Q1.1

((lambda (x1 y1) (if (> x1 y1) #t #f)) 8 3)-

Assign type variables for every sub expression:

((lambda (x1 y1) (if (> x1 y1) #t #f)) 8 3)	T_0
(lambda (x1 y1) (if (> x1 y1) #t #f))	T_1
(if (> x1 y1) #t #f)	T_2
(> x1 y1)	T_4
#t	T_5
#f	T_6
>	T_7
x1	T_8
x2	T_9
8	T_{10}
3	T_{11}

Construct type equations:

((lambda (x1 y1) (if (> x1 y1) #t #f)) 8 3)	$T_1 = [T_{10} * T_{11} \rightarrow T_0]$
(lambda (x1 y1) (if (> x1 y1) #t #f))	$T_1 = [T_8 * T_9 \rightarrow T_2]$
(if (> x1 y1) #t #f)	$T_2 = T_5$
(> x1 y1)	$T_4 = \text{Boolean}$
#t	$T_5 = \text{Boolean}$
#f	$T_6 = \text{Boolean}$
>	$T_7 = [\text{Number} * \text{Number} \rightarrow \text{Boolean}]$
8	$T_{10} = \text{Number}$
3	$T_{11} = \text{Number}$

Solving the equations:

~~$T_1 = [T_{10} * T_{11} \rightarrow T_0]$~~	$T_1 = [T_{10} * T_{11} \rightarrow T_0]$
$T_1 = [T_8 * T_9 \rightarrow T_2]$	
$T_2 = T_5$	
$T_4 = \text{Boolean}$	
$T_5 = \text{Boolean}$	
$T_6 = \text{Boolean}$	
$T_7 = [\text{Number} * \text{Number} \rightarrow \text{Boolean}]$	
$T_{10} = \text{Number}$	
$T_{11} = \text{Number}$	

~~$T_1 = [T_8*T_9 \rightarrow T_2]$~~	$T_1 = [T_{10}*T_{11} \rightarrow T_0]$
$T_2 = T_5$	
$T_4 =$ Boolean	
$T_5 =$ Boolean	
$T_6 =$ Boolean	
$T_7 =$ [Number*Number→Boolean]	
$T_{10} =$ Number	
$T_{11} =$ Number	
$T_8 = T_{10}$	
$T_9 = T_{11}$	
$T_2 = T_0$	

~~$T_2 = T_5$~~	$T_1 = [T_{10}*T_{11} \rightarrow T_0]$ $T_2 = T_5$ $T_4 =$ Boolean
~~$T_4 =$ Boolean~~	
$T_5 =$ Boolean	
$T_6 =$ Boolean	
$T_7 =$ [Number*Number→Boolean]	
$T_{10} =$ Number	
$T_{11} =$ Number	
$T_2 = T_0$	

~~$T_5 =$ Boolean~~	$T_1 = [T_{10}*T_{11} \rightarrow T_0]$ $T_2 =$ Boolean $T_4 =$ Boolean $T_5 =$ Boolean $T_6 =$ Boolean
~~$T_6 =$ Boolean~~	
$T_7 =$ [Number*Number→Boolean]	
$T_{10} =$ Number	
$T_{11} =$ Number	
$T_2 = T_0$	

~~$T_7 =$ [Number*Number→Boolean]~~	$T_1 = [T_{10}*T_{11} \rightarrow T_0]$ $T_2 =$ Boolean $T_4 =$ Boolean $T_5 =$ Boolean $T_6 =$ Boolean $T_7 =$ [Number*Number→Boolean]
$T_{10} =$ Number	
$T_{11} =$ Number	
$T_2 = T_0$	

~~$T_{10} =$ Number~~	$T_1 =$ [Number*Number→ T_0] $T_2 =$ Boolean $T_4 =$ Boolean $T_5 =$ Boolean $T_6 =$ Boolean $T_7 =$ [Number*Number→Boolean] $T_{10} =$ Number $T_{11} =$ Number
~~$T_{11} =$ Number~~	
$T_2 = T_0$	

~~$T_2 = T_0$~~	$T_1 =$ [Number*Number→ Boolean] $T_2 =$ Boolean $T_4 =$ Boolean $T_5 =$ Boolean $T_6 =$ Boolean $T_7 =$ [Number*Number→Boolean] $T_{10} =$ Number $T_{11} =$ Number $T_0 =$ Boolean

$T_0 =$ Boolean => ((lambda (x1 y1) (if (> x1 y1) #t #f)) 8 3) is of type Boolean.

Q1.2

a. {f:[T1->T2], x: T1} |- (f x): T2

True.

In the given environment; x is from type T1, and f is a function from T1 to T2. Hence, the output of provoking f on x, is from type T2.

b. {f:[T1->T2] ,g: [T2->T3]}, x: T2} |- (f g x): T3

False.

(f g x) is not a valid expression because f accept 1 operand of type T1, hence False

c. {f:[T2->T1],g: [T1->T2], x: T1}|- (f (g x)): T1

True.

In the given environment x is from type T1, and g operates on T1 and output T2 => g(x) : T2

f is a function from T2 to T1 => g(x) is in the range of f so the operation is valid, and the output will be from type T1

=> f (g(x))

x:T1

=>g(x) :T2

=>f(g(x)) : T1

d. {f:[T2->Number], x: Number}|- (f x x): Number

False.

f doesn’t operate on 2 operands, hence the expression is invalid.

3.

a. cons :: [T1*T2 -> Pair <T1,T2>]

b. car :: [Pair<T1,T2> -> T1]

c. cdr :: [Pair<T1,T2> -> T2]

4.

(Define f (lambda (x) (values x x x)))

[T1-> (T1 * T1 * T1)]

If x:T1 => (values x x x) = (x * x * x)

5. Write the MGU of the following expressions, or state that there is no such MGU.

a. T1 , T2 => {T1 = T2}

b. Number , Number => {}

c. [T1*[T1->T2]->Number] , [[T3->Number]*[T4->Number]->Number]

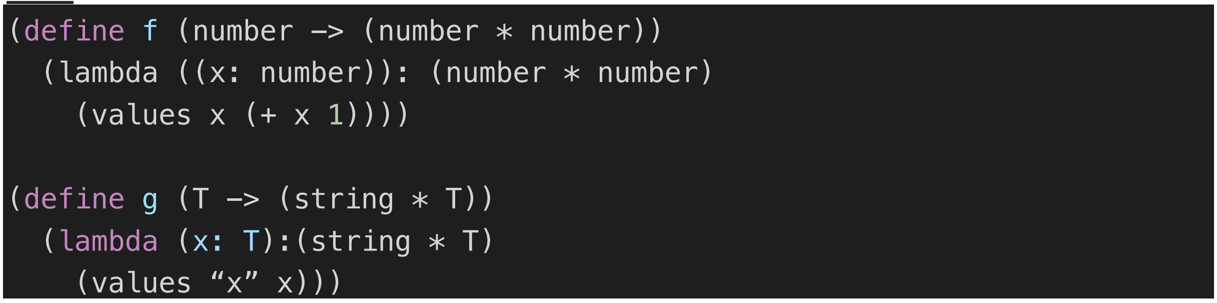
=> {T1=T4=[T3->Number],T2=Number}

d. [T1->T1] , [T1->[Number->Number]] =>

{T1 = [Number -> Number]}

There is no unifier that will satisfies those expressions.

Q2.3



```
(define f (number -> (number * number))
  (lambda ((x: number)): (number * number)
    (values x (+ x 1))))

(define g (T -> (string * T))
  (lambda (x: T):(string * T)
    (values “x” x)))
```

Q4.b

The use of promises helps us write a cleaner code, using chained promises.
Also we have the ability to handle errors of few promises with one catch.