Problem 1 – Scheme

As we discussed in class, let does not add anything to the expressiveness of the language, i.e., they are only a convenient notation. For instance, (let ((x v1) (y v2)) e) can be rewritten as ((lambda (x y) e) v1 v2).

How can you rewrite (let* ((x v1) (y v2) (z v3)) e) in terms of λ-abstractions and function applications?

Problem 2 – Lambda Calculus

Use β-reductions to compute the final answer for the following λ-terms. Note: Use a “fake” reduction step for the “+” operator. Identify each redex for β-reduction steps. Your computation ends with a final result if no more reductions can be applied. Does the order in which you apply the β-reduction make a different whether you can compute a final result? Justify your answer.

1. (((λx.x) (λx.1)) (λy.y))
2. (((λx.((λx.((λx.(x x)) 3)) (λy.(+ x y)))) 1)
3. ((λz. ((λy.z) ((λx.(x x))(λx.(x x))))) 5)

Problem 3 – Programming in Lambda Calculus

In the lecture, we discussed a possible representations of truth values true and false in the lambda calculus, together with lambda calculus implementations of some logical operators.

1. Compute the value of ((or true) false) using β-reductions.

2. Define the and operator in lambda calculus. Prove that your definition is correct, i.e., your lambda term for and implements the logical and operation.
Problem 4 – Lambda Calculus and Combinators S & K

In the lecture, we introduced the S and K combinators:

- \( K \equiv \lambda xy.x \)
- \( S \equiv \lambda xyz.((xz)(yz)) \)

Prove that the identify function \( I \equiv \lambda x.x \) is equivalent to \( ((S \ K) \ K) \), i.e.,

\( I \equiv SKK \)