

## Manual tecnico

Traductor a riscv assembler, partes esenciales, el compilador, el generador, y las utils que son herramientas para hacer la vida mas fácil.

```
1  import { FrameVisitor } from "../tools/Frame.js";
2  import ReferenceVariable from "../nodes/ReferenceVariable.js"
3  import { registers as r, floatRegisters as f } from "../tools/registers.js";
4  import { Generator } from "../tools/Generator.js";
5  import Visitor from "../abstract/Visitor.js";
6
7
8  export default class Compiler extends Visitor {
9
10   constructor() {
11     super();
12     this.code = new Generator();
13
14     this.continueLabel = null;
15     this.breakLabel = null;
16
17     this.functionMetadata = {};
18     this.insideFunction = false;
19     this.frameDclIndex = 0;
20     this.returnLabel = null;
21     this.table = [];
22   }
23
24   visitExpressionStmt(node) {
25     node.exp.accept(this);
26     const isFloat = this.code.getTopObject().type === 'float';
27     this.code.popObject(isFloat ? f.FA0 : r.R0);
28   }
29
30   visitPrimitive(node) {
31     this.code.comment(`Primitivo: ${node.valor}`);
32     this.code.pushConstant({ type: node.tipo, valor: node.valor });
33     this.code.comment(`Fin Primitivo: ${node.valor}`);
34   }
35
36   visitBinaryOperation(node) {
37     this.code.comment(`Operacion: ${node.op}`);
38
39     if (node.op === '&&') {
40       node.izq.accept(this); // izq
41       this.code.popObject(r.R0); // izq
42
43       const labelFalse = this.code.getLabel();
44       const labelEnd = this.code.getLabel();
45
46       this.code.pushConstant({ type: 'boolean', valor: false });
47       this.code.pushConstant({ type: 'boolean', valor: true });
48     }
49   }
50 }
```

Este código implementa un compilador en JavaScript que genera código de máquina simulando instrucciones en un entorno de bajo nivel. Las características principales incluyen:

**Clase Compiler:** Extiende de Visitor y maneja la generación de código usando la clase Generator. Los atributos permiten gestionar etiquetas, tablas de símbolos, y funciones de control (por ejemplo, break y continue).

**Visitas a distintos nodos de AST:**

**Expresiones y Operaciones:** Procesa operaciones binarias (+, -, \*, &&, ||, etc.), unarias y primitivas, generando el código correspondiente.

**Declaraciones y Asignaciones:** Administra las variables y actualiza la tabla de símbolos.

Control de Flujo: Genera código para estructuras de control como if, while, for, break, y continue.

Funciones y Llamadas a Funciones:

Declara funciones, maneja metadatos de marco de pila, y administra llamadas (incluyendo algunas funciones embebidas como parseInt y parseFloat).

Gestión de Bloques y Alcance: Introduce y cierra bloques de código, manejando el alcance de variables y liberando espacio en la pila cuando se abandona el bloque.

En conjunto, este código proporciona las bases para un compilador que convierte un AST de alto nivel en instrucciones más cercanas al ensamblador, facilitando la interpretación o ejecución posterior.

```
1 import { builtins } from "../builtins.js";
2 import { registers as r } from "../registers.js";
3 import { stringToByteArray, numberToF32 } from "../utils.js";
4 import Instruction from "../Instruction.js";
5
6 export class Generator {
7
8   constructor() {
9     this.instrucciones = []
10    this.objectStack = []
11    this.instruccionesDeFunciones = []
12    this.depth = 0
13    this._usedBuiltins = new Set()
14    this._labelCounter = 0;
15  }
16
17  getLabel() {
18    return `L${this._labelCounter++}`
19  }
20
21  addLabel(label) {
22    label = label || this.getLabel()
23    this.instrucciones.push(new Instruction(`${label}:`))
24    return label
25  }
26
27  add(rd, rs1, rs2) {
28    this.instrucciones.push(new Instruction('add', rd, rs1, rs2))
29  }
30
31  sub(rd, rs1, rs2) {
32    this.instrucciones.push(new Instruction('sub', rd, rs1, rs2))
33  }
34
35  mul(rd, rs1, rs2) {
36    this.instrucciones.push(new Instruction('mul', rd, rs1, rs2))
37  }
38
39  div(rd, rs1, rs2) {
40    this.instrucciones.push(new Instruction('div', rd, rs1, rs2))
41  }
42
43  addi(rd, rs1, inmediato) {
44    this.instrucciones.push(new Instruction('addi', rd, rs1, inmediato))
45  }
46
47  sw(rs1, rs2, inmediato = 0) {
48    this.instrucciones.push(new Instruction('sw', rs1, `${inmediato}({rs2})`))
49  }
50}
```

El archivo Generator.js define una clase Generator que se encarga de generar código en un lenguaje de bajo nivel simulando instrucciones ensambladoras. Esta clase se integra con el compilador y maneja tanto instrucciones aritméticas, de salto, carga y almacenamiento, así como funciones auxiliares para impresión y control de flujo.

#### Descripción General:

**Gestión de Instrucciones:** La clase permite crear instrucciones como add, sub, beq, entre otras, y las organiza en una lista (instrucciones). También soporta instrucciones de punto flotante (como fadd.s, fdiv.s).

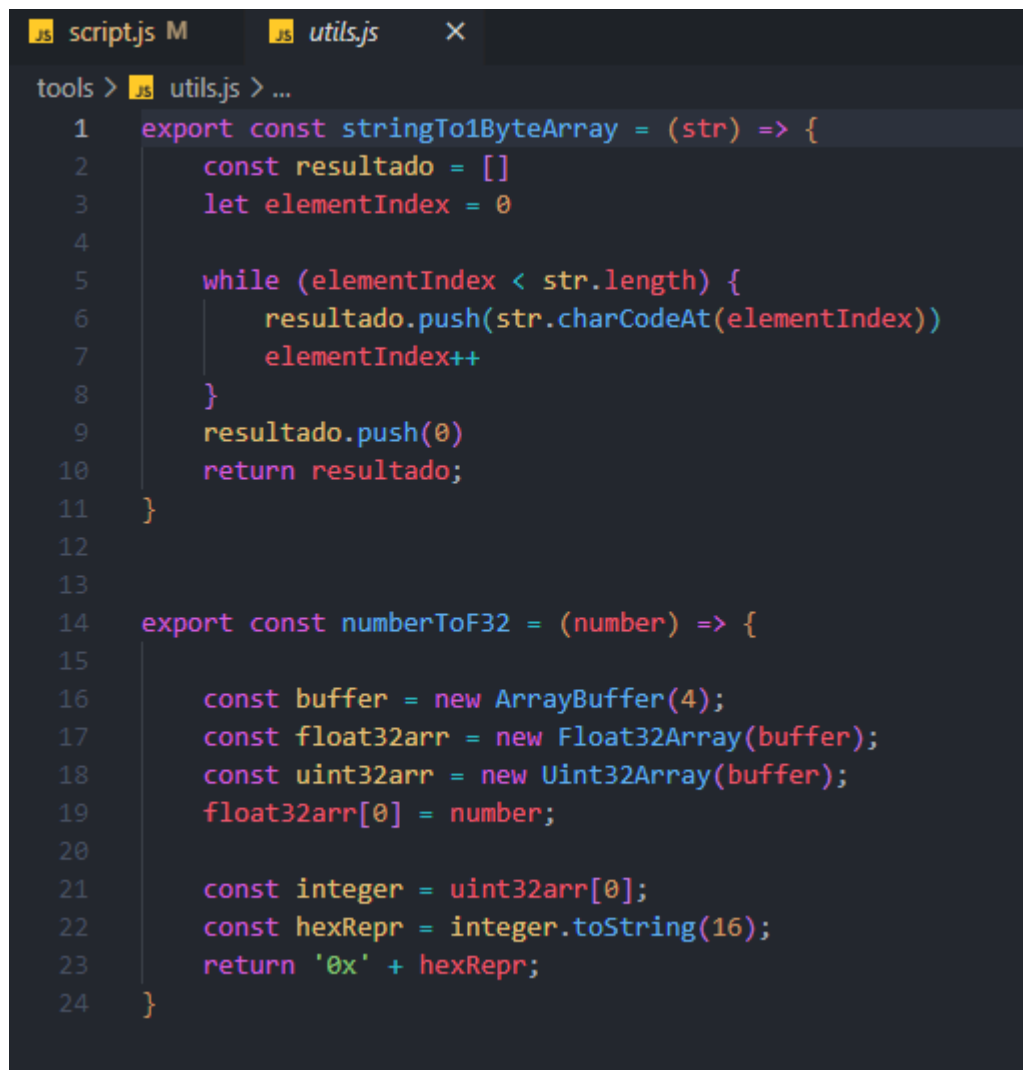
**Pila de Objetos (objectStack):** Almacena objetos durante la ejecución y permite el manejo de variables en distintos ámbitos, controlando su almacenamiento y recuperación.

**Manejo de Etiquetas y Llamadas:** Soporta etiquetas (getLabel) y llamadas a subrutinas, incluyendo llamadas a funciones definidas en el código y algunas funciones embebidas (builtins).

**Utilidades de Impresión:** Proporciona métodos para imprimir distintos tipos de datos (enteros, booleanos, cadenas y caracteres) y formatea literales para facilitar la depuración y visualización.

**Control de Ámbitos:** Implementa control de ámbito, lo que permite definir y limpiar el espacio de variables locales al entrar y salir de bloques de código, ajustando los punteros y el espacio de pila.

Esta clase es fundamental en el compilador, ya que transforma las instrucciones de alto nivel en una representación más cercana al código de máquina, permitiendo su interpretación o ejecución.



```
1  export const stringTo1ByteArray = (str) => {
2      const resultado = []
3      let elementIndex = 0
4
5      while (elementIndex < str.length) {
6          resultado.push(str.charCodeAt(elementIndex))
7          elementIndex++
8      }
9      resultado.push(0)
10     return resultado;
11 }
12
13
14 export const numberToF32 = (number) => {
15
16     const buffer = new ArrayBuffer(4);
17     const float32arr = new Float32Array(buffer);
18     const uint32arr = new Uint32Array(buffer);
19     float32arr[0] = number;
20
21     const integer = uint32arr[0];
22     const hexRepr = integer.toString(16);
23     return '0x' + hexRepr;
24 }
```

El archivo `utils.js` define funciones auxiliares que facilitan la manipulación de datos de tipos específicos dentro del compilador. Estas funciones convierten cadenas y números en formatos compatibles con la representación de datos en bajo nivel, usados en el código generado por el compilador.

#### Descripción de Funciones:

`stringTo1ByteArray`: Convierte una cadena (string) en un arreglo de bytes, donde cada carácter se representa en un byte (ASCII). Al final del arreglo añade un byte de valor 0 como delimitador. Este formato es adecuado para manejar cadenas en entornos de bajo nivel o ensamblador.

Parámetro: str (cadena de texto a convertir).

Retorno: Arreglo de bytes (números enteros) que representan los caracteres de la cadena y terminan en 0.

numberToF32: Convierte un número de JavaScript en su representación IEEE 754 de 32 bits (formato de punto flotante), ideal para compatibilidad en operaciones de punto flotante en ensamblador.

Parámetro: number (número decimal a convertir).

Retorno: Cadena en formato hexadecimal que representa el valor en 32 bits según el estándar IEEE 754.

Estas funciones son útiles en el contexto del compilador para traducir datos complejos (cadenas y números de punto flotante) a formatos básicos que el ensamblador o lenguaje de máquina puede interpretar directamente.