Solution notes

Comparative Programming Languages 2005 (MR)
Paper 5 Question 7, Paper 12 Question 7

This question tests understanding of many parts of this course, from problems in ambiguity, type systems, standards, reasons why programs contain errors, formal specification, machine dependence etc.

a) Classification of kinds of programming error. Eg:

*) Simple typos (1000 instead of 1001).
*) Failing to initialise a varible, which by chance get initialised to a safe value.
*) Type errors and interface errors (f(g, n) instead of f(n, g)).
*) Misunderstanding of the language specification (eg x & 1 = 0 in BCPL).
*) Bugs arising from migration to a machine with a different word length, or floating point representation.
*) Bugs arising from ambiguities in the standard, eg features qualified by words such as: implementation dependent, undefined, optional.
*) There may be bugs in the compiler, library or operating system, and these may appear when late in the application's life when the underlying system software is upgraded.

b) The following ideas should help the programmer make fewer errors.

*) Be very familiar with the language standard.

*) Use only features that conform to the standard.

*) Don't write code that might produce different results because of undefined features such as order of evaluation of operands in arithmetic expressions.

*) Don't rely on the precision and rounding properties of floating point.

*) Don't rely on thread priority, or the exact behaviour of synchronisation primitives.

*) Don't assume that x++; or even x=0; are atomic operations.

*) Don't rely on speed of execution or size of compiled code.

*) If possible program cautiously so as to reduce that chance that errors are catastrophic (eg dynamically check for subscript out of range, if this is not done within the language).

*) Use good program development tools that can check that good programming practice is being used.

*) Possibly use a language that make formal proof of correctness possible. This may require the inclusion of annotations (eg assertions) to help a theorem prover.

*) Use a program development system that helps the programmer to avoid syntactic and semantic errors. Let it help to programmer understand what he/she has written. Eg for PL/1 it would be good to warn the programmer that the type of 9+8/3 is DEC(15,14).

The program development system should behave like a tactful super intelligent assistant programmer who is thoroughly familiar with the language standard. Such systems are typically not currently available.

*) When upgrading the system onto new hardware, use an architecture that is upward compatible, or even exactly compatible so that word lengths, character codes, floating point properties etc are exactly the same.

*) Possibly use a machine independent abstract machine so that compiled programs always use the same compiler over the lifetime of the application even though the underlying hardware may change (several times).

c) Anwers with a reasonable number of points such as the following will receive full marks.

*) Less ambiguity in language design at the expense of efficiency of compiled code.

*) More emphasis on compile time checking thatis more advanced that simple type checking.

*) Possible insistence on annotations to help automatic proof of correctness.

*) More reliance on carefully written and checked library code. Good interface checking to ensure the separate modules of a large system are compatible. This will hopefully reduce the chance of errors caused by relinking as system with a new (‘‘better’’) version of the library.

Overall, answers containing the equivalent of more than about 80% of the points made above will obtain full marks provided they are argued sensibly and don’t say anything that is patently wrong.