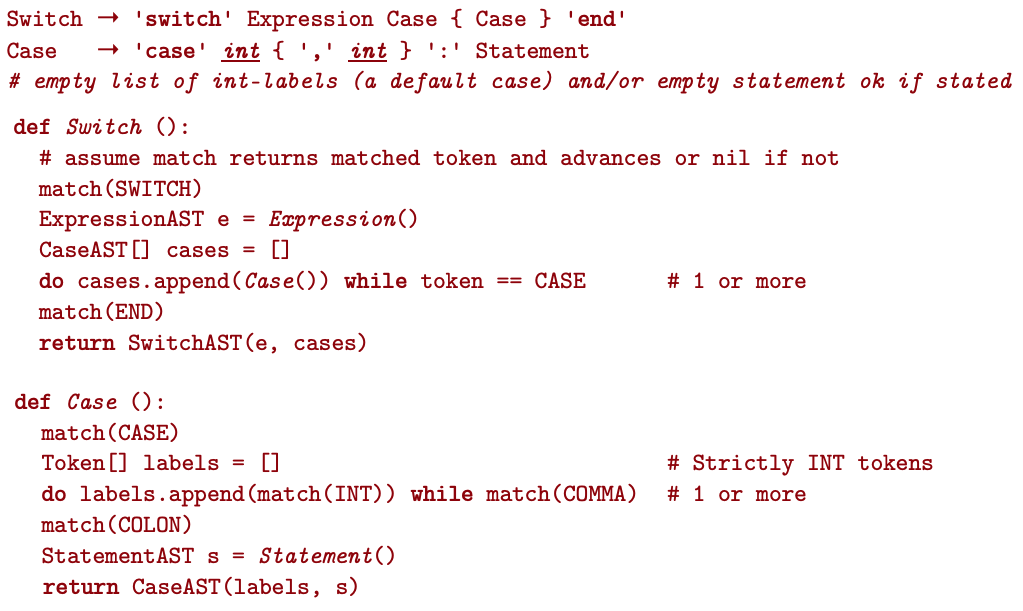
1a)

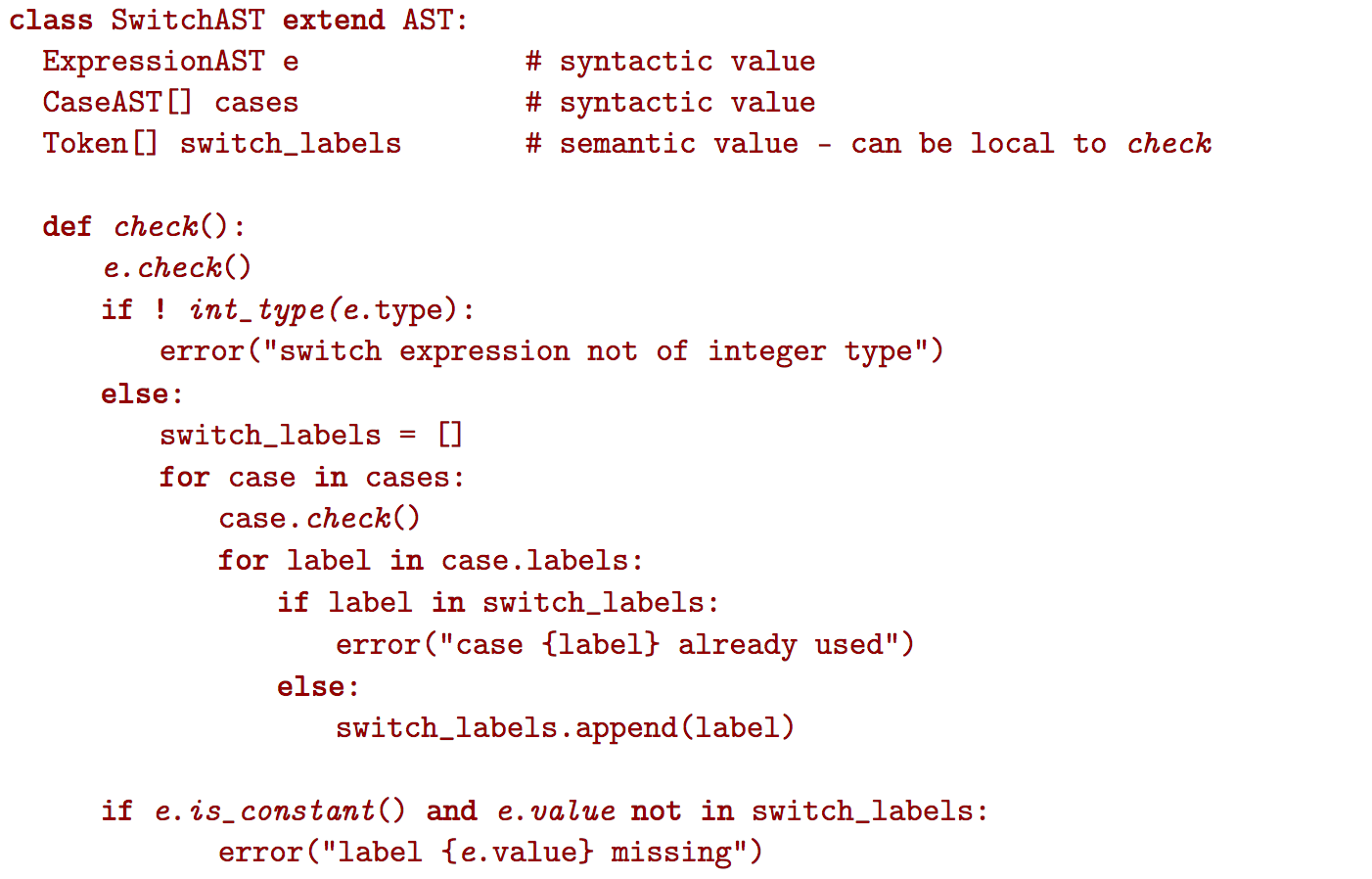
Switch ➝ 'switch' Expression Case { Case } 'end'

Case ➝ 'case' int { ',' int } ':' Statement

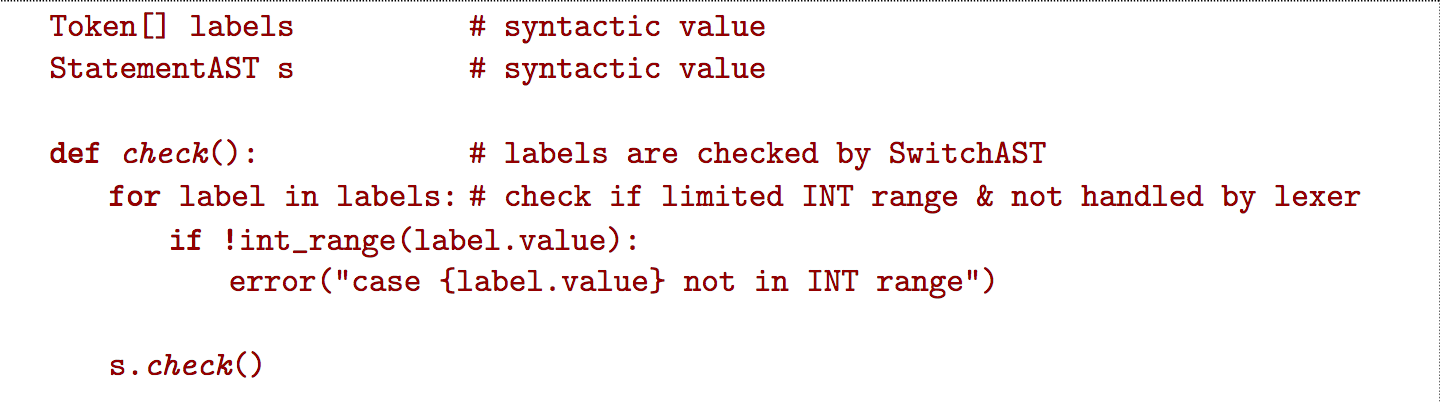
1b)

[[[

1c)







1d)

cmp r0,1

jmpeq stat1

cmp r0,2

jmpeq stat1

cmp r0,3

jmpeq stat1

cmp r0,10

jmpeq stat2

cmp r0,100

jmpeq stat2

jmp error

stat1:

…

jmp end

stat2:

…

jmp end

error:

...

jmp end

end:

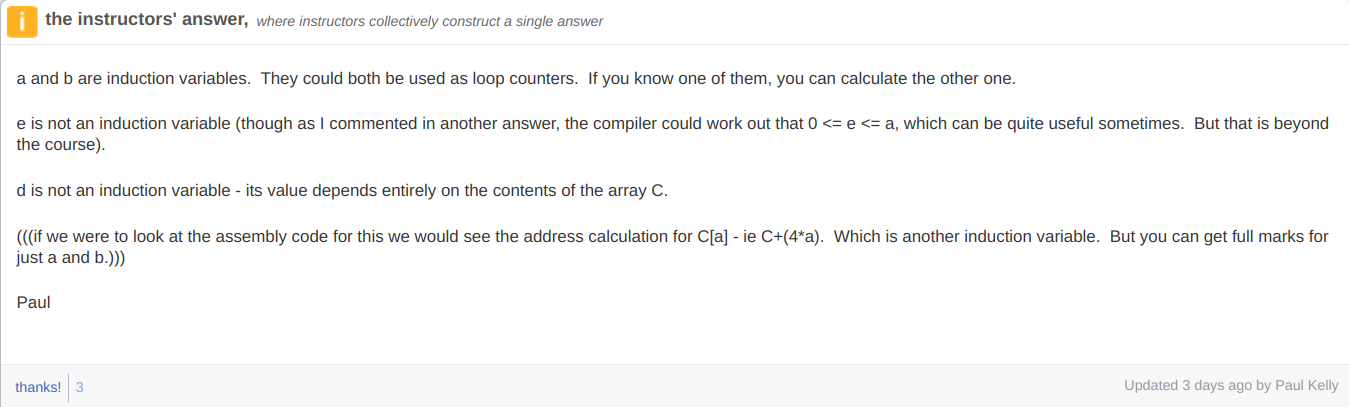
2a)

{S9, S7, S6, S5, S4, S1, S0}

2b)

{S7, S8, S9, S10, S11}

2c)

{a, b}

2d)

d, as the value in it isn’t written it’s only read so its usage should be replaced with c[a]

N, as it’s not written so the cost is only to read and not read and write

Also e can technically be kept on stack since we only have an if check that’s not constant on the loop

‘b’ by replacing “while (b > N)” with “while (100 – a > N)”?

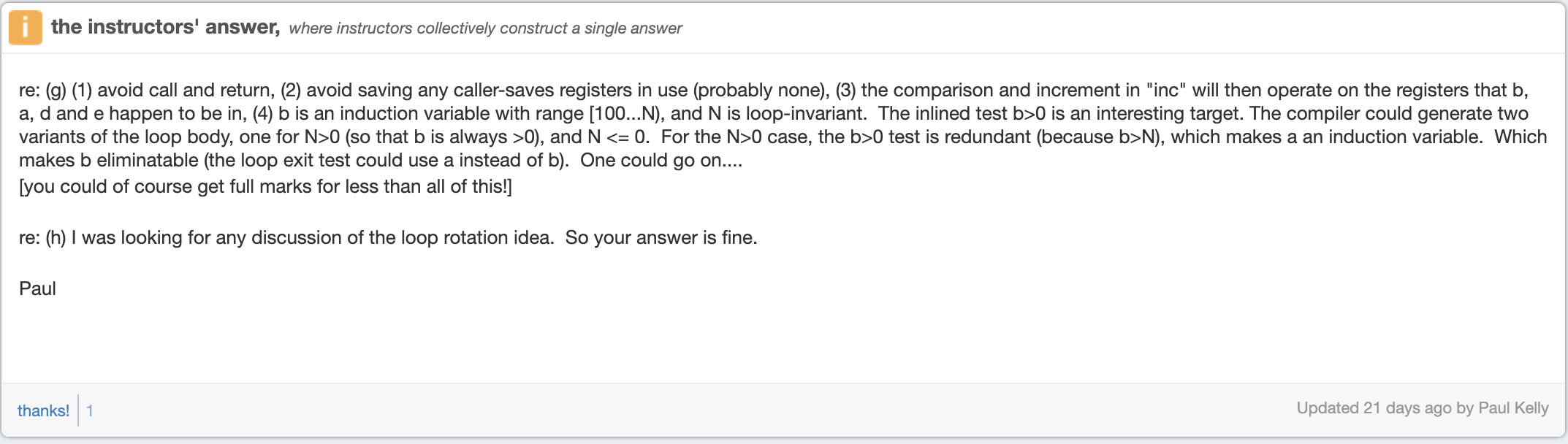
2e) {S13 - S1} S4?

I thought we don’t consider inter-procedural stuff => {S5 - S13}

2f)

Callee saves as it only uses a few registers while count2 uses multiple so if count2 had to save them, it would save registers that woul dn’t have been overwritten.

2g) Not sure about this one ngl but you could potentially unroll the loop, propagate the constants to the conditional jumps and evaluate the conditions at compile-time. You could then apply control flow analysis on top of that to eliminate the if (true) and if (false) that would be left after evaluating.



2h)

The while loop could contain duplicated code so it checks the condition and then has the body of the loop and then code to check the condition at the bottom of the loop as well.

The inc function calls could become inline so there isn’t an unconditional when calling the function.

But then the “if then else” statement requires an unconditional unless as the body of the if statement doesn’t affect the condition, you could have repeated code so if A then B else C becomes if A then B, if not A then C. However, in this case the inc could be implemented using conditional move so it would not need an unconditional jump.