REPORTE TÉCNICO

COMPILADOR DEL LENGUAJE APICRAFT

Autores:

Anahi Hull
Betsy Arceo

1 de diciembre de 2025

Índice

1. Introducción

El presente documento describe el desarrollo completo de un compilador funcional para un lenguaje diseñado específicamente para modelar servicios web y rutas HTTP de manera declarativa. El proyecto integra las fases de análisis léxico, análisis sintáctico y análisis semántico, empleando una arquitectura modular en C# que permite identificar errores en tiempo de compilación y validar programas escritos en el lenguaje ApiCraft.

Este reporte presenta el diseño del lenguaje, la estructura del compilador, los mecanismos de detección y manejo de errores, así como conclusiones derivadas del desarrollo y prueba del sistema.

2. Diseño del Lenguaje

2.1. Nombre del Lenguaje

ApiCraft — Un lenguaje declarativo y minimalista para describir servicios web REST, sus rutas, métodos HTTP y la lógica básica dentro de cada endpoint.

2.2. Propósito del Lenguaje

ApiCraft nace como un DSL (Domain-Specific Language) orientado a simplificar la definición de APIs. Su objetivo es:

- Expresar servicios y rutas HTTP de forma legible.
- Reducir la complejidad de plantillas repetitivas en backend.
- Permitir validaciones semánticas comunes: variables declaradas antes de uso, tipos correctos, expresiones válidas.
- Servir como puente educativo para aprender conceptos clave de compiladores aplicados a casos modernos.

2.3. Casos de Uso

- Definición rápida de prototipos REST.
- Generación futura de código backend (expansión posible).
- Validación temprana de lógica simple (condiciones, variables y retornos).
- Uso académico para mostrar cómo un compilador real puede aplicarse a un dominio actual.

2.4. Ejemplo de Programa Válido

```
service MiAPI {
    route "/usuarios" {
        GET {
            let x = 10 + 20;
            let mensaje = "Hola Mundo";
            if (x > 5) {
                return mensaje;
            }
        }

        POST {
            let nombre = "Anahi";
            return nombre;
        }
    }

    route "/auth" {
        POST {
            let usuario = "Pringle";
            if (usuario == "Pringle") {
                return "Login correcto";
            }
            else {
                return error;
            }
        }
    }
}
```

Listing 1: Ejemplo de programa en ApiCraft

2.5. Tokens del Lenguaje

2.5.1. Palabras clave

`service`, `route`, `let`, `if`, `else`, `elif`, `return`, `true`, `false`, `null`, `GET`, `POST`, `PUT`, `PATCH`, `DELETE`, `HEAD`, `OPTIONS`.

2.5.2. Operadores

- **Aritméticos:** `+`, `-`, `*`, `/`, `%`
- **Comparación:** `==`, `!=`, `>`, `<`, `>=`, `<=`
- **Lógicos:** `&&`, `||`, `!`
- **Asignación:** `=`

2.5.3. Delimitadores

`{`, `}`, `(`, `)`, `[`, `]`, `;`, `:`, `,`

2.5.4. Literales

- Números: `123`, `45.6`
- Strings: `"texto"`
- Booleanos: `true`, `false`

2.5.5. Especiales

- Parámetros de ruta: `:id`
- Identificadores: `nombreDeVariable`, `MiAPI`

2.6. Gramática Sintáctica (LL(1))

$$
\begin{aligned}
\text{PROGRAMA} &\rightarrow \text{SERVICIO EOF} \\
\text{SERVICIO} &\rightarrow \texttt{service ID \{ RUTAS \}} \\
\text{RUTAS} &\rightarrow \text{RUTA RUTAS} \mid \epsilon \\
\text{RUTA} &\rightarrow \texttt{route (STRING | PATHPARAM) \{ METODOS \}} \\
\text{METODOS} &\rightarrow \text{METODO METODOS} \mid \epsilon \\
\text{METODO} &\rightarrow \texttt{(GET | POST | PUT | PATCH | DELETE) \{ SENTENCIAS \}} \\
\text{SENTENCIAS} &\rightarrow \text{SENTENCIA SENTENCIAS} \mid \epsilon \\
\text{SENTENCIA} &\rightarrow \text{DECLARACION} \mid \text{IF} \mid \texttt{RETURN ;} \\
\text{DECLARACION} &\rightarrow \texttt{let ID = EXPRESION ;} \\
\text{IF} &\rightarrow \texttt{if (EXPRESION) \{ SENTENCIAS \} ELSEOPT} \\
\text{ELSEOPT} &\rightarrow \texttt{else \{ SENTENCIAS \}} \mid \epsilon \\
\text{RETURN} &\rightarrow \texttt{return EXPRESION} \mid \texttt{return error} \\
\text{EXPRESION} &\rightarrow \text{TERMINO EXPRESION'} \\
\text{EXPRESION'} &\rightarrow \text{OPERADOR\_BINARIO TERMINO EXPRESION'} \mid \epsilon \\
\text{TERMINO} &\rightarrow \texttt{ID | NUMBER | STRING | true | false | (EXPRESION)}
\end{aligned}
$$

2.7. Reglas Semánticas Clave

1. Variables deben declararse antes de usarse.
2. Los tipos de las expresiones deben ser compatibles (p.ej., comparaciones requieren operandos del mismo tipo).
3. Las condiciones del `if` deben ser booleanas.
4. Las rutas dentro de un mismo servicio no pueden duplicarse.
5. Las operaciones aritméticas deben involucrar tipos numéricos.

3. Análisis Léxico

3.1. Implementación

El analizador léxico está implementado en la clase `Lexer`, que ejecuta un Autómata Finito Determinista (AFD) construido mediante un diccionario de transiciones.

- Reconoce palabras clave usando un diccionario de keywords.
- Reconoce operadores, números, strings, delimitadores y comentarios.
- Genera objetos `Token` con: tipo, lexema, línea y columna.

3.2. Manejador de Errores Léxicos

Cuando el lexer encuentra un carácter desconocido:

- Genera un token `INVALID`.
- Registra el lexema problemático.
- La fase sintáctica puede decidir detenerse o continuar.

Esto cumple con la rúbrica de detección básica de errores léxicos.

3.3. Tabla de Tokens

El programa `Program.cs` imprime todos los tokens generados, lo cual facilita su verificación manual.

4. Análisis Sintáctico

4.1. Enfoque

Se implementó un Parser Descendente Recursivo basado en la gramática LL(1) previamente definida.

Cada no terminal tiene un método:

- `ParseService()`
- `ParseRoute()`
- `ParseMethodBody()`
- `ParseStatement()`
- `ParseExpression()`

El parser consume los tokens producidos por el lexer y valida si la estructura del programa coincide con la gramática.

4.2. Detección de Errores Sintácticos

El parser utiliza la función `Expect(TokenType)`:
Si el token actual no coincide, se lanza una excepción explicando:

- tipo esperado,
- tipo encontrado,
- línea y lexema correspondiente.

Esto asegura una detección clara y localizada de errores.
Ejemplo:

```
Error sintáctico: Se esperaba '{' pero se encontró 'GET' en línea 12.
```

4.3. Recuperación Sintáctica (Básica)

La estrategia elegida es detección temprana: en caso de error se detiene el análisis sintáctico. Esto simplifica la implementación y es adecuado para un compilador educativo.

5. Análisis Semántico

El analizador semántico recorre la estructura reconocida por el parser (en este caso, el flujo lineal del programa ya tokenizado) y valida las reglas del lenguaje. Todo está implementado en la clase `SemanticAnalyzer`.

5.1. Tabla de Símbolos

Para cada método HTTP se abre un nuevo scope:

- Uso de un diccionario `symbolTable`.
- Cada `let ID = EXPRESION` registra una variable y su tipo.
- Las variables solo existen dentro del método.

5.2. Validaciones Semánticas Implementadas

5.2.1. Declaración antes de uso

Si un identificador aparece sin haber sido declarado:

```
Error semántico: Variable 'x' no declarada en este scope.
```

5.2.2. Tipos compatibles en operaciones

El analizador valida: comparaciones, operaciones aritméticas, booleanos en operadores lógicos, concatenación válida de strings.
Errores típicos:

```
Error: No se pueden comparar tipos 'string' y 'int'.
Error: Operador '+' no permitido entre 'string' y 'int'.
```

5.2.3. Condiciones del if

La expresión dentro del paréntesis debe ser de tipo booleano:

```
Error: La condición del if debe ser booleana, pero es de tipo 'string'.
```

5.2.4. Rutas duplicadas

Si un servicio contiene:

```
route "/usuarios" { ... }
route "/usuarios" { ... }
```

Se produce:

```
Error: Ruta duplicada '/usuarios' en el servicio.
```

5.3. Reglas de Inferencia de Tipos

El analizador implementa una función de combinación:

- `int + float` → `float`
- `int + int` → `int`
- `string + string` → `string`
- comparadores → `bool`
- lógicos → `bool`
- incompatibles → error semántico

6. Arquitectura Completa del Compilador

Entrada → Lexer → Lista de Tokens → Parser → Estructura Válida → Semantic Analyzer → Reporte

6.1. Componentes principales

Módulo	**Responsabilidad**
Lexer.cs	Tokeniza el archivo de entrada
Parser.cs	Valida estructura del programa mediante gramática
SemanticAnalyzer.cs	Comprueba reglas semánticas y tipos
Token.cs	Estructura unificada para tokens
Tools.cs	Utilidades para coloreo y mensajes
Program.cs	Orquesta la ejecución de las fases

Cuadro 1: Componentes del compilador

7. Manejo de Errores

7.1. Errores Léxicos

- Carácter desconocido → token `INVALID`

7.2. Errores Sintácticos

- Tokens inesperados
- Mensajes detallados con línea/columna
- Aborta la ejecución sintáctica

7.3. Errores Semánticos

- Variables no declaradas
- Tipos incompatibles
- Condiciones inválidas
- Rutas duplicadas

Todos los errores se muestran con: tipo de error, descripción, línea exacta y lexema involucrado.

8. Demostración Funcional

Para la demo en video:

8.1. Compilación correcta

Mostrar el programa `MiAPI`. Ver tokens, validación sintáctica y validación semántica correcta.

8.2. Error léxico

Incluir un símbolo prohibido:

```
let x = 10 @ 2;
```

8.3. Error sintáctico

Quitar una llave:

```
GET {
   let x = 10
```

8.4. Error semántico

Usar variable sin declarar:

```
return x;
```

9. Conclusiones

El desarrollo de ApiCraft permitió comprender y aplicar los principios principales del diseño de compiladores, integrando:

- Autómatas para el lexer
- Gramáticas LL(1) y parsing predictivo
- Tablas de símbolos, reglas de tipo y validaciones semánticas
- Manejo claro de errores en cada fase del proceso de compilación

Además, se demostró que los compiladores no son solo herramientas académicas, sino que pueden aplicarse a resolver problemas modernos como la definición estructurada de APIs y servicios web.

10. Áreas de Mejora y Trabajo Futuro

El compilador puede expandirse agregando:

- Generación de código (Node.js, Python, C#, etc.).
- AST formal en vez del flujo basado en tokens.
- Soporte para funciones definidas por el usuario.
- Tipos compuestos (listas, objetos).
- Optimización semántica.
- Módulos y archivos externos.
- Manejo avanzado de errores con recuperación.