Documentación del Proyecto: Analizador LR(1)

Puntos Extras Examen 2
Curso de Compiladores
Universidad de Ingeniería y Tecnología (UTEC)

Octubre 2024

Índice

1. Introducción

Este documento describe de manera detallada la implementación de un *analizador sintáctico LR(1)* desarrollado como parte de los puntos extras del Examen 2 del curso de Compiladores de la UTEC. El proyecto implementa un parser LR(1) desde cero y provee una interfaz web moderna construida con React. La aplicación es capaz de leer gramáticas libres de contexto, construir el autómata LR(1) canónico, generar la tabla de análisis `ACTION/GOTO` y analizar cadenas de entrada mostrando la traza completa del proceso. Además, incluye visualizaciones profesionales del autómata mediante `Graphviz`.

1.1. Objetivos

- Implementar un parser LR(1) canónico desde cero, superando los requisitos del proyecto (que solicitaba un LALR(1)).
- Proporcionar una interfaz web intuitiva para interactuar con el parser y visualizar resultados.
- Permitir la visualización del autómata LR(1) y de la tabla de análisis de manera clara y profesional.
- Facilitar el análisis de cadenas con trazabilidad paso a paso.
- Soportar sintaxis con el operador "—" para especificar múltiples producciones en una sola línea.

2. Fundamentos teóricos

2.1. Parsers LR(1)

El término LR(1) significa **L**eft-to-right parsing con construcción de una derivación por la **R**everse rightmost derivation y utilizando un símbolo de **lookahead**. Los parsers LR(1) son capaces de analizar un superconjunto de gramáticas que las variantes SLR(1) o LALR(1); aunque generan más estados, resuelven más conflictos *shift/reduce* y *reduce/reduce*. Utilizan un *item LR(1)* de la forma $[A \to \alpha \bullet \beta, a]$, donde a es el *lookahead*. Cada item indica que se ha visto la parte α de la producción y se espera leer β con a como símbolo de anticipación.

2.2. Componentes clave

Un parser LR(1) se construye a partir de los siguientes componentes:

- **Conjuntos FIRST**. Para cada símbolo se determina el conjunto de terminales con los que puede comenzar cualquier derivación; permiten calcular *lookaheads* correctos.
- **Conjuntos FOLLOW**. Para cada no terminal se determina el conjunto de terminales que pueden aparecer inmediatamente a su derecha; se utilizan en la construcción de las reglas de reducción.
- **Items LR(1)**. Expresan el progreso en una producción y el lookahead permitido.

- **Función** `closure`. Calcula la clausura de un conjunto de items, añadiendo nuevos items cuando el punto se encuentra antes de un no terminal, propagando correctamente los *lookaheads*.
- **Función** `goto`. Calcula la transición al avanzar el punto sobre un símbolo; define las transiciones del autómata.
- **Tablas ACTION y GOTO**. Determinan, para cada estado y símbolo, si la acción es desplazar, reducir, aceptar o error, y a qué estado transicionar después de una reducción.

2.3. Algoritmo de construcción

A continuación se describe de forma resumida el algoritmo para construir un parser LR(1). La construcción utiliza un algoritmo de punto fijo para calcular los conjuntos `FIRST` y `FOLLOW`, tras lo cual se construye el autómata canónico y las tablas de análisis.

1. Calcular los conjuntos FIRST y FOLLOW para todos los símbolos de la gramática.
2. Crear una *gramática aumentada* añadiendo una producción $S' \to S$ donde S es el símbolo inicial original.
3. Construir el autómata LR(1): se toma como estado inicial `closure`$(\{[S' \to \bullet S, \$]\})$ y se generan sucesivos estados aplicando `goto` sobre todos los símbolos posibles; cada nuevo conjunto de items genera un nuevo estado.
4. Construir las tablas ACTION y GOTO: para cada estado y símbolo se asigna una acción de desplazamiento, reducción o aceptación según la posición del punto y el lookahead en los items.
5. Utilizar las tablas para analizar cadenas; se emplea una pila de estados y un apuntador de entrada, desplazando y reduciendo conforme lo indique la tabla ACTION.

3. Arquitectura del proyecto

El proyecto se divide en tres módulos principales: el backend en Python, el parser propiamente dicho (implementado en un paquete `parser/`) y el frontend en React. La tabla 1 muestra la estructura de directorios.

Directorio	**Descripción**
`backend/`	API REST con Flask
`frontend/react-app/`	Aplicación React + Vite
`parser/`	Implementación del parser LR(1)
`requirements.txt`	Dependencias Python
`README.md`	Documentación de usuario

Cuadro 1: Estructura general del proyecto.

3.1. Backend

El backend está construido con **Flask 3.0.0** y expone una API REST para construir el parser, generar visualizaciones y analizar cadenas. El archivo `app.py` define los siguientes endpoints principales:

- `/api/build_parser` (POST): recibe una gramática, construye el parser e informa sobre el autómata (número de estados y transiciones, terminales, no terminales, producciones y conjuntos FIRST/FOLLOW).
- `/api/parse_string` (POST): recibe una cadena de entrada y devuelve si es aceptada junto con la traza de análisis.
- `/api/generate_graphviz` (POST): genera una representación visual del autómata en formato `SVG` y `PNG` mediante Graphviz.
- `/api/get_parsing_table` (GET): devuelve la tabla ACTION/GOTO completa para inspección o visualización.

El backend utiliza las clases implementadas en `parser/lr1_parser.py` para procesar la gramática y analizar cadenas. Para la visualización se hace uso de `graphviz` y `networkx`, y se soporta `CORS` para permitir el consumo desde el frontend.

3.2. Parser

El núcleo del proyecto se encuentra en el paquete `parser/`. La clase central es `LR1Parser`, implementada en `lr1_parser.py`, que encapsula toda la lógica de construcción del parser LR(1). A continuación se muestran algunos de sus elementos más importantes (extracto):

```python
from dataclasses import dataclass, field
from typing import List, Set, Dict, Tuple

@dataclass
class Production:
    """Representa una producci n de la gram tica"""
    left: str
    right: List[str]
    number: int = 0

@dataclass
class LR1Item:
    """Representa un item LR(1) con lookahead"""
    production: int
    dot_position: int
    lookahead: str

class LR1Parser:
    """Parser LR(1) completo"""
    def __init__(self):
        self.grammar: List[Production] = []
        self.terminals: Set[str] = set()
        self.non_terminals: Set[str] = set()
```

```
        self.first_sets: Dict[str, Set[str]] = {}
        self.follow_sets: Dict[str, Set[str]] = {}
        self.states: List[Set[LR1Item]] = []
        self.transitions: Dict[Tuple[int,str], int] = {}
        self.action_table: Dict[Tuple[int,str], str] = {}
        self.goto_table: Dict[Tuple[int,str], int] = {}
```

Listing 1: Definición de clases y atributos principales del parser.

El método `_compute_first_sets()` implementa el cálculo de los conjuntos FIRST mediante un algoritmo de punto fijo, mientras que `_compute_follow_sets()` calcula los conjuntos FOLLOW. Posteriormente `_build_lr1_automaton()` genera los estados del autómata aplicando clausura y transiciones y `_build_parsing_table()` construye las tablas ACTION y GOTO. El método `parse_string()` emplea las tablas para analizar cadenas, manteniendo una pila de estados y una traza de pasos.

3.3. Frontend

La interfaz de usuario está construida con **React 18** y utiliza `Vite` como herramienta de construcción y servidor de desarrollo. Los componentes principales son:

- `GrammarEditor.jsx`: permite ingresar una gramática, cargar ejemplos y solicitar la construcción del parser. Utiliza un área de texto con estado interno y un botón que invoca la API REST.
- `AutomatonInfo.jsx`: muestra información resumida del autómata (número de estados, transiciones, terminales y no terminales) en tarjetas.
- `VisualizationTabs.jsx`: organiza en pestañas la visualización Graphviz y la tabla ACTION/GOTO. Al cambiar de pestaña se realizan peticiones a la API para obtener la última versión de la visualización.
- `StringParser.jsx`: permite ingresar una cadena de entrada y consultar si es aceptada, mostrando además la traza de análisis en una tabla.

A modo de ejemplo, el código de `GrammarEditor.jsx` luce así:

Listing 2: Componente para editar y construir la gramática.

```
function GrammarEditor({ onBuild, loading, error }) {
  const [grammar, setGrammar] = useState(DEFAULT_GRAMMAR);

  const handleBuild = () => {
    onBuild(grammar);
  };

  return (
    <div className="section">
      <h2>Gram tica </h2>
      <textarea
        value={grammar}
        onChange={(e) => setGrammar(e.target.value)}
        rows={10}
      />
```

```
      <button onClick={handleBuild} disabled={loading}>
        {loading ? 'Construyendo...' : 'Construir Parser'}
      </button>
      {error && <div className="alert alert-error">{error}</div>}
    </div>
  );
}
```

4. Instalación y uso de la aplicación

4.1. Requisitos previos

Para ejecutar la aplicación se requiere tener instalado lo siguiente:

- Python 3.9 o superior
- Node.js 18 o superior
- npm 9 o superior
- Graphviz (instalado en el sistema)

4.2. Instalación de Graphviz

Graphviz debe instalarse en el sistema operativo antes de ejecutar la aplicación:

```
# macOS
brew install graphviz

# Linux (Ubuntu/Debian)
sudo apt-get install graphviz

# Windows
# Descargar desde https://graphviz.org/download/
```

4.3. Instalación de dependencias

Dependencias de Python:

```
pip3 install -r requirements.txt
```

Las dependencias incluidas son:

- Flask 3.0.0
- Flask-CORS 4.0.0
- matplotlib 3.8.2
- networkx 3.2.1
- graphviz $\geq$ 0.16

- Pillow $\geq$ 10.1.0

Dependencias de React:

```
cd frontend/react-app
npm install
cd ../..
```

4.4. Ejecución de la aplicación

La aplicación consta de dos componentes que deben ejecutarse en terminales separadas:
Backend (Terminal 1):

```
python3 -m backend.app
```

El backend estará disponible en `http://localhost:5001`
Frontend (Terminal 2):

```
cd frontend/react-app
npm run dev
```

El frontend estará disponible en `http://localhost:5173`

4.5. Funciones de la interfaz

- **Editor de gramáticas**. Permite escribir gramáticas libres de contexto, separando símbolos con espacios y utilizando ε o `epsilon` para producciones vacías. Además, soporta el uso del operador "—" para especificar múltiples alternativas en una sola línea (por ejemplo, `C ->a | b`).
- **Construcción del parser**. Al pulsar «Construir Parser» se envía la gramática al backend; el sistema responde con el número de estados, transiciones y los conjuntos FIRST y FOLLOW.
- **Visualización del autómata**. La pestaña «Graphviz» muestra los items LR(1) con sus lookaheads. La visualización incluye una transición especial con el símbolo $ hacia un estado final marcado como ACCEPT.
- **Tabla ACTION/GOTO**. Permite inspeccionar la tabla de análisis completa; las acciones `shift`, `reduce` y `accept` se colorean para mayor claridad.
- **Análisis de cadenas**. Permite ingresar una cadena y conocer si es aceptada; se presenta la traza paso a paso con la pila de estados, la entrada restante y la acción tomada.

5. Ejemplos y resultados

Se incluye a continuación un ejemplo de gramática y los conjuntos FIRST y FOLLOW correspondientes, además de un ejemplo de traza de análisis. Esta información sirve para comprobar el funcionamiento del parser.

5.1. Gramática de ejemplo

```
S -> q * A * B * C
A -> a
A -> b * b * D
B -> a |
C -> b |
D -> C |
```

Nótese el uso del operador "—" para especificar alternativas en las producciones de B, C y D.

5.2. Conjuntos FIRST

Símbolo	FIRST( )
S	$\{q\}$
A	$\{a, b\}$
B	$\{a, \epsilon\}$
C	$\{b, \epsilon\}$
D	$\{b, \epsilon\}$

Cuadro 2: Conjuntos FIRST del ejemplo.

5.3. Conjuntos FOLLOW

Símbolo	FOLLOW( )
S	$\{\$\}$
A	$\{*\}$
B	$\{*\}$
C	$\{\$, *\}$
D	$\{*\}$

Cuadro 3: Conjuntos FOLLOW del ejemplo.

5.4. Información del autómata

Para la gramática anterior se obtienen 19 estados y 18 transiciones. El autómata reconoce cinco terminales ($, $*$, a, b, q) y seis no terminales (S, S', A, B, C, D). La tabla ACTION/GOTO generada tiene 19 filas (estados) y columnas para cada símbolo. La figura 1 ilustra un ejemplo de visualización del autómata mediante Graphviz.

Figura 1: Vista del autómata LR(1) generado con Graphviz para la gramática de ejemplo. Nótese la transición con $ hacia el estado ACCEPT.

5.5. Traza de análisis de cadena

A continuación se muestra la traza generada al analizar la cadena `q * a * a * b`, donde cada fila indica el contenido de la pila, la entrada restante y la acción tomada:

Paso	Pila	Entrada	Acción
1	0	$q * a * a * b\$$	shift 1
2	0 1	$*a * a * b\$$	shift 3
3	0 1 3	$a * a * b\$$	shift 4
4	0 1 3 4	$*a * b\$$	reduce 2 ($A \to a$)
5	0 1 3 5	$*a * b\$$	shift 7
6	0 1 3 5 7	$a * b\$$	shift 10
7	0 1 3 5 7 10	$*b\$$	reduce 4 ($B \to a$)
8	0 1 3 5 7 12	$*b\$$	shift 13
9	0 1 3 5 7 12 13	$b\$$	shift 14
10	0 1 3 5 7 12 13 14	\$	reduce 6 ($C \to b$)
11	0 1 3 5 7 12 13 15	\$	reduce 1 ($S \to q * A * B * C$)
12	0 2	\$	ACCEPT

6. Características destacadas

El proyecto presenta varias funcionalidades que exceden los requisitos del examen:

- **Parser LR(1) completo**. Se implementa un algoritmo LR(1) canónico en lugar de LALR(1), con cálculo correcto de lookaheads, manejo de producciones *epsilon* y construcción de tablas ACTION/GOTO.
- **Soporte para sintaxis con alternativas**. El parser reconoce el operador "—" para especificar múltiples producciones en una sola línea, simplificando la escritura de gramáticas.
- **Visualización profesional con Graphviz**. Se generan visualizaciones detalladas que muestran los items LR(1) completos con lookaheads, estados coloreados (verde para inicial, rojo para aceptación) y una transición explícita con \$ hacia el estado ACCEPT.
- **Interfaz web moderna**. La aplicación utiliza React con arquitectura de componentes y proporciona una experiencia de usuario fluida; la tabla ACTION/GOTO se colorea según el tipo de acción.
- **Manejo de errores**. El backend valida gramáticas y cadenas, devolviendo mensajes descriptivos; la traza de análisis se detiene en caso de error sintáctico.

7. Comparación con los requisitos

Se presenta a continuación una comparación entre los requisitos del proyecto y las funcionalidades implementadas.

Requisito	Estado	Detalles
Parser LALR(1)	Superado	Se implementó un LR(1) canónico, más potente que LALR(1).
Interfaz gráfica	Cumplido	Aplicación React con secciones para gramática, autómata, tabla y análisis de cadenas.
Reporte pequeño	Cumplido	Este documento incluye una explicación completa del proyecto.
Uso exclusivo de Python	Cumplido	El backend está escrito en Python; el frontend se implementó adicionalmente para mejorar la experiencia.
Presentación	Cumplido	El proyecto cuenta con visualización en vivo del autómata y de las tablas de análisis.

Cuadro 5: Estado de los requisitos del proyecto.

8. Limitaciones y trabajo futuro

Aunque el parser LR(1) implementado es completo y funcional, existen limitaciones que podrían abordarse en versiones posteriores:

- **Detección de ambigüedad**. El sistema no detecta automáticamente si una gramática es ambigua; en caso de conflictos toma la primera acción disponible.
- **Minimización de estados**. No se ha implementado la fusión de estados típica de LALR(1); por ello se generan más estados que los necesarios.
- **Rendimiento de visualización**. Para gramáticas grandes, la visualización con Graphviz puede ser lenta y producir diagramas extensos.
- **Optimización del algoritmo**. La construcción del autómata podría paralelizarse o guardarse en caché para gramáticas repetidas.

En cuanto al trabajo futuro se proponen las siguientes mejoras:

- Implementar detección de conflictos *shift/reduce* y *reduce/reduce*, así como la conversión a LALR(1).
- Integrar un editor con resaltado de sintaxis y un modo oscuro en la interfaz web.
- Permitir exportar la tabla de análisis a diferentes formatos y generar código de parser a partir de la tabla.
- Añadir un histórico de gramáticas y la posibilidad de compartirlas mediante enlaces.

9. Conclusiones

Se ha desarrollado un analizador sintáctico LR(1) completo y se ha integrado con una interfaz web moderna, superando ampliamente los requisitos iniciales del proyecto. El parser maneja adecuadamente producciones vacías, soporta sintaxis con alternativas mediante

el operador "—", y genera lookaheads precisos, produciendo tablas ACTION/GOTO correctas. La arquitectura modular del código facilita la reutilización y el mantenimiento. Este proyecto no sólo demuestra conocimientos teóricos de análisis sintáctico, sino que también integra conceptos de desarrollo full-stack, y puede servir como base para diseñar lenguajes de programación o validar gramáticas en contextos académicos y profesionales.

10. Referencias

1. Aho, A. V., Lam, M. S., Sethi, R., & Ullman, J. D. *Compilers: Principles, Techniques, and Tools* (2nd ed.). Pearson, 2006.

2. Cooper, K., & Torczon, L. *Engineering a Compiler* (2nd ed.). Morgan Kaufmann, 2011.

3. Appel, A. W. *Modern Compiler Implementation in ML*. Cambridge University Press, 2004.

4. Documentación oficial de Python: `https://docs.python.org/3/`

5. Documentación de React: `https://react.dev/`

6. Documentación de Flask: `https://flask.palletsprojects.com/`

7. Documentación de Graphviz: `https://graphviz.org/documentation/`