1.

(a) false, because we dont know if type variable T1 equals to number
, therefor we cant know if we can activate function g on the number a.

(b)false, function f takes one parameter of the type T2,but in (f x) f takes parameter x of the type T1
, therefor we cant know if we can activate function f on the T1 x.

(c) false, (f x) f takes parameter x of the type T1 and return T2 type,(f x):T2.
((lambda () (f x))) takes no parameters and returns T2,then((lambda () (f x))):() -> T2.
Proof By Contradiction ((lambda () (f x))):T2, then T2 = () -> T2.for every type-variable T
, type expression of it does not include T.Contradiction.

(d) true, f takes two parameters of the type T1 and T2 and return T3 type.(f x y), y: T2 thus parameter x should be of the type T1,(lambda (x) (f x y)) takes parameter x and returns the type of (f x y), therefor (lambda (x) (f x y)):T1 -> T3

2.

(a)

((lambda (f x1) (f 1 x1)) + #t)
Rename bound variables.
((lambda (f x) (f 1 x)) + #t)

Assign type variables for every sub expression:
((lambda (f x) (f 1 x)) + #t) : T0
(lambda (f x) (f 1 x)) : T1
(f 1 x): T2
f : Tf
x1 : TX
1 : Tnum1
+ : T+
#t : T#t

Construct type equations.

((lambda (f x) (f 1 x)) + #t) : T1 = [T+*T#t -> T0]
(lambda (f x) (f 1 x)): T1 = [Tf*TX -> T2]
(f 1 x1): Tf = [Tnum1*Tx1 -> T2]
1 : Tnum1 = Number
+ : T+= [Number * Number -> Number]
#t : T#t = boolean

Solve the equations.
Step 1:
Substitution:
{T1 = [T+*T#t -> T0]}
Equation:
T1 = [Tf*TX -> T2]
Tf = [Tnum1*Tx -> T2]
Tnum1 = Number
T+= [Number * Number -> Number]
T#t = boolean

Step 2:
Substitution:
{T1 = [T+*T#t -> T0]}
Equation:
Tf = [Tnum1*Tx -> T2]
Tnum1 = Number
T+= [Number * Number -> Number]
T#t = boolean
Tf = T+
TX = T#t
T2 = T0

Step 3:
Substitution:
{T1 = [T+*T#t -> T0],
Tf = [Tnum1*Tx -> T2]}
Equation:
Tnum1 = Number
T+= [Number * Number -> Number]
T#t = boolean
Tf = T+
TX = T#t
T2 = T0

Step 4:
Substitution:
{T1 = [T+*T#t -> T0],
Tf = [Number*Tx -> T2],
Tnum1 = Number}
Equation:
T+= [Number * Number -> Number]
T#t = boolean
Tf = T+
TX = T#t
T2 = T0

Step 5:
Substitution:
{T1 = [[Number * Number -> Number]*T#t -> T0],
Tf = [Number*Tx -> T2],
Tnum1 = Number,
T+= [Number * Number -> Number]}
Equation:
T#t = boolean
Tf = T+
TX = T#t
T2 = T0

Step 6:
Substitution:
{T1 = [[Number * Number -> Number]*boolean -> T0],
Tf = [Number*Tx -> T2],
Tnum1 = Number,
T+= [Number * Number -> Number],
T#t = boolean}

Equation:
Tf = T+
TX = T#t
T2 = T0

Step 6:
Substitution:
{T1 = [[Number * Number -> Number]*boolean -> T0],
Tf = [Number*Tx -> T2],
Tnum1 = Number,
T+= [Number * Number -> Number],
T#t = boolean}
Equation:
TX = T#t
T2 = T0
Tx = Number
T2 = Number

Step 7:
Substitution:
{T1 = [[Number * Number -> Number]*boolean -> T0],
Tf = [Number*boolean -> T2],
Tnum1 = Number,
T+= [Number * Number -> Number],
T#t = boolean,
TX = boolean}
Equation:
T2 = T0
Tx = Number
T2 = Number

Step 8:
Substitution:
{T1 = [[Number * Number -> Number]*boolean -> T0],
Tf = [Number*boolean -> T0],
Tnum1 = Number,
T+= [Number * Number -> Number],
T#t = boolean,
TX = boolean,
T2 = T0}
Equation:
Tx = Number
T2 = Number

We get the conflicting equation:
number = boolean and we can say that the expression is not well typed.

(b)

((lambda (f1 x1) (f1 x1 1)) + *)
Rename bound variables.
((lambda (f x) (f x 1)) + *)

Assign type variables for every sub expression:
((lambda (f x) (f x 1)) + *) : T0
(lambda (f x) (f x 1)) : T1
(f x 1): T2
f : Tf
x : TX
1 : Tnum1
+ : T+
* : T*

Construct type equations.

((lambda (f x) (f x 1)) + *) : T1 = [T+*T* -> T0]
(lambda (f x) (f x 1)): T1 = [Tf*TX -> T2]
(f x 1): Tf = [Tx*Tnum1 -> T2]
1 : Tnum1 = Number
+ : T+= [Number * Number -> Number]
#t : T* = [Number * Number -> Number]

Solve the equations.
Step 1:
Substitution:
{T1 = [T+*T* -> T0]}
Equation:
T1 = [Tf*TX -> T2]
Tf = [Tx*Tnum1 -> T2]
Tnum1 = Number
T+= [Number * Number -> Number]
T* = [Number * Number -> Number]

Step 2:
Substitution:
{T1 = [T+*T* -> T0]}
Equation:
Tf = [Tx*Tnum1 -> T2]
Tnum1 = Number
T+= [Number * Number -> Number]
T* = [Number * Number -> Number]
Tf = T+
TX = T*
T2 = T0

Step 3:
Substitution:
{T1 = [T+*T* -> T0],
Tf = [Tx*Tnum1 -> T2]}
Equation:
Tnum1 = Number
T+= [Number * Number -> Number]
T* = [Number * Number -> Number]
Tf = T+
TX = T*
T2 = T0

Step 4:

Substitution:
{T1 = [T+*T* -> T0],
Tf = [Tx*Number -> T2],
Tnum1 = Number}
Equation:
T+= [Number * Number -> Number]
T* = [Number * Number -> Number]
Tf = T+
TX = T*
T2 = T0

Step 5:
Substitution:
{T1 = [[Number * Number -> Number]*T* -> T0],
Tf = [Tx*Number -> T2],
Tnum1 = Number,
T+= [Number * Number -> Number],
}
Equation:
T* = [Number * Number -> Number]
Tf = T+
TX = T*
T2 = T0

Step 6:
Substitution:
{T1 = [[Number * Number -> Number]*[Number * Number -> Number] -> T0],
Tf = [Tx*Number -> T2],
Tnum1 = Number,
T+= [Number * Number -> Number],
T* = [Number * Number -> Number]}
Equation:
Tf = T+
TX = T*
T2 = T0

Step 7:
Substitution:
{T1 = [[Number * Number -> Number]*[Number * Number -> Number] -> T0],
Tf = [Tx*Number -> T2],
Tnum1 = Number,
T+= [Number * Number -> Number],
T* = [Number * Number -> Number]}
Equation:
TX = T*
T2 = T0
Tx = Number
T2 = Number

Step 8:
Substitution:
{T1 = [[Number * Number -> Number]*[Number * Number -> Number] -> T0],
Tf = [[Number * Number -> Number]*Number -> T2],
Tnum1 = Number,
T+= [Number * Number -> Number],

T* = [Number * Number -> Number],
TX = [Number * Number -> Number]}
Equation:
T2 = T0
Tx = Number
T2 = Number

Step 9:
Substitution:
{T1 = [[Number * Number -> Number]*[Number * Number -> Number] -> T0],
Tf = [[Number * Number -> Number]*Number -> T0],
Tnum1 = Number,
T+= [Number * Number -> Number],
T* = [Number * Number -> Number],
TX = [Number * Number -> Number],
T2 = T0}
Equation:
Tx = Number
T2 = Number

We get the conflicting equation:
Number = [Number * Number -> Number], and we can say that the expression is not well typed.

Typing rule set!:
(set! var val)
tenv-val = extend-tenv(var:texp; tenv)
If type<val>(tenv) = texp
then type<(set! var val)>(tenv) = void

Typing rule quote:
(quote val)
If type<val>(tenv) = texp
then type<(quote val)> = texp

Typing rule Define-type:
(define-type UD (R1 ...) ... (Rn ...))
For every: type environment Tenv,
type define UD, n >= 0 , and
records R1, ..., Rn,
If Ri != Rj for all 1 <= i != j <= n
Then Tenv |- ((define-type UD (R1 ...) (R2 ...))) : void

Typing rule Type-case:
(type-case UD val (R1 (...) e1) ... (Rn (...) en))
For every: type environment Tenv,
type define UD, n >= 0 , and
cases R1, ..., Rn,
expressions e1, ..., en
If type<val>(Tenv) = Ri, 1 <= i <= n, and
type<ei>(Tenv) = texp

then type<(type-case UD val (R1 (...) e1) ... (Rn (...) en))> = texp