Curry and Function Composition

Why do we curry?

Curried functions are particularly useful in the context of function **composition**.

In algebra, given two functions, f and g:

$$\mathbf{f : a \to b,\ g : b \to c}$$

You can compose those functions together to create a new function, h from a directly to c:

$$\mathbf{h : a \to c}$$

$$\mathbf{h(x) = f(g(x))}$$

$$\mathbf{h : f \circ g}$$

Composition

In typescript **h(x)** = **f(g(x))**

Composition

In typescript **h(x)** = **f(g(x))**

```
const g =( n :number)=> n + 1
const f =( n :number)=> n * 10

const h = (n:number) => f(g(n))
```

这种组合方式并没有限制只需要一个参数，或者说，这种组合方式需要自己来处理参数。

Composition

In typescript **h(x)** = **f(g(x))**

```
const g =( n :number)=> n + 1
const f =( n :number)=> n * 10

const h = (n:number) => f(g(n))
```

这种组合方式并没有限制只需要一个参数，或者说，这种组合方式需要自己来处理参数。

如何能够让函数的组合仅仅包含函数?

h : f ∘ g

Composition

In typescript **h(x)** = **f(g(x))**

```
const g =( n :number)=> n + 1
const f =( n :number)=> n * 10

const h = (n:number) => f(g(n))
```

这种组合方式并没有限制只需要一个参数，或者说，这种组合方式需要自己来处理参数。

如何能够让函数的组合仅仅包含函数?

h : f ∘ g

```
const h = compose(f,g)
```

What is point-free style

Point-free style is a style of programming where function definitions do not make reference to the function's arguments.

```
const resultEquals20 =
    filter(
        compose(
            equals(20),
            prop("result")
        )
    )
```

What is a curried function

A curried function is a function that takes multiple arguments one at a time.

What is a curried function

A curried function is a function that takes multiple arguments one at a time.

Given a function with 3 parameters:

normal style

```
const add = (x:number,y:number) => x + y;
```

What is a curried function

A curried function is a function that takes multiple arguments one at a time.

Given a function with 3 parameters:

normal style

```
 const add = (x:number,y:number) => x + y;
```

curried style:

```
const add = (x:number) => (y:number) => x + y
//adds 1 to a number
const inc = add(1)
```

What is a partial application

A partial application is a function which has been applied to some, but not yet all of its arguments.

Partial applications can take as many or as few arguments a time as desired.

Curried functions on the other hand always return a unary function: a function which takes one argument.

Use function curry to curry a function

Is it possible to reuse existing multi-parameters function?

Use function curry to curry a function

Is it possible to reuse existing multi-parameters function?

```
const add = (x:number,y:number) => x + y;
const inc = curry(add)(1)
```

Trace function

How to trace the value between each step during the composition?

```
const traceWithLabel = <T>(label:string, value : T)=>{
    console.log(`label:${value}`);
    return value;

}
const trace  = curry(traceWithLabel)
```

Implementation of function compose

Implementation of function compose

with type information:

```
const compose = <T1,T2,T3>(f:(v:T2)=>T3,g:(v:T1)=>T2) => (v:T1)=>f(g(v))
```

without type information:

```
const compose = (f,g) => (v)=>f(g(v))
```

Composition in other direction : pipe

While compose combines functions from right-to-left, function pipe build a pipe flowing left-to-right, calling each function with the output of the last one.

```
const h = compose(f,g)
const i = pipe(g,f)
```

exercise

Implement a type pipe function that combine two functions.

```
const g =( n :number)=> n + 1
const f =( n :number)=> n * 10

const i = pipe(g,f)
console.log(i(10));//output 110
```