Assignment 4

1.

a) $S = \{T2 = number, T1 = [T3 \rightarrow T4], T5 = [T3 \rightarrow T4]\}$
b) $There\ is\ no\ unifier\ because\ T1 = number\ and\ T1 = symbol.$
c) $S = \{T2 = T1\}$
d) $S = \{\}$

2. we can typecheck `letrec` expressions without specific problems related to recursion and without the need for a recursive environment because the type checker passes over the body of each expression only once, whereas the interpreter might pass over each body anywhere from zero to infinite times. That's because the type environment binds identifiers to types and all it has to do is to check the relevant constraints, whereas the interpreter's environment binds identifiers to values or locations (boxes). In simple words, when the interpreter evaluates a letrec expression, it needs to bind variable to a procedure expression. The evaluation of the procedure expression happens on run time and it's on the same scope of the declared variable – thus it might require dealing with recursive environment. The type checker knows the variable and the procedure types in advance, all it needs to do is to compare the two to see their types are equal. No run time calculation is needed. As consequence of that, no need to support recursion.

3. answer: **"(lambda (x) (lambda (y) (lambda (z) (lambda (w) 5))))"**

Test:

```
let array = [];
const ans = JSON.stringify(p(I.inferTypeOf("(lambda (x) (lambda (y) (lambda (z) (lambda (w) 5))))")));

for(let n of flatten(["T_93","-> ",["T_95","-> ",["T_97","-> ",["T_99","-> ","number"]]]]))
      array = array.concat(n);

let i = 0;
let pos = 0;
while((pos = array.indexOf("-> ")) > -1){
  i++;
  array = array.slice(pos + 1);
}
assert.equal(i, 4); // True
```