Por: Diego Alejandro Angarita Arboleda
Documentación Proyecto final: Analizador sintáctico con algoritmos LL(1) y SLR(1)

Objetivo del proyecto: Desarrollar un parser en python para leer GLC's (gramáticas libres de contexto) y leer cadenas usando los algoritmos LL(1) y SLR(1) de forma eficiente.

-Decisiones de diseño:

Separar cada algoritmo en un archivo distinto: esto se hace con el fin de mejorar la legibilidad y permite la aplicación de los algoritmos de forma más clara.

-Estructuras de Datos Clave:

Diccionarios (Hash Maps): Son la estructura principal para todas las tablas de análisis (Tabla LL(1), Tablas ACCIÓN e IR_A de SLR(1)), así como para los conjuntos FIRST y FOLLOW. Se eligieron por su eficiencia de búsqueda promedio de O(1).

Conjuntos (Sets): Se utilizan para almacenar los conjuntos FIRST y FOLLOW. Esto es fundamental ya que estos conjuntos no admiten duplicados y requieren operaciones eficientes de unión y diferencia.

Listas (Arrays): Cumplen dos roles vitales:

Como Pilas (LIFO): Son el motor de ambos analizadores para procesar la cadena de entrada (usando .append() y .pop()).

Como Colas (FIFO): Se usa una lista como cola en la construcción del autómata SLR(1) (usando .append() y .pop(0)) para explorar estados en amplitud (BFS).

Tuplas: Se usan como claves inmutables para los diccionarios de las tablas de análisis (ej. (Estado, Símbolo) o (No Terminal, Terminal)).

-Explicaciones de los algoritmos:

Cálculo de FIRST y FOLLOW:

Ambos se calculan usando un algoritmo iterativo de punto fijo. Se inicializan los conjuntos (vacíos para FIRST, y FOLLOW(S) = {$} para el símbolo inicial S).

Se itera sobre todas las producciones, aplicando las reglas de FIRST y FOLLOW y añadiendo símbolos a los conjuntos.

El bucle se repite hasta que en una iteración completa no se añade ningún símbolo nuevo a ningún conjunto. Esto garantiza que todos los símbolos se han propagado correctamente.

Analizador LL(1) (Descendente):

Construcción de Tabla: Se crea una tabla M[NoTerminal, Terminal]. Para cada producción A -> a:

Para cada terminal 'a' en FIRST(a), se añade A -> a a M[A, a].

Si épsilon ('e') está en FIRST(a), para cada terminal 'b' en FOLLOW(A), se añade A -> a a M[A, b].

Detección de Conflictos: Si al intentar añadir una producción a una celda M[A, a], esta ya contiene otra producción, la gramática NO es LL(1).

Análisis: Utiliza una pila (iniciada con S$) y un puntero de entrada. Compara el tope de la pila (X) con la entrada (a). Si X es un no terminal, se desapila X y se apila la producción M[X, a] en orden inverso. Si X es un terminal, se compara con 'a' y se avanza.

Analizador SLR(1) (Ascendente):

Construcción del Autómata LR(0): Este es el núcleo.

Se aumenta la gramática (S' -> S).

Se crea un estado inicial a partir de la "Clausura" del item [S' -> ·S].

Se exploran recursivamente nuevas transiciones usando la función "IR_A" (GOTO). Cada nuevo conjunto de items generado por "Clausura" es un nuevo estado.

Se utiliza una cola para procesar los estados y una estructura de datos (diccionario) para no repetir estados ya calculados.

Construcción de Tablas ACCION e IR_A: Se recorren todos los estados generados:

Acción de Desplazamiento (Shift): Si un item [A -> a·ab] está en el estado 'i' y la transición IR_A(i, a) lleva al estado 'j', entonces ACCION[i, a] = "desplazar j".

Acción de Reducción (Reduce): Si un item completo [A -> a] está en el estado 'i', para cada terminal 'b' en FOLLOW(A), entonces ACCION[i, b] = "reducir A -> a".

Acción de Aceptación: Si [S' -> S·] está en el estado 'i', ACCION[i, $] = "aceptar".

Tabla IR_A (GOTO): Almacena las transiciones para No Terminales.

Detección de Conflictos:

- Shift-Reduce: Si en un estado 'i' y terminal 'a', una regla pide "desplazar" (Regla 1) y otra pide "reducir" (Regla 2).

Reduce-Reduce: Si en un estado 'i' y terminal 'a', dos reglas diferentes piden "reducir" (Regla 2).

Análisis: Utiliza una pila de estados. En cada paso, mira el estado en el tope ('i') y el símbolo de entrada ('a'). Ejecuta la acción de ACCION[i, a] (desplazar, reducir o aceptar).

-Problemas al Desarrollar el Proyecto:

Iteración de Punto Fijo: Asegurar que los bucles de FIRST y FOLLOW converjan correctamente fue un desafío. La solución fue usar una bandera de "cambio" que rastrea si se añadió algún elemento nuevo en una pasada completa.

Gestión de Estados LR(0): El autómata SLR(1) puede tener muchos estados. Fue crucial implementar una forma de saber si un estado recién calculado ya existía. La solución fue usar un diccionario que mapea un frozenset (conjunto inmutable) de ítems a su ID de estado.

Detección de Conflictos: Implementar correctamente la lógica para registrar un conflicto (shift-reduce o reduce-reduce) fue clave. El código lo maneja verificando si una celda de la tabla ACCION ya está ocupada antes de escribir en ella.

Problemas al medir los tiempos de ejecución de los algoritmos:
Ya que python, a diferencia de c++ es un lenguaje interpretado, hace que el código funcione, ejecutándose parte por parte, y por mi parte que no estoy acostumbrado a trabajar con este tipo de lenguajes, tuve problemas al medir tiempos en las partes correctas, ya que yo pensaba que se deban implementar en los archivos correspondientes a los documentos, pero la solución fue medir los tiempos en el archivo principal al llamar las funciones.

-BigO de los algoritmos:

Cálculo de FIRST/FOLLOW: La complejidad es polinomial, relacionada con el tamaño de la gramática |G| y el número de no terminales |N|. En el peor caso, puede ser O(N * |G|), pero en la práctica es muy rápido.

Construcción de Tabla LL(1): Es rápido. Su complejidad es proporcional al número de producciones multiplicado por el número de terminales.

Construcción de Tabla SLR(1): Esta es la fase más costosa. El número de estados LR(0) puede, en el peor caso teórico, ser exponencial con respecto al tamaño de la gramática. Sin embargo, para la mayoría de las gramáticas de lenguajes de programación, es polinomial.

Análisis de Cadena (Ambos): Ambos analizadores tienen un rendimiento O(n), donde 'n' es la longitud de la cadena de entrada. Esto es lineal y extremadamente eficiente, ya que solo realizan una pasada sobre la entrada.

-Tabla de algoritmos y tiempos:

Estructura / Algoritmo	Caso de entrada (n)	Resultado (Tiempo en ms)	Complejidad Esperada	Observaciones
Cálculo de FIRST	Gramática (5 prod)	0.000150	O(N * G)	Algoritmo iterativo, rápido.
Cálculo de FOLLOW	Gramática (5 prod)	0.000180	O(N * G)	Similar a FIRST, depende de él.
Construcción Tabla LL(1)	Gramática (5 prod)	0.000120	O(N * G * T)	Más rápida que SLR(1), pero no se puede aplicar a gramáticas complejas.
Construcción Tabla SLR(1)	Gramática (5 prod)	0.000850	O(G^3 - G^4)	Construir un autómata es más costoso.

-Bibliografía:

-Aho - Compilers - Principles, Techniques, and Tools
-https://es.scribd.com/document/380893509/Lenguaje-Libre-Del-Contexto
-https://dlsiis.fi.upm.es/procesadores/Documentos/FirstFollow
-https://www.geeksforgeeks.org/compiler-design-ll1-parser-in-python
-https://www.geeksforgeeks.org/compiler-design-slr1-parser-using-python/
-https://web.unizar.es/listas/doc/pld/teoria/pli-08
-http://cartagena99.com/recursos/alumnos/apuntes/PL_Analisis_Sintactico_II_SLR
-https://ocw.unican.es/pluginfile.php/1201/course/section/1487/PL-OCW-Analizador-SLR-y-LR-canonico-v1.0

-Usos de la IA:

modelo de IA usado: claude sonnet 4.5

-que estructura de datos puedo usar para medir tiempos de ejecución en python

-de qué manera puedo hacer para solo medir los tiempos de ejecución de los algoritmos

-Videos:

https://www.loom.com/share/7df6805f07344c0f9e45d0ae637fbce4
https://www.loom.com/share/561f85e33eb0474b83bc5380a3c19bb6