

# Manual del Programador: Microcompilador Web

## Introducción

Este documento técnico describe la implementación y funcionamiento interno del Microcompilador Web, una herramienta educativa diseñada para introducir a los usuarios en los conceptos básicos de compilación y ejecución de código. Este manual está dirigido a desarrolladores que deseen entender, mantener o extender la funcionalidad del Microcompilador.

## Arquitectura General

El Microcompilador Web sigue una arquitectura de front-end pura, ejecutándose completamente en el navegador del cliente sin necesidad de un servidor backend. Está compuesto por tres archivos principales:

1. **index.html**: Define la estructura de la interfaz de usuario
2. **styles.css**: Define los estilos visuales de la aplicación
3. **main.js**: Contiene toda la lógica de compilación y ejecución

## Componentes Principales

### 1. Editor de Código

Utilizamos la biblioteca CodeMirror para implementar un editor de código con resaltado de sintaxis. El editor está configurado con un modo personalizado para resaltar las palabras reservadas del lenguaje.

javascript

```
CodeMirror.defineMode("customMode", function() {
  return {
    token: function(stream) {
      if (stream.eatWhile(/\w+/)) {
        const cur = stream.current();
        if (reservedWords.includes(cur)) {
          return "keyword";
        }
      }
      stream.next();
      return null;
    }
  };
});


const editor = CodeMirror.fromTextArea(document.getElementById("editor"), {
  lineNumbers: true,
  mode: "customMode",
  theme: "default",
  extraKeys: { "Ctrl-Space": "autocomplete" }
});
```

## 2. Analizador Léxico

El analizador léxico identifica tokens en el código fuente. Las palabras reservadas, operadores y símbolos se definen en constantes al inicio del archivo:

javascript

```
const reservedWords = ['write', 'capture', 'if', 'then', 'end-if', 'and', 'or', 'not', 'end', '
const operadoresRelacionales = ['<', '>', '<=', '>=', '=', '<>'];
const operadoresLogicos = ['and', 'or', 'not'];
```



La función `verTokens()` implementa la funcionalidad básica de conteo de tokens:

javascript

```
function verTokens() {  
  const code = editor.getValue();  
  const tokens = code.match(/\w+|\S/g) || [];  
  const counts = {};  
  tokens.forEach(t => counts[t] = (counts[t] || 0) + 1);  
  let output = 'TOKENS:\n';  
  for (const [token, count] of Object.entries(counts)) {  
    output += `${token} : ${count}\n`;  
  }  
  output += `\nTOTAL DE TOKENS: ${tokens.length}`;  
  document.getElementById('output').textContent = output;  
}
```

### 3. Analizador Sintáctico

El analizador sintáctico verifica que el código cumpla con la gramática del lenguaje. Esto se implementa principalmente en la función `compilar()`, que recorre cada línea del código y verifica su estructura según las reglas definidas.

Las reglas principales del lenguaje incluyen:

- Cada instrucción debe terminar con `::`
- Las estructuras de control (`if`, `while`) tienen sus propias reglas de sintaxis
- Las expresiones y condiciones deben ser válidas

Funciones auxiliares como `verificarParentesis()`, `verificarOperadores()`, `verificarCondicion()` y `verificarExpresion()` realizan comprobaciones específicas para garantizar que el código sea sintácticamente correcto.

### 4. Intérprete

El intérprete ejecuta el código paso a paso después de una compilación exitosa. Está implementado mediante las siguientes funciones:

- `ejecutar()`: Inicia la ejecución paso a paso
- `ejecutarSiguientePaso()`: Ejecuta una línea y actualiza el estado de la ejecución
- `interpretarLinea()`: Interpreta una línea específica de código
- `evaluarExpresion()`: Evalúa expresiones matemáticas y de cadena
- `evaluarCondicion()`: Evalúa condiciones lógicas con operadores relacionales y lógicos

Las variables y el flujo del programa se controlan a través de:

```
javascript
```

```
let programaActual = [];  
let variablesGlobales = {};  
let lineaActual = 0;  
let puntosDeControl = [];  
let enEjecucion = false;
```

## 5. Sistema de Manejo de Archivos

El Microcompilador incluye funcionalidades para crear, abrir, eliminar y descargar archivos:

```
javascript
```

```
function nuevoArchivo() { ... }  
function abrirArchivo() { ... }  
function eliminarArchivo() { ... }  
function descargarArchivo() { ... }
```

## Gramática del Lenguaje

El lenguaje implementado sigue una gramática simple:

### 1. Instrucciones de salida:

```
write(expresión) ::
```

### 2. Instrucciones de entrada:

```
capture(identificador) ::
```

### 3. Asignaciones:

```
identificador = expresión ::
```

### 4. Instrucciones condicionales:

```
if (condición) then  
  instrucciones  
end-if
```

### 5. Instrucciones repetitivas:

```
while (condición)
    instrucciones
end-while
```

Las condiciones pueden incluir operadores relacionales (`<`, `>`, `<=`, `>=`, `=`, `<>`) y operadores lógicos (`and`, `or`, `not`).

## Verificación de Errores

El compilador verifica varios tipos de errores:

1. **Errores léxicos:** Palabras reservadas mal escritas
2. **Errores sintácticos:** Estructura incorrecta de instrucciones
3. **Errores de paréntesis:** Paréntesis no balanceados
4. **Errores en expresiones:** Expresiones sin sentido o mal formadas
5. **Errores en estructuras de control:** `if` o `while` sin sus correspondientes cierres

## Ejecución Paso a Paso

La ejecución se realiza interpretando cada línea del código y actualizando las variables globales según corresponda. Para estructuras de control, se utilizan "puntos de control" para gestionar saltos en la ejecución:

javascript

```
function ejecutarSiguientePaso() {
  if (!enEjecucion || lineaActual >= programaActual.length) {
    finalizarEjecucion();
    return;
  }

  const linea = programaActual[lineaActual];
  let resultado = '';

  try {
    resultado = interpretarLinea(linea, lineaActual);
    // ... actualizar la visualización ...

    // Avanzamos a la siguiente línea
    lineaActual++;

    // Procesamos saltos si es necesario
    if (puntosDeControl.length > 0) {
      const ultimoPunto = puntosDeControl.pop();
      if (ultimoPunto.tipo === 'salto') {
        lineaActual = ultimoPunto.lineaDestino;
      }
    }

    // Programamos la ejecución del siguiente paso
    setTimeout(ejecutarSiguientePaso, 1000);
  } catch (error) {
    document.getElementById('output').textContent = `Error en ejecución: ${error.message}`;
    enEjecucion = false;
  }
}
```

## Documentación

El sistema incluye una función para acceder a la documentación:

javascript

```
function verDocumentacion() {  
    // Crear un modal para mostrar opciones  
    // ...  
  
    // Eventos para cada tipo de documentación  
    document.getElementById('manualProgramador').addEventListener('click', function() {  
        window.open('manuales/manual_programador.pdf', '_blank');  
    });  
  
    document.getElementById('manualUsuario').addEventListener('click', function() {  
        window.open('manuales/manual_usuario.pdf', '_blank');  
    });  
}
```

## Extensiones Posibles

El Microcompilador puede extenderse en varias direcciones:

1. **Añadir más estructuras de control:** `for`, `switch`, etc.
2. **Implementar funciones:** Permitir la definición y llamada a funciones
3. **Tipos de datos adicionales:** Arreglos, estructuras, etc.
4. **Optimización:** Mejorar el rendimiento del intérprete
5. **Exportación a otros lenguajes:** Traducir a JavaScript, Python, etc.

## Limitaciones Conocidas

1. No se admiten funciones o procedimientos
2. No existe un sistema de tipos estricto
3. No hay manejo de arreglos o estructuras de datos complejas
4. La evaluación de expresiones se basa en `eval()` de JavaScript, lo que puede provocar comportamientos inesperados en ciertos casos
5. No hay persistencia de datos entre sesiones más allá de guardar/cargar archivos

## Consideraciones de Seguridad

El uso de `eval()` para la evaluación de expresiones presenta riesgos de seguridad si se ejecuta código malicioso. En un entorno educativo controlado, este riesgo es mínimo, pero debe tenerse en cuenta para posibles extensiones o despliegues más amplios.

## Conclusión

Este Microcompilador Web ofrece una plataforma educativa para entender los conceptos básicos de compilación e interpretación. Su diseño modular permite extensiones y mejoras según las necesidades específicas.