PPL assignment 4: Theoretical questions

Notes:

1) We implemented values as a PrimOp.

Answers:

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

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

Stage 2: To suit type variable to each expression

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)	T-if
(> x y)	T-test
x	T-x
Y	T-y
>	T>
#t	T-#t
#f	T-#f
8	Tnum8
3	Tnum3

Stage 3: Build types equations

Expression	Equation
((lambda (x y) (if (> x y) #t #f))8 3)	T1 = [Tnum8 * Tnum3 -> T0]
(lambda (x y) (if (> x y) #t #f))	T1 = [T-x * T-y->T-if]
(if (> x y) #t #f)	T-if=T-#t, T-test=Boolean, T-#t=T-#f
(> x y)	T> = [T-x * T-y -> T-test]
>	T> = [number * number -> boolean]
#t	T-#t = boolean
#f	T-#f = boolean
8	Tnum8 = number
3	Tnum3 = number

Stage 4: Solutions for the equations

Equation	Substitution
	T1 = [number * number -> boolean]
	T-if = boolean
	T> = [number * number -> boolean]
	T-#t = boolean
	T-#f = boolean
	Tnum8 = number
	Tnum3 = number
	T-x = number
	T-y = number
	T0 = boolean
	T-test = boolean

We received T0 = boolean so the Texp of the whole expression is a 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 does not 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.
In addition, 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. <u>Cons type:</u>

[T1 * T2 -> Pair(T1, T2)].

b. <u>Car type:</u>

[Pair(T1, T2) -> T1].

c. <u>Cdr type:</u>

[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]}

2.3. Fully type annotated 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)))
```

4.1b. Promises have 3 main benefits over the structure that callbacks do:

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

Code of question 3:

```
import { isBoolean } from "../shared/type-predicates";

export function* braid(gen1: () => Generator, gen2: () => Generator): Generator {
  let arr = [gen1(), gen2()];
  while (arr.length > 0) {
    for (let x = 0; x < arr.length; x++) {
      const { value, done } = arr[x].next();
      if (!done) {yield value;}
      else {arr.splice(x, 1);}}}}

export function* biased(gen1: () => Generator, gen2: () => Generator): Generator {
  let Gen1 = gen1();
  let Gen2 = gen2();
  let isGen1,isGen2 = false;
  while (!(isGen1 && isGen2)) {
    for (let x = 0; x < 2; x++) {
      const { value, done } = Gen1.next();
      isBoolean(done) ? isGen1 = done : isGen1 = true;
      if (!isGen1) {yield value;}}
    const { value, done } = Gen2.next();
    isBoolean(done) ? isGen2 = done : isGen2 = true;
    if (!isGen2) {yield value;}}}
```

Code for question 4:

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

export const divideZero = new Error("Dividing zero");

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

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 WP = <T>(promise: Promise<T>, place: number): Promise<SlowerResult<T>> =>
   new Promise<SlowerResult<T>>((resolve, reject) =>
      promise.then((res) => resolve([place, res])).catch((exeption) => reject(exeption)));

export const slower = <T>(promises : Promise<T>[]): Promise<SlowerResult<T>> => {
   const w1 = WP(promises[0], 0);
   const w2 = WP(promises[1], 1);
   return new Promise<SlowerResult<T>>((resolve, reject) =>Promise.race([w1,
w2]).then((fasterValue) => {Promise.all([w1, w2])
      .then((values) => resolve(values.find(element => element[0] != fasterValue[0]))).catch((e)
=> reject(e))})
      .catch((e) => reject(e)));};
```