Submission Guidelines

This assignment includes programming in TypeScript. Please refer to this link for installation instructions.

Updates:

- **17/5 23:30** - Published.

Responsible Lecturer: Yaron Gonen
Responsible TA: Ben Eyal

- Read this carefully.
- You are provided with the template here: `id1_id2.zip`. Please refer for each question for a specific instructions.
- There is no structure test for this homework.
- Zip all your answers in a file `id1_id2.zip` - replace `id1` and `id2` with your id (e.g.: `123456789_987654321.zip`). The zip file should not create any additional directories when inflated.
- You have a forum you can use to ask questions.
- Solution for Part 1 (Concept Questions): hw4-concept-solution.pdf
- Tests for Parts 2, 3: hw4-tests.zip

Your implementations are tested automatically. Please make sure you strictly abide by the instructions given in the assignment specification and here

Part 1: Concept Questions (30 Points)

Submission instructions: Write all answers in the file `hw4.pdf`

1. Find MGUs for the following pairs of type expressions (if exists):
   a. `[T1*[T1->T2]->N]`, `[[T3->T4]*[T5->Number]->N]`
   b. `[T1*[T1->T2]->N]`, `[Number * [Symbol->T3]->N]`
   c. `T1`, `T2`
   d. `Number`, `Number`
2. Explain why we can typecheck `letrec` expressions without specific problems related to recursion and without the need for a recursive environment like we had in the interpreter.

3. In the type equation implementation - we represent Type Variables (TVar) with a content field (which is a box which contains a Type Expression value or #f when empty). In this representation, we can have a TVar refer in its content to another TVar - repeatedly, leading to a chain of TVars. Design a program which, when we pass it to the type inference algorithm, creates a chain of length 4 of Tvar1 → Tvar2 → Tvar3 → Tvar4. Write a test to demonstrate this configuration.

Part 2: Type Checker (30 points)

Support `define` and `program` expressions

Modify the files under the `part-2-define-program` directory in the given template.

Define

Description

Add support for `define` expressions in the type checker. For example, the following code should be typed void:

```
1. (define (foo : number) 5)
```

and the following code should raise a type error:

```
1. (define (foo : number) (lambda (x y) (+ x y)))
```

Guidelines

Think what is the typing rule for `define` expressions, and complete the function `typeofDefine` in the file `L5-typecheck.ts`.

Program

Description

Add support for checking the type of a whole program.

Guidelines

The main issue is to update the type-environment after a `define` expression, so later on when we encounter a `var-ref` expression such that it's variable was defined using a `define` expression, it's type can be found. To extend the type-environment without mutation (functional style), use the same method for extending the environment we used in L1 language.

Complete the function `typeofProgram` in the file `L5-typecheck.ts`

Type Inference (30 points)

Support `Pair(T1,T2)` Compound Type

Modify the files under the `part-2-pair` directory in the given template.

Add support for the Pair type in the type inference system. Notice that pair type expression is denoted `(Pair T1 T2)` and **not** `Pair(T1, T2)`.

Follow the following steps:

- Add Pair as part of the TExp type language (modify the file `TExp.ts`):
  - Modify the TExp type itself
  - Modify the parser: modify the function `parseCompoundTExp`
- Modify `unparseTExp` to support Pair
- Modify `matchTVarsInTE` to support Pair
- Add primitives `cons`, `car`, `cdr` to the type checker (function `typeofPrim` in `L5-typecheck.ts`, `isPrimitiveOp` in `L5-ast.ts`)
- Modify `checkNoOccurrence` to support pairs
- Add quote special form and its typing rule
- Extend the type language implementation to support comparison of type expressions including Pair.

Example:

```
1. (lambda ([a : number] [b : number]): (Pair number number)
2.   (cons a b))
```

Part 3: Promises and Generators (10 points)

In order to make testing your generators easier, the function `take` takes a generator `g` and a natural number `n` and returns an array of the first `n` elements of `g`. If `g` is exhausted before reaching `n` elements, less than `n` elements are returned.

```
                                10 lines ...
1. function take(g, n) {
2.     const result = [];
3.     for (let i = 0; i < n; i++) {
4.         const { value, done } = g.next();
5.         if (done) {
6.             break;
7.         }
8.         result.push(value);
9.     }
10.    return result;
11. }
```

Question 3.1: `race`

Implement the `race` function.

From Mozilla Developer Network : The `race(promises)` function returns a promise that resolves or rejects as soon as one of the promises in the array resolves or rejects, with the value or reason from that promise.

Example:

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

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

race([promise1, promise2]).then(function(value) {
  console.log(value);
  // Both resolve, but promise2 is faster
});
// expected output: "two"
```

Question 3.2: `flatten(array)`

Write a function that takes an arbitrarily nested array and generates the sequence of values from the array.

Example:
`[...flatten([1, [2, [3]], 4, [[5, 6], 7, [[[8]]]])] => [1, 2, 3, 4, 5, 6, 7, 8]`

Question 3.3: `interleave(g1, g2)`

Given two generators, write a function that generates the interleaved sequence of elements of both generators.

Example: given generators for even and odd numbers,
`take(interleave(evens(), odds()), 8) => [0, 1, 2, 3, 4, 5, 6, 7]`

Question 3.4: `cycle(array)`

Write a function that continuously generates elements of a given array in a cyclic manner.

Example: `take(cycle([1, 2, 3]), 8) => [1, 2, 3, 1, 2, 3, 1, 2]`

Question 3.5: `chain(arrays)`

Write a function that returns all elements from the first array, then all elements from the next array, etc. This function lets us to treat an array of arrays as a single collection.

Example: `[...chain([['A', 'B'], ['C', 'D']])] => ['A', 'B', 'C', 'D']`