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

Programming Languages
Depth Examination, Part II

Tuesday, February 9, 1993
5:00 – 7:00 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 *three* of the *four* questions 6-9.

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 6

Recall that lambda expressions can be reduced either using *normal order reduction (NOR)*: reduce the leftmost outermost redex, or using *applicative order reduction (AOR)*: reduce the leftmost innermost redex.

Two reduction strategies, S_1 and S_2 are considered to be *equivalent*, if for every lambda expression *e*, either both S_1 and S_2 reduce *e* to normal form, or neither does (*i.e.*, neither terminates).

Part A.

What are the advantages of NOR over AOR and vice versa? Give examples to illustrate your answers.

Part B.

Is the strategy: reduce the rightmost outermost redex equivalent to NOR? If yes, briefly justify your answer. If no, give an example of a lambda expression for which one strategy leads to a normal form while the other strategy fails to terminate.

Part C.

Is the strategy: reduce the rightmost innermost redex equivalent to AOR? If yes, briefly justify your answer. If no, give an example of a lambda expression for which one strategy leads to a normal form while the other strategy fails to terminate.

Question 7

Assume that we are dealing with a language that has only integer-valued scalar variables (*i.e.*, there are no pointers or arrays) and that a program in the language can be represented with a control-flow graph whose nodes represent either assignment statements or control predicates. A control-flow graph also contains a unique entry node and a unique exit node. The exit node is labeled with a distinguished variable that represents the "final answer" computed by the program.

A variable *x* is defined to be *live* at a program point if there exists a path from that point to the end of the program on which *x* is used before being defined. A variable that is not live is *dead*.

An idea related to that of a dead variable is the notion of a *faint* variable. The faint variables at a point are a superset of the dead variables at that point. A variable *x* is faint at a program point if *x* is dead at that point, or if *x* is live only because it is used to define a faint variable. The variable that labels the exit node (the program's "final answer") is not faint at the point just before the end of the program.

These concepts are illustrated in the following two example programs; the comments refer to the program point just after the corresponding statement.

```
begin
   i := 0;              -- i is live and not faint
   x := i;              -- x is live and faint
   y := x               -- y is dead and faint
end(i).                 -- i is the program's final answer

begin
   i := 0;              -- i is live and not faint
   x := 0;              -- x is live and faint; i is live and not faint
   while i ≤ 10 do      -- x is live and faint; i is live and not faint
      x := x + 1;       -- x is live and faint; i is live and not faint
      i := i + 1        -- x is live and faint; i is live and not faint
   end
end(i).                 -- i is the program's final answer
```

Part A.

Recall that the results of live-variable analysis can be used to find and remove useless assignments. What are the advantages of finding and removing useless assignments based on information about faint variables rather than information about live variables?

Part B.

Let's call a variable that is not faint *truly live*. True-liveness analysis seeks to identify the minimal set of truly-live variables at each program point. Give equations for solving the true-liveness problem (*e.g.*, in terms of *trulyLiveBefore*[*n*] and *trulyLiveAfter*[*n*], where *n* is a control-flow graph node).

Part C.

Give an algorithm that computes true-liveness information.

Question 8

For the purposes of this question, assume we are working with a strict functional language. That is, operators such as +, –, =, *etc.* are strict in their arguments; however, the **if**-**then**-**else**-**fi** operator is strict only in the condition argument.

Part A.

Explain what is meant by the term "function caching". (Note: function caching is also known as "tabulation" or "memoization".)

Part B.

Consider the following functional definition of the Fibonacci function:

$$\begin{array}{l} fib : \mathbf{Nat} \to \mathbf{Nat}_\perp \\ fib = \lambda x \,.\, \textbf{if } x = 0 \textbf{ or } x = 1 \textbf{ then} \\ \qquad\qquad 1 \\ \qquad \textbf{else} \\ \qquad\quad fib\,(x-1) + fib\,(x-2) \\ \qquad \textbf{fi} \end{array}$$

Using some imperative language of your own choosing, give a recursive program that corresponds to *fib*, but uses function caching. For this example, what is the performance benefit of using function caching?

Part C.

Let domain **D** be the set { 0, 1, 2, 3 }. Consider the recursive function *f* defined as follows:

$$\begin{array}{l} f : \mathbf{D} \to \mathbf{D}_\perp \\ f = \lambda x \,.\, \textbf{if } x = 0 \textbf{ or } x{=}3 \textbf{ then} \\ \qquad\qquad x^2 \\ \qquad \textbf{else} \\ \qquad\quad (f\,(x-1) + f\,(x+1)) \,/\, 2 \\ \qquad \textbf{fi} \end{array}$$

Recursive function *f* corresponds to the least fixed point of the following functional *F*:

$$\begin{array}{l} F : (\mathbf{D} \to \mathbf{D}_\perp) \to (\mathbf{D} \to \mathbf{D}_\perp) \\ F = \lambda f \,.\, \lambda x \,.\, \textbf{if } x = 0 \textbf{ or } x{=}3 \textbf{ then} \\ \qquad\qquad\quad x^2 \\ \qquad\quad \textbf{else} \\ \qquad\qquad (f\,(x-1) + f\,(x+1)) \,/\, 2 \\ \qquad\quad \textbf{fi} \end{array}$$

What is the least fixed point of *F* in the domain $\mathbf{D} \to \mathbf{D}_\perp$? (Give your answer in the form: $f\,(0) = ?$, $f\,(1) = ?$, $f\,(2) = ?$, $f\,(3) = ?$.) Give a second fixed point of *F*, as well. How many fixed points does *F* have in $\mathbf{D} \to \mathbf{D}_\perp$?

Part D.

Suppose you were to create a function-caching version of *f* (*i.e.*, the analogue of what you did in Part B for function *fib*). Which of the fixed points of *F* would this program correspond to?

Question 9

This question concerns implementation techniques for recursive functions in functional languages.

Part A.

Consider the following recursive function definition, which returns the length of a list:

$$
\begin{array}{l}
\mathit{Length} : \textbf{IntList} \rightarrow \textbf{Int} \\
\mathit{Length}\,(\mathit{list}) = \textbf{cases}\ \mathit{list} \\
\qquad\qquad \mathit{nil}: \qquad\quad 0 \\
\qquad\qquad \mathit{cons}\,(\mathit{hd},\ \mathit{tail}): \quad 1 + \mathit{Length}\,(\mathit{tail}) \\
\qquad\quad \textbf{end}
\end{array}
$$

What is the pattern of storage consumption for a call $\mathit{Length}\,(\mathit{list})$, where *list* is of length n?

Part B.

Consider the following function definition, which returns true or false depending on whether x is a member of *list*:

$$
\begin{array}{l}
\mathit{MemberOf} : \textbf{Int} \times \textbf{IntList} \rightarrow \textbf{Boolean} \\
\mathit{MemberOf}\ (x,\ \mathit{list}) = \textbf{cases}\ \mathit{list} \\
\qquad\qquad\quad \mathit{nil}: \qquad\qquad \mathit{false} \\
\qquad\qquad\quad \mathit{cons}\,(\mathit{hd},\ \mathit{tail}): \quad \textbf{if}\ x = \mathit{hd}\ \textbf{then}\ \mathit{true}\ \textbf{else}\ \mathit{MemberOf}\ (x,\ \mathit{tail}) \\
\qquad\qquad \textbf{end}
\end{array}
$$

Note that function MemberOf is tail recursive. Explain what optimization or optimizations are possible for tail-recursive functions. What is the pattern of storage consumption for a call $\mathit{MemberOf}\ (x,\ \mathit{list})$, where *list* is of length n? In general, what is the pattern of storage consumption for your proposed implementation of tail-recursive functions?

Part C.

"Continuation-passing style" is a paradigm for writing functional programs in which functions receive explicit "continuation" arguments that are invoked within the function instead of returning from the function. In other words, each "return" from a function looks like a call on another function.

> **Aside**. The functional definition obtained as the meaning of a program in a language whose denotational-semantics definition uses "continuations" is one example of a function in continuation-passing style; this question happens to be about functions written directly in continuation-passing style. **End Aside**.

Suppose we rewrite the Length function from Part A as follows:

$$
\begin{array}{l}
\mathit{Length} : \textbf{IntList} \rightarrow \textbf{Int} \\
\mathit{Length}(\mathit{list}) = \mathit{Length}'(\mathit{list},\ \lambda z.z) \\
\\
\textbf{continuation} = \textbf{Int} \rightarrow \textbf{Int} \\
\mathit{Length}' : \textbf{IntList} \times \textbf{continuation} \rightarrow \textbf{Int} \\
\mathit{Length}'(\mathit{list},\ k) = \textbf{cases}\ \mathit{list} \\
\qquad\qquad\quad \mathit{nil}: \qquad\qquad k\,(0) \\
\qquad\qquad\quad \mathit{cons}\,(\mathit{hd},\ \mathit{tail}): \quad \mathit{Length}'(\mathit{tail},\ \lambda\, z\,.\,k\,(1+z)) \\
\qquad\qquad \textbf{end}
\end{array}
$$

In particular, function Length′ is tail recursive and thus the method you described in Part B should apply. In terms of the overall pattern of storage consumption, did we "get something for free" by using this transformation into tail-recursive form? For instance, what is the pattern of storage consumption for a call $\mathit{Length}(\mathit{list})$, where *list* is of length n? In general, by transforming a function into continuation-passing style do we necessarily gain performance benefits?