Theoritical Questions

Part1:

Q1:
((lambda (x1 y1) (if (> x1 y1) #t #f)) 8 3)
Stage 1: Rename bound variables.
((lambda (x y) (if (> x y) #t #f)) 8 3)
Stage 2: Assign type variables for every sub expression:

((lambda (x y) (if (> x y) #t #f)) 8 3)	T0
(lambda (x y) (if (> x y) #t #f))	T1
(if (> x y) #t #f)	T2
(x y)	T3=[T5*T6]
(> x y)	T4
x	T5
y	T6
8	T7
3	T8
#t	T9
#f	T10

Stage 4: Construct type equations. The equations for the sub-expressions are:

((lambda (x y) (if (> x y) #t #f)) 8 3)	T1= [T7*T8 -> T0]
(lambda (x y) (if (> x y) #t #f))	T1=[T3 -> T2]
(if (> x y) #t #f)	T9=T10
	T2=T9
(> x y)	T4= [T5* T6 -> Boolean]
(x y)	T3=[T5*T6]

x	T5
y	T6
8	T7=Number
3	T8= Number
#t	T9= Boolean
#f	T10= Boolean

Stage 5:

T1=[T7*T8 -> T0]
T1=[T5*T6 ->T2]
T3= [T5 * T6]
T4= [T5 * T6 -> Boolean]
T5=T7
T6=T8
T0=T2
T9=T10
T2= T9
T7= Number
T8= Number
T9= Boolean
T10= Boolean

Stage6:

T1= [Number*Number -> Boolean]
T3= [Number*Number]
T4=[Number*Number-> Boolean]
T5=Number
T6=Number
T0=Boolean
T2=Boolean
T7=Number
T8=Number
T9=Boolean
T10=Boolean

<u>Q2:</u>

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

This typing statement is true because the function gets two parameters: function which gets value of type T1 and returns value of type T2, and parameter of value T1. The body evaluates the parameter f on the value x which indeed is the type f supposed to get. and then f returns T2 and the main function returns T2 as it is shown in the typing statement.

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

This typing statement is false. because the body first evaluates the second parameter g (which is supposed to get T2 and return T3) on x which is of type T2. so g returns value of T3 and then f tries to evaluate itself on g returned value, but f supposed to get T1 and g returns T3 so we get a failure.

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

This typing statement is true. because the body first evaluates the second parameter g (which is supposed to get T1 and return T2) on x which is of type T1. so g returns value of T2 and then f evaluates itself on g returned value, which is indeed of type T2. so f returns T1 and that's the returned value that specified in the statement.

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

This statement is true because in the body f is activated on the parameter xx which is number, and f gets value of T2 (which is any) so it can get a parameter of type number. f returns number and so the main function returns Number as it is written in the statement.

Q3**:**

a. cons return the full exp so its type will be:
   [T1*T2*T3…. Tn]- > [T1*T2*T3… Tn]
b. car return the first exp so its type will be:
   [T1*T2*T3…. Tn]- > T1
c. cdr return the all the exp without the first one so its type will be:
   [T1*T2*T3…. Tn]- > [T2*T3… Tn]

Q4:

The type of the following function: (Define f (lambda (x) (values x x x))) is a tuple of three x's type. Because the type of every function is it's returning value and f's returning value is (values x x x) which is a tuple of three values of x's type.

Q5**:**

a. T1 , T2
T1=T2
b. Number , Number
any MGU will solve this even empty mgu {}
c. [T1*[T1->T2]->Number] , [[T3->Number]*[T4->Number]->N]:
T1=T3, T1=[T3->Number], T1=T4
d. [T1->T1] , [T1->[Number->Number]
T1=[Number->Number]

Part 2:

1. (define f (lambda (x:**Number**) (values x (+ x 1))))
2. (define g (lambda (x:**T1**) (values "x" x)))

Part 4a:

Callbacks generally require a seperate level for each step in a sequence of operations, and the error handling branches are also nested inside the callbacks. When we use promises we write code that his structure reflects the sequence of calls we intend.
Error handling in promise can also be centralized into a single catch handler instead of repeated for each invocation in callback.