PPL - Assignment 4

We implemented values as PrimOp, and no support for nested tuples.

Part 1: Theoretical Questions

1.1.

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

Stage 1: ((lambda (x y) (if (> x y) #t #f))8 3)

Stage 2:

expression	**var**
((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
x	TX
Y	TY
>	T>
#t	T#T
#f	T#F
8	Tnum1
3	Tnum2

Stage 3:

expression	**equation**
((lambda (x y) (if (> x y) #t #f))8 3)	T1 = [Tnum1*Tnum2->T0]
(lambda (x y) (if (> x y) #t #f))	T1 = [TX*TY->T2]
(if (> x y) #t #f)	T2 = T#T
	T#T = T#F
(> x y)	T> = [TX*TY-> T3]
>	T> = [Number*Number->Boolean]
#t	T#T = Boolean
#f	T#F = Boolean
8	Tnum1 = Number
3	Tnum2 = Number

Stage 4:

equation	substitution
	T1 = [Number * Number -> Boolean]
	T2 = Boolean
	T> = [Number * Number -> Boolean]
	T#T = Boolean
	T#F = Boolean
	Tnum1 = Number
	Tnum2 = Number
	TX=Number
	TY= Number
	T0= Boolean
	T3= Boolean

Therefore, the Texp of the whole expression is a Boolean, since we received T0=Boolean

1.2.

a. Yes. We can apply f on x, since under the stated assumptions x is T1, and f expect to receive T1 as input. Moreover, (f x) type is indeed T2 since f return T2. Therefore, the statement is true.

b. No. Under the stated assumptions f expect to receive T1 as input. Yet, f doesn't receive T1 in the (f g x) expression. Therefore, the statement is false.

c. Yes. We can apply g on x, since under the stated assumptions x is T1 and g expect to receive T1 as input. Furthermore, we can apply f on (g x) since g return T2 and f expect to receive T2 as input. Moreover, (f (g x)) type is indeed T1 since f return T1. Therefore, the statement is true

d. No. The expression (f x x) apply f on two numbers, while under the stated assumptions f expects to receive T2 as input. Therefore, the statement is false.

1.3.

a. Cons type: [T1 * T2 -> Pair(T1, T2)]

b. Car type: [Pair(T1, T2) -> T1]

c. Cdr type: [Pair(T1, T2) -> T2]

1.4.

The **function type** is: [T1 -> (T1 * T1 * T1)]

1.5.

a. {T1=T2}

b. { }

c. {T1=[T3->Number], T4 = [T3->Number], T2=Number}

d. {T1=[Number->Number]}

Part 2: Theoretical Questions

2.3. The **fully type annotated version** of the function is:

```
(define f: [number -> (number * number)]
        (lambda (x: number): (number * number)
                (values x (+ x 1))))
(define g: [T1 -> (string * T1)]
        (lambda (x: T1): (string * T1)
                (values "x" x)))
```

Part 4: Theoretical Questions

4.1b. Promises are a general programming pattern designed to simplify asynchronous composition, in particular error handling.

Using promises, we can achieve 3 main benefits over the structure that callbacks only would require:

- The type of functions returning Promises is more informative and is similar to the simple types of synchronous versions.
- We can chain sequences of asynchronous calls in a chain of .then() calls, instead of using the nested method which is less intuitive.
- We can aggregate error handling in a single handler for a chain of calls, in a way similar to exception handling, instead of handling errors separately.

Part 3: Code

```
import { range } from "ramda";

let checkGenrator : Generator;
type Gen = Generator | (() => Generator);

const isGenerator = (x: Gen): x is Generator => typeof(checkGenrator) === typeof(
x);
const isGeneratorFunc = (x: Gen): x is () => Generator => "function" === typeof(x
);

export function* braid(generator1: Gen, generator2: Gen): Generator {
    let a: Generator;
    let b: Generator;
    if (isGeneratorFunc(generator1)) a = generator1();
    else a = generator1;
    if (isGeneratorFunc(generator2)) b = generator2();
    else b = generator2;
    let c = a.next();
    let d = b.next();
    while (!c.done && !d.done) {
        yield c.value;
        yield d.value;
        c = a.next();
        d = b.next();
    }
    while (!c.done){
        yield c.value;
        c = a.next();
    }
    while (!d.done){
        yield d.value;
        d = b.next();
    }
}

export function* biased(generator1 : Gen, generator2: Gen): Generator {
    let a: Generator;
    let b: Generator;
    if (isGeneratorFunc(generator1)) a = generator1();
    else a = generator1;
    if (isGeneratorFunc(generator2)) b = generator2();
    else b = generator2;
    let c = a.next();
```

```
    let d = b.next();
    while (!c.done && !d.done) {
        yield c.value;
        c = a.next();
        if (c.done) break;
        yield c.value;
        yield d.value;
        c = a.next();
        d = b.next();
    }
    while (!c.done){
        yield c.value;
        c = a.next();
    }
    while (!d.done){
        yield d.value;
        d = b.next();
    }
}
```

Part 4: Code

```
import { KeyValuePair } from "ramda";

export function f(x: number): Promise<number> {
    return new Promise<number>((resolve,reject) => {
        try {
            if(x===0)
                reject(divisionByZero)
            else
                resolve(1 / x)
        } catch (err) {
            reject(err)
        }
    })
}

export const divisionByZero = new Error("error: division by zero")

export function g(x: number): Promise<number> {
    return new Promise<number>((resolve,reject) => {
        try {
            resolve(x * x)
```

```
        } catch (err) {
            reject(err)
        }
    })
}

export function h(x: number): Promise<number> {
    return new Promise<number>((resolve,reject)=> {
    g(x)
        .then((x) => f(x) )
        .then((x) => resolve(x) )
        .catch((err) => reject(err));
    })
}

export type slowerResult<T> = KeyValuePair<number, T>;

const indexForPromises = <T>(p: Promise<T>, i: number): Promise<slowerResult<T>>
=>
    new Promise<slowerResult<T>>((resolve, reject) =>
        p.then((x) => resolve([i, x]))
            .catch((err) => reject(err)));

export const slower = <T>(p: Promise<T>[]): Promise<slowerResult<T>> => {
    const p1 = indexForPromises(p[0], 0);
    const p2 = indexForPromises(p[1], 1);

    return new Promise<slowerResult<T>>((resolve, reject) =>
        Promise.race([p1, p2])
            .then((fasterResult) => {
                Promise.all([p1, p2])
                    .then((x) => resolve(x.find(element => element[0] != fasterRe
sult[0])))
                    .catch((err) => reject(err))
            })
            .catch((err) => reject(err))
    );
};
```