*Grupo 5 21/03/2025*

**Explicación de Estructuras**

El proyecto que se desarrollo consiste en un intérprete básico para un subconjunto del lenguaje LISP, con ayuda del lenguaje de programación Java. Esta compuesto de varias clases, cada una con diferentes funciones, que en conjunto permiten realizar la tokenización, parsing y evaluación de expresiones. "El lenguaje Lisp, diseñado por John McCarthy en 1958, es uno de los lenguajes funcionales más antiguos y sigue siendo relevante en la programación moderna debido a su simplicidad y flexibilidad.” (Soler, 2024).

***Clase Main (Controlador Principal)***

Es la clase que contiene el método *main()* y actúa como punto de entrada de la aplicación. Su objetivo es crea en un contexto de ejecución, tokenizar las expresiones LISP, parsearlas y evaluarlas. Este es un patrón clásico de separación entre entrada, procesamiento y salida en la programación estructurada. “(…) crear objetos mutar sus propiedades y mostrarlas en el método main() de la clase Main.” (Bonaparte, 2012).

***Clase Lexer***

“(…) dividir este código en componentes que contengan las palabras claves del código u operadores; estas palabras claves se van a llamar token. Este proceso realizado para obtener los tokens es lo que se conoce como Analizador Léxico o Lexer en inglés.” (Lamas, 2024). Para este proyecto implementa:

* Extracción de números, símbolos y paréntesis.
* Validación de paréntesis balanceados con el método *BalanceParentesis()* para evitar errores en la fase de parsing.

***Clase Token***

Define la estructura de cada token con un enum Tipo (NUMBER, SYMBOL, LPAREN, RPAREN) para clasificar su tipo, y almacena su valor original. Incluye la lógica para determinar si un String es numérico, usando *Double.parseDouble()*. Esta clasificación es importante en compiladores e intérpretes, donde cada token se asocia a un tipo léxico. “La información de los tokens generados se procesará por diferentes estructuras de datos (…).” (Lamas, 2024)

***Clase Parser***

Implementa el parser recursivo para construir un árbol de expresiones a partir de la lista de tokens. Este árbol es una representación jerárquica de la expresión original, siguiendo el estilo clásico de los lenguajes funcionales.

* Distingue entre átomos y listas de expresiones *(ListExpression)*
* Lanza errores cuando encuentra expresiones mal formadas (Ejemplo: Listas vacías)

“(…) función diseñada para interactuar eficazmente con las estructuras y enumeraciones definidas por el Analizador Léxico, distinguiendo las diferentes propiedades y atributos.”) (Lamas, 2024)

***Clase abstracta Expression***

Esta es la clase abstracta que obliga a todas las expresiones a implementar el método *evaluar(Context)*. Esto muestra que permite que ambas expresiones compartan la interfaz de evaluación. “(…) se genera un árbol de sintaxis abstracta donde se mostrarán todos los datos generados en la fase del analizador sintáctico.” (Lamas, 2024)

***Clase Atom***

Representa átomos individuales (números o símbolos). Esto pasa cuando se evalúa:

* Si es un número, devuelve el valor
* Si es un símbolo, busca el valor en el contexto actual
* Si es “T”, retorna 1, que es true en LISP
* Si es “NIL”, retorna 0, que es false

***Clase ListExpression***

Modela las listas de expresiones LISP, depende del operador recibido reacciona:

* Soporta operadores básicos, como +, -, \*, /
* Define funciones con defun y variables con setq
* Soporta recursión a través de la evaluación de funciones definidas por el usuario.

“(…) hacen posible generar unidades básicas en las expresiones, según cierta prioridad, o asociatividad.” (Jeltsch, s.f.)

***Clase Evaluator***

Encapsula la lógica de evaluación de primitivas LISP como:

* Listp, atom, equals, cond, <, >
* Llama a funciones definidas con defun

***Clase Context***

Esta clase almacena variables definidas y funciones definidas. Es un entorno de ejecución similar al concepto de **ambiente léxico** en lenguajes funcionales. Esta clase asegura la gestión del **estado** entre las expresiones evaluadas.

El proyecto demuestra la implementación de un intérprete LISP, destacando la importancia de estructuras como tokens, análisis léxico y sintáctico, y la ejecución de expresiones. Las clases Parser, Token, Lexer y las representaciones de expresiones ilustran como organizar y evaluar el código de manera estructurada. Se refleja como conceptos abstractos de compilación se aplican para resolver problemas concretos, mostrando la importancia de herramientas como Lisp en el desarrollo de intérpretes y sistemas de programación

# Referencias

Bonaparte, U. J. (2012). *Proyectos UML Diagramas de clases y aplicaciones JAVA en NetBeans 6.9.1*. Obtenido de chrome-extension://efaidnbmnnnibpcajpcglclefindmkaj/https://d1wqtxts1xzle7.cloudfront.net/40272178/uml\_JAVA-libre.pdf?1448226852=&response-content-disposition=inline%3B+filename%3DProyectos\_UML.pdf&Expires=1742528879&Signature=EY~YBXJ87qYY0VqS3kADLYlNqQoG

Jeltsch, E. (s.f.). *Prácticas docentes en Teoría de Autómatas y Lenguajes Formales (TALF) con la herramienta ANTLR/ANTLRWorks*. Obtenido de chrome-extension://efaidnbmnnnibpcajpcglclefindmkaj/https://infonorchile2012.uta.cl/infonor2012\_23a63e.pdf

Lamas, M. E. (Abril de 2024). *Herramienta de métricas de calidad de diseño software en aplicaciones para iPhone: análisis sintáctico y AST*. Obtenido de chrome-extension://efaidnbmnnnibpcajpcglclefindmkaj/https://oa.upm.es/82566/1/TFG\_MARIA\_ESTHER\_COLMENAR\_LAMAS.pdf

Soler, F. O. (2024). *Intérpretes y Diseño de Lenguajes de Programación*. Obtenido de chrome-extension://efaidnbmnnnibpcajpcglclefindmkaj/https://reflection.uniovi.es/ortin/publications/interpretes.pdf