
od;
x := k
```

Program 2

```
for k := 10 to 1 do
  x := 0;
od;
x := k
```

**Question 3.**

Assume we are generating code for a basic block composed of quadruples of the form:

(OP,LEFT,RIGHT,RESULT).

OP is any integer arithmetic operation. LEFT and RIGHT are integer scalar variables or literals. RESULT is an integer scalar variable.

We are generating code for a typical load/store architecture in which operands (including literals) must always be in registers, and results are always computed into registers. One register can be used multiple times in an instruction; for example:

```
Add R1, R2, R1          // R1 = R1 + R2
```

Our goal is to assign registers to operands and results optimally, so that the fewest number of loads and stores are generated. Initial loads of variables and literals are unavoidable. After the last computation of a variable in a basic block, a store is unavoidable (because the variable may be used in succeeding blocks). However, reloads of variables and literals previously loaded are avoidable. Stores of dead variables are also unnecessary.

Part (a)

Assume that each quadruple maps to a single machine-level instruction, and that these instructions are not reordered or altered. Give an algorithm that determines the fewest registers needed to translate a basic block without generating any unnecessary loads or stores. Illustrate your algorithm on the following basic block:

```
(+,A,1,D)
(+,D,A,B)
(+,B,2,C)
(+,A,2,A)
(+,C,B,B)
```

Part (b)

If we have fewer than the minimum number of registers needed, extra loads and/or stores will be generated. A typical "on the fly" register allocator loads and computes values into registers until all registers contain a live value. Then it spills one of the registers (storing the value from the register only if necessary, then loading the new value). A common rule is to spill that register whose value has the most distant next use.

However, this rule is not always optimal. It sometimes leads to more stores than are really necessary. This problem arises for the example from Part (a) when only 3 registers are available. Explain what goes wrong.

**Question 4.**

Most new language designs include some notion of exception handling. In particular, during execution an "exception handler" may be activated. If a corresponding exception is raised, the exception handler is executed.

Part (a)

The scope of an exception handler may be either static or dynamic. Illustrate the difference and explain the strengths and weakness of each approach.

Part (b)

After an exception handler has executed, the program must continue execution somewhere. Explain the possible choices as to where execution might continue. What are the strengths and weakness of each choice? Which might the user prefer? Which is easiest to implement?

Part (c)

An exception raised within an exception handler can be problematic. What semantics would you recommend for such a situation?

**Question 5.**

This question concerns the conversion of a regular expression *R* to a (nondeterministic) finite-state automaton (NDFA). One way to accomplish this is as follows:

(i) Label each occurrence of an alphabet symbol in *R* with a number to distinguish it from all other occurrences of that symbol in *R*. For example, if *R* is $a^*bb(a \mid b)^*$, then the labeled version of *R* would be $a_1^*b_1b_2(a_2 \mid b_3)^*$.

(ii) Create an NDFA with (a) a state for each labeled symbol of *R*, and (b) a transition labeled "*y*" from the state for labeled-symbol $x_i$ to the state for labeled-symbol $y_j$ iff symbol-occurrence $y_j$ is in the "Follow" set of symbol-occurrence $x_i$.

(The machine will also have a start state and some of the states will be labeled as final states (*i.e.*, accepting states). See Part (b) below.)

In order to simplify things somewhat, assume in what follows that regular-expression *R* does not contain any occurrences of the special symbol that denotes the empty string (denoted by "ε", "λ", or "1" in various regular-expression languages).

Part (a)

Describe how to compute the Follow set of a labeled symbol in the labeled version of *R*. (In your answer, you may use any notation that you find convenient, but you should briefly explain the notation used if it is not self-explanatory. You should also explain any auxiliary constructs/functions/etc. that you introduce as part of your method to compute the Follow set.)

Part (b)

How should the start state be handled? How do we determine which states in the machine are final states? How do we handle the case when *R* accepts the empty string?

Part (c)

Demonstrate your construction from Parts (a) and (b) on the regular expression *R* from (i) above.

**Question 6.**

In languages with automatic storage reclamation (i.e., garbage collection), a frequently profitable optimization is to allocate objects with limited lifetimes in a stack frame, rather than on the heap.

Part (a)

What is the advantage of allocating in a stack frame?

Part (b)

Under what conditions can an object be allocated in a stack frame?

Part (c)

Consider a simple language in which a program contains one or more functions, defined as follows:

<function> ::= **function** <fn-name> ( <Id-list> ) <decl-list> <stmt-list>
;
<stmt> ::= Id := <expr>
| Id := ALLOC(Id)
| **while** <cond> **do** <stmt-list> **od**
| **if** <cond> **then** <stmt-list> **else** <stmt-list>
| **return** <expr>
;
<expr> ::= Id
| Int
| <fn-name> ( <expr-list> )
;
<cond> ::= Id
| <expr> == <expr>
;

Arguments to a function are passed by value. There are no non-local variables. ALLOC allocates a new memory cell containing the value of the given identifier, and returns a pointer to the new cell. Note that pointers cannot be dereferenced (e.g., neither "x := *y" nor "*x := ALLOC(y)" is allowed). Note further that passing a pointer by value only copies the pointer, not the structure pointed to.

Describe an intraprocedural static program analysis that can determine whether it is safe for each call on ALLOC in the function to allocate its memory in the current stack frame. Your description should include details such as the data flow equations and how they apply to the statements and expressions in the language.

Part (d)

Explain why allowing pointer dereferencing complicates the analysis. How would you extend your analysis to handle this?