Part 1: Theoretical

1.

a. false- number is not T1.

b. false- f gets a type of T2 and x is type of T1.

c. true-f gets a type of T1 and x is type of T1, f returns type of T2 and we get apply of lambda.

d. true - In this case if the lamda is called with x of type T1 we get the type of return value is the type of activation f on x and y which is actually a type of appliction i.e. the type of return value of procedure f, which is T3.

2.

a.

stage 1: rename bound variables

((lambda (f x) (f 1 x)) + #t)

Stage 2: assign type variables to all sub-exp.

expression	var
((lambda (f x) (f 1 x)) + #t)	T0
(lambda (f x) (f 1 x))	T1
(f 1 x)	T2
f	Tf
x	Tx
+	T+
#t	Tt
1	Tnum1

Stage 3: construct type equations

expression	equation
((lambda (f x) (f 1 x)) + #t)	T1=[T+*Tt->T0]
(lambda (f x) (f 1 x))	T1=[Tf*Tx->T2]
+	T+=[Number*Number->Number]
#t	Tt=Boolean
(f 1 x)	Tf=[Tnum1*Tx->T2]
1	Tnum1=Number

Stage 4: solving the equations

equation	substitution
//1. T1=[T+*Tt->T0]	T1=[[Number*Number->Number]* Boolean->T0]
//2. T1=[Tf*Tx->T2]	T+=[Number*Number->Number]
//3. T+=[Number*Number->Number]	Tt=Boolean
//4. Tt=Boolean.	Tf=[Number* Boolean -> T0]
//5. Tf=[Tnum1*Tx->T2]	Tnum1=Number
//6. Tnum1=Number	Tx=Boolean
//7. Tf=T+	T2=T0
//8. Tx=Tt	
//9.T2=T0	
//10.Number=Number	
11.Tx=Number	
12.T2=Number	

Both sides of the equation are atomic types but they not equal, output FAIL.

b. stage 1: rename bound variables

((lambda (f x) (f x 1)) + *)

Stage 2: assign type variables to all sub-exp.

expression	var
((lambda (f x) (f x 1)) + *)	T0
(lambda (f x) (f x 1))	T1
+	T+
*	T*
(f x 1)	T2
f	Tf
x	Tx
1	Tnum1

Stage 3: construct type equations

expression	equation
((lambda (f x) (f x 1)) + *)	T1=[T+ * T* ->T0]
(lambda (f x) (f x 1))	T1=[Tf * Tx -> T2]
+	T+=[Number*Number->Number]
*	T*=[Number*Number->Number]
(f x 1)	Tf=[Tx*Tnum1->T2]
1	Tnum1=Number

Stage 4: solving the equations

equation	substitution
//1. T1=[T+ * T* ->T0]	T1=[[Number*Number->Number]* [Number*Number->Number]->T0]
//2. T1=[Tf * Tx -> T2]	T+=[Number*Number->Number]
//3. T+=[Number*Number->Number]	T*=[Number*Number->Number]
//4. T*=[Number*Number->Number]	Tf=[[Number*Number->Number] * Number ->T0]
//5. Tf=[Tx*Tnum1->T2]	Tnum1=Number
//6. Tnum1=Number	Tx=[Number*Number->Number]
//7. Tf=T+	T2=T0
//8. Tx=T*	
//9. T2=T0	
10.Tx=Number	
11.Number=Number	
12.T2=Number	

[Number*Number->Number]=Number

We get a conflicting equation and neither of the further steps is valid.

Part 3.1:

Typing rule for lit:

For every: type environment _Tenv,
expressions _e1:
If _Tenv |- _e1 : Number then _Tenv |- _e1 : Number
_Tenv |- _e1 : string then _Tenv |- `_e1 : string
_Tenv |- _e1 : Boolean then _Tenv |- _e1 : Boolean
_Tenv |- _e1 : Symbol then _Tenv |- `(_e1) : Symbol
_Tenv |- _e1 : Empty then _Tenv |- `(_) : Symbol
_Tenv |- _e1 : Compound then _Tenv |- `(cons _e1) : Pair

Typing rule for set! :

For every: type environment _Tenv,
variable reference _v1 and
expressions _e1 and
type expressions _S1:
If _Tenv |- _e1 : _S1 and
_Tenv |- _v1 : _S1_
Then _Tenv |- (set! _v1 _e1) : void

Part 3.2.1:

Typing rule for define-type:

For every: type environment _Tenv,
expressions _UD, _r1, … , _rn
type expressions UD, t1, … ,tn
If _Tenv |- _UD:UD,
_Tenv |- _r1:t1, … , _Tenv |- _rn:tn
Then _Tenv |- (define-type _UD (r1…) , … , (rn…)):UD

Typing rule for type-case:

For every: type environment _Tenv,
expressions _x,_TC, _UD, _r1, … , _rn
type expressions TC, UD, t1, … ,tn
type expressions [x1, … ,xm]
type expressions T1, … , Tn
If _Tenv |-x:ti
Then _Tenv|- _TC:[x1* … *xm -> Ti]