**Informe del Trabajo Práctico N°1: Desarrollo de un Analizador Léxico**

**Integrantes:** Agustin Arispe, Ladislao Sabarnik, Luca Pianzola

**Materia:** Diseño de Compiladores

**Fecha:** 7 de Octubre de 2025

**1. Introducción**

El presente informe detalla el proceso de diseño, implementación y depuración del Analizador Léxico correspondiente al primer trabajo práctico. El objetivo principal fue construir un escáner capaz de procesar un código fuente y convertirlo en una secuencia de tokens, basándose en un Autómata Finito Determinista (AFD).

Nuestra implementación se realizó en Java, utilizando un enfoque de motor guiado por matrices para representar el autómata. El desarrollo abarcó el reconocimiento de los tokens estándar del lenguaje, así como los temas particulares asignados a nuestro grupo: constantes enteras long (32 bits), constantes de punto flotante dfloat (64 bits), cadenas de caracteres de una línea y comentarios multilínea. El proceso fue marcadamente iterativo, consistiendo en ciclos de diseño, codificación, pruebas y corrección de errores que resultaron fundamentales para el aprendizaje.

**2. Arquitectura y Diseño del Analizador**

Desde el inicio, optamos por una arquitectura modular para garantizar la claridad, mantenibilidad y escalabilidad del proyecto.

**2.1. Estructura de Paquetes**

El proyecto se organizó en tres paquetes principales para separar responsabilidades:

* **CompiladoresMain**: Contiene las clases centrales del proyecto, incluyendo el AnalizadorLexico, las clases para representar un Token y sus AtributosTokens, y la clase de constantes TiposToken.
* **conjuntoSimbolos**: Alberga un conjunto de clases cuyo único propósito es el mapeo de caracteres de entrada a las columnas de las matrices del autómata.
* **accionesSemanticas**: Contiene una clase individual para cada Acción Semántica (AS1, AS2, etc.), aislando la lógica específica de cada transición del autómata.

**2.2. El Autómata Finito: Matrices y Mapeo**

El núcleo de nuestro analizador es el AFD, implementado a través de dos componentes principales:

* **Matrices (matrizTransicion y matrizAcciones)**: La matrizTransicion actúa como el "mapa" del autómata, dictando el estado siguiente (proximo\_estado) a partir del estado actual y el símbolo de entrada. Por su parte, la matrizAcciones determina qué procedimiento o efecto secundario se debe ejecutar durante esa transición. Esta separación nos permitió aislar la lógica de control del flujo de las operaciones sobre los tokens.
* **Mapeo de Símbolos (iniciarColumnas)**: Para evitar un bloque if-else o switch masivo y poco mantenible, adoptamos un enfoque orientado a objetos. Creamos una lista de objetos ConjuntoSimbolos, donde cada objeto representa una columna de la matriz y es responsable de determinar si un carácter de entrada le pertenece. Esto nos proporcionó una solución flexible y ordenada para clasificar los símbolos.

**2.3. El Motor del Analizador (nextToken)**

El método nextToken() es el corazón funcional del proyecto. Nuestra implementación inicial se basó en bucles anidados que resultaron complejos para manejar correctamente el fin de línea y de archivo. Por ello, refactorizamos el método a una estructura más robusta y canónica: un único bucle while(true).

Este bucle centraliza la lógica: en cada iteración, se encarga de la lectura de caracteres, el mapeo a una columna, la consulta a las matrices y la ejecución de la acción semántica. La decisión final se toma en un bloque if-else if-else que inspecciona el proximo\_estado para manejar limpiamente los tres resultados posibles: transicionar a un nuevo estado, finalizar y retornar un token (ESTADO\_FINAL), o iniciar la recuperación de un error (ESTADO\_ERROR).

**3. Implementación de Acciones Semánticas (AS)**

Cada acción semántica se implementó como una clase separada, permitiéndonos enfocar su lógica de manera aislada.

* **AS simples (AS1, AS3)**: Fueron las más directas, encargadas de inicializar y concatenar caracteres al lexema del token en construcción.
* **AS de Finalización (AS4, AS7, AS5)**: Estas acciones resultaron más complejas, ya que debían incluir la lógica de backtracking (retroceder el puntero de lectura). La AS4, en particular, requirió implementar el chequeo de longitud de identificadores (20 caracteres), la generación de WARNINGS y la interacción con la Tabla de Símbolos para agregar nuevos identificadores o actualizar el contador de los existentes.
* **AS de Constantes Numéricas (AS8, AS9)**: Estas fueron las más desafiantes. Para AS8 (long), fue necesario parsear la cadena, remover el sufijo 'L' y utilizar la clase BigInteger para realizar una validación de rango robusta contra los límites de un long de 32 bits. Para AS9 (dfloat), la lógica implicó separar la base del exponente, validar la presencia obligatoria del signo en el exponente y usar try-catch para manejar posibles NumberFormatException, además de realizar la validación de rango correspondiente.

**4. Proceso de Pruebas y Depuración**

El testeo fue una parte fundamental y reveladora del proceso.

* **Entorno de Pruebas**: Desarrollamos una clase ProbadorSimple.java que nos permitió automatizar la ejecución de una batería de casos de prueba definidos como Strings. Esta clase crea archivos temporales para cada caso, lo que nos permitió probar nuestro analizador (que esperaba rutas de archivo) de forma rápida y repetible, acelerando significativamente la depuración.
* **Depuración del Comportamiento del Autómata**: Los errores más complejos no fueron de compilación, sino de lógica. Los resultados iniciales mostraban tokens fragmentados o mal clasificados. El proceso de depuración nos llevó a identificar dos causas raíz principales:
  1. **El Problema de la "Pizarra Sucia"**: Detectamos que el estado del objeto Token (su lexema) persistía entre llamadas a nextToken. La solución fue mover la instanciación (new Token()) al inicio del método nextToken, asegurando un estado limpio para cada nuevo token.
  2. **Desincronización de Columnas**: El error más persistente fue el ArrayIndexOutOfBoundsException y los errores de "carácter no válido". Mediante el uso de System.out.println para depurar el estado del autómata, descubrimos que nuestra lista mapeoColumnas no coincidía exactamente con la estructura de las matrices. Corregir este desfasaje para lograr una correspondencia 1 a 1 fue el paso final y clave para estabilizar el analizador.
* **Ajuste Fino de la Lógica**: Finalmente, tuvimos que refinar la matrizTransicion para corregir comportamientos específicos, como el reconocimiento de 123L como un único token en lugar de dos. Esto implicó agregar nuevos estados intermedios para asegurar que el autómata completara el lexema antes de transicionar a un estado final.

**5. Conclusión**

El desarrollo de este trabajo práctico fue un ejercicio integral que abarcó desde el diseño teórico de un autómata hasta la implementación y depuración de un software funcional. El proceso iterativo de corregir errores, desde problemas de configuración de paquetes hasta fallos sutiles en la lógica del autómata, fue la instancia de aprendizaje más significativa. El resultado es un analizador léxico robusto y modular que cumple con los requisitos de la consigna y nos proporcionó una comprensión práctica y profunda de la primera fase del proceso de compilación.