Taller Typescript

1. Clases de criptografía

- Crear una interfaz para clases que implementan algoritmos de criptografía, es decir, que permiten cifrar y descifrar cadenas. Se necesita dos métodos, cifrar y descifrar, que toman una cadena como parámetro y retornan una cadena.

- Implementar una clase que implementa la interfaz previa, con el cifrado César (https://es.wikipedia.org/wiki/Cifrado_C%C3%A9sar). El constructor de esa clase debe tomar un parámetro entero D que representa el desplazamiento.
  Cada carácter del rango a-z se codifica con un entero del rango [0;25].

  Ciframiento: $c_i = m_i+D \bmod 26$

  Desciframiento: $m_i = c_i-D \bmod 26$

  m_i siendo el carácter con índice i del mensaje a cifrar, y c_i el carácter con índice i del mensaje cifrado.
- Implementar una clase que implementa la interfaz previa, con el cifrado One Time Pad (https://es.wikipedia.org/wiki/Libreta_de_un_solo_uso). El constructor de esa clase debe tomar una cadena como parámetro. Esa cadena p debería ser de un tamaño mayor o igual a los mensajes a cifrar.

  Ciframiento: $c_i = m_i-p_i \bmod 26$

  Desciframiento: $m_i = c_i+p_i \bmod 26$

Nota: Estos dos algoritmos trabajan con el alfabeto [a-z], y se debe ignorar todos los caracteres que no pertenecen a ese rango.

- Utilizar el principio del polimorfismo para cifrar y descifrar una cadena con varios algoritmos y varios parámetros: Ejemplo:

```
let s = "the quick brown fox jumps over the lazy dog";
let algos =[new CifradoCesar(0), new CifradoCesar(2),
           new CifradoCesar(-2),new CifradoCesar(26)
           , new OneTimePad("aaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaa")
           , new OneTimePad("jklmnopqrstuvwxyzabcdefghijklmnopqr")
           , new OneTimePad("thequickbrownfoxjumpsoverthelazydog")
           ];
for(var i in algos){
    var a = algos[i];
    console.log(a);
    console.log(a.cifrar(s))
    console.log(a.descifrar(a.cifrar(s)))
    console.log("")
}
```

Salida esperada:

```
CifradoCesar { n: 0 }
thequickbrownfoxjumpsoverthelazydog
thequickbrownfoxjumpsoverthelazydog

CifradoCesar { n: 2 }
vjgswkemdtqyphqzlworuqxgtvjgncbafqi
thequickbrownfoxjumpsoverthelazydog

CifradoCesar { n: -2 }
rfcosgaizpmuldmvhsknqmtcprfcjyxwbme
thequickbrownfoxjumpsoverthelazydog

CifradoCesar { n: 26 }
thequickbrownfoxjumpsoverthelazydog
thequickbrownfoxjumpsoverthelazydog

OneTimePad { mask: 'aaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaa' }
thequickbrownfoxjumpsoverthelazydog
thequickbrownfoxjumpsoverthelazydog

OneTimePad { mask: 'jklmnopqrstuvwxyzabcdefghijklmnopqr' }
kxtehunukzvcsjrzkulnpkqyklyuaomkoyp
thequickbrownfoxjumpsoverthelazydog

OneTimePad { mask: 'thequickbrownfoxjumpsoverthelazydog' }
aaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaa
thequickbrownfoxjumpsoverthelazydog
```

2. Clase de Tabla Hash

Implementar una clase genérica de tabla hash con los métodos siguientes:

- constructor(h:(T) => number, size:number)
    - El parámetro h es una función de hash para el tipo de datos T.
    - El parámetro size determina el tamaño del arreglo que almacena los elementos.
- set(key:T, value:any)
    - Calcula la función h(key)%size, y añade el valor value en la casilla del arreglo que corresponde. Si la clave ya existe, se debe reemplazar el valor. Si hay una colisión de hash, se debe almacenar todos los valores en un arreglo.
- get(key:T):any
    - Devuelve el valor asociado a la clave.
- iterator()
    - Devuelve un iterador que permite enumerar las claves presentes en la tabla hash. El iterador implementa el método next():{done:bool, value:T}.

Ejemplo:

```
var hm = new Hashmap<string>(h, 10);
hm.set("abc", "v")
hm.set("abc", "testabc")
hm.set("abcd", "h")
hm.set("abcd", "abcd")
hm.set("k1", "v1")
hm.set("k2", "v2")
console.log(hm.get("abc"))
console.log(hm.get("abcd"))
console.log(hm.get("unknown"))
console.log("Iterate:")
var iterator = hm.iterate();
var v;
while(!(v = iterator.next()).done){
   console.log(v.value+" => "+ hm.get(v.value))
}
```

Salida:

```
testabc
abcd
null
Iterate:
abc => testabc
abcd => abcd
k1 => v1
k2 => v2
```

3. Clase de BigInteger

Implementar una clase que permite manipular números de un tamaño arbitrario. Esos números son almacenados como cadenas de caracteres.
Los siguientes métodos deben ser implementados:

- constructor(s:string)
    - El parámetro es una cadena de inicialización del número decimal. Ejemplo: s=”123456789”.
- add(b:Biginteger):Biginteger
    - Calcula la suma entre el la actual instancia de Biginteger y el Biginteger pasado en parámetro. Este método devuelve una nueva instancia de Biginteger, sin modificar la actual ni b.
    - Se debe sumar cada dígitos de los números de la derecha a la izquierda, tomando en cuenta el acomulador.
- public mult(b:Biginteger):Biginteger
    - Calcula la multiplicación entre la instancia actual y b. Se devuelve una nueva instancia de Biginteger, sin modificar los parámetros.
- (**Opcional**) public pow(b:number):Biginteger
    - Calcula la potencia de la instancia actual b veces, utilizando el método *mult*.

Ejemplo de utilización de la clase:

```
let a = new Biginteger("900000")
let b = new Biginteger("1100000")
let c = new Biginteger("100")
let d = new Biginteger("39")
let e = new Biginteger("142")
console.log(c+"+"+e+"="+c.add(e))
console.log(a+"+"+b+"="+a.add(b))
console.log(c+"*"+d+"="+c.mult(d))
let test = new Biginteger("00001000")
let t2 = new Biginteger("00")
console.log(test+"")
console.log(t2+"")
console.log(e+"^7="+e.pow(7))
console.log(e+"^10="+e.pow(10))
```

Salida esperada:

```
Biginteger (100)+Biginteger (142)=Biginteger (242)
Biginteger (900000)+Biginteger (1100000)=Biginteger (2000000)
Biginteger (100)*Biginteger (39)=Biginteger (3900)
Biginteger (1000)
Biginteger (0)
Biginteger (142)^7=Biginteger (1164175380274048)
Biginteger (142)^10=Biginteger (3333639396234118349824)
```

4. Intérprete de un subconjunto de LISP

- Este ejercicio tiene el objetivo de implementar un intérprete para un subconjunto del lenguaje de programación LISP. Se limitará solamente a la evaluación de expresiones aritméticas sencillas (+, -, *, /) en notación prefija, por ejemplo: (+ 5 (- 2 (/ 6 (* 2 1)))).
- Implementar una clase *LispInterprete*, con los métodos siguientes:
    - tokenize(s:string)
        - Cree un arreglo de tokens a partir de la entrada, y lo almacena en una variable de la clase.
        - existen 4 tipos de token: Paréntesis derecho ‘(‘, paréntesis izquierdo ‘)’, número (entero o flotante), o un identificador (que no contiene espacios, paréntesis, y que no sea un número).
    - parse()
        - Genera el AST a partir del arreglo de tokens, con la gramática siguiente:
            - FCALL -> ‘(‘ ID ‘,’ PARAMS ’)’
            - PARAMS -> PARAM PARAMS | epsilon
            - PARAM -> NUMBER | FCALL
        - El AST se compone de instancias de clases que heredan de la interfaz INode (interface INode{execute():any;}.
            - NodeFCall, que tiene un identificador (nombre de la función, por ejemplo ‘+’ o ‘*’), y una lista de parámetros de tipo INode.
            - NodeID, que contiene el lexema del identificador
            - NodeNumber, que contiene el lexema de un número
    - execute()
        - Se recorre el AST, ejecutando el método execute():any de cada nodo, empezando por la raiz del arbol.
            - La función execute de la clase NodeFCall devuelve un número, que es el resultado de la aplicación de la funcion a sus parametros
            - La funcion execute de la clase NodeID devuelve una cadena, el lexema del identificador (‘+’ o ‘*’, por ejemplo).
            - La funcion execute de la clase NodeNumber devuelve una representacion numerica del lexema.

Ejemplo:

```
let program = "(+ 5 (- 2 (/ 6 (* 2 1))))"
let li = new LispInterprete();
li.tokenize(program);
li.parse();
console.log(program+"="+li.execute())
```

Salida:

```
[ Token { t: 2, lexema: '(' },
  Token { t: 1, lexema: '+' },
  Token { t: 0, lexema: '5' },
  Token { t: 2, lexema: '(' },
  Token { t: 1, lexema: '-' },
  Token { t: 0, lexema: '2' },
  Token { t: 2, lexema: '(' },
  Token { t: 1, lexema: '/' },
  Token { t: 0, lexema: '6' },
  Token { t: 2, lexema: '(' },
  Token { t: 1, lexema: '*' },
  Token { t: 0, lexema: '2' },
  Token { t: 0, lexema: '1' },
  Token { t: 3, lexema: ')' },
  Token { t: 3, lexema: ')' },
  Token { t: 3, lexema: ')' },
  Token { t: 3, lexema: ')' } ]
(+ 5 (- 2 (/ 6 (* 2 1))))=4
```