Universidad de Ingeniería y Tecnología

Compiladores (CS3025) - Teoría 2 - 2025 - 2

PARSER LR(1)

Analizador Sintáctico LR(1)

con Interfaz Web y API REST

Profesor:
Yarasca Moscol, Julio

Estudiantes:

Nombres y Apellidos	Código Estudiante
Joel David Miguel Fernández	202310186
Luis Fabiano Rivadeneira Fernández	202310074
Luis Fernando Lopez Chambi	202310244

Lima – Perú
Octubre, 2025

Índice

1. Introducción

El análisis sintáctico es una etapa fundamental en el proceso de compilación de lenguajes de programación. El presente proyecto implementa un **analizador sintáctico LR(1)** completo, diseñado para construir y validar gramáticas libres de contexto mediante el método ascendente (bottom-up).

1.1. Objetivos

Los objetivos principales del proyecto son:

- Implementar un generador de tablas LR(1) con construcción de la colección canónica de items.
- Desarrollar un parser shift-reduce con trazabilidad completa de cada paso del análisis.
- Proporcionar una API REST para integración con interfaces de usuario.
- Crear una interfaz web interactiva que permita visualizar estados, tablas y árboles de derivación.

1.2. Alcance

El sistema desarrollado permite:

1. Cargar gramáticas en formato textual estándar (producciones con flecha $\rightarrow$).
2. Generar automáticamente la colección canónica de estados LR(1).
3. Construir las tablas ACTION y GOTO con detección de conflictos.
4. Parsear cadenas de entrada con traza detallada paso a paso.
5. Visualizar el árbol de derivación resultante en formato JSON y ASCII.

2. Desarrollo del Proyecto

2.1. Arquitectura del Sistema

El proyecto se divide en dos componentes principales:

- **Backend (Python + FastAPI)**: Motor del analizador LR(1) y API REST.
- **Frontend (Next.js + React)**: Interfaz web interactiva para usuarios finales.

2.2. Backend - Analizador LR(1)

El backend está compuesto por módulos especializados que implementan cada fase del análisis sintáctico LR(1).

2.2.1. Módulos Principales

grammar.py: Módulo de carga y procesamiento de gramáticas.
Funcionalidades:

- Carga gramáticas desde archivos (`load_from_file(filename)`) o desde strings.
- Detecta automáticamente terminales y no terminales analizando las producciones.
- Identifica el símbolo inicial (primera producción).
- Añade automáticamente el símbolo de fin de entrada `$` a la lista de terminales.
- Expone públicamente: `terminals`, `nonTerminals`, `initialState` y `rules`.

lr1.py: Núcleo del generador LR(1).
Estructuras de datos:

- `Production`: Producción estructurada con lado izquierdo (lhs) y derecho (rhs).
- `LR1Item`: Item LR(1) que contiene producción, posición del punto (`dot`) y símbolo de lookahead.
- `LR1State`: Estado formado por un conjunto de items LR(1) y sus transiciones.

Algoritmos implementados:

- `closure(items)`: Calcula el cierre de un conjunto de items aplicando las reglas de producción.
- `goto(items, X)`: Calcula el estado destino al consumir el símbolo X desde un conjunto de items.
- `build_canonical_collection()`: Construye la colección canónica completa de estados LR(1) mediante iteración hasta alcanzar un punto fijo.
- `build_tables()`: Genera las tablas ACTION (shift/reduce/accept) y GOTO (transiciones de no terminales) con detección automática de conflictos shift-reduce y reduce-reduce.

Utilidades de visualización:

- `print_states()`: Imprime todos los estados con sus items y transiciones.
- `print_tables()`: Muestra las tablas ACTION y GOTO en formato legible.
- `print_closure_table()`: Genera una tabla detallada del cierre canónico.

lr_parser.py: Motor de parseo shift-reduce.
Características:

- Clase `LRParser` que ejecuta el análisis mediante el método `parse(tokens, collect_trace=False`
- Mantiene una pila de estados y símbolos para simular el autómata LR(1).
- Implementa las cuatro acciones fundamentales:

1. **Shift**: Empuja el token y su estado en la pila.
2. **Reduce**: Aplica una producción reduciendo símbolos de la pila.
3. **Goto**: Transición tras una reducción (registrada como paso adicional en la traza).
4. **Accept**: Finalización exitosa del análisis.

- Genera trazas estructuradas en `LRParser.last_trace` (formato JSON) con información detallada de cada paso: pila de estados, pila de símbolos, entrada restante y acción ejecutada.
- Construye el árbol de derivación almacenado en `LRParser.last_tree`, disponible en formato JSON y como string ASCII para visualización en consola.

parser.py: Parser genérico basado en tablas (módulo legado/referencia).
Utilizado como referencia para comparación con implementaciones LL(1) o análisis basado en tablas preconstruidas con IDs numéricos de símbolos.

2.2.2. API REST con FastAPI

La API expone dos endpoints RESTful que permiten la construcción de tablas y el análisis sintáctico:

Listing 1: Endpoint 1: Construcción de tablas

```
POST /build
Body: {
    "grammar": "S␣->␣C␣C\nC␣->␣c␣C\nC␣->␣d"
}
Response: {
    "initial": "S",
    "terminals": ["$", "c", "d"],
    "nonterminals": ["C", "S"],
    "rules": [...],
    "states": [...],
    "tables": { "action": {...}, "goto": {...} },
    "conflicts": []
}
```

Listing 2: Endpoint 2: Análisis sintáctico

```
POST /parse
Body: {
    "grammar": "S␣->␣C␣C\nC␣->␣c␣C\nC␣->␣d",
    "input": "c␣d␣d␣$"
}
Response: {
    "accepted": true,
    "trace": [...],
    "tree": {...},
    "tree_ascii": "S\n␣␣          ␣C\n␣␣    ␣␣␣          ␣c\n..."
}
```

2.3. Frontend - Interfaz Web

Desarrollado con Next.js 14 y React, proporciona una experiencia de usuario moderna e interactiva.

2.3.1. Vista Build Grammar

Interfaz para construcción y visualización de tablas LR(1):

- Textarea para edición de gramáticas con formato de texto plano.
- Visualización expandible/colapsable de estados LR(1) con sus items y transiciones.
- Renderizado dinámico de las tablas ACTION y GOTO en formato tabular.
- Panel de conflictos que muestra errores shift-reduce o reduce-reduce si se detectan.

2.3.2. Vista Parse Input

Interfaz para análisis de cadenas:

- Campo de entrada para la cadena de tokens a analizar.
- Visualización paso a paso de la traza del parseo con actualización dinámica de la pila.
- Renderizado del árbol de derivación en formato visual jerárquico.
- Indicador visual de aceptación (verde) o rechazo (rojo) de la cadena.

3. Resultados

3.1. Construcción de Tablas LR(1)

Para la gramática de ejemplo:

```
S -> C C
C -> c C
C -> d
```

El sistema genera correctamente:

- 10 estados LR(1) (I0 a I9) con sus correspondientes items y lookaheads.
- Tabla ACTION completa con 9 acciones shift, 6 reduce y 1 accept.
- Tabla GOTO con 4 transiciones de no terminales.
- Sin conflictos detectados (gramática LR(1) válida).

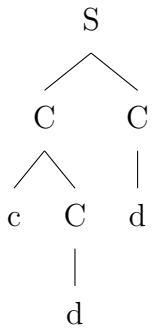
La colección canónica cubre todos los posibles puntos de análisis, garantizando que cualquier cadena válida de la gramática será correctamente reconocida.

3.2. Análisis de Cadenas

Para la entrada `c d d $`, el parser ejecuta exitosamente 12 pasos:

1. Shift del token `c` → estado 3
2. Shift del token `d` → estado 4
3. Reduce: $C \to d$
4. Goto estado 5 sobre `C`
5. Reduce: $C \to c\ C$
6. Goto estado 1 sobre `C`
7. Shift del token `d` → estado 8
8. Reduce: $C \to d$
9. Goto estado 6 sobre `C`
10. Reduce: $S \to C\ C$
11. Goto estado 2 sobre `S`
12. Accept

El árbol de derivación resultante representa correctamente la estructura jerárquica:



3.3. Integración Backend-Frontend

La arquitectura cliente-servidor mediante API REST ofrece:

- **Separación de responsabilidades**: Lógica de compilación aislada de la presentación.
- **Reutilización**: El backend puede ser consumido por múltiples clientes (web, móvil, CLI).
- **Escalabilidad**: Cada componente puede escalar independientemente según demanda.
- **Mantenibilidad**: Actualizaciones del parser no afectan al frontend y viceversa.

La comunicación mediante JSON estructurado facilita la depuración y permite el registro detallado de todas las operaciones.

4. Conclusiones

1. Se implementó exitosamente un analizador LR(1) completo que genera tablas correctas, detecta conflictos automáticamente y produce análisis sintácticos precisos para gramáticas libres de contexto.

2. La arquitectura cliente-servidor con API REST demuestra ser una solución robusta que facilita la integración, permite que múltiples interfaces consuman el mismo motor de análisis y garantiza la escalabilidad del sistema.

3. La visualización paso a paso del proceso de parsing, incluyendo la traza detallada y el árbol de derivación, proporciona una herramienta educativa invaluable para comprender el funcionamiento interno del algoritmo LR(1) y el análisis sintáctico ascendente.