# Manual Técnico

#### 1. Descripción General

Este compilador toma como entrada un programa en un lenguaje de alto nivel y lo traduce a instrucciones de lenguaje ensamblador RISC-V, que luego es leído por el simulador RARS. Su objetivo es realizar la traducción a través de varias fases: análisis léxico, análisis sintáctico, análisis semántico y generación de código. Para lograr esto, el compilador utiliza clases de visitantes para recorrer el árbol de sintaxis abstracta (AST) y construir instrucciones RISC-V.

## 2. Estructura del Compilador

El compilador se organiza en varios módulos:

- **Gramática PeggyJS**: define las reglas sintácticas del lenguaje de entrada.
- Clase CompilerVisitor: implementa el recorrido del AST y la generación de instrucciones RISC-V.
- Clases Primitivo, Print, ReferenciaVariable: representan nodos específicos del AST.
- Clase Generador: contiene métodos para emitir instrucciones en RISC-V.
- Constantes RISC-V (registers, floatRegisters): definen registros utilizados para operaciones de enteros y flotantes.

#### 3. Fases del Compilador

#### 3.1. Análisis Léxico

El **análisis léxico** descompone el código fuente en tokens. Cada token representa una unidad léxica, como palabras clave, operadores o identificadores. Esta fase es manejada principalmente por el generador de tokens en la gramática de entrada.

#### 3.2. Análisis Sintáctico

El **análisis sintáctico** verifica que la estructura de los tokens cumpla con las reglas de la gramática, y construye el AST, representando la estructura jerárquica del programa.

Ejemplo de una regla en PeggyJS:

```
ExpresionTernaria = left:Expresion "?" middle:Expresion ":"
right:Expresion ";"
3.3. Análisis Semántico
```

En el **análisis semántico**, se verifican los tipos y las reglas semánticas. Esta fase asegura que las operaciones se realicen entre tipos compatibles y detecta errores de uso de variables o funciones.

#### 4. Generación de Código

La clase CompilerVisitor convierte el AST en código ensamblador RISC-V. A continuación, se describen algunas de las funciones más importantes:

#### 4.1. Visitantes de Expresiones

Las funciones visitExpresionStmt y visitOperacionBinaria manejan expresiones y operadores. Cada función utiliza registros específicos, como r.T0 y r.T1, para almacenar resultados temporales.

Ejemplo de manejo de operación & &:

```
visitOperacionBinaria(node) {
   if (node.op === '&&') {
        // Cargar y evaluar la expresión izquierda
        node.izq.accept(this);
        this.code.popObject(r.T0);

        const labelFalse = this.code.getLabel();
        const labelEnd = this.code.getLabel();

        // Evaluar la expresión derecha solo si la izquierda es verdadera
        this.code.beq(r.T0, r.ZERO, labelFalse);
        node.der.accept(this);

        // Guardar el resultado final de la operación
        this.code.li(r.T0, 1);
        this.code.push(r.T0);
   }
}
```

#### 4.2. Primitivos y Funciones

La clase visitPrimitivo agrega constantes al código generado, y visitFuncDol maneja la declaración de funciones y la administración de frames en la pila de ejecución.

#### Ejemplo de visitFuncDcl:

```
visitFuncDcl(node) {
    // Calcular y reservar espacio en el frame para parámetros y
variables locales
    const totalSize = baseSize + paramSize + localSize + returnSize;
    this.functionMetada[node.id] = {
        frameSize: totalSize,
        returnType: node.tipo,
    };

    // Manejo de instrucciones
    const instruccionesDeDeclaracionDeFuncion = [];
    node.params.forEach((param, index) => {
        this.code.pushObject({
```

#### 4.3. Llamadas a Funciones Integradas

Las funciones integradas (parseInt, toString, toLowerCase, etc.) están implementadas para realizar operaciones comunes. La función visitLlamada ejecuta la llamada y utiliza el registro de retorno r.A0.

### Ejemplo de llamada a parseInt:

```
visitLlamada(node) {
   const nombreFuncion = node.callee.id;
   if (nombreFuncion === "parseInt") {
      node.args[0].accept(this);
      switch (this.code.getTopObject().type) {
        case "float":
            this.code.callBuiltin("parseFloatToInt");
            break;
      case "string":
            this.code.callBuiltin("parseStringToInt");
            break;
    }
}
```

### 5. Administración de Memoria y Registros

La administración de memoria se realiza en el **stack frame**, donde cada función reserva espacio para parámetros y variables locales. Los registros se asignan de la siguiente manera:

• Registros Temporales (r. T0, r. T1): usados para cálculos temporales.

- Registros de Flotantes (f. FTO, f. FT1): usados para operaciones de punto flotante.
- Puntero de Pila (r.sp) y Puntero de Marco (r.sp): controlan la administración de frames de función.

#### 6. Built-ins y Librerías

El compilador soporta varias funciones integradas para operaciones de tipo, manipulación de strings y conversiones entre tipos (como parseInt y toString). Estas funciones utilizan llamadas a subrutinas predefinidas, como compareStrings para comparar cadenas o parseStringToFloat para convertir cadenas en flotantes.

## 7. Ejecución y Ejemplo de Código Generado

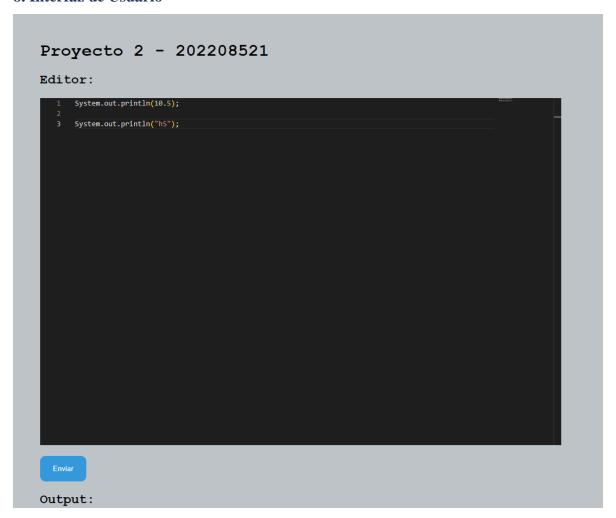
Un ejemplo de código RISC-V generado para una operación de suma sería:

```
# Operación de suma
li t1, 5  # Cargar el primer operando
li t0, 3  # Cargar el segundo operando
add t0, t1, t0 # Sumar los operandos
```

#### Para una función simple:

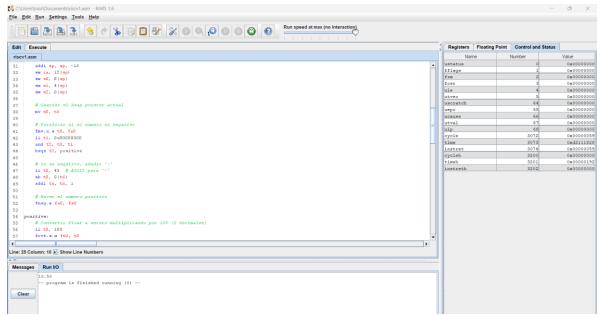
```
# Declaración de función ejemplo
example:
   addi sp, sp, -4 # Reservar espacio en el stack
   sw ra, 0(sp) # Guardar el retorno
   li a0, 42 # Cargar el valor de retorno
   lw ra, 0(sp) # Recuperar el retorno
   addi sp, sp, 4 # Liberar el stack
   jr ra # Retornar
```

### 8. Interfaz de Usuario



Finalmente, aquí es donde usted puede interactuar, aquí usted pone su código, le da clic a enviar, y le genera un output del código en RISC V, que usted luego debe de meter en el simulador RARS

### 9. Simulador



Aquí usted ingresa su código, lo compila y finalmente lo ejecuta.