## SOLUTION NOTES

### Foundations of Functional Programming 2003 Paper 6 Question 10 (ACN)

I think that (a) is most naturally handled by describing a rather direct lambda-reducer that uses environments to cope with the association between names and values. It would probably be reasonable to include at least pseudo-code for it in an answer. Here is a rough version!

```
fun lookup x [] = FAIL
    lookup x ((x,y) :: _ = y
    lookup x (_ :: z) = lookup x z

fun eval id env = lookup id env
    eval (f,g) env = apply f (eval g env) env
    eval (lambda x,a) env = closure(x,a,b)

fun apply (closure x a b) g env = eval b ((x,b)::a)
    apply f g env = apply (eval f env) g env
```

For (b) one can either carefully adjust the code above so that the argument is passed in unevaluated. The code changes are small but the commentary to explain what is being done needs to be clear. Or one can mention the idea of converting to combinators and doing graph reduction on them.

Graph reduction on combinators will NOT be exactly normal both because changes may happen during conversion to combinator form and because overwriting shared tree nodes is a mess. The adapted lambda calculus reduce can be exactly normal order.

For the comparison I mostly expect to observe that the eager version can sometimes do unnecessary work which may slow calculations down but the really heavy use of closures or the fine granularity of combinators carries a heavy cost overhead for the lazy calculator.