UNIVERSIDAD EAFIT

Proyecto final

Lenguajes Formales y Compiladores (2025-2)

Group Number: C2566-ST0270-3952

Nombre Completo

Jean Carlo Ardila Acevedo
Andrés Restrepo Giraldo
Juan Manuel Young Hoyos

Tutor: Oscar Rodriguez Cifuentes

Medellín, 15 de noviembre de 2025

Índice

1 | Introducción al Proyecto

Este proyecto implementa un analizador sintáctico (parser) para gramáticas libres de contexto que puede determinar si una gramática es LL(1), SLR(1), ambas o ninguna. Además, una vez clasificada la gramática, el programa puede validar si cadenas de entrada son válidas según las reglas de la gramática.

1.1 | ¿Qué es una gramática libre de contexto?

Una gramática libre de contexto es un conjunto de reglas que nos permite generar cadenas de texto válidas. Imagina que son las reglas del juego para construir expresiones o sentencias válidas.
Por ejemplo, una gramática simple podría ser:

- `S` → `aS` (S puede convertirse en "a" seguido de otra S)
- `S` → `b` (o S puede convertirse en "b")

Con estas reglas, podríamos generar cadenas como: `b`, `ab`, `aab`, `aaab`, etc.

1.2 | Tipos de parsers implementados

1.2.1 | Parser LL(1)

- **L**: Lee la entrada de izquierda a derecha (Left-to-right)
- **L**: Usa derivación por la izquierda (Leftmost derivation)
- **1**: Usa un símbolo de lookahead

Es un parser *top-down* (de arriba hacia abajo), como si fueras construyendo un árbol desde la raíz hacia las hojas.

1.2.2 | Parser SLR(1)

- **S**: Simple
- **L**: Lee de izquierda a derecha
- **R**: Usa derivación por la derecha (Rightmost derivation)
- **1**: Un símbolo de lookahead

Es un parser *bottom-up* (de abajo hacia arriba), como si fueras armando las piezas desde las hojas hasta formar el árbol completo.

2 | Arquitectura del Sistema

El proyecto está organizado en módulos que trabajan juntos como una cadena de producción. Vamos a ver cada uno paso a paso.

2.1 | Flujo general del programa

1. El usuario proporciona una gramática
2. El programa la analiza y calcula información necesaria
3. Intenta construir parsers LL(1) y SLR(1)
4. Determina qué tipo de gramática es
5. Permite validar cadenas de entrada

2.2 | Módulos principales

El código está dividido en módulos, cada uno con una responsabilidad específica:

- `main.py`: Punto de entrada del programa
- `cli.py`: Interfaz de línea de comandos
- `utils.py`: Tipos de datos básicos (símbolos)
- `grammar.py`: Representación de gramáticas
- `first_follow.py`: Cálculo de conjuntos FIRST y FOLLOW
- `ll1.py`: Implementación del parser LL(1)
- `slr1.py`: Implementación del parser SLR(1)

3 | Componentes Fundamentales

Antes de entender los parsers, necesitamos conocer los bloques básicos con los que trabaja el programa.

3.1 | Símbolos (utils.py)

Un símbolo es la unidad básica en una gramática. Piensa en ellos como las piezas de Lego con las que construimos.

3.1.1 | Tipos de símbolos

1. **Terminales**: Son símbolos que aparecen en el resultado final
   - Se representan con letras minúsculas, dígitos o símbolos
   - Ejemplos: `a`, `b`, `+`, `*`, `(`, `)`
2. **No terminales**: Son símbolos que se pueden expandir usando reglas
   - Se representan con letras MAYÚSCULAS
   - Ejemplos: `S`, `A`, `B`, `E`, `T`
3. **Epsilon (ε)**: Representa la cadena vacía
   - En el código se representa con la letra `'e'`
   - Significa “nada” o “cadena vacía”
4. **Marcador de fin ($)**: Indica el final de la entrada
   - Se usa para saber cuándo terminamos de leer

3.1.2 | Implementación de Symbol

La clase `Symbol` identifica automáticamente el tipo de símbolo basándose en su carácter:

- Si el carácter está entre ’A’ y ’Z’, es un no terminal
- Si es ’e’ con bandera especial, es epsilon
- Si es ’$’ con bandera especial, es marcador de fin
- Todo lo demás es un terminal

3.2 | Producciones (grammar.py)

Una producción es una regla de la gramática. Tiene la forma: `A` $\rightarrow \alpha$

- **Lado izquierdo (lhs)**: Un no terminal (ej: `S`)
- **Lado derecho (rhs)**: Una secuencia de símbolos (ej: `a S b`)

Ejemplo: La producción `S` $\rightarrow$ `aS` significa que el símbolo `S` puede reemplazarse por `a` seguido de `S`.

3.3 | Gramática (grammar.py)

Una gramática completa consiste en:

1. Un conjunto de **no terminales**
2. Un conjunto de **terminales**
3. Un conjunto de **producciones**
4. Un **símbolo inicial** (siempre es `S` en este proyecto)

3.3.1 | Formato de entrada

El programa espera la gramática en un formato específico:

```
3
S -> S+T T
T -> T*F F
F -> (S) i
```

Donde:

- Primera línea: número de líneas de producción (3)
- Siguientes líneas: cada línea define producciones para un no terminal
- Las alternativas se separan por espacios

En el ejemplo anterior:

- `S` tiene dos producciones: `S` $\rightarrow$ `S+T` y `S` $\rightarrow$ `T`
- `T` tiene dos producciones: `T` $\rightarrow$ `T*F` y `T` $\rightarrow$ `F`
- `F` tiene dos producciones: `F` $\rightarrow$ `(S)` y `F` $\rightarrow$ `i`

4 | Conjuntos FIRST y FOLLOW

Antes de construir los parsers, necesitamos calcular dos conjuntos importantes que nos ayudan a tomar decisiones durante el análisis sintáctico.

4.1 | Conjunto FIRST

Definición: `FIRST(X)` es el conjunto de terminales que pueden aparecer al inicio de una cadena derivada desde `X`.

4.1.1 | ¿Para qué sirve?

Imagina que estás leyendo una cadena y ves el símbolo `a`. El conjunto FIRST te dice qué producciones podrías usar basándote en ese primer símbolo.

4.1.2 | Reglas de cálculo

1. Si `X` es un terminal, entonces `FIRST(X) = {X}`
2. Si `X` es un no terminal con producción `X` $\rightarrow$ $Y_1Y_2 \ldots Y_n$:
   - Agrega `FIRST(`Y_1`) - {`ε`}` a `FIRST(X)`
   - Si Y_1 puede derivar ε, agrega `FIRST(`Y_2`) - {`ε`}`
   - Continúa mientras los símbolos puedan derivar ε
   - Si todos los Y_i pueden derivar ε, agrega ε a `FIRST(X)`

4.1.3 | Ejemplo

Para la gramática:

```
S -> aB | c
B -> b | e
```

Los conjuntos FIRST son:

- `FIRST(S) = {a, c}` (S puede empezar con 'a' o con 'c')
- `FIRST(B) = {b,` ε`}` (B puede empezar con 'b' o ser vacío)
- `FIRST(a) = {a}`, `FIRST(b) = {b}`, `FIRST(c) = {c}`

4.1.4 | Implementación

El algoritmo usa **iteración de punto fijo**:

1. Inicializa los conjuntos FIRST para todos los símbolos
2. Repite hasta que no haya cambios:
   - Para cada producción, calcula el FIRST del lado derecho
   - Agrégalo al FIRST del lado izquierdo

4.2 | Conjunto FOLLOW

Definición: `FOLLOW(A)` es el conjunto de terminales que pueden aparecer inmediatamente después de `A` en alguna derivación.

4.2.1 | ¿Para qué sirve?

Si el parser está procesando un no terminal `A` y necesita decidir qué hacer, mira el siguiente símbolo de entrada. Si ese símbolo está en `FOLLOW(A)`, el parser puede reducir (terminar de procesar `A`).

4.2.2 | Reglas de cálculo

1. `FOLLOW(S)` contiene $ (el símbolo inicial siempre puede estar seguido por fin de entrada)
2. Para cada producción `A` $\rightarrow$ α`B`β:
   - Agrega `FIRST(`β`) - {`ε`}` a `FOLLOW(B)`
   - Si β puede derivar ε (o está vacío), agrega `FOLLOW(A)` a `FOLLOW(B)`

4.2.3 | Ejemplo

Para la gramática:

```
S -> AB
A -> a
B -> b
```

Los conjuntos FOLLOW son:

- `FOLLOW(S) = {$}` (S es el inicio, puede terminar la entrada)
- `FOLLOW(A) = {b}` (A está seguido por B, y B puede empezar con 'b')
- `FOLLOW(B) = {$}` (B está al final, hereda FOLLOW de S)

4.2.4 | Implementación

Similar al cálculo de FIRST, usa iteración de punto fijo:

1. Inicializa todos los FOLLOW como vacíos
2. Agrega $ a FOLLOW del símbolo inicial
3. Repite hasta que no haya cambios:
   - Para cada producción y cada posición en el lado derecho
   - Si encuentra un no terminal, actualiza su FOLLOW según las reglas

5 | Parser LL(1)

El parser LL(1) es un analizador sintáctico *predictivo* que trabaja de arriba hacia abajo.

5.1 | Idea principal

El parser usa una **pila** y una **tabla de análisis sintáctico** para decidir qué producciones aplicar.

- **Pila**: Almacena símbolos que esperamos procesar
- **Tabla M[A, a]**: Para no terminal `A` y terminal `a`, indica qué producción usar
- **Lookahead**: Miramos un símbolo adelante para decidir

5.2 | Construcción de la tabla LL(1)

La tabla se construye usando los conjuntos FIRST y FOLLOW:
Para cada producción `A` $\rightarrow \alpha$:

1. Para cada terminal `a` en `FIRST(`α`)`:
   - Agrega la producción `A` $\rightarrow \alpha$ a `M[A, a]`
2. Si ε está en `FIRST(`α`)`:
   - Para cada terminal `b` en `FOLLOW(A)`
   - Agrega la producción `A` $\rightarrow \alpha$ a `M[A, b]`

5.3 | Conflictos en LL(1)

Una gramática **NO es LL(1)** si:

- Una celda `M[A, a]` tiene más de una producción
- Esto se llama un **conflicto**

Ejemplo de conflicto:

```
S -> aS | aB
```

Ambas producciones empiezan con 'a', así que `M[S, a]` tendría dos entradas. El parser no sabría cuál elegir.

5.4 | Algoritmo de parsing LL(1)

1. **Inicialización**:
   - Pila = `[$, S]` (símbolo de fin y símbolo inicial)
   - Entrada = cadena + `$`
2. **Bucle principal**:
   - `top` = tope de la pila
   - `current` = símbolo actual de entrada
3. **Casos**:
   - [a] Si `top == current`:
     - Sacar de la pila
     - Avanzar en la entrada
   - [b] Si `top` es no terminal:
     - Buscar `M[top, current]`
     - Si no existe: **rechazar**
     - Si existe: sacar top, poner símbolos del lado derecho (en orden inverso)
   - [c] Si `top` es terminal pero `top` $\neq$ `current`:
     - **Rechazar**
4. **Aceptación**:
   - Aceptar si pila = `[$]` y entrada = `[$]`

5.5 | Ejemplo de parsing LL(1)

Gramática:

```
S -> aS | b
```

Cadena de entrada: `aab`

Paso	Pila	Entrada	Acción
1	`[$, S]`	`aab$`	M[S,a] = S→aS
2	`[$, a, S]`	`aab$`	Coincidir a
3	`[$, S]`	`ab$`	M[S,a] = S→aS
4	`[$, a, S]`	`ab$`	Coincidir a
5	`[$, S]`	`b$`	M[S,b] = S→b
6	`[$, b]`	`b$`	Coincidir b
7	`[$]`	`$`	**Aceptar**

Cuadro 5.1: Traza de ejecución del parser LL(1)

6 | Parser SLR(1)

El parser SLR(1) es un analizador sintáctico de tipo *shift-reduce* que trabaja de abajo hacia arriba.

6.1 | Idea principal

En lugar de predecir qué producción usar, el parser SLR(1):

- Lee símbolos de entrada y los acumula (shift)
- Cuando reconoce el lado derecho de una producción, lo reduce al lado izquierdo (reduce)
- Usa un **autómata LR(0)** para guiar las decisiones

6.2 | Items LR(0)

Un **item LR(0)** es una producción con un punto (•) que indica el progreso del parsing.
Ejemplos:

- `[S → • aS]` - No hemos visto nada aún
- `[S → a • S]` - Ya vimos 'a', esperamos ver S
- `[S → aS •]` - Vimos todo, listos para reducir

6.3 | Construcción del autómata LR(0)

6.3.1 | Operación CLOSURE

Dado un conjunto de items, `CLOSURE` agrega items relacionados.
Regla: Si tenemos `[A → α • B β]`, donde B es no terminal, agregamos todos los items `[B → • γ]` para cada producción de B.
Ejemplo:

```
CLOSURE({[S -> • AB]}) = {
    [S -> • AB],
    [A -> • a],    (porque A está después del punto)
    [A -> • e]
}
```

6.3.2 | Operación GOTO

`GOTO(I, X)` nos dice a qué estado vamos si vemos el símbolo X estando en el estado I.
Proceso:

1. Toma todos los items con X después del punto
2. Mueve el punto sobre X
3. Calcula el CLOSURE del resultado

Ejemplo:

```
I_0 = {[S -> • aB], [S -> • c]}
GOTO(I_0, a) = CLOSURE({[S -> a • B]})
             = {[S -> a • B], [B -> • b], [B -> • e]}
```

6.3.3 | Construcción del autómata

1. **Estado inicial**:
   - Crear gramática aumentada: `S' → S`
   - Estado inicial = `CLOSURE({[S' → • S]})`
2. **Generar estados**:
   - Para cada estado y cada símbolo posible
   - Calcular `GOTO(estado, símbolo)`
   - Si el resultado es un estado nuevo, agregarlo
   - Marcar la transición
3. **Repetir** hasta que no haya estados nuevos

6.4 | Tablas ACTION y GOTO

El parser SLR(1) usa dos tablas:

6.4.1 | Tabla ACTION

Para el estado `i` y terminal `a`, indica la acción:

1. **Shift j**: Mover el símbolo a la pila, ir al estado `j`
2. **Reduce A→α**: Reducir usando la producción A→α
3. **Accept**: La entrada es válida
4. **Error**: La entrada no es válida

Reglas de construcción:

- Si `[A → α • a β]` está en estado i y `GOTO(i, a) = j`:
  - `ACTION[i, a] = shift j`
- Si `[A → α •]` está en estado i:
  - Para cada `a` en `FOLLOW(A)`
  - `ACTION[i, a] = reduce A→α`
- Si `[S' → S •]` está en estado i:
  - `ACTION[i, $] = accept`

6.4.2 | Tabla GOTO

Para estado `i` y no terminal `A`:

- Si `GOTO(i, A) = j`, entonces `GOTO[i, A] = j`

6.5 | Conflictos en SLR(1)

Una gramática **NO es SLR(1)** si hay conflictos:

1. **Shift/Reduce**: Una celda tiene shift y reduce
2. **Reduce/Reduce**: Una celda tiene dos reduces diferentes

6.6 | Algoritmo de parsing SLR(1)

1. **Inicialización**:
   - Pila de estados = `[0]`
   - Pila de símbolos = `[]`
   - Entrada = cadena + `$`
2. **Bucle principal**:
   - `estado` = tope de pila de estados
   - `símbolo` = símbolo actual de entrada
   - `acción` = `ACTION[estado, símbolo]`
3. **Ejecutar acción**:
   - [a] **Shift j**:
     - Poner símbolo en pila de símbolos
     - Poner j en pila de estados
     - Avanzar entrada
   - [b] **Reduce A→α**:
     - Sacar $|\alpha|$ símbolos y estados
     - Poner A en pila de símbolos
     - `nuevo_estado` = `GOTO[tope_estados, A]`
     - Poner nuevo_estado en pila de estados
   - [c] **Accept**: **Aceptar cadena**
   - [d] **Error**: **Rechazar cadena**

6.7 | Ejemplo de parsing SLR(1)

Gramática:

```
S -> AB
A -> a
B -> b
```

Cadena: `ab`

Paso	Pila Estados	Pila Símbolos	Entrada	Acción
1	`[0]`	`[]`	`ab$`	shift 2
2	`[0,2]`	`[a]`	`b$`	reduce A→a
3	`[0,3]`	`[A]`	`b$`	shift 4
4	`[0,3,4]`	`[A,b]`	`$`	reduce B→b
5	`[0,3,5]`	`[A,B]`	`$`	reduce S→AB
6	`[0,1]`	`[S]`	`$`	**accept**

Cuadro 6.1: Traza de ejecución del parser SLR(1)

7 | Flujo de Ejecución del Programa

Ahora que entendemos todos los componentes, veamos cómo trabajan juntos.

7.1 | Punto de entrada (main.py)

El archivo `main.py` es muy simple:

1. Configura el path para importar los módulos
2. Llama a la función `run()` de `cli.py`

7.2 | Interfaz de línea de comandos (cli.py)

La función `run()` ejecuta el flujo principal:

7.2.1 | Paso 1: Leer la gramática

- Lee de la entrada estándar
- Espera el formato: primera línea = número de producciones, luego las producciones
- Llama a `parse_grammar()` para convertir el texto en un objeto Grammar

7.2.2 | Paso 2: Calcular conjuntos FIRST y FOLLOW

- `first_sets = compute_first_sets(grammar)`
- `follow_sets = compute_follow_sets(grammar, first_sets)`
- Estos conjuntos se usarán para construir los parsers

7.2.3 | Paso 3: Intentar construir parser LL(1)

```
try:
    ll1_parser = LL1Parser(grammar, first_sets, follow_sets)
except NotLL1Exception:
    ll1_parser = None
```

- Si se construye exitosamente: la gramática es LL(1)
- Si lanza excepción: la gramática NO es LL(1)

7.2.4 | Paso 4: Intentar construir parser SLR(1)

```
try:
    slr1_parser = SLR1Parser(grammar, first_sets, follow_sets)
except NotSLR1Exception:
    slr1_parser = None
```

Similar al paso anterior pero para SLR(1).

7.2.5 | Paso 5: Clasificar y actuar

Hay 4 casos posibles:

1. **Ambos parsers exitosos** (LL(1) Y SLR(1)):
   - Entrar en modo interactivo
   - El usuario elige qué parser usar:
     - `T` o `t`: Usar LL(1)
     - `B` o `b`: Usar SLR(1)
     - `Q` o `q`: Salir
   - Leer cadenas y validarlas con el parser elegido
2. **Solo LL(1)**:
   - Imprimir: "Grammar is LL(1)."
   - Leer cadenas y validarlas con el parser LL(1)
3. **Solo SLR(1)**:
   - Imprimir: "Grammar is SLR(1)."
   - Leer cadenas y validarlas con el parser SLR(1)
4. **Ninguno**:
   - Imprimir: "Grammar is neither LL(1) nor SLR(1)."
   - Terminar el programa

7.2.6 | Paso 6: Validar cadenas

La función `parse_strings_until_empty()` lee cadenas hasta encontrar una línea vacía:

- Para cada cadena, llama a `parser.parse(cadena)`
- Si retorna `True`: imprime "yes"
- Si retorna `False`: imprime "no"

8 | Ejemplos Completos

Veamos ejemplos reales de cómo funciona el programa.

8.1 | Ejemplo 1: Gramática solo SLR(1)

Entrada:

```
3
S -> S+T T
T -> T*F F
F -> (S) i
i+i
(i)
(i+i)*i)
```

Proceso:

1. Parser lee la gramática (expresiones aritméticas)
2. Calcula FIRST y FOLLOW
3. Intenta LL(1): **Falla** (tiene recursión izquierda)
4. Intenta SLR(1): **Éxito**
5. Imprime: "Grammar is SLR(1)."
6. Procesa las cadenas:
   - `i+i`: yes (válida)
   - `(i)`: yes (válida)
   - `(i+i)*i)`: no (paréntesis mal balanceados)

8.2 | Ejemplo 2: Gramática ambos tipos

Entrada:

```
3
S -> AB
A -> aA d
B -> bBc e
T
d
adbc
a

Q
```

Proceso:

1. Parser lee la gramática

2. Calcula FIRST y FOLLOW
3. Intenta LL(1): **Éxito**
4. Intenta SLR(1): **Éxito**
5. Entra en modo interactivo
6. Usuario presiona `T` (elegir LL(1))
7. Procesa cadenas con LL(1):
   - `d`: yes
   - `adbc`: yes
   - `a`: no
8. Usuario presiona `Q` (salir)

9 | Consideraciones de Implementación

9.1 | Eficiencia

- **Conjuntos FIRST/FOLLOW**: Iteración de punto fijo puede requerir múltiples pasadas
- **Autómata LR(0)**: En el peor caso, número de estados exponencial
- **Parsing**: Ambos parsers son lineales en el tamaño de la entrada

9.2 | Manejo de errores

- **Gramática inválida**: Se captura con `ValueError`
- **Conflictos LL(1)**: Se lanza `NotLL1Exception`
- **Conflictos SLR(1)**: Se lanza `NotSLR1Exception`
- **Entrada malformada**: Se valida y rechaza apropiadamente

9.3 | Estructuras de datos

- **Símbolos**: Hashables para usar en conjuntos y diccionarios
- **Items LR(0)**: También hashables para el autómata
- **Estados**: Conjuntos congelados (frozenset) para inmutabilidad
- **Tablas**: Diccionarios con tuplas como claves

10 | Resumen

Este proyecto implementa un sistema completo de análisis sintáctico que:

1. **Lee** gramáticas libres de contexto en formato texto
2. **Calcula** conjuntos FIRST y FOLLOW necesarios para el análisis
3. **Construye** parsers LL(1) (top-down) y SLR(1) (bottom-up)
4. **Clasifica** automáticamente la gramática
5. **Valida** cadenas de entrada según las reglas de la gramática

La arquitectura modular permite entender cada componente de forma independiente, mientras que el flujo de ejecución muestra cómo se integran para resolver el problema completo del análisis sintáctico.

11 | Ejemplo Paso a Paso: Walkthrough Completo

En esta sección vamos a seguir la ejecución del programa con un ejemplo concreto, como si estuviéramos haciendo debugging. Veremos exactamente qué pasa en cada paso del código.

11.1 | Ejemplo Seleccionado

Vamos a usar esta gramática simple:

```
3
S -> AB
A -> aA d
B -> bBc e
```

Y vamos a validar la cadena: `adbc`

11.2 | Paso 1: Lectura de la Entrada (cli.py)

11.2.1 | Función read_grammar_input()

El programa comienza leyendo la entrada estándar línea por línea:

```
def read_grammar_input():
    lines = []
    first_line = input().strip()  # Lee "3"
    lines.append(first_line)
    n = int(first_line)            # n = 3

    for _ in range(n):             # Lee 3 líneas
        line = input()
        lines.append(line)
```

Estado después de este paso:

```
lines = [
    "3",
    "S -> AB",
    "A -> aA d",
    "B -> bBc e"
]
```

11.3 | Paso 2: Parseo de la Gramática (grammar.py)

11.3.1 | Función parse_grammar(lines)

Ahora el programa convierte el texto en objetos Python:

1. **Lee el número de producciones:**

   ```
   n_str = lines[0].strip()  # "3"
   n = int(n_str)             # 3
   ```

2. **Procesa cada línea de producción:**

   Línea 1: "S ->AB"

   ```
   def parse_production_line("S -> AB"):
       parts = "S -> AB".split(" -> ")
       # parts = ["S", "AB"]

       lhs_str = "S"
   ```

```
            rhs_str = "AB"

            lhs = char_to_symbol('S')  # Symbol('S', nonterminal)

            alternatives = "AB".split()  # ["AB"]

            # Para "AB":
            rhs = string_to_symbols("AB")
            # rhs = [Symbol('A'), Symbol('B')]

            return [Production(S, [A, B])]
```

Línea 2: "A ->aA d"

```
    alternatives = "aA d".split()  # ["aA", "d"]

    # Producción 1: A -> aA
    rhs = [Symbol('a', terminal), Symbol('A', nonterminal)]

    # Producción 2: A -> d
    rhs = [Symbol('d', terminal)]
```

Línea 3: "B ->bBc e"

```
    alternatives = "bBc e".split()  # ["bBc", "e"]

    # Producción 1: B -> bBc
    rhs = [Symbol('b'), Symbol('B'), Symbol('c')]

    # Producción 2: B -> e (epsilon)
    rhs = [Symbol('e', is_epsilon=True)]
```

Estado después de este paso:

```
Grammar:
  productions = [
    S -> AB,
    A -> aA,
    A -> d,
    B -> bBc,
    B -> e (epsilon)
  ]

  nonterminals = {S, A, B}
  terminals = {a, b, c, d}
  start_symbol = S

  production_map = {
    S: [S -> AB],
    A: [A -> aA, A -> d],
    B: [B -> bBc, B -> e]
  }
```

11.4 | Paso 3: Cálculo de Conjuntos FIRST (first_follow.py)

11.4.1 | Función compute_first_sets(grammar)

Inicialización:

```
# Paso 1: FIRST de terminales
FIRST(a) = {a}
FIRST(b) = {b}
FIRST(c) = {c}
FIRST(d) = {d}
FIRST(epsilon) = {epsilon}
FIRST($) = {$}

# Paso 2: FIRST de no terminales (vacíos al inicio)
FIRST(S) = {}
FIRST(A) = {}
FIRST(B) = {}
```

Iteración 1:

```
Procesando: S -> AB
  RHS = [A, B]
  compute_first_of_string([A, B], first_sets):
    - FIRST(A) está vacío aún, pero A es el primero
    - result = {} (todavía)
  FIRST(S) = {} (sin cambios)

Procesando: A -> aA
  RHS = [a, A]
  compute_first_of_string([a, A], first_sets):
    - Primer símbolo es 'a' (terminal)
    - FIRST(a) = {a}
    - 'a' no es epsilon, así que paramos
    - result = {a}
  FIRST(A) = {} U {a} = {a}

Procesando: A -> d
  RHS = [d]
  compute_first_of_string([d], first_sets):
    - result = {d}
  FIRST(A) = {a} U {d} = {a, d}

Procesando: B -> bBc
  RHS = [b, B, c]
  compute_first_of_string([b, B, c], first_sets):
    - Primer símbolo es 'b' (terminal)
    - result = {b}
  FIRST(B) = {} U {b} = {b}

Procesando: B -> e (epsilon)
  RHS = [epsilon]
  compute_first_of_string([epsilon], first_sets):
    - result = {epsilon}
  FIRST(B) = {b} U {epsilon} = {b, epsilon}
```

Hubo cambios, así que continuamos...
Iteración 2:

```
Procesando: S -> AB
  RHS = [A, B]
  compute_first_of_string([A, B], first_sets):
    - FIRST(A) = {a, d}
    - Agregamos {a, d} - {epsilon} = {a, d}
    - epsilon in FIRST(A)? NO
```

```
    - Paramos (no seguimos a B)
    - result = {a, d}
  FIRST(S) = {} U {a, d} = {a, d}

Procesando: A -> aA (sin cambios)
Procesando: A -> d (sin cambios)
Procesando: B -> bBc (sin cambios)
Procesando: B -> e (sin cambios)
```

Hubo cambios, seguimos...
Iteración 3:

```
Procesando todas las producciones: sin cambios
```

¡Convergencia! El algoritmo termina.
Resultado Final:

```
FIRST(S) = {a, d}
FIRST(A) = {a, d}
FIRST(B) = {b, epsilon}
FIRST(a) = {a}
FIRST(b) = {b}
FIRST(c) = {c}
FIRST(d) = {d}
```

11.5 | Paso 4: Cálculo de Conjuntos FOLLOW (first_follow.py)

11.5.1 | Función compute_follow_sets(grammar, first_sets)

Inicialización:

```
FOLLOW(S) = {}
FOLLOW(A) = {}
FOLLOW(B) = {}

# Agregar $ al símbolo inicial
FOLLOW(S) = {$}
```

Iteración 1:

```
Procesando: S -> AB
  Para cada símbolo en RHS:

  Posición 0: símbolo = A
    beta = [B] (lo que sigue)
    FIRST(beta) = FIRST([B]) = FIRST(B) = {b, epsilon}

    Agregar FIRST(beta) - {epsilon} a FOLLOW(A):
    FOLLOW(A) = {} U {b} = {b}

    epsilon in FIRST(beta)? SÍ
    Agregar FOLLOW(S) a FOLLOW(A):
    FOLLOW(A) = {b} U {$} = {b, $}

  Posición 1: símbolo = B
    beta = [] (vacío, está al final)
    FIRST(beta) = {epsilon}

    FOLLOW(B) = {} U ({epsilon} - {epsilon}) = {}
```

```
    beta está vacío, agregar FOLLOW(S):
    FOLLOW(B) = {} U {$} = {$}

Procesando: A -> aA
  Posición 0: símbolo = a (terminal, saltamos)

  Posición 1: símbolo = A
    beta = [] (vacío)

    Agregar FOLLOW(A) a FOLLOW(A):
    FOLLOW(A) = {b, $} U {b, $} = {b, $} (sin cambios)

Procesando: A -> d
  Posición 0: símbolo = d (terminal, saltamos)

Procesando: B -> bBc
  Posición 0: símbolo = b (terminal, saltamos)

  Posición 1: símbolo = B
    beta = [c]
    FIRST(beta) = FIRST(c) = {c}

    Agregar {c} - {epsilon} a FOLLOW(B):
    FOLLOW(B) = {$} U {c} = {$, c}

    epsilon in FIRST(beta)? NO, no agregamos FOLLOW(B)

  Posición 2: símbolo = c (terminal, saltamos)

Procesando: B -> e (epsilon)
  RHS solo tiene epsilon, saltamos
```

Hubo cambios, continuamos...
Iteración 2:

```
Procesando todas las producciones: sin cambios
```

¡Convergencia!
Resultado Final:

```
FOLLOW(S) = {$}
FOLLOW(A) = {b, $}
FOLLOW(B) = {c, $}
```

11.6 | Paso 5: Construcción del Parser LL(1) (ll1.py)

11.6.1 | Función _build_table()

Ahora construimos la tabla de análisis sintáctico M[A, a]:

```
Procesando: S -> AB
  RHS = [A, B]
  FIRST([A, B]) = FIRST(A) = {a, d}

  Para cada símbolo en FIRST - {epsilon}:
    M[S, a] = S -> AB
    M[S, d] = S -> AB

  epsilon in FIRST([A, B])? NO
```

```
Procesando: A -> aA
  RHS = [a, A]
  FIRST([a, A]) = {a}

  M[A, a] = A -> aA

  epsilon in FIRST? NO

Procesando: A -> d
  RHS = [d]
  FIRST([d]) = {d}

  M[A, d] = A -> d

  epsilon in FIRST? NO

Procesando: B -> bBc
  RHS = [b, B, c]
  FIRST([b, B, c]) = {b}

  M[B, b] = B -> bBc

  epsilon in FIRST? NO

Procesando: B -> e (epsilon)
  RHS = [epsilon]
  FIRST([epsilon]) = {epsilon}

  FIRST no tiene terminales (solo epsilon)

  epsilon in FIRST? SÍ
  Para cada símbolo en FOLLOW(B) = {c, $}:
    M[B, c] = B -> e
    M[B, $] = B -> e
```

Tabla LL(1) Completa:

No Terminal	a	b	c	d	$
S	S→AB	-	-	S→AB	-
A	A→aA	-	-	A→d	-
B	-	B→bBc	B→e	-	B→e

Cuadro 11.1: Tabla LL(1) construida

¿Hay conflictos? NO. Cada celda tiene máximo una producción.
Conclusión: La gramática ES LL(1)

11.7 | Paso 6: Construcción del Parser SLR(1) (slr1.py)

11.7.1 | Gramática Aumentada

Primero creamos la gramática aumentada:

```
S' -> S
S -> AB
A -> aA | d
B -> bBc | e
```

11.7.2 | Construcción del Autómata LR(0)

Estado 0: CLOSURE({[S' → • S]})

```
Items iniciales:
  [S' -> • S]

Aplicar CLOSURE:
  [S' -> • S] tiene S después del punto
  Agregar producciones de S:
    [S -> • AB]

  [S -> • AB] tiene A después del punto
  Agregar producciones de A:
    [A -> • aA]
    [A -> • d]

Estado 0 = {
  [S' -> • S],
  [S -> • AB],
  [A -> • aA],
  [A -> • d]
}
```

Transiciones desde Estado 0:

```
GOTO(0, S):
  Items con S después del punto: [S' -> • S]
  Mover punto: [S' -> S •]
  CLOSURE([S' -> S •]) = {[S' -> S •]}

  Estado 1 = {[S' -> S •]}

GOTO(0, A):
  Items con A después del punto: [S -> • AB]
  Mover punto: [S -> A • B]
  CLOSURE:
    [S -> A • B] tiene B después del punto
    Agregar: [B -> • bBc], [B -> • e]

  Estado 2 = {
    [S -> A • B],
    [B -> • bBc],
    [B -> • e]
  }

GOTO(0, a):
  Items: [A -> • aA]
  Mover: [A -> a • A]
  CLOSURE:
    Agregar: [A -> • aA], [A -> • d]

  Estado 3 = {
    [A -> a • A],
    [A -> • aA],
    [A -> • d]
  }

GOTO(0, d):
  Items: [A -> • d]
```

```
  Mover: [A -> d •]

  Estado 4 = {[A -> d •]}
```

Continuamos desde Estado 2:

```
GOTO(2, B):
  Items: [S -> A • B]
  Mover: [S -> AB •]

  Estado 5 = {[S -> AB •]}

GOTO(2, b):
  Items: [B -> • bBc]
  Mover: [B -> b • Bc]
  CLOSURE:
    Agregar: [B -> • bBc], [B -> • e]

  Estado 6 = {
    [B -> b • Bc],
    [B -> • bBc],
    [B -> • e]
  }
```

Continuamos desde Estado 3:

```
GOTO(3, A):
  Items: [A -> a • A]
  Mover: [A -> aA •]

  Estado 7 = {[A -> aA •]}

GOTO(3, a): va al Estado 3 (mismo estado)
GOTO(3, d): va al Estado 4
```

Continuamos desde Estado 6:

```
GOTO(6, B):
  Items: [B -> b • Bc]
  Mover: [B -> bB • c]

  Estado 8 = {[B -> bB • c]}

GOTO(6, b): va al Estado 6 (mismo)

GOTO(6, c):
  Items: [B -> bB • c]
  (esto se hace desde Estado 8)
```

Desde Estado 8:

```
GOTO(8, c):
  Items: [B -> bB • c]
  Mover: [B -> bBc •]

  Estado 9 = {[B -> bBc •]}
```

Autómata LR(0) Completo:

```
Estados: 0, 1, 2, 3, 4, 5, 6, 7, 8, 9

Transiciones:
  (0, S) -> 1
  (0, A) -> 2
  (0, a) -> 3
  (0, d) -> 4
  (2, B) -> 5
  (2, b) -> 6
  (3, A) -> 7
  (3, a) -> 3
  (3, d) -> 4
  (6, B) -> 8
  (6, b) -> 6
  (8, c) -> 9
```

11.7.3 | Construcción de Tablas ACTION y GOTO

Tabla ACTION:

```
Estado 0:
  [A -> • aA]: shift en 'a' -> ACTION[0, a] = shift 3
  [A -> • d]: shift en 'd' -> ACTION[0, d] = shift 4

Estado 1:
  [S' -> S •]: accept -> ACTION[1, $] = accept

Estado 2:
  [B -> • bBc]: shift en 'b' -> ACTION[2, b] = shift 6

Estado 3:
  [A -> • aA]: shift en 'a' -> ACTION[3, a] = shift 3
  [A -> • d]: shift en 'd' -> ACTION[3, d] = shift 4

Estado 4:
  [A -> d •]: reduce A->d en FOLLOW(A) = {b, $}
  ACTION[4, b] = reduce A->d
  ACTION[4, $] = reduce A->d

Estado 5:
  [S -> AB •]: reduce S->AB en FOLLOW(S) = {$}
  ACTION[5, $] = reduce S->AB

Estado 6:
  [B -> • bBc]: shift en 'b' -> ACTION[6, b] = shift 6

Estado 7:
  [A -> aA •]: reduce A->aA en FOLLOW(A) = {b, $}
  ACTION[7, b] = reduce A->aA
  ACTION[7, $] = reduce A->aA

Estado 8:
  [B -> bB • c]: shift en 'c' -> ACTION[8, c] = shift 9

Estado 9:
  [B -> bBc •]: reduce B->bBc en FOLLOW(B) = {c, $}
  ACTION[9, c] = reduce B->bBc
  ACTION[9, $] = reduce B->bBc
```

Tabla GOTO:

```
GOTO[0, S] = 1
GOTO[0, A] = 2
GOTO[2, B] = 5
GOTO[3, A] = 7
GOTO[6, B] = 8
```

¿Hay conflictos? NO. Cada celda tiene solo una acción.
Conclusión: La gramática ES SLR(1)

11.8 | Paso 7: Decisión del Programa (cli.py)

```
ll1_parser = LL1Parser(...)  # ÉXITO
slr1_parser = SLR1Parser(...) # ÉXITO

# Ambos parsers se construyeron exitosamente
if ll1_parser and slr1_parser:
    interactive_mode(ll1_parser, slr1_parser)
```

El programa entra en modo interactivo y espera que el usuario elija:

- `T` o `t`: Usar LL(1)
- `B` o `b`: Usar SLR(1)
- `Q` o `q`: Salir

Supongamos que el usuario elige `T` (LL(1)).

11.9 | Paso 8: Parsing de la Cadena "adbc" con LL(1)

11.9.1 | Inicialización

```
input_string = "adbc"
input_symbols = [a, d, b, c, $]
stack = [$, S]
input_idx = 0
```

11.9.2 | Traza de Ejecución

Paso 1:

```
Stack: [$, S]
Input: adbc$
input_idx: 0 (apuntando a 'a')

top = S (no terminal)
current = a

Buscar M[S, a] = S -> AB
Pop S
Push B, A (en orden inverso)

Stack: [$, B, A]
```

Paso 2:

```
Stack: [$, B, A]
Input: adbc$
input_idx: 0 (apuntando a 'a')

top = A (no terminal)
current = a
```

```
Buscar M[A, a] = A -> aA
Pop A
Push A, a (en orden inverso)

Stack: [$, B, A, a]
```

Paso 3:

```
Stack: [$, B, A, a]
Input: adbc$
input_idx: 0 (apuntando a 'a')

top = a (terminal)
current = a

¡Coinciden! Pop a, avanzar input
input_idx = 1

Stack: [$, B, A]
```

Paso 4:

```
Stack: [$, B, A]
Input: dbc$
input_idx: 1 (apuntando a 'd')

top = A (no terminal)
current = d

Buscar M[A, d] = A -> d
Pop A
Push d

Stack: [$, B, d]
```

Paso 5:

```
Stack: [$, B, d]
Input: dbc$
input_idx: 1 (apuntando a 'd')

top = d (terminal)
current = d

¡Coinciden! Pop d, avanzar input
input_idx = 2

Stack: [$, B]
```

Paso 6:

```
Stack: [$, B]
Input: bc$
input_idx: 2 (apuntando a 'b')

top = B (no terminal)
current = b

Buscar M[B, b] = B -> bBc
```

```
Pop B
Push c, B, b (en orden inverso)

Stack: [$, c, B, b]
```

Paso 7:

```
Stack: [$, c, B, b]
Input: bc$
input_idx: 2 (apuntando a 'b')

top = b (terminal)
current = b

¡Coinciden! Pop b, avanzar input
input_idx = 3

Stack: [$, c, B]
```

Paso 8:

```
Stack: [$, c, B]
Input: c$
input_idx: 3 (apuntando a 'c')

top = B (no terminal)
current = c

Buscar M[B, c] = B -> e (epsilon)
Pop B
Push epsilon (pero epsilon se ignora)

Stack: [$, c]
```

Paso 9:

```
Stack: [$, c]
Input: c$
input_idx: 3 (apuntando a 'c')

top = c (terminal)
current = c

¡Coinciden! Pop c, avanzar input
input_idx = 4

Stack: [$]
```

Paso 10:

```
Stack: [$]
Input: $
input_idx: 4 (apuntando a '$')

top = $ (end marker)
current = $

¡Coinciden! Pop $, avanzar input
input_idx = 5

Stack: []
```

Resultado:

```
Stack vacío y entrada consumida completamente
¡ACEPTAR!

El programa imprime: "yes"
```

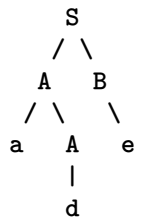
11.10 | Resumen de la Traza Completa

Paso	Pila	Entrada	Acción	Producción
1	[$, S]	adbc$	expand	S→AB
2	[$, B, A]	adbc$	expand	A→aA
3	[$, B, A, a]	adbc$	match	-
4	[$, B, A]	dbc$	expand	A→d
5	[$, B, d]	dbc$	match	-
6	[$, B]	bc$	expand	B→bBc
7	[$, c, B, b]	bc$	match	-
8	[$, c, B]	c$	expand	B→e
9	[$, c]	c$	match	-
10	[$]	$	match	-
11	[]	-	**ACCEPT**	-

Cuadro 11.2: Traza completa del parsing LL(1) para “adbc”

11.11 | Árbol de Derivación

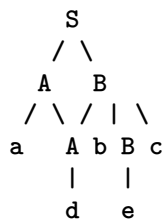
El proceso de parsing construye implícitamente este árbol de derivación:



Que se puede leer como:

- S deriva AB
- A deriva aA, que deriva ad
- B deriva e (epsilon)
- Resultado final: a + d + (nada) = `ad`

Pero espera... ¿dónde están la ’b’ y la ’c’?
¡Error de análisis! Revisemos...
Ah, en el paso 6 expandimos B → bBc, lo que agrega ’b’ y ’c’ como terminales a verificar. Entonces:



Cadena generada: a + d + b + (nada) + c = `adbc`

11.12 | Puntos Clave del Ejemplo

1. **Epsilon es especial**: La producción B → e (epsilon) no agrega nada a la cadena, pero es importante para el análisis.
2. **FOLLOW es crucial**: Sin FOLLOW, no sabríamos cuándo usar B → e. El símbolo 'c' en el input está en FOLLOW(B), por eso sabemos que podemos reducir B a epsilon.
3. **La pila trabaja al revés**: Cuando expandimos una producción, ponemos el lado derecho en orden inverso en la pila para que el primer símbolo quede arriba.
4. **LL(1) es predictivo**: En cada paso sabemos exactamente qué hacer solo mirando el tope de la pila y el siguiente símbolo de entrada. No hay que retroceder ni probar opciones.
5. **La gramática acepta múltiples cadenas**: Por ejemplo, también aceptaría:
   - "d" (A→d, B→e)
   - "aaadbc" (A→aA múltiples veces)
   - "dbbcc" (A→d, B→bBc con B→bBc dentro)

11.13 | Comparación: ¿Cómo sería con SLR(1)?

Si hubiéramos elegido el parser SLR(1), el proceso sería diferente:

- **Bottom-up**: Empezaríamos leyendo símbolos y haciendo *shift*
- **Reduce**: Cuando reconocemos el lado derecho de una producción, lo reducimos
- **Estados**: Usamos una pila de estados en vez de símbolos de gramática
- **Resultado**: El mismo, pero el camino es opuesto

LL(1) construye el árbol de arriba hacia abajo (de la raíz a las hojas). SLR(1) construye el árbol de abajo hacia arriba (de las hojas a la raíz).

11.14 | Conclusión del Walkthrough

Este ejemplo muestra cómo el programa:

1. Lee y parsea la entrada
2. Construye estructuras de datos (Grammar, Symbol, Production)
3. Calcula conjuntos FIRST y FOLLOW mediante iteración de punto fijo
4. Construye las tablas de parsing (LL(1) y SLR(1))
5. Valida cadenas de entrada paso a paso

Cada componente del código tiene un propósito específico y todos trabajan juntos para realizar el análisis sintáctico completo.

12 | Referencias