B Exam January 1997: Programming Languages

Answer 4 of the following 5 questions.

Problem 1: C implements call-by-value (pass-by-value) by copying the bits of an argument's representation into the local storage holding the corresponding formal parameter. This implementation works well for atomic values like numbers, booleans, and characters, but produces anomalous behavior for objects such as structures and arrays.

Explain how this implementation of call-by-value interferes with a semantic interpretation of an object as a value; include an example. (Hint: Consider how both Scheme and Java treat objects and call-by-value.)

Problem 2: Some experts claim that all garbage collection algorithms are "conservative". More specifically, they assert that no automatic memory manager can reclaim all the memory that a program will not access in the remainder of an evaluation.

Question 1: Explain what portion of allocated memory is difficult to reclaim. Give an example showing reclaimable memory that "precise" (non-conservative) garbage collectors miss.

Question 2: Can a C programmer, who has far more control over memory allocation than, say, a Java or a Scheme programmer, overcome this problem?

Problem 3: A former 311 student recalls Milner's famous theorem, which says that "typed programs can't go wrong." Explain what the theorem really asserts and what it does *not* assert. Address the following questions:

Question 1: Consider the Scheme subset consisting of the simply typed *lambda*-calculus where the only ground type is *integer* and the only functions are (curried) addition, *if0* (a conditional expression operator where the test expression is compared against 0), and Y_σ (fixed-point) operators of every type σ. Is it true that any typable program in this language (using the standard type checking rules taken from the simply typed λ-calculus) cannot generate a run-time error.

Question 2: Extend the preceding Scheme subset to include array operations (allocation, access, and update) and the corresponding type system to include an array type constructor. Is it true that a typable program in this expanded language cannot raise a run-time error?

Question 3: Is it true that an untypable program in the array language must raise a run-time error?

Problem 4: Convert the following Scheme function into CPS:

```
(define (quick-sort comp alon)
  (if (null? alon)
      null
      (append
        (quick-sort (lesser comp (cdr alon) (car alon)))
        (equals comp (cdr alon) (car alon))
        (quick-sort (greater comp (cdr alon) (car alon))))))
```

Assume `append`, `lesser`, `equals`, and `greater` are atomic functions.

Problem 5: An object-oriented language supporting inheritance such as Java, C++, or SmallTalk, does not lexically close the object (instance) methods in a class over the other methods defined in the class. (In contrast, Scheme `letrec` performs precisely this form of closure for the procedures introduced in the `letrec`.) Why do OO languages fail to perform this closure? Give an example showing why performing the closure would have undesirable consequences.