Assignment 4

Submission Instructions

Submit your answers to the theoretical questions and your code for programming questions inside the provided files in the correct places. Zip those files together (including the pdf file, and only those files) into a file called id1_id2.zip. The id1_id2.zip file should include the following files:

1. part1.pdf - which will include all your answers for theoretical questions.
2. A folder named part2, which includes all the files from git (interpreters/src/L5 at master · bguppl/interpreters), and the following files should be extended to solve the problem in part2:
   - L5-ast.ts
   - TExp.ts
   - L5-eval.ts
   - L5-typecheck.ts
3. A folder named part3 which includes the file part3.ts. In part3.ts write the solutions for both the functions in 1 and 2.
4. A folder named part4 which include:
   - part4.ts answers for 1.a and 2.
   - Notice: the answer for 1b which is theoretical should be included in part1.pdf

Part 1: Theoretical questions:

1. **Perform typing inference for the expression:**

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

2. **Are these typing statements true? Explain.**

   a. {f:[T1->T2], x: T1} |- (f x)}: T2

   b. {f:[T1->T2],g: [T2->T3]}, x: T2}|- (f g x): T3

   c. {f:[T2->T1],g: [T1->T2], x: T1}|- (f (g x)): T1

   d. {f:[T2->Number],, x: Number}|- (f x x): Number

3. What is the type of the following primitive operators:

   a. cons

   b. car

   c. cdr

4. Write the type of the following function: (Define f (lambda (x) (values x x x)))
   (see question 2 for the definition of values).

5. Write the MGU of the following expressions, or state that there is no such MGU.

   a. T1 , T2
   b. Number , Number
   c. [T1*[T1->T2]->Number] , [[T3->Number]*[T4->Number]->N]
   b. [T1->T1] , [T1->[Number->Number]

Part 2: Type Checking

In the type language for L5 - we define compound type expressions to be:

```
;; <compound-te>  ::= <proc-te> | <tuple-te>
export type CompoundTExp = ProcTExp | TupleTExp;
export const isCompoundTExp = (x: any): x is CompoundTExp =>
    isProcTExp(x) || isTupleTExp(x);
```

But in fact - in all of the code - the only possible compound type expression we process is ProcTExp. That is, tuples are only used as parameters to functions, and we cannot have an L5 expression which has tuple-texp as type.

In principle, tuples can be used as first-class values, and this is the case in many programming languages, such as Python and Scheme. For example, in Racket - the following syntactic constructs rely on tuple values: https://docs.racket-lang.org/reference/values.html

```
(let-values ([(x y) (quotient/remainder 10 3)])
  (list y x))
⇒ '(1 3)

(define f
  (lambda (x)
    (values 1 2 3)))

(let-values (((a b c) (f 0)))
  (+ a b c))
⇒ 6

(let-values (((n s) (values 1 “string”))) n)
⇒ 1
```

The "**values**" special form returns a tuple value.
For example, the expression (values 1 2 3) evaluates into a tuple of three numbers, and (values 1 “string”) to a tuple containing a number and a string.

let-values binds the tuple returned by the right-hand-side of the binding (f 0) to the variables on the left-hand-side (a b c).

The primitive "**quotient/remainder**" performs division of two numbers, and returns the result as a tuple containing the quotient and the remainder of the integer division.

The goal of this assignment is to extend L5 to support values and let-values. To do so, perform the following steps:

2.1. Extend L5-ast.ts to support "**values**" and "**let-values**".

2.2. Extend TExp.ts to support tuples in places which are not only the parameters of an AppExp. (The tuple TExp is already defined in TExp.ts -- but it cannot be used in all the places that the new extension allows.) Modify the concrete syntax and the abstract syntax accordingly, and update the parser parseTExp().

2.3 Write the fully type-annotated version of this function:

```
(define f
   (lambda (x)
      (values x (+ x 1))))


(define g
   (lambda (x)
      (values “x” x)))
```

2.4 Extend L5-eval.ts to support “**values**” and “**let-values**” with direct evaluation of **let-values** (not as a syntactic transformation).

2.5 Extend the L5 type checker (L5-typecheck.ts) to support tuple composite expressions.

Part 3: Generators

1. Write the function **function*** braid(generator1, generator2) that accepts two generators and returns a generator that combines both generators by interleaving their values.

   For example:
   ```
   function* gen1() {
         yield 3;
         yield 6;
         yield 9;
         yield 12;
     }
   function* gen2() {
         yield 8;
         yield 10;
     }

   for (let n of take(4, braid(gen1,gen2))) {
     console.log(n);
   }
   // 3, 8, 6, 10
   ```

2. Write the function **function*** biased(generator1, generator2) that accepts two generators and returns a generator that combines both generators by taking two elements from gen1 and one from the gen2.
   Example :

   ```
   for (let n of take(4, biased(gen1,gen2))) {
     console.log(n);
   }
   // 3, 6, 8, 9
   ```

Part 4. Promises

a. Use the promise interface to write an asynchronous code that performs the following computation. Handle possible errors by printing them to the screen.

```
function f (x : number): number {
    return 1/x
}
function g (x : number): number {
    return x*x
}
function h (x : number): number {
    return f(g(x))
}
```

b. What are the benefits of the promise interface compared to the callback interface.

2. Implement the promise **slower** that accepts two promises (p1 and p2), and succeeds only if both promises succeed. The return value is (0, value) or (1, value) where 0 indicates that the first promise was **slower**, and 1 indicates that the second promise was slower, value is the return value of the promise that was resolved last.

12 lines ...

```
const promise1 = new Promise(function(resolve, reject) {
setTimeout(resolve, 500, 'one');
});

const promise2 = new Promise(function(resolve, reject) {
setTimeout(resolve, 100, 'two');
});

slower([promise1, promise2]).then(function(value) {
  console.log(value);
  });
```

// (0, ‘one’)