Desarrollo de un Parser en Python con ANTLR4 para el Análisis Eficiente de Datos: Bases, Gramática y Componentes Clave

Carlos Rojas, Sebastian Florido, Luis Rodriguez.
Escuela de ciencias exactas e ingeniería, Universidad Sergio Arboleda, Bogotá, Colombia
correos: luis.rodriguez05@usa.edu.co, steven.florido01@usa.edu.co, carlos.rojas02@usa.edu.co

07 de Junio del 2024

Resumen - Este documento tiene la intención de profundizar en los conceptos que implementa nuestro parser, así como en su funcionamiento y capacidad informática. Se desea explicar a detalle sobre qué herramientas está construido, cómo funciona y en qué contextos puede trabajar con él.
Glosario: Analizador lexico, sintactico, semantico

I. Herramientas

Antlr4: Es una herramienta que se usa para generar analizadores léxicos y sintácticos. Funciona mediante la definición de gramáticas formales, las cuales describen la estructura y las reglas de un lenguaje en particular.

Python: Es un lenguaje de programación versátil y fácil de entender. Se utiliza en una amplia gama de aplicaciones, de entre las cuales destacamos el análisis de datos, útil para el objetivo de este semillero.

Linux: Ubuntu es una distribución del sistema operativo Linux. Está diseñada para ser fácil de usar y accesible para todo tipo de usuarios. Cuenta con características como: flexibilidad, capacidad, control, potencia, estabilidad, actualizaciones, soporte, entre otras.

II. Problemática

La falta de herramientas flexibles, eficientes y personalizables en el ámbito del análisis de datos y la optimización de recursos computacionales, capaces de manejar grandes volúmenes de datos sin redundancias y con una carga computacional mínima, es un problema identificado en diversos estudios. Por ejemplo, Bandler et al. (1988) describen un algoritmo flexible y eficiente para la optimización que integra aproximaciones de gradientes, mejorando significativamente la eficiencia computacional (Bandler et al., 1988). Además, Andrade et al. (2004) proponen un marco de optimización de bases de datos que soporta la reutilización de datos y cálculos, así como el almacenamiento en caché semántico activo para acelerar la evaluación de cargas de trabajo de múltiples consultas (Andrade et al., 2004). Chu y Kemere (2021) introducen GhostiPy, una herramienta de análisis espectral y procesamiento de señales que prioriza la eficiencia y el rendimiento mediante algoritmos paralelos y bloqueados, superando en eficiencia a software comercial (Chu & Kemere, 2021). Finalmente, Jerez (2013) presenta técnicas de optimización personalizada para la implementación eficiente en hardware, enfocadas en la toma de decisiones óptima en tiempo real y la reducción de limitaciones de memoria en plataformas embebidas (Jerez, 2013).

III. Objetivos

Objetivo general:

- Desarrollar un parser eficiente y versátil utilizando Python y ANTLR4, basado en programación funcional, que permita realizar una amplia gama de operaciones matemáticas, trigonométricas, matriciales y de análisis de datos, incluyendo visualización gráfica, manipulación de archivos y optimización de datos, fomentando el aprendizaje y la colaboración en el desarrollo de parsers y análisis avanzado de datos.

Objetivos específicos:

- Diseñar e implementar un parser utilizando Python y ANTLR4 capaz de manejar operaciones aritméticas básicas, funciones trigonométricas, cálculos matriciales y otras operaciones avanzadas de manera eficiente.

- Implementar funciones de generación de gráficos, permitiendo la visualización de datos mediante representaciones gráficas de puntos, barras y rectas.

- Mejorar la eficiencia y rendimiento del código del parser, optimizando algoritmos y estructuras de datos para garantizar un análisis rápido y preciso.

- Realizar pruebas exhaustivas para verificar la precisión y confiabilidad del parser en diferentes escenarios, asegurando que las funciones implementadas generan resultados correctos y consistentes.

IV. Estado del Arte

El desarrollo de parsers, o analizadores sintácticos, ha sido un área crucial en la ciencia de la computación desde sus inicios. Los parsers descomponen estructuras complejas en sus componentes constitutivos, permitiendo una interpretación y procesamiento más eficiente de los datos.

El desarrollo de modelos computacionales que emulan el parsing humano ha permitido un entendimiento más profundo de cómo las personas procesan el lenguaje. Estos modelos han ayudado a mejorar la precisión de los parsers en situaciones de ambigüedad y errores en la entrada, acercando la tecnología a la comprensión humana del lenguaje (Abney, 1989).

Los primeros parsers se desarrollaron en el contexto de la compilación de programas informáticos. La tecnología de parsing se centraba en la descomposición automática de programas en sus partes constitutivas, facilitando la traducción de lenguajes de alto nivel a código máquina (Bunt & Nijholt, 2000). Esta tecnología también se aplicó rápidamente a otros campos, como el procesamiento de lenguajes naturales, debido a la similitud entre los programas informáticos y los textos escritos en términos de su estructura basada en símbolos.

El parsing también se ha extendido a lenguajes multidimensionales, mejorando la comprensión de información de layout en textos complejos como diagramas y figuras (Tomita, 2001).

El parsing ha sido central en el desarrollo de sistemas de traducción automática, donde se combinan métodos estadísticos y reglas para mejorar la precisión del parsing en múltiples lenguajes (Yao, 2002).

El procesamiento del lenguaje natural (NLP) ha impulsado muchas de las innovaciones en la tecnología de parsing. Los parsers han sido fundamentales para la implementación de teorías lingüísticas y el desarrollo de aplicaciones prácticas de NLP. Esto incluye la traducción automática y la extracción de información, donde la interpretación correcta del texto es crucial (Sadler, 2004).

Las innovaciones en tecnología de parsing también han impactado la compilación de lenguajes de programación, permitiendo una mejor extracción de relaciones gramaticales y procesamiento de gramáticas complejas (Riezler, 2006).

La evolución de las técnicas de parsing ha sido continua y significativa. Desde el uso inicial de parsers de lenguajes libres de contexto hasta los algoritmos avanzados para lenguajes de programación complejos, la tecnología ha mejorado en términos de eficiencia y precisión. Los parsers se utilizan en una variedad de productos de software, incluyendo navegadores web y programas de compresión de datos (Grune & Jacobs, 2007).

El deep learning ha permitido avances en el parsing lingüístico profundo, especialmente en la robustez y velocidad del análisis de oraciones largas. Estas técnicas se han aplicado en plataformas que soportan búsquedas semánticas robustas y clasificación de citas en corpus científicos (Schäfer & Kiefer, 2009).
Los avances en tecnología de parsing han permitido el desarrollo de sistemas adaptativos que pueden interactuar de manera natural con usuarios humanos y extenderse fácilmente a nuevos dominios. Estas técnicas son esenciales para aplicaciones como la

traducción automática, el reconocimiento de voz y la minería de textos (Merlo, Bunt & Nivre, 2010).

Las técnicas modernas de inteligencia artificial han integrado el parsing con otros métodos avanzados, creando sistemas híbridos con capacidades de resolución de problemas más flexibles y autónomas (Tweedale & Jain, 2014).

Para manejar entradas incorrectas o imprevistas, se han desarrollado técnicas de parsing robusto y fuzzy. Estas técnicas permiten que los parsers manejen ambigüedades y errores en la entrada, mejorando la robustez y la aplicabilidad de los sistemas de parsing en entornos del mundo real (Carvalho, Oliveira & Henriques, 2014).

El parsing también se ha resaltado como un problema de modelado del lenguaje, utilizando técnicas avanzadas de redes neuronales para lograr nuevos estados del arte en parsing de constituyentes (Choe & Charniak, 2016).

Los avances recientes en tecnología de parsing han sido impulsados por el desarrollo de técnicas de aprendizaje profundo y la combinación de métodos tradicionales de parsing con redes neuronales. Esto ha resultado en arquitecturas híbridas que mejoran significativamente el rendimiento en tareas de procesamiento de lenguaje natural (Zanzotto, Satta & Cristini, 2017).

V. Definición de Conceptos

- **Diseño de una gramática:** El primer paso fue desarrollar una gramática que cumpla con un amplio margen, las necesidades preestablecidas, sin embargo, para facilitar y asegurar el correcto desarrollo de la gramática, se optó por diseñar poco a poco las estructuras de las operaciones necesarias, por ejemplo, antes de diseñar una regla que permitiera la multiplicacion de matrices, se realizó una operación que fuera capaz de sumar dos números, luego una capaz de multiplicar y así sucesivamente hasta alcanzar el cálculo necesario. Finalmente el resultado de nuestra gramática es el archivo “LabeledExpr.g4”, disponible en el repositorio.

- **Ejecución con Antlr4:** Cada vez que se realiza un cambio importante en la gramática, es necesario compilarla mediante antlr4, esto se hace para que los archivos que genera la herramienta se actualicen, y pueda funcionar correctamente el analizador léxico y sintáctico. En este paso, se pueden realizar pruebas para verificar la funcionalidad de los resultados, sin embargo, hay que tener que esas pruebas solo aseguran el correcto despliegue del árbol de análisis (Parser Tree), en otras palabras, se verifica que las operaciones estén bien escritas pero aún no realiza ningún cálculo.

- **Construcción del main:** Implementamos el parser generado por antlr4 en el lenguaje python, mediante la construcción de un main que sea capaz de interactuar con los analizadores. Por otro lado, se establece que el main sea capaz de recibir un archivo de texto como parámetro, este archivo (t.expr, presente en el repositorio) será alguna instrucción que esté definida en la gramática.

- **Desarrollo del Visitor:** Este archivo (My Visitor, presente en el repositorio) realizará el trabajo de un analizador semántico, dotando de propósito a las reglas examinadas por los otros 2 analizadores. En este se definen y realizan los cálculos de todas las operaciones matemáticas que se deben realizar, según lo establecido en la gramática.

- **Pruebas:** Una vez se cuente con un bosquejo general del proyecto, se realizan varias pruebas para corroborar el correcto funcionamiento de varios conjuntos de instrucciones, se tiene en cuenta aspectos como: correcta interpretación de los analizadores, verificación de casos específicos, resultados esperados, manejo de errores, entre otros.

- **Optimizaciones:** Una vez que se logra construir una regla y se verifica su correcto funcionamiento, se busca mejorar la manera en la que está trabaja, definimos los siguientes parámetros para medir qué tan viable es reestructurar algún elemento que permite la

ejecución de la regla: Legibilidad en el código, tiempo de procesamiento, cantidad de cálculos necesarios, relevancia de funciones y variables, y complemento con otras reglas. Si se considera que es posible mejorar algún apartado de la regla, mediante los parámetros definidos anteriormente, se realizarán y ejecutarán para analizar los resultados.

VI. Desarrollo

Fase 1: Diseño de la Gramática

Paso 1.1: Definición de la Gramática Base

Acciones:

- Iniciamos diseñando reglas para operaciones aritméticas simples, como suma y resta.
- Gradualmente, añadimos reglas para operaciones más complejas, como multiplicación y división.
- La gramática se extendió para incluir operaciones trigonométricas, cálculos matriciales y manipulación de archivos.
- Finalmente, incluimos estructuras para condicionales y bucles, asegurando una sintaxis clara y funcional.

Paso 1.2: Pruebas Iniciales de Gramática

Acciones:

- Compilamos la gramática utilizando ANTLR4 para generar los analizadores léxicos y sintácticos.
- Realizamos pruebas con ejemplos simples para asegurar que el árbol de análisis se generará correctamente.
- Ajustamos la gramática según los resultados de las pruebas para corregir errores y mejorar la precisión.

Fase 2: Ejecución con ANTLR4

Paso 2.1: Compilación de la Gramática

Acciones:

- Ejecutamos el comando de compilación de ANTLR4 para generar los archivos necesarios (analizadores léxicos y sintácticos).
- Verificamos que los archivos generados se actualizarán correctamente y que no hubiera errores en la compilación.

Paso 2.2: Pruebas de Árbol de Análisis

- Realizamos pruebas con varias entradas para verificar que el árbol de análisis refleja correctamente la estructura de las operaciones definidas en la gramática.
- Identificamos y corregimos cualquier problema en la generación del árbol de análisis, ajustando las reglas de la gramática según fuera necesario.

Fase 3: Construcción del Main

Paso 3.1: Implementación del Parser en Python

Acciones:

- Escribimos un script principal (main) en Python que interactúa con los analizadores léxicos y sintácticos generados.
- Configuramos el main para aceptar un archivo de texto como parámetro de entrada, que contuviera las instrucciones a ejecutar.

Paso 3.2: Desarrollo de la Lógica de Procesamiento

Acciones:

- Implementamos la lógica para leer y procesar el archivo de entrada, descomponiendo las instrucciones según las reglas de la gramática.
- Aseguramos que el main manejara correctamente los errores de sintaxis y proporciona mensajes claros para facilitar la depuración.

Fase 4: Desarrollo del Visitor

Paso 4.1: Implementación del Analizador Semántico

Acciones:

- Desarrollamos el Visitor (MyVisitor) en Python, definiendo métodos para cada regla de la gramática.
- Implementamos los cálculos y operaciones matemáticas, asegurando que cada regla se ejecutará correctamente.

- Añadimos soporte para operaciones avanzadas, como funciones trigonométricas, manipulación de matrices y visualización de datos.

Paso 4.2: Pruebas y Verificación

Acciones:

- Realizamos pruebas exhaustivas con diversas entradas para asegurar la precisión de los cálculos.
- Ajustamos el Visitor según los resultados de las pruebas para corregir errores y mejorar la eficiencia.

Fase 5: Pruebas Exhaustivas

Paso 5.1: Pruebas Funcionales

Acciones:

- Diseñamos conjuntos de pruebas para evaluar la funcionalidad de todas las operaciones definidas en la gramática, incluyendo aritmética básica, funciones trigonométricas, operaciones matriciales, condicionales, bucles y manipulación de archivos.
- Ejecutamos las pruebas y documentamos los resultados, asegurando que cada operación generará los resultados esperados.
- Identificamos y corregimos cualquier problema, optimizando las reglas y el código según fuera necesario.

Paso 5.2: Manejo de Errores

Acciones:

- Implementamos manejo de errores en el Visitor y el main para proporcionar mensajes claros y útiles en caso de entradas incorrectas.
- Realizamos pruebas con entradas intencionalmente incorrectas para verificar que el parser respondiera adecuadamente y facilitara la depuración.

Paso 5.3: Pruebas de Rendimiento

Acciones:

- Ejecutamos pruebas de rendimiento utilizando grandes volúmenes de datos y operaciones complejas para medir el tiempo de procesamiento y el uso de recursos.
- Analizamos los resultados de las pruebas para identificar cuellos de botella y áreas de mejora.
- Optimizamos el código y las estructuras de datos para mejorar la eficiencia y reducir la carga computacional.

Fase 6: Optimizaciones

Paso 6.1: Mejoras de Código

Acciones:

- Revisamos el código del Visitor y el main para mejorar la legibilidad, eliminando redundancias y simplificando las estructuras.
- Documentamos el código para facilitar su comprensión y mantenimiento futuro.

Paso 6.2: Optimización de Algoritmos

Acciones:

- Analizamos los algoritmos utilizados en el Visitor y el main para identificar áreas de mejora.
- Implementamos mejoras en los algoritmos para reducir el tiempo de procesamiento y el uso de recursos, asegurando que las operaciones se realizarán de manera eficiente.

Paso 6.3: Evaluación de Parámetros

Acciones:

- Definimos parámetros clave para medir la viabilidad de las reglas, como la legibilidad del código, tiempo de procesamiento, cantidad de cálculos necesarios y relevancia de funciones y variables.
- Realizamos ajustes en las reglas y el código según los resultados de la evaluación para optimizar el rendimiento del parser.

Fase 7: Implementación de Funciones Avanzadas

Paso 7.1: Visualización de Datos

Acciones:

- Añadimos soporte en el Visitor para la generación de gráficos de puntos, barras y rectas, permitiendo la visualización de datos.
- Realizamos pruebas para verificar que las funciones de visualización generan gráficos correctos y precisos.

Paso 7.2: Manipulación de Archivos
Acciones:

- Implementamos funciones en el Visitor para leer y escribir archivos de datos, asegurando la correcta manipulación de archivos en diferentes formatos.
- Realizamos pruebas para verificar que las funciones de manipulación de archivos funcionaran correctamente en diversos escenarios.

Paso 7.3: Integración de Cálculos Lambda
Acciones:

- Añadimos reglas en la gramática para definir y utilizar funciones lambda.
- Implementamos la lógica en el Visitor para evaluar y ejecutar cálculos lambda, permitiendo operaciones dinámicas y flexibles.
- Realizamos pruebas para verificar la correcta ejecución de cálculos lambda, asegurando su precisión y eficiencia.

VII. Resultados

Definimos múltiples instrucciones para poner a prueba la capacidad de nuestro proyecto, de menor a mayor complejidad, estas fueron algunas operaciones que realizamos:

- Suma, resta, multiplicación, división, potenciación, módulo.

Código: *((10 * 5) / (8+2)) ^ 2*

```
25.0

Tiempo de ejecución: 0.0018644332885742188
```

Imagen 1. Resultado de operaciones matemáticas.

```
Resultado: 25,00, Tiempo: 0,0000201710 segundos
```

Imagen 1.1. Resultado del código obtenido en Java.

- Cálculo de funciones trigonométricas.

Código: *SIN(90,Radians) + (COS(0) / (COS(0)+1))*

```
1.5

Tiempo de ejecución: 0.006634950637817383
```

Imagen 2. Resultado de funciones trigonométricas.

```
Resultado: 1,5000000000, Tiempo: 0,0000245340 segundos
```

Imagen 2.1. Resultado del código obtenido en Java.

- Operaciones matriciales.

Código: *a = [[1,2],[3,4]]*
b = INV(a)
*c = TRAS(a*b)*
c

```
luis@luis-nodo:~/antlr-pruebas/Respaldo_
Tiempo de ejecucion: 1.3429670333862305
[[-2.   4.5]
 [ 2.  -2. ]]
```

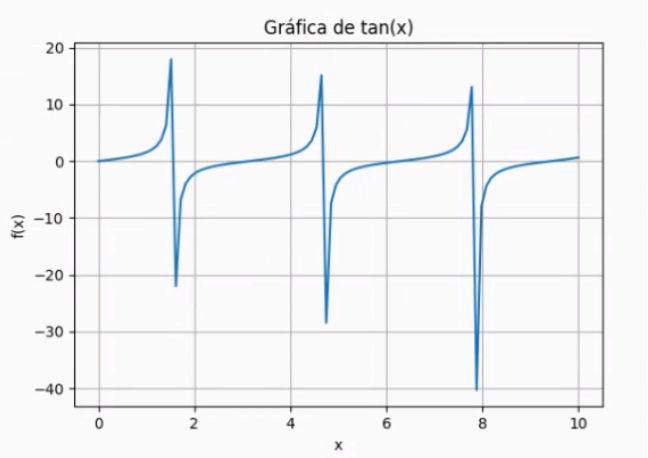
Imagen 3. Resultado de operaciones matriciales.

```
-2,00000 1,50000
1,00000 -0,50000
Tiempo total: 0,0000133770 segundos
```

Imagen 3.1. Resultado del código obtenido en Java.

- Dibujo de funciones.

Código: *DRAW(TAN(X),0,10,100)*




```
Tiempo de ejecución: 1.3299627304077148
```

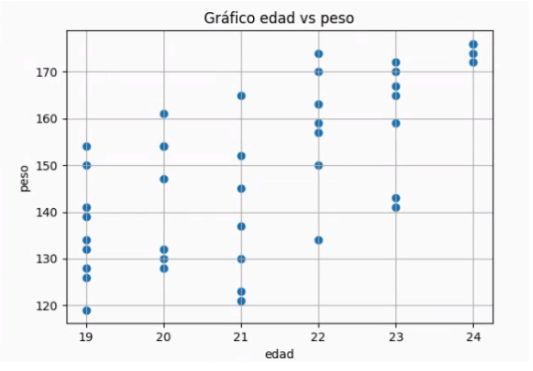
Imagen 4. Resultado de dibujo de funciones.

```
Tiempo de ejecución: 0.3551797867 segundos
```

Imagen 4.1. Resultado del código obtenido en Python.

- Dibujos Datasets

Código: *a = READ(open,Header)*
SCATTER(a,edad,peso)



```
Tiempo de ejecución: 1.5191755294799805
```

Imagen 5. Resultado de dibujo de datasets.

```
Tiempo de ejecución: 0.7869408131 segundos
```

Imagen 5.1. Resultado del código obtenido en Python.

- Condicionales y bucle

Código:*a = 10*
b = 10
if a > b {
*c = a*b*
}
else{
c = a/b
}
for(0,10,1){
c = c+1
}
c

```
11.0

Tiempo de ejecución: 0.007056236267089844
```

Imagen 6. Resultado de condicionales y bucles.

```
11.0
Tiempo de ejecución total: 0,0025328200 segundos
```

Imagen 6.1. Resultado del código obtenido en Java.

- Apertura y escritura de archivos.

Código: a = READ(open,Header)
b = [[1,2],[3,4],[5,6]]
c = a*b
WRITE(open,c,Header)

Open inicial: edad,peso
15,60
20,70
25,

Open resultado: edad,peso
15,120.0
60,280.0
125,
Tiempo de ejecucion: 0.0796422

- Operaciones sobre datasets.

Código: *a = READ(open,Header)*
a = FILLNA(a)
a
SCATTER(a,edad,peso)

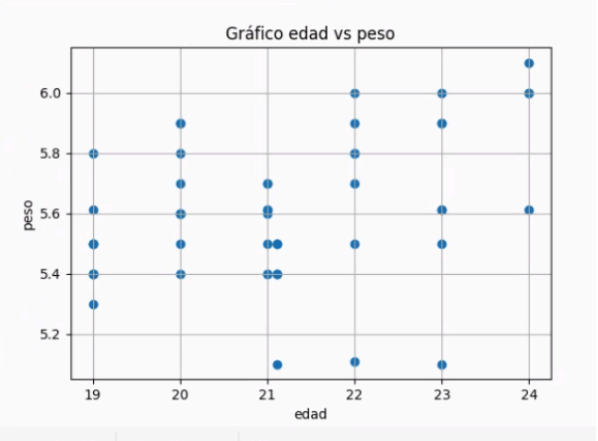


Imagen 7. Resultado de grafica de operaciones sobre datasets.

```
31     Alejandra Peña  21.000000  137.00000  5.500000
32      Rafael Lozano  24.000000  172.00000  6.000000
33       Adriana Cruz  20.000000  128.00000  5.400000
34    Antonio Cabrera  23.000000  159.00000  5.900000
35     Natalia Marín   19.000000  126.00000  5.400000
36     Fernando León   21.111111  167.00000  5.100000
37        Irene Soto   21.000000  130.00000  5.611591
38     Eduardo Bravo   20.000000  148.26087  5.700000
39     Paula Vázquez   23.000000  143.00000  5.500000
40      Diego Rivera   24.000000  176.00000  5.611591
41    Carla Castillo   19.000000  134.00000  5.500000
42      Raúl Fuentes   22.000000  170.00000  5.110000
43       Rocío Luna    21.111111  121.00000  5.400000
44      Víctor Rojas   20.000000  148.26087  5.900000
45  Beatriz Iglesias   23.000000  141.00000  5.611591
46    Martín Escobar   19.000000  154.00000  5.800000
47     Teresa Guzmán   21.000000  145.00000  5.600000
48   Samuel Guerrero   22.000000  174.00000  6.000000
49     Laura Mendoza   21.111111  139.00000  5.500000

Tiempo de ejecución: 1.3641893863677979
```

Imagen 8. Resultado de operaciones sobre datasets.

```
Tiempo total: 0.5896129608154297 segundos
```

Imagen 8.1. Resultado del código obtenido en Python.

- Definición de funciones con cálculo lambda

Código: *b = lambda x,y,z : x+y*x/z*
c = b(2,2,3)
c

```
3.333333333333333

Tiempo de ejecución: 0.0039713382720947266
```

Imagen 9. Resultado de funciones con cálculo lambda.

```
Resultado de c: 3.333333333333333
Tiempo de ejecución: 0,0000836600 segundos
```

Imagen 9.1. Resultado del código obtenido en Java.

- Calcular Indice masa corporal

Código: *data = READ(csvPesos,Header)*
data = DROP(data,edad)
data = FILLNA(data)

nombres = SELECT_COLUMN(data,nombre)
masas = SELECT_COLUMN(data,peso)
estatura = SELECT_COLUMN(data, estatura)
*LibrasAkilos = lambda x: x*0.453592*
*PiesAmetros = lambda x: x*0.3048*
masas = LibrasAkilos(masas)
estatura = PiesAmetros(estatura)
*IndiceMasaCorporal = (masas / (estatura*estatura))*
resultado =
CONCAT_COLUMN(nombres,IndiceMasaCorporal)
resultado
PesosYestaturas =
CONCAT_COLUMN(estatura,masas)
SCATTER(PesosYestaturas, estatura, peso)

```
31     Alejandra Peña  22.112134
32      Rafael Lozano  23.327135
33       Adriana Cruz  21.431764
34    Antonio Cabrera  22.301217
35      Natalia Marín  21.096892
36      Fernando León  31.348126
37         Irene Soto  20.156114
38      Eduardo Bravo  22.279852
39      Paula Vázquez  23.080548
40       Diego Rivera  27.288277
41     Carla Castillo  21.627926
42       Raúl Fuentes  31.786491
43         Rocío Luna  20.259714
44       Víctor Rojas  20.794955
45   Beatriz Iglesias  21.861631
46     Martín Escobar  22.351167
47      Teresa Guzmán  22.574982
48    Samuel Guerrero  23.598381
49      Laura Mendoza  22.434938

Tiempo de ejecución: 2.3886194229125977
```

Imagen 10. Resultado de Calculo IMC

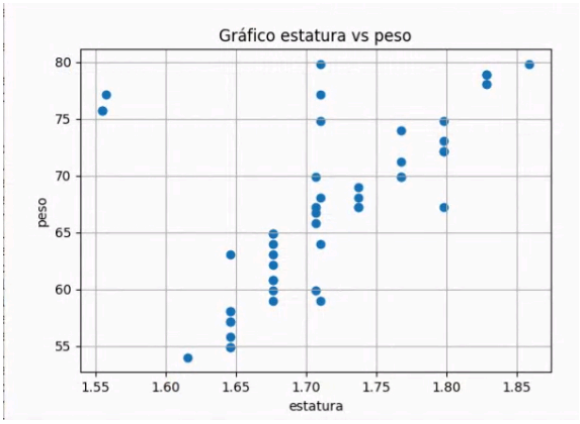


Imagen 11. Resultado de gráfica de IMC

```
Tiempo total: 0.5044748783111572 segundos
```

Imagen 11.1. Resultado del código obtenido en Java.

Todos los problemas ingresados arrojaron el resultado esperado, esto incluye casos donde el proyecto debía detenerse por algún error de sintaxis, operaciones indeterminadas (0 dividido 0 o similares) o mala implementación del proyecto. Al ser todos los resultados positivos concluimos que el proyecto funciona adecuadamente, según los parámetros preestablecidos para el semillero.

VIII. Cómo usarlo

Dar uso al proyecto es bastante simple, primero es necesario verificar que se tengan todas las herramientas necesarias para ejecutar la aplicación (véase el readme disponible en el repositorio), posteriormente se debe dirigir al archivo t.expr, el cual es un txt con las instrucciones a realizar, si se desea se puede crear otro txt y enviarlo como parámetro, solo no olvide modificar la instruccion para ejecutar el archivo en la consola.

En cuanto al diseño de las instrucciones, se recomienda ver los ejemplos que están en el repositorio, y si es posible ver la gramática para entender un poco más acerca de cómo construir una instrucción.

IX. Beneficios de construir e implementar un parser

El desarrollo de un parser presenta numerosos beneficios, como se ha demostrado en este informe. Uno de los principales beneficios es el aprendizaje y exploración. La creación de un parser permite profundizar en el análisis sintáctico y las gramáticas formales, proporcionando una comprensión más detallada de cómo se construyen y funcionan los analizadores léxicos y sintácticos. Este conocimiento es crucial para cualquier profesional en el campo de la informática y la programación.

Pruebas:

Imagen 12. Imagen prueba de fecha anterior gramática.

https://drive.google.com/file/d/1xTyxIMs28aBdQkDjkVzbVcQHt3_K_K3c/view?usp=drive_link

Link 1. Primer gramática creada(hace 6 meses)

https://drive.google.com/file/d/1a3OrJJN4fHX8cf-1kvLyngOKXcvDBYR0/view?usp=drive_link

Link 2. Segunda gramática creada (actual)

Un beneficio significativo de implementar un parser propio es la flexibilidad y personalización que ofrece. Permite un control total sobre la sintaxis y las reglas de la gramática, adaptando el comportamiento del proyecto a necesidades específicas. En nuestro desarrollo, hemos incorporado operaciones avanzadas como funciones trigonométricas, cálculos matriciales y visualización de datos, mostrando así la capacidad del parser para manejar una amplia gama de operaciones y contextos. La gramática es tan adaptable que puede ser utilizada para otros análisis sintácticos y semánticos, como se demuestra en este ejemplo implementado en Java. Además, el uso de ANTLR4 facilita la reutilización de gramáticas al generar analizadores para múltiples lenguajes de programación, promoviendo la interoperabilidad y reutilización en diferentes entornos. Esto es particularmente útil en proyectos que requieren integración con otros sistemas y tecnologías, demostrando la eficiencia del parser en el manejo de operaciones complejas y grandes volúmenes de datos.

Pruebas:

```java
public class Calc {
    public static void main(String[] args) throws Exception {
        // Verificar si se proporciona un archivo como argumento de línea de comandos
        if (args.length != 1) {
            System.err.println("Uso: java Calc archivo");
            return;
        }

        // Obtener el nombre del archivo de los argumentos de línea de comandos
        String fileName = args[0];

        // Crear un flujo de entrada para leer desde el archivo
        InputStream inputStream = new FileInputStream(fileName);
        CharStream input = CharStreams.fromStream(inputStream);

        // Crear un analizador léxico y sintáctico
        LabeledExprLexer lexer = new LabeledExprLexer(input);
        CommonTokenStream tokens = new CommonTokenStream(lexer);
        LabeledExprParser parser = new LabeledExprParser(tokens);

        // Parsear la entrada para obtener el árbol de análisis sintáctico
        ParseTree tree = parser.prog();

        // Crear un visitor y visitar el árbol de análisis sintáctico
        EvalVisitor evalVisitor = new EvalVisitor();
        evalVisitor.visit(tree);


        // Cerrar el flujo de entrada
        inputStream.close();
    }
}
```

Imagen 13. Prueba código de Calc(Analizador sintáctico) de Java

```java
class EvalVisitor extends LabeledExprBaseVisitor<Float> {
    public Float visitAddSub(LabeledExprParser.AddSubContext ctx) {
        Float left = visit(ctx.expr(0));  // Visitamos el lado izquierdo de la suma
        Float right = visit(ctx.expr(1)); // Visitamos el lado derecho de la suma
        Float suma = left + right;
        System.out.println("Resultado: "+suma);
        return suma;              // retornamos el resultado
    }

    public Float visitNumber(LabeledExprParser.NumberContext ctx) {
        return Float.parseFloat(ctx.NUMBER().getText());
    }
}
```

Imagen 14. Prueba código de MyVisitorc(Analizador semántico) de Java

Imagen 15. Prueba código de Ejecutado de Java

La eficiencia y rendimiento son también beneficios importantes, como se ha visto en las pruebas de rendimiento realizadas. Las optimizaciones en los algoritmos y estructuras de datos han permitido reducir significativamente el tiempo de procesamiento y el uso de recursos, asegurando que el parser puede manejar grandes volúmenes de datos rápidamente y con precisión. Esto es crucial para aplicaciones que requieren análisis de datos en tiempo real.

Código: *((10 * 5) / (8+2)) ^ 2*

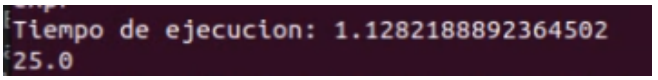


Imagen 16. Prueba código ejecucion mismo código que en imagen 1. Se evidencia una clara diferencia en los tiempos.

Finalmente, la robustez y manejo de errores del parser ha sido demostrada a través de pruebas exhaustivas. El parser ha mostrado una capacidad notable para manejar errores de sintaxis y operaciones indeterminadas, proporcionando mensajes claros y útiles para facilitar la depuración. Esta robustez asegura que el sistema pueda operar de manera confiable en diversos entornos y condiciones, mejorando la experiencia del usuario y la fiabilidad del sistema.

Código: a = b+5

a

```
Error en linea 1, columna 4
'b' no existe
```

Imagen 17. Prueba error en código.

Código: a = [[1,2],[3,4]
b = INVa)
c = TRASa*b)
c

```
Ingrese el nombre del archivo de instrucciones o escriba 'salir' pa
ra finalizar: t.expr
Error de análisis sintáctico en linea 1, columna 4: token recogniti
on error at: '[[1,2],[3,4]\n'
Error de análisis sintáctico en linea 2, columna 2: mismatched inpu
t '=' expecting NEWLINE
Error de análisis sintáctico en linea 3, columna 11: extraneous inp
ut ')' expecting NEWLINE
```

Imagen 18. Prueba error en código.

X. Conclusiones

El desarrollo del parser en Python utilizando ANTLR4 ha demostrado ser una solución efectiva y versátil para el análisis de datos y la ejecución de operaciones complejas. Las pruebas exhaustivas realizadas han demostrado que el parser es capaz de manejar operaciones aritméticas, trigonométricas, matriciales y de manipulación de archivos de manera eficiente y precisa. Todos los resultados obtenidos en las pruebas fueron correctos y consistentes con los valores esperados, lo que confirma la confiabilidad del sistema.

La capacidad del parser para integrar nuevas reglas y operaciones ha sido fundamental. La implementación de funciones avanzadas, como el cálculo de funciones lambda y la visualización gráfica, ha demostrado la flexibilidad y escalabilidad del sistema, permitiendo adaptarse a diversas necesidades y contextos de uso. Esta flexibilidad ha permitido al parser evolucionar y mejorar constantemente, ofreciendo una solución robusta y adaptable para el análisis de datos.

El sistema ha mostrado una robustez significativa en el manejo de errores, proporcionando mensajes claros y útiles que facilitan la depuración. Las pruebas con entradas incorrectas confirmaron que el parser puede gestionar adecuadamente errores de sintaxis y operaciones indeterminadas, mejorando la experiencia del usuario y la fiabilidad del sistema. Esta robustez es crucial para asegurar que el parser pueda operar de manera confiable en entornos reales y manejar datos diversos y complejos sin fallos.

Las optimizaciones realizadas en los algoritmos y estructuras de datos han mejorado notablemente el rendimiento del parser. Las pruebas de rendimiento indicaron una reducción significativa en el tiempo de procesamiento y el uso de recursos, lo que permite manejar grandes volúmenes de datos de manera eficiente. Estas mejoras aseguran que el parser no solo sea preciso, sino también rápido y capaz de operar en tiempo real, ofreciendo resultados inmediatos y precisos.

La capacidad de ANTLR4 para generar analizadores en múltiples lenguajes de programación y la modularidad del diseño del parser facilitan la interoperabilidad y reutilización de la gramática en diferentes entornos. Esto es especialmente beneficioso para proyectos que requieren integración con otros sistemas y tecnologías, asegurando que el parser pueda ser utilizado en una amplia gama de aplicaciones y contextos sin necesidad de reescribir código.

El parser desarrollado no solo es una herramienta educativa valiosa para comprender el análisis sintáctico y semántico, sino que también tiene aplicaciones prácticas en la optimización de datos, visualización gráfica y análisis de grandes volúmenes de información. Esto fomenta la colaboración y el desarrollo continuo en el ámbito del análisis avanzado de datos, proporcionando una base sólida para futuros proyectos de machine learning y exploraciones en este campo.

XI. Bibliografías

1. Bunt, H., & Nijholt, A. (2000). Parsing Technologies.
2. Grune, D., & Jacobs, C. (2007). Parsing Techniques: A Practical Guide.
3. Sadler, L. (2004). Current Issues in Parsing Technology.
4. Abney, S. P. (1989). A Computational Model of Human Parsing.
5. Carvalho, P., Oliveira, N., & Henriques, P. (2014). Unfuzzying Fuzzy Parsing.

6. Zanzotto, F. M., Satta, G., & Cristini, G. (2017). CYK Parsing over Distributed Representations.
7. Turian, J. P., & Melamed, I. D. (2006). Advances in Discriminative Parsing.
8. Schäfer, U., & Kiefer, B. (2009). Advances in Deep Parsing of Scholarly Paper Content.
9. Choe, D. K., & Charniak, E. (2016). Parsing as Language Modeling.
10. Merlo, P., Bunt, H., & Nivre, J. (2010). Current Trends in Parsing Technology.
11. Tweedale, J., & Jain, L. (2014). Advances in Modern Artificial Intelligence.
12. Yao, M. (2002). Research on Parsing Technology in Machine Translation System.
13. Tomita, M. (2001). Recent Advances in Parsing Technology.
14. Riezler, S. (2006). New Developments in Parsing Technology.
15. Tomita, M., Bunt, H., & Carroll, J. (2004). New Developments in Parsing Technology.
16. Bandler, J., Chen, S. H., Daijavad, S., & Madsen, K. (1988). Efficient optimization with integrated gradient approximations. *IEEE Transactions on Microwave Theory and Techniques*, 36, 444-455.
17. Andrade, H., Kurç, T., Sussman, A., & Saltz, J. (2004). Optimizing the execution of multiple data analysis queries on parallel and distributed environments. *IEEE Transactions on Parallel and Distributed Systems*, 15, 520-532.
18. Chu, J. P., & Kemere, C. T. (2021). GhostiPy: An efficient signal processing and spectral analysis toolbox for large data. *eNeuro*.
19. Jerez, J. (2013). Custom optimization algorithms for efficient hardware implementation. *PhD Thesis*.