Proyecto II – Implementación y Evaluación

LIS-3042 Teoría de la Computación

Adriana de Jesús Jerónimo Alarcón (178002)

3 de diciembre de 2025

1 Resumen Ejecutivo de Proyecto I

El lenguaje propuesto es un mini-lenguaje para comandos de editor de texto que soporta las instrucciones `find`, `replace` y `delete`. El objetivo es validar que un comando tenga estructura sintáctica correcta antes de ejecutarse.

El alfabeto definido fue:

$$\Sigma = \{a \dots z,\, 0 \dots 9,\, :\}.$$

El lenguaje:

$$L = \{w \in \Sigma^* \mid w \text{ es un comando válido de tipo find, replace o delete}\}.$$

Ejemplos válidos: `find:hello`, `replace:old:new`, `delete:123`. Ejemplos inválidos: `find`, `replace:old`, `delete:abc`.

2 Gramática Formal del Lenguaje

2.1 Gramática en BNF

```
S        ::= FIND | REPLACE | DELETE
FIND     ::= "find" ":" ARG
REPLACE  ::= "replace" ":" ARG ":" ARG
DELETE   ::= "delete" ":" DIGITS
ARG      ::= ALNUM | ALNUM ARG
DIGITS   ::= DIGIT | DIGIT DIGITS
ALNUM    ::= a|b|...|z|0|1|...|9
```

2.2 Explicación del diseño

La gramática impone exactamente las mismas restricciones que el AFD del Proyecto I, permitiendo un argumento alfanumérico después de `find`, dos argumentos después de `replace`, y solo dígitos para `delete`.

3 Descripción de la Implementación

La implementación del lenguaje propuesto se dividió en tres componentes principales: el analizador léxico, el analizador sintáctico y la arquitectura general del sistema. La decisión de mantener estos módulos separados permitió un diseño más claro, extensible y alineado con los principios clásicos de construcción de compiladores. En particular, se buscó que la implementación reflejara lo más fielmente posible los modelos formales desarrollados en el Proyecto I.

3.1 Analizador Léxico (Lexer)

El analizador léxico es responsable de transformar una cadena de entrada en una secuencia de tokens interpretables por el analizador sintáctico. Se implementó como una clase en Python que recorre la cadena carácter por carácter, identificando patrones correspondientes a palabras reservadas, caracteres de separación y componentes alfanuméricos.

El lexer reconoce los siguientes tipos de tokens: `FIND`, `REPLACE`, `DELETE`, `COLON`, `ALNUM`, `DIGIT` y `EOF`. Para las palabras reservadas se emplea un reconocimiento por prefijos, lo cual permite distinguir correctamente entre comandos completos y secuencias que solo coinciden parcialmente. Una ventaja de este enfoque es su transparencia: el lexer no requiere expresiones regulares externas ni bibliotecas adicionales, lo cual facilita su validación manual.

3.2 Analizador Sintáctico (Parser)

El analizador sintáctico se implementó siguiendo un enfoque descendente recursivo. Esta técnica fue seleccionada debido a que la gramática del lenguaje es simple, no ambigua y perfectamente compatible con este tipo de reconocimiento. El parser consume la secuencia de tokens proporcionada por el lexer y verifica si la estructura corresponde a una de las producciones válidas del lenguaje.

Cada comando fue implementado como una función de parsing especializada:

- `parse_find()` valida la estructura `find:ARG`.
- `parse_replace()` valida la estructura `replace:ARG:ARG`.
- `parse_delete()` valida la estructura `delete:DIGITS`.

Estas funciones emplean la operación `consume()`, que permite verificar de manera precisa el tipo de token esperado en cada paso. Si se encuentra un token distinto al esperado, se genera una excepción `SyntaxError` con un mensaje explicativo. Esto facilita la retroalimentación hacia el usuario y permite una depuración más sencilla del sistema.

Una característica importante del parser es la validación estricta del fin de entrada. Después de procesar un comando válido, se exige que el siguiente token sea `EOF`. Esto previene que secuencias como `find:hello123extra` sean clasificadas incorrectamente como válidas.

3.3 Arquitectura General del Sistema

La estructura del sistema se organizó en módulos independientes con responsabilidades claras:

- **lexer.py**: encargado del análisis léxico y generación de tokens.
- **parser.py**: realiza el análisis sintáctico e implementa la gramática definida.
- **errors.py**: contiene las excepciones especializadas para errores léxicos y sintácticos.
- **main.py**: actúa como interfaz principal del sistema y coordina lexer y parser.
- **tests.py**: incluye un conjunto de pruebas unitarias creadas para validar el comportamiento del sistema.

Esta separación modular sigue la arquitectura clásica utilizada en compiladores e intérpretes reales. Además, permite extender o modificar partes del sistema sin afectar a las demás. Por ejemplo, sería posible reemplazar el lexer actual por uno basado en expresiones regulares o integrar un sistema de logging sin alterar las funciones principales del parser.

3.4 Decisiones de diseño

Algunas decisiones importantes tomadas durante la implementación incluyen:

- **Uso de un lexer manual**: Se privilegió la claridad y simplicidad sobre herramientas automatizadas.
- **Parser descendente recursivo**: Ideal para la gramática simple y estructurada del lenguaje.
- **Validación estricta de tokens**: Se exige correspondencia exacta con los comandos del lenguaje.
- **Excepciones claras**: Permiten detectar errores tempranamente y mejorar la experiencia de usuario.

Estas decisiones favorecen un sistema transparente, fácil de mantener y alineado con los principios teóricos estudiados en el curso.

4 Evidencia de Funcionamiento

4.1 Cadenas Válidas

```
find:hello   → VALID
replace:old:new → VALID
delete:123 → VALID
```

4.2 Cadenas Inválidas

`find` → ERROR
`replace:old` → ERROR
`delete:abc` → ERROR

4.3 Casos Borde

`""` → ERROR
`find:a` → VALID
`replace:a:b:c` → ERROR

5 Evaluación y Discusión

La implementación realizada en esta fase permitió poner en práctica el diseño formal elaborado en el Proyecto I y evaluar su funcionamiento dentro de un sistema real de análisis

léxico y sintáctico. En esta sección se examina la correspondencia entre el modelo formal y la implementación, se comparan las aproximaciones mediante autómatas y gramáticas, se evalúa el comportamiento del sistema, se exponen las principales limitaciones encontradas y se proponen posibles mejoras futuras.

5.1 Correspondencia con el modelo formal

Durante el desarrollo se verificó que el comportamiento del sistema coincide rigurosamente con el autómata finito determinista (AFD) especificado previamente. Cada comando válido del lenguaje—`find`, `replace` y `delete`—posee una estructura determinista y lineal que permitió mapear de manera directa las transiciones del AFD hacia las funciones del analizador sintáctico.

El analizador léxico garantiza que únicamente se consideren caracteres pertenecientes al alfabeto $\Sigma = \{a \dots z,\ 0 \dots 9,\ :\}$, mientras que el analizador sintáctico comprueba que la secuencia de tokens se ajuste a las reglas formales establecidas. Esto demuestra que los conceptos teóricos de lenguajes regulares y autómatas de estado finito pueden traducirse eficazmente a una implementación funcional sin pérdida de generalidad.

5.2 Comparación entre reconocimiento por autómatas y por gramáticas

La implementación también permitió reflexionar sobre las diferencias prácticas entre el reconocimiento basado en autómatas y el basado en gramáticas. El AFD ofrece una descripción operacional explícita del proceso de validación: cada carácter leído desencadena una transición específica hacia un estado siguiente. Esto resulta útil cuando se desea visualizar el flujo del procesamiento paso a paso.

Por otro lado, la gramática formal proporciona una representación declarativa más concisa y estructurada del lenguaje. En particular, el comando `replace`, que requiere dos argumentos separados por dos puntos, se expresa más elegantemente mediante producciones gramaticales que mediante múltiples estados en un AFD.

5.3 Evaluación del rendimiento y corrección

El sistema mostró un comportamiento estable y consistente durante las pruebas. Todas las cadenas válidas fueron aceptadas y todas las inválidas generaron errores significativos y apropiados. No se detectaron casos de falsos positivos. La corrección del sistema se comprobó mediante pruebas unitarias y mediante la evaluación manual con cadenas especialmente diseñadas para cubrir casos borde.

Tanto el analizador léxico como el sintáctico operan con complejidad temporal. Esto asegura un rendimiento óptimo incluso para cadenas de longitud considerable. La claridad de los mensajes de error contribuye además a mejorar la interpretabilidad del sistema desde el punto de vista del usuario.

5.4 Limitaciones encontradas

Aunque la implementación cumple con los lineamientos del proyecto, se identificaron varias limitaciones derivadas tanto del diseño del lenguaje como de la implementación:

- No se permiten espacios ni caracteres especiales en los argumentos.
- Los comandos deben coincidir estrictamente con las palabras reservadas del lenguaje (`find`, `replace`, `delete`).
- El sistema no soporta cadenas entre comillas ni argumentos vacíos deliberados.
- Los nombres de argumentos deben ser estrictamente alfanuméricos, lo que limita su flexibilidad.
- El analizador léxico utiliza reconocimiento manual de prefijos, lo cual complica la escalabilidad ante la inclusión de nuevos comandos.

Estas restricciones son deliberadas para mantener el lenguaje dentro del ámbito de los lenguajes regulares y facilitar su análisis mediante autómatas y gramáticas de tipo 3.

6 Conclusiones

El desarrollo de este proyecto permitió integrar los conceptos teóricos de la asignatura con una implementación práctica y funcional. A lo largo del proceso se observó cómo los elementos fundamentales de la teoría de lenguajes formales pueden trasladarse de manera directa a un sistema real capaz de validar comandos estructurados.

La construcción del analizador léxico y sintáctico confirmó que el lenguaje propuesto pertenece al ámbito de los lenguajes regulares, ya que su estructura es completamente determinista y no requiere memoria adicional para su procesamiento. Además, implementar el AFD y la gramática permitió comprender de manera más profunda la equivalencia entre estos modelos y reforzó su relación con los paradigmas clásicos de diseño de compiladores.

La implementación resultante es estable, modular y suficientemente robusta para reconocer tanto cadenas válidas como errores comunes de uso. La arquitectura por módulos facilita el desarrollo y dejó las bases listas para posibles extensiones futuras del lenguaje. Asimismo, las pruebas realizadas demostraron la confiabilidad del sistema y su comportamiento consistente frente a diferentes tipos de entradas, incluyendo casos borde y errores sintácticos.