1. **Find MGUs for the following pairs of type expressions (if exists):**
   1. [T1\*[T1->T2]->N], [[T3->T4]\*[T5->Number]->N]

{ T1 = [T3->T4] , T5 = [T3->T4] ,T2=Number }

* 1. [T1\*[T1->T2]->N], [Number \* [Symbol->T3]->N]

T1 = number & T1 = symbol

number ≠ symbol

Therefore no unifier.

* 1. T1, T2

{T1=T2}

* 1. Number, Number

{}

1. Explain why we can typecheck letrec expressions without specific problems related to recursion and without the need for a recursive environment like we had in the interpreter.
2. In the type equation implementation - we represent Type Variables (TVar) with a content field (which is a box which contains a Type Expression value or #f when empty). In this representation, we can have a TVar refer in its content to another TVar - repeatedly, leading to a chain of TVars. Design a program which, when we pass it to the type inference algorithm, creates a chain of length 4 of Tvar1 → Tvar2 → Tvar3 → Tvar4. Write a test to demonstrate this configuration.

(let ((a 6))

(let ((b a))

(let ((c b))

(let ((d c)) d)

)))

>6

Td → Tc→ Tb → Ta. Ta is number.