Part 1:

1. false- g expect to receive a variable of type T1, and indeed a is of type T1. so g:[T1 ->T2] returns a T2, but f also expects to receive a variable of type T1, that is why this typing statement is false.
2. true- f is a function that takes a T1 variable and returns a variable of type T2. as we activate f on variable y of type T1, we indeed receive a result of type T2 and that is why this typing statement is true.
3. false- this is false because having no type assumption on x might not satisfy the well-typing rules of Scheme(x may not be of type T1), and create a runtime error while trying to apply f on x.
4. false- this is false because having no type assumption on x might not satisfy the well-typing rules of Scheme(x may not be of type T1), and create a runtime error while trying to apply f on x and 100. Also, there are no statements in the environment that tells us that TNumber is T2.

2.

a) **Stage I**: Rename: ((lambda (x1) (+ x1 1)) 4) **To**: ((lambda (x) (+ x 1)) 4)

Stage II: Assign type variables for every sub expression:

Expression	Variable
((lambda (x) (+ x 1)) 4)	T0
(lambda (x) (+ x 1))	T1
(+ x 1)	T2
+	T+
x	Tx
1	Tnum1
4	Tnum4

Stage III: Construct type equations.

The equations for the sub-expressions are:

Expression	Equation

((lambda (x) (+ x 1)) 4)	T1 =[Tnum4 -> T0]
((lambda (x) (+ x 1))	T1 = [Tx -> T2]
(+ x 1)	T+ = [Tx * Tnum1 -> T2]

The equations for the primitives are:

Expression	Equation
+	T+ = [Number * Number -> Number]
1	Tnum1 = Number
4	Tnum4 = Number

Stage IV: Solve the equations.

Equation	Substitution
1. T1 =[Tnum4 -> T0]	{}
2. T1 = [Tx -> T2]	
3. T+ = [Tx * Tnum1 -> T2]	

4. T+ = [Number * Number -> Number]	
5. Tnum1 = Number	
6. Tnum4 = Number	

step 1:

Equation	Substitution
2. T1 = [Tx -> T2]	**{ T1 := [Tnum4 -> T0] }**
3. T+ = [Tx * Tnum1 -> T2]	
4. T+ = [Number * Number -> Number]	
5. Tnum1 = Number	
6. Tnum4 = Number	

step 2:

Equation	Substitution
3. T+ = [Tx * Tnum1 -> T2]	{ T1 := [Tnum4 -> T0] }

4. T+ = [Number * Number -> Number]	
5. Tnum1 = Number	
6. Tnum4 = Number	
7. Tx = Tnum4	
8. T2 = T0	

step 3:

Equation	Substitution
4. T+ = [Number * Number -> Number]	{ T1 := [Tnum4 -> T0] }, **{ T+ = [Tx * Tnum1 -> T2]}**
5. Tnum1 = Number	
6. Tnum4 = Number	
7. Tx = Tnum4	
8. T2 = T0	

step 4:

Equation	Substitution

5. Tnum1 = Number	{ T1 := [Tnum4 -> T0] }, { T+ = [Tx * Tnum1 -> T2]}
6. Tnum4 = Number	
7. Tx = Tnum4	
8. T2 = T0	
9. **T2 = Number**	

Skipping 2 trivial steps, Tnum1 and Tnum4:

step 5:

Equation	Substitution
7. Tx = Tnum4	{ T1 := [**Number** -> T0] }, { T+ = [Tx * **Number-**> T2]}, **Tnum1 := Number** **Tnum4 := Number**
8. T2 = T0	
9. T2 = Number	

step 6:

<table>
<tr><th>Equation</th><th>Substitution</th></tr>
<tr><td>8. T2 = T0</td><td>{ T1 := [Number -> T0] }, { T+ =
[Number * Number-> T2]},
Tnum1 := Number
Tnum4 := Number
Tx := Number</td></tr>
<tr><td>9. T2 = Number</td><td></td></tr>
</table>

step 7:

<table>
<tr><th>Equation</th><th>Substitution</th></tr>
<tr><td>9. T2 = Number</td><td>{ T1 := [Number -> T0}, { T+ =
[Number * Number-> T0]},
Tnum1 := Number
Tnum4 := Number
Tx := Number,
T2 = T0</td></tr>
</table>

step 8 :

<table>
<tr><th>Equation</th><th>Substitution</th></tr>
<tr><td></td><td>{ T1 := [Number -> T0}, { T+ =
[Number * Number-> T0]},
Tnum1 := Number,
Tnum4 := Number,
Tx := Number,
T2 = Number,
T0 :=Number</td></tr>
</table>

The type inference succeeds, meaning that the expression is well typed. Because there are no free variables, the inferred type of T0 is: Number.

b) stage 1: rename bound variables.
((lambda (f1 x1) (f1 x1 1)) 4 +) turns to ((lambda (f x) (f x 1)) 4 +).
stage 2:Assign type variables for every sub expression:

Expression	Variable
((lambda (f x) (f x 1)) 4 +)	T0
(lambda (f x) (f x 1))	T1
(f x 1)	T2
f	Tf
x	Tx
1	Tnum1
4	Tnum4
+	T+

stage 3: Construct type equations. The equations for the sub-expressions are:

Expression	Equation
((lambda (f x) (f x 1)) 4 +)	T1 = [Tnum4 * T+] -> T0]

(lambda (f x) (f x 1))	T1 = [Tf * Tx -> T2]
(f x 1)	Tf = [Tx * Tnum1 -> T2]

The equations for the primitives are:

Expression	Equation
1	Tnum1 = Number
4	Tnum4 = Number
+	T+ = [Number * Number -> Number]

stage 4: Solve the equations:

Equation	Substitution
1.T1 = [Tnum4 * T+] -> T0]	{}
2.T1 = [Tf * Tx -> T2]	
3.Tf = [Tx * Tnum1 -> T2]	

4.Tnum1 = Number	
5. Tnum4 = Number	
6. T+ = [Number * Number -> Number]	

step 1:

Equation	Substitution
2.T1 = [Tf * Tx -> T2]	{ T1 = [Tnum4 * T+] -> T0] }
3.Tf = [Tx * Tnum1 -> T2]	
4.Tnum1 = Number	
5. Tnum4 = Number	
6. T+ = [Number * Number -> Number]	

step 2:

Equation	Substitution
3.Tf = [Tx * Tnum1 -> T2]	{T1 = [Tnum4 * T+] -> T0]}
4.Tnum1 = Number	
5. Tnum4 = Number	
6. T+ = [Number * Number -> Number]	
7. Tf = Tnum4	
8. Tx = T+	
9.T2 = T0	

step 3:

Equation	Substitution
4.Tnum1 = Number	{ T1 = [Tnum4 * T+] -> T0], Tf = [Tx * Tnum1 -> T2] }
5. Tnum4 = Number	
6. T+ = [Number * Number -> Number]	

7. Tf = Tnum4	
8. Tx = T+	
9.T2 = T0	

step 4:

Equation	Substitution
5. Tnum4 = Number	{ T1 = [Tnum4 * T+] -> T0], Tf = [Tx * **Number** -> T2], Tnum1 = Number}
6. T+ = [Number * Number -> Number]	
7. Tf = Tnum4	
8. Tx = T+	
9.T2 = T0	

step 5:

Equation	Substitution

6. T+ = [Number * Number -> Number]	{ T1 = [**Number** * T+] -> T0], Tf = [Tx * Number -> T2], Tnum1 = Number, Tnum4 = Number }
7. Tf = Tnum4	
8. Tx = T+	
9.T2 = T0	

step 6:

Equation	Substitution
7. Tf = Tnum4	{ T1 = [Number * **[Number * Number -> Number]**] -> T0], Tf = [Tx * Number -> T2], Tnum1 = Number, Tnum4 = Number, T+ = [Number * Number -> Number]}
8. Tx = T+	
9.T2 = T0	

step 7: (Tf = Tnum4) ○ Substitution = ([Tx * Number -> T2] = Number).

we got a conflicting equation, so we can say that the expression is **not** well typed.

<u>Question 2.2</u>

b)The wrapped function asycMemo returns a Promise<R> type because it is an async function (or in our case, the helper function is the async one), and every async function must return a type Promise (even if we didn't return a Promise by intention, because the function is async it would wrap the return value in a Promise).

<u>Part 3- Typing rules for Define and Set!:</u>

<u>Typing rule define:</u>

for every: type environment _Tenv,

variable _x1,

expression _e1 and

type expressions _S1,_U1:

If _Tenv o { _x1 : _S1) |- _e1 : U1

then _Tenv |- (define _x1 _e1) : void

<u>Set!:</u>

for every: type environment _Tenv,

variable _x1,

expression _e1 and

type expression _S1:

If _Tenv |- _x1:S1

_Tenv |- _e1:S1

Then _Tenv |- (Set! _x1 _e1) : Void