Lenguajes Formales

Proyecto Final

Estudiantes: Hever Andre Alfonso Jimenez y Moises Vergara Garces

Profesor: Adolfo Andres Castro Sanchez

Mayo 2025

Introducción

Durante el desarrollo del proyecto, uno de los mayores retos fue entender y aplicar correctamente los algoritmos para calcular los conjuntos **FIRST** y **FOLLOW**, así como implementar un **parser LL(1)** basado en tabla. Aunque las definiciones teóricas son claras, llevarlas a código y aplicarlas a gramáticas reales nos generó confusión en algunos puntos.

Antes de implementar cualquier parte del código, nos enfocamos en revisar los conceptos en la práctica. Para esto, nos apoyamos en asesorías con el monitor del curso y repasamos los ejemplos presentados en el libro base del curso (Aho et al.). Esto nos permitió identificar con mayor claridad los pasos necesarios para implementar correctamente los algoritmos.

Cálculo del conjunto FIRST

El conjunto **FIRST**(α) de una cadena α es el conjunto de terminales que pueden aparecer como primer símbolo en alguna derivación de α. Si α puede derivar ε, entonces $\varepsilon \in$ FIRST(α).

Regla práctica:

$$\text{FIRST}(\alpha\beta) = (\text{FIRST}(\alpha) \setminus \{\varepsilon\}) \cup \begin{cases} \text{FIRST}(\beta), & \text{si } \varepsilon \in \text{FIRST}(\alpha) \\ \emptyset, & \text{si } \varepsilon \notin \text{FIRST}(\alpha) \end{cases}$$

Implementamos esto con la función `first_of_string` que recorre los símbolos de la cadena y construye el conjunto correspondiente, manejando correctamente la aparición de ε.

Problema: Inicialmente confundíamos cuándo detenernos al recorrer una cadena y cómo tratar múltiples ε. *Solución:* Agregamos pruebas y consultamos con el monitor para entender cómo propagar ε adecuadamente en cadenas compuestas.

Cálculo del conjunto FOLLOW

El conjunto **FOLLOW**(A) de un no terminal A es el conjunto de terminales que pueden aparecer justo después de A en alguna derivación. También se incluye el símbolo \$ en el FOLLOW del símbolo inicial.

Regla práctica: Si hay una producción de la forma:

$$A \to \alpha B \beta$$

Entonces:

- Agregar FIRST(β) $\setminus \{\varepsilon\}$ a FOLLOW(B)
- Si $\varepsilon \in$ FIRST(β) o $\beta = \varepsilon$, entonces agregar FOLLOW(A) a FOLLOW(B)

Problema: Al principio, no propagábamos correctamente el FOLLOW cuando β era vacío. *Solución:* Iteramos sobre las producciones varias veces y ajustamos el algoritmo para garantizar que se acumularan correctamente los FOLLOW.

Chequeo de propiedad LL(1)

Para que una gramática sea LL(1), sus producciones deben cumplir que:

- Los conjuntos FIRST de cada producción de un no terminal deben ser disjuntos.
- Si una producción puede derivar ε, su conjunto FIRST debe ser disjunto con el FOLLOW del no terminal.

Regla práctica:

- Para cada par de producciones $A \to \alpha_i$ y $A \to \alpha_j$, verificar que $FIRST(\alpha_i) \cap FIRST(\alpha_j) = \emptyset$
- Si $\varepsilon \in FIRST(\alpha_i)$, verificar que $FIRST(\alpha_i) \cap FOLLOW(A) = \emptyset$

Problema: Algunas gramáticas ambiguas pasaban como LL(1) por errores en la intersección de conjuntos. *Solución:* Añadimos validaciones explícitas entre pares de producciones y depuramos con casos extremos del libro.

Construcción de la tabla LL(1)

La tabla LL(1) indica qué producción usar al ver un par (no terminal, símbolo de entrada). Para construirla:

- Para cada producción $A \to \alpha$, para cada $a \in$ FIRST(α) $- \{\varepsilon\}$, ponemos $A \to \alpha$ en `table[A][a]`.
- Si $\varepsilon \in$ FIRST(α), agregamos la producción en todas las entradas $b \in$ FOLLOW(A).

Problema: Las entradas de la tabla se sobrescribían al haber conflictos. *Solución:* Añadimos verificación previa y reportamos errores si una celda ya estaba ocupada por otra producción.

Parsing con tabla LL(1)

Una vez construida la tabla LL(1), el parsing se hace con una pila. Se inicializa con \$ y el símbolo inicial. Luego:

- Si el tope de la pila es un terminal y coincide con el símbolo de entrada, se consume.
- Si es un no terminal A, se consulta `table[A][a]` y se reemplaza por el lado derecho (en orden inverso).
- Si no hay entrada válida, se rechaza.

Problema: Al invertir las producciones, algunas no se cargaban correctamente. *Solución:* Usamos `reversed()` para mantener el orden correcto y añadimos impresión de pila para depuración.

Construcción de la tabla SLR(1)

El enfoque SLR(1) utiliza conjuntos de items LR(0) para construir una tabla de análisis más poderosa. Primero, generamos los **items** y los agrupamos en **estados** (con cierre).

Regla práctica:

- Si el punto en un item puede avanzar con un símbolo, creamos una transición.
- Si el punto está al final de la producción y $A \to \alpha$ está completa, para cada $a \in FOLLOW(A)$ ponemos una reducción.

Problema: Algunos items se duplicaban por error en el cierre. *Solución:* Cambiamos listas por `set()` y verificamos que el cierre no duplicara tuplas ya presentes.

Construcción del parser SLR(1)

Luego de construir los estados y transiciones, generamos la tabla `ACTION` y `GOTO`:

- Si hay transición por un terminal, es `shift`.
- Si el punto está al final, y el item es $A \to \alpha$, y $a \in FOLLOW(A)$, se aplica `reduce`.
- Si es la producción inicial y está completa con \$, es `accept`.

Problema: En gramáticas ambiguas, aparecían conflictos de shift/reduce. *Solución:* Separamos las funciones LL(1) y SLR(1), mostrando el mensaje "no es LL(1)" o "no es SLR(1)" según aplicara.

Integración del sistema completo

El sistema se integra con una función principal `main()` que:

- Lee la gramática y verifica productividad.
- Decide si es LL(1), SLR(1) o ambos.
- Permite al usuario escoger el tipo de parser para cada cadena de prueba.

Problema: Inicialmente se intentaban aplicar ambos analizadores sin distinción. *Solución:* Añadimos menús interactivos para seleccionar entre LL(1) y SLR(1), y organizamos la ejecución en etapas claras.

Conclusión

En este proyecto abordamos paso a paso la construcción de un analizador sintáctico predictivo y un parser más general tipo SLR(1). Comenzamos implementando los conjuntos **FIRST** y **FOLLOW**, luego validamos si una gramática era **LL(1)**, construimos la **tabla LL(1)** y desarrollamos el **parser** correspondiente. Extendimos nuestro sistema con el cálculo del **cierre** y los **estados LR(0)**, construimos la **tabla SLR(1)** y su simulador basado en pila, y finalizamos integrando todo en la función `main()` que permite operar el sistema completo de manera interactiva.

Cada etapa presentó desafíos conceptuales y técnicos, pero al trabajar con ejemplos concretos, consultas con el monitor y validaciones sistemáticas, logramos implementar un sistema funcional que refuerza nuestro entendimiento sobre el procesamiento de lenguajes formales y su aplicación práctica en compiladores.