Assignment 4

*In part2 values is defined as 'special form'

Part 1

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

According to the 4 steps defined in class:

Stage I: Rename bound variables:

((lambda (x y) (if (> x y) #t #f)) 8 3)

Stage II: Assign type variables for every sub expression:

Expression	Variable
((lambda (x y) (if (> x y) #t #f)) 8 3)	T_0
(lambda (x y) (if (> x y) #t #f))	T_1
(if (> x y) #t #f)	T_{if}
(> x y)	T_{cond}
>	$T_{>}$
#t	$T_{\#t}$
#f	$T_{\#f}$
x	T_x
y	T_y
8	T_{num8}
3	T_{num3}

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

Expression	Equation
((lambda (x y) (if (> x y) #t #f)) 8 3)	$T_1 = [T_{num8} * T_{num3} \rightarrow T_0]$
(lambda (x y) (if (> x y) #t #f))	$T_1 = [T_x * T_y \rightarrow T_{if}]$
(if (> x y) #t #f)	$T_{if} = T_{\#t} = T_{\#f}$
(> x y)	$T_{cond} = [T_x * T_y \rightarrow bool]$

The equations for the primitives are:

Expression	Equation
>	$T_> = [number * number \rightarrow bool]$
#t	$T_{\#t} = bool$
#f	$T_{\#f} = bool$
8	$T_{num8} = number$
3	$T_{num3} = number$

Stage IV: Solve the equations:

1:

Equation	Substitution
$T_1 = [T_{num8} * T_{num3} \rightarrow T_0]$	1) $T_1 = [T_{num8} * T_{num3} \rightarrow T_0]$
$T_1 = [T_x * T_y \rightarrow T_{if}]$	
$T_{if} = T_{\#t} = T_{\#f}$	
$T_{cond} = [T_x * T_y \rightarrow bool]$	
$T_> = [number * number \rightarrow bool]$	
$T_{\#t} = bool$	
$T_{\#f} = bool$	
$T_{num8} = number$	
$T_{num3} = number$	

2:

Equation	Substitution
$T_1 = [T_x * T_y \rightarrow T_{if}]$	1) $T_1 = [T_{num8} * T_{num3} \rightarrow T_0]$
$T_{if} = T_{\#t} = T_{\#f}$	
$T_{cond} = [T_x * T_y \rightarrow bool]$	
$T_> = [number * number \rightarrow bool]$	
$T_{\#t} = bool$	
$T_{\#f} = bool$	
$T_{num8} = number$	
$T_{num3} = number$	
$T_x := T_{num8}$	
$T_y := T_{num3}$	

$T_{if} := T_0$	

3:

Equation	Substitution
$T_{if} = T_{\#t} = T_{\#f}$	1) $T_1 = [T_{num8} * T_{num3} \rightarrow T_0]$ 2) $T_{if} = T_{\#t} = T_{\#f}$
$T_{cond} = [T_x * T_y \rightarrow bool]$	
$T_> = [number * number \rightarrow bool]$	
$T_{\#t} = bool$	
$T_{\#f} = bool$	
$T_{num8} = number$	
$T_{num3} = number$	
$T_x := T_{num8}$	
$T_y := T_{num3}$	
$T_{if} := T_0$	

4:

Equation	Substitution
$T_{cond} = [T_x * T_y \rightarrow bool]$	1) $T_1 = [T_{num8} * T_{num3} \rightarrow T_0]$ 2) $T_{if} = T_{\#t} = T_{\#f}$ 3) $T_{cond} = [T_x * T_y \rightarrow bool]$
$T_> = [number * number \rightarrow bool]$	
$T_{\#t} = bool$	
$T_{\#f} = bool$	
$T_{num8} = number$	
$T_{num3} = number$	
$T_x := T_{num8}$	
$T_y := T_{num3}$	
$T_{if} := T_0$	

5:

Equation	Substitution
$T_{>} = [number * number \rightarrow bool]$	1) $T_1 = [T_{num8} * T_{num3} \rightarrow T_0]$ 2) $T_{if} = T_{\#t} = T_{\#f}$ 3) $T_{cond} = [T_x * T_y \rightarrow bool]$ 4) $T_{>} = [number * number \rightarrow bool]$
$T_{\#t} = bool$	
$T_{\#f} = bool$	
$T_{num8} = number$	
$T_{num3} = number$	
$T_x := T_{num8}$	
$T_y := T_{num3}$	
$T_{if} := T_0$	

6: (Fast forward...)

Equation	Substitution
$T_{\#t} = bool$	1) $T_1 = number * number \rightarrow T_0]$ 2) $T_{if} = bool$ 3) $T_{cond} = [T_x * T_y \rightarrow bool]$ 4) $T_{>} = [number * number \rightarrow bool]$ 5) $T_{\#t} = bool$ 6) $T_{\#f} = bool$ 7) $T_{num8} = number$ 8) $T_{num3} = number$
$T_{\#f} = bool$	
$T_{num8} = number$	
$T_{num3} = number$	
$T_x := T_{num8}$	
$T_y := T_{num3}$	
$T_{if} := T_0$	

7:

Equation	Substitution
$T_x := T_{num8}$	1) $T_1 = number * number \rightarrow T_0]$ 2) $T_{if} = bool$ 3) $T_{cond} = [number * number \rightarrow bool]$ 4) $T_> = [number * number \rightarrow bool]$ 5) $T_{\#t} = bool$ 6) $T_{\#f} = bool$ 7) $T_{num8} = number$ 8) $T_{num3} = number$ 9) $T_x := number$ 10) $T_y := number$
$T_y := T_{num3}$	
$T_{if} := T_0$	

8:

Equation	Substitution
$T_{if} := T_0$	1) $T_1 = number * number \rightarrow bool]$ 2) $T_{if} = bool$ 3) $T_{cond} = [number * number \rightarrow bool]$ 4) $T_> = [number * number \rightarrow bool]$ 5) $T_{\#t} = bool$ 6) $T_{\#f} = bool$ 7) $T_{num8} = number$ 8) $T_{num3} = number$ 9) $T_x := number$ 10) $T_y := number$ 11) $T_0 := bool$

2.

a. `{f:[T1->T2], x: T1} |- (f x):T2`
true because the type of a function is it's return type.

b. `{f:[T1->T2],g: [T2->T3]}, x: T2}|- (f g x):T3`
false because the types are incompatible. The type of g is `[T2->T3]` but the function f gets the type T1.

c. `{f:[T2->T1],g: [T1->T2], x: T1}|- (f (g x)): T1`
true. g(x) returns T2 and f goes from T2 to T1, so the type of the expression is T1.

d. `{f:[T2->Number], x: Number}|- (f x x): Number`
false because f gets 1 parameter, but is given 2 parameters..

3.

a. cons: `(x: T1, y: T2)-> Pair(T1, T2)`
b. car: `(Pair(T1, T2) )-> T1`
c. cdr: `(Pair(T1, T2) )-> T2`

4.

`f(x: T1) -> Tuple(T1, T1, T1)`

5.

a. T1, T2 ⇒
{T1 = T2}

b. Number, Number ⇒
{} - (since number always equals 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]}

Part 2.3

```
(define f
  (lambda (x)
    (values x (+ x 1))))
     [(Number) => Tuple(Number, Number)]

(define g
  (lambda (x)
    (values "x" x)))
     [(T) => Tuple(string,T)]
```

Part 4

As we learned in class:

- The type of functions returning Promises is more informative and similar to the simple types of synchronous versions
- We can chain sequences of asynchronous calls in a chain of `.then()` calls.
- We can aggregate error handling in a single handler for a chain of calls, in a way similar to exception handling.

Code from the tasks

Part 3

```
export function* braid(generator1: Generator, generator2: Generator){
    const gen1Op = generator1;
    const gen2Op = generator2;
    for(let n=0;; n++){
        let x = gen1Op.next()
        let y = gen2Op.next()
        if(! (x.value === undefined) )
            yield x.value
        if(! (y.value === undefined) )
            yield y.value
        if ((x.value === undefined)&&(y.value === undefined))
            return x.value
    }
}


export function* biased(generator1: Generator, generator2: Generator){
    const gen1Op = generator1;
    const gen2Op = generator2;
    for(let n=0;; n++){
        let x = gen1Op.next()
        let y = gen2Op.next()
        if(! (x.value === undefined) )
            yield x.value;
        x = gen1Op.next()
        if(! (x.value === undefined) )
            yield x.value;
        if(! (y.value === undefined) )
            yield y.value;
        if ((x.value === undefined) && (y.value === undefined))
            return x.value
    }
}
```

Part 4

a-

```
export function f(x: number): Promise<number>{
  return new Promise<number>( (resolve : any, reject: any) => {
    if( x != 0)
      resolve(1/x);
    else
      reject("x = 0\n");
  })
}

export function g (x: number): number{
    return (x * x);
}

export function h (x: number): Promise<number> {
    return f(g(x))
}
```

b-

```
export const slower = (p1: Promise<any>, p2: Promise<any>):Promise<[number,
string]> =>{
  const pArr = [p1,p2];
  return new Promise<[number, string]>( (resolve,reject) => {
    Promise.race([p1,p2]).then((value)=> {
      Promise.all(([p1,p2])).then((valsArr)=>{
        pArr[getTheOtherOne(value,valsArr)].then((val)=> {
          resolve ([getTheOtherOne(value,valsArr),val])}).catch((v)=>{reject(new
Error("error"))})
      }).catch((v)=>{reject(new Error("error"))})
    }).catch((v)=>{reject(new Error("error"))})
  })
}

const getTheOtherOne = (x:any,arr:any[2])=>{
  if(arr.indexOf(x)===0){
    return 1;
  }
  else
    return 0;
}
```