Cambios que hicimos en el proyecto

- Añadimos la función `extract_terminals` para extraer los terminales automáticamente desde las producciones. Esto evita duplicar lógica, reduce errores y facilita construir las tablas LL(1) y SLR(1).
- En el archivo `main.py`, agregamos bloques para imprimir las tablas:
  - Si la gramática es LL(1), se muestra la tabla LL(1).
  - Si es SLR(1), se imprimen las tablas ACTION y GOTO.
- Ejemplo de código:

```
if is_ll1:
    print("LL(1) Table:")
    for nonterminal, rules in ll1_table.items():
        for terminal, production in rules.items():
            print(f"{nonterminal} -> {terminal}: {production}")

if is_slr1:
    print("ACTION Table:")
    for state, actions in slr_ACTION.items():
        print(f"State {state}: {actions}")

    print("GOTO Table:")
    for state, gotos in slr_GOTO.items():
        print(f"State {state}: {gotos}")
```

Aspectos teóricos fundamentales (en lenguaje de estudiante)

analizador sintáctico

Un analizador sintáctico (parser) es un programa que se encarga de revisar si una cadena de símbolos cumple con las reglas de una gramática. Existen analizadores **ascendentes** (como SLR(1)) y **descendentes** (como LL(1)).

Tabla LL(1)

La tabla LL(1) se usa para analizar una cadena **de arriba hacia abajo**, sin retroceso.

- Para construirla se usan los conjuntos **FIRST** y **FOLLOW**.
- Si un terminal está en FIRST de una producción, esa producción se agrega a la tabla.
- Si una producción puede ir a vacío (ε), también se usa FOLLOW del no terminal.
- Si una celda tiene más de una producción, hay **conflicto** y la gramática no es LL(1).

Conjuntos FIRST y FOLLOW

- **FIRST(X)**: todos los terminales que pueden aparecer al principio de las derivaciones de X.
- **FOLLOW(X)**: todos los terminales que pueden aparecer inmediatamente después de X en alguna producción.
- Estos conjuntos se usan para construir tanto la tabla LL(1) como para llenar correctamente la tabla ACTION del SLR(1).

¿Qué es un analizador SLR(1)?

Un analizador SLR(1) funciona **de abajo hacia arriba**. Usa dos tablas: **ACTION** y **GOTO**.

Tabla ACTION (SLR(1))

- Le dice al parser qué hacer según el estado actual y el símbolo de entrada.
- Puede tener tres acciones:
  - **shift**: avanzar y pasar a un nuevo estado.
  - **reduce**: aplicar una producción (según el FOLLOW del no terminal).
  - **accept**: cuando la cadena es reconocida.
- Se llena usando el autómata LR(0) y los conjuntos FOLLOW.

Tabla GOTO (SLR(1))

- Dice a qué estado ir cuando se ha hecho una reducción a un no terminal.
- Por ejemplo, si desde el estado 0 se reduce a un T, GOTO[0, T] dice a qué estado ir ahora.

¿Cuándo es LL(1) o SLR(1)?

- Una gramática es **LL(1)** si:
  - No tiene recursión por la izquierda.
  - No tiene ambigüedad.
  - La tabla LL(1) no tiene conflictos.
- Una gramática es **SLR(1)** si:
  - El autómata LR(0) no produce conflictos en la tabla ACTION.
  - Las reducciones se pueden decidir solo con FOLLOW.

Sustentación teórica y técnica del proyecto

Este proyecto permite identificar si una gramática es LL(1), SLR(1), ambas o ninguna. Luego construye las tablas correspondientes y permite analizar cadenas paso a paso.

1. Implementación LL(1)

- En `ll1_parser.py` se construye la tabla LL(1).
- Se usa `first_follow.py` para calcular los conjuntos FIRST y FOLLOW.
- Si hay conflicto en una celda, la gramática no es LL(1).

2. Implementación SLR(1)

- En `slr1_parser.py` se construye el autómata LR(0).
- Se crean las tablas ACTION y GOTO en `build_slr_tables`.
- Las acciones se asignan según si hay un `shift`, una `reduce` o un `accept`, dependiendo de la posición del punto y los FOLLOW.

3. Integración en el sistema

- En `main.py` se orquesta todo:
  - Se lee la gramática.
  - Se calculan FIRST y FOLLOW.
  - Se verifica si es LL(1), SLR(1), ambas o ninguna.
  - Se imprimen las tablas LL(1), ACTION y GOTO.
  - Se permite al usuario ingresar una cadena para analizarla paso a paso.