QUARTZ LANG

Tienes binarios para GNU/Linux (los que no terminan en .exe) y para windows (los que terminan en .exe). Todos son de 64 bits. Los binarios disponibles son:

- quartz: Intérprete.
- quartz-debug: Muestra el proceso de compilación y ejecución.
- quartz-sgc: Versión con el modo "stress garbage collector" activado. Va más lento pero puedes detectar fallos en el GC.
- quartz-debug-sgc: Mezcla de quartz-debug y quartz-sgc.

También hay una carpeta con programas de prueba. Tenga en cuenta que los programas que incluyen otros ficheros se hace relativo a la posición del binario de quartz (como en PHP, por ejemplo). Si queremos ejecutar el programa 'brainfuck.qz' que esta en 'programs/brainfuck/' debemos ir a la carpeta 'programs/brainfuck/' y ejecutar el intérprete así:

```
cat add.bf | ../../quartz brainfuck.qz
```

Esto es porque el contenido de 'brainfuck.qz' es:

```
/**
* A brainfuck interpreter written in my own programming language (Quartz)
*/


import "stdio";
import "./vm.qz";

var program = stdin();
var vm = new VM();
vm.interpret(program);
println("");
vm.dump();
```

El programa importa el fichero './vm.qz' relativo al path actual. Si el programa no importase nada relativo se puede ejecutar desde donde quieras. Ejemplo:

```
./quartz programs/fibonacci_imperative.qz
```

Documentación de la stdlib.

```
import 'stdio'; // importa la librería de entrada y salida.

// ofrece las siguientes funciones:

fn print(str: String): Void          // Imprime el string sin un salto de línea al final
fn println(str: String): Void        // Imprime el string con un salto de línea al final
fn readstr(Void): String             // Lee una línea del teclado
fn stdin(Void): String               // Lee el buffer completo de stdin (útil para leer de un pipe)
```

```
import 'stdconv'; // importa la librería de conversiones.

// ofrece las siguientes funciones:

fn ntos(n: Number): String               // Convierte un número a string
fn btos(b: Bool): String                 // Convierte un booleano a string
fn ston(str: String): Number             // Parsea un número
fn typeof(any_var: Any): Void            // Imprime por pantalla el tipo de cualquier variable
fn parse_ascii(arr: []Number): String    // Transforma un array con números codificados en ascii y devuelve el String correspondient
```

```
import 'stdtime'; // importa la librería de tiempo.

// ofrece las siguientes funciones:

fn time(): Number               // Segundos que han pasado desde que arrancó el programa
```

Documentación de clases internas:

```
// Cualquier string tiene los siguientes métodos:

fn length(): Number                     // Longitud del string
fn get_char(index: Number): String      // Obtiene un caracter del string. Si el index no es correcto se produce un error
                                        // de ejecución.
fn to_ascii(): []Number                 // Retorna un array con la codificación en ascii de la cadena

Ejemplo:

var str = "hola";

str.length(); // 4
str.get_char(2); // l
std.to_ascii(); // [104, 111, 108, 97] (decimales de la tabla ascii para h o l a)
```

```
Cualquier array tiene los siguientes métodos:

fn length(): Number                             // Número de elementos del array
fn get(index: Number): Any                      // Obtiene un elemento del array. Si el index no es correcto se produce un
                                                // error de ejecución.
fn set(index: Number, element: Any): Void       // Setea un elemento del array. Si el index no es correcto se produce un
                                                // error de ejecución.
fn push(element: Any): Void                     // Pushea un elemento (como en una pila, se queda al final)
fn pop(): Any                                   // Popea un elementos (el del final). Además acorta el array.

Ejemplo:

var arr: []Number = []Number{1, 1, 2, 3, 5, 8}

arr.length(); // 6
arr.get(4); // 5
arr.set(0, 35); // [35, 1, 2, 3, 5, 8]
arr.push(13); // [35, 1, 2, 3, 5, 8, 13]
arr.pop(); // [35, 1, 2, 3, 5, 8]

// Para iterar sobre el array (en este caso imprime cada elemento)

import 'stdio';
import 'stdconv';

for (var i = 0; i < arr.length(); i = i + 1) {
    var current: Number = cast<Number>(out.get(i)); // El valor de retorno de get(Number) es Any. Hay que castear explícitamente a
                                                    // Number.
    println(ntos(current)); // Transforma el Número actual en un String y luego lo imprime
}
```

Tipos

Tipos básicos:

- Number
- String
- Any

- Void

Tipos definidos por el usuario:

- Type alias
- Nombres de clases
- Funciones (ejemplo de una fucnción que no toma parámetros y devuelve void: '(): Void')
- Arrays (ejemplo: []Number. Para crear uno se usa la expresión []Number{...(elemento)...})

Casteos

Principalmente los casteos están pensados para transformar un tipo Any a su tipo real. Su sintaxis es:

```
cast<TIPO>(EXPRESIÓN)
```

Cualquier tipo de datos es casteable implicitamente a Any, pero al revés debe ser explícito. Si el tipo real de la variable con Any no es el tipo al que se castea da un error de ejecución.

Ejemplo:

```
var num: Any = 5; // Casteo implícito. La expresión '5' es de tipo Number, pero se asigna a una variable de tipo Any.

var ok: Number = cast<Number>(num); // OK, La variable 'ok' es de tipo Number y contiene un 5.
var mal: String = cast<String>(num);  // Runtime error. El tipo real de la variable num es Number.
var horrible: String = cast<Number>(num); // Error de compilación. El tipo de la expresión 'cast<Number>(num)' es Number y
                                          // se asigna a una variable de tipo String (tipos incompatibles).
```

Además los casteos pueden servir para saber si una expresión es verdadera o falsa si se castea a Bool.

Ejemplo:

```
var yes = cast<Bool>(1+1); // true
var zero_is_false = cast<Bool>(0); // false
var empty_string_is_true = cast<Bool>(""); // true
var nil_is_false = cast<Bool>(nil); // false
```

Algunas consideraciones:

- En este documento sólo se explican las peculiaridades de Quartz. El resto del lenguaje es muy parecido a cualquier lenguaje estilo C. Para conocer más mira los programas de prueba.
- No existen los operadores unarios ++ y --. Pero sí están disponibles los operadores binarios +, -, *, / y % (módulo) y los operadores unarios - y + (ej: +5 o -5. +5 = 5 y +(-5) = -5)
- Están disponibles los operadores binarios de comparación y booleanos ==, <, >, !=, <=, >=, && y ||. También el operador unario !.
- El operador binario + si es usado para dos strings los concatena.
- Hay clases pero no hay herencia ni interfaces, por lo que tampoco hay polimorfismo.
- Existe el ciclo for y while pero no el do-while.
- Todas las funciones son ciudadanos de primera clase. Incluso los métodos también funcionan como ciudadanos de primera clase.
- No existe una expresión de función. Por lo que no hay funciones anónimas (igual que en python). Esto esta mal:

```
var func = fn() {};
var arrow = () => {}; // Tampoco hay arrow functions.
```

- Si un tipo te parece pesado de escribir (por ejemplo, el tipo de una función) puedes definir un alias:

```
fn do_something(n: Number, s: String, b: Bool): []String {...}

var my_func: (Number, String, Bool): []String = do_something; // Si esto lo tienes que escribir muchas veces puede ser pesado.

typedef MyFunc = (Number, String, Bool): []String; // Crea un alias del tipo.

var my_func: MyFunc = do_something; // El casteo es implícito tanto del tipo real al alias como del alias al tipo real.
```

- Es indiferente el uso de las " y ' para los strings. Incluso se pueden usar en un mismo string ("este string es correcto')
- Es un lenguaje de juguete. No apto para su uso real.
- Es una versión de prueba. Puede contener errores.