Compilador del Lenguaje ApiCraft

Un DSL para Servicios Web REST

Anahi Hull Betsy Arceo

1 de diciembre de 2025

Contenido

Introducción

Objetivo del Proyecto

Desarrollar un compilador completo para **ApiCraft**, un lenguaje diseñado para modelar servicios web REST de manera declarativa.

Fases Implementadas:

- Análisis Léxico
- Análisis Sintáctico
- Análisis Semántico

Tecnología:

- Lenguaje: C#
- Arquitectura modular
- Detección de errores

ApiCraft: ¿Qué es?

Definición

ApiCraft es un DSL (Domain-Specific Language) para definir APIs REST de forma simple y legible.

Propósito:

- Expresar servicios HTTP claramente
- Reducir código repetitivo
- Validar lógica tempranamente
- Herramienta educativa

Casos de Uso:

- Prototipos rápidos
- Generación de código
- Validación de APIs
- Aprendizaje de compiladores

Ejemplo de Código

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

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

Tokens del Lenguaje

Palabras Clave:

- `service`, `route`
- `let`, `if`, `else`
- `return`, `true`, `false`
- `GET`, `POST`, `PUT`, `DELETE`

Operadores:

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

Literales:

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

Delimitadores:

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

Gramática LL(1)

Reglas Principales

$$
\begin{aligned}
\text{PROGRAMA} &\rightarrow \text{SERVICIO EOF} \\
\text{SERVICIO} &\rightarrow \texttt{service ID \{ RUTAS \}} \\
\text{RUTA} &\rightarrow \texttt{route STRING \{ METODOS \}} \\
\text{METODO} &\rightarrow \texttt{HTTP\_VERB \{ SENTENCIAS \}} \\
\text{DECLARACION} &\rightarrow \texttt{let ID = EXPRESION ;}
\end{aligned}
$$

Característica Importante

Gramática **LL(1)** permite parsing descendente recursivo eficiente.

Implementación

Clase `Lexer` que ejecuta un **AFD** (Autómata Finito Determinista)

Funcionalidades:

- Reconoce palabras clave
- Identifica operadores
- Procesa números y strings
- Maneja comentarios
- Genera tokens con:
  - Tipo
  - Lexema
  - Línea y columna

Manejo de Errores:

- Detecta caracteres inválidos
- Genera token `INVALID`
- Registra posición exacta
- Permite continuar o abortar

Análisis Sintáctico

Enfoque: Parser Descendente Recursivo

Cada no-terminal de la gramática tiene un método dedicado

Métodos Principales:

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

Validación:

- Verifica estructura
- Función `Expect()`
- Mensajes detallados
- Indica línea exacta

Ejemplo de Error

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

Análisis Semántico

Clase SemanticAnalyzer

Valida reglas del lenguaje y coherencia de tipos

Tabla de Símbolos:

- Scope por método HTTP
- Registro de variables y tipos
- Variables locales al método

Validaciones Implementadas:

1. Variables declaradas antes de uso
2. Tipos compatibles en operaciones
3. Condiciones booleanas en `if`
4. Rutas únicas por servicio
5. Operaciones aritméticas válidas

Reglas Semánticas

Inferencia de Tipos

- `int + int` → `int`
- `int + float` → `float`
- `string + string` → `string`
- Comparadores → `bool`
- Lógicos → `bool`

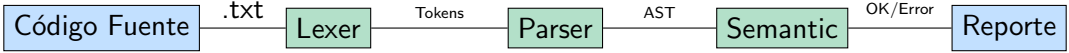
Errores Detectados

- Variable no declarada
- Tipos incompatibles
- Condición no booleana
- Rutas duplicadas

Ejemplo

```
Error: No se pueden comparar tipos 'string' y 'int'.
```

Arquitectura Completa



Módulos Principales

Lexer.cs	Tokenización
Parser.cs	Validación sintáctica
SemanticAnalyzer.cs	Validación semántica
Token.cs	Estructura de tokens
Program.cs	Orquestación

Sistema de Manejo de Errores

Errores Léxicos

- Carácter inválido
- Token `INVALID`
- Posición exacta

Errores Sintácticos

- Token inesperado
- Estructura incorrecta
- Detención temprana

Errores Semánticos

- Variables no declaradas
- Tipos incompatibles
- Lógica inválida

Información de Errores

Todos los errores incluyen: **tipo**, **descripción**, **línea** y **lexema**.

Caso 1: Compilación Exitosa

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

Resultado

Análisis léxico: OK

Análisis sintáctico: OK

Análisis semántico: OK

Caso 2: Error Léxico

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

Error Detectado

```
Error léxico: Carácter inválido '@' en línea 1
```

Caso 3: Error Sintáctico

```
GET {
    let x = 10
    // Falta punto y coma
```

Error Detectado

```
Error sintáctico: Se esperaba ';' pero se encontró '}' en línea 3
```

Caso 4: Error Semántico

```
GET {
    if (x > 5) {
        return "error";
    }
}
```

Error Detectado

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

Conclusiones

Logros Principales

- Compilador funcional completo para ApiCraft
- Integración exitosa de las tres fases de compilación
- Sistema robusto de detección de errores
- Aplicación práctica de teoría de compiladores

Aprendizajes

- Diseño e implementación de autómatas finitos
- Construcción de gramáticas LL(1)
- Manejo de tablas de símbolos y tipos
- Los compiladores resuelven problemas reales y modernos

Trabajo Futuro

Mejoras Técnicas:

- Generación de código (Node.js, Python, C#)
- AST formal completo
- Funciones definidas por usuario
- Tipos compuestos (listas, objetos)

Optimizaciones:

- Análisis semántico avanzado
- Módulos y archivos externos
- Recuperación de errores
- Optimización de código

Visión

Convertir ApiCraft en una herramienta práctica para el desarrollo ágil de APIs REST.

Gracias por su atención

¿Preguntas?

Anahi Hull y Betsy Arceo