Compiler Design - Visualization Project

Implementation of Lexical Analysis and Parsing with Web Visualization

CS23I1043 - VISHWA S
CS23I1044 - PRAVEEN CHOUTHRI

Course: Compiler Design
IIITDMK

November 11, 2025

Presentation Outline

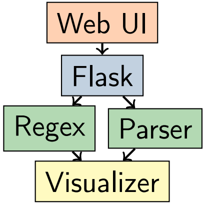
What We Built

Project Goal

A web-based compiler visualization system that demonstrates lexical analysis and parsing in real-time

Key Components:

- **Regex to DFA Converter** (C - LEX and YACC)
- **CLR Parser** (Python based)
- **Web Interface** (HTML/CSS/JavaScript)
- **Flask Backend** (REST API)
- **Real-time Visualization** (HTML tables + DOT graphs)



Why We Built It This Way

Design Decisions:

- **Web-based**: Accessible from any device, no installation required.
- **DFA Visualizer**: Used the LEX and YACC tools to convert the given regular expression to DFA and visualized it.
- **CFG Parser**: Used purely python code to built the CLR parser.
- **Real-time Visualization**: Immediate feedback for custom inputs learning

Technology Stack:

- **Frontend**: HTML5, CSS3, JavaScript (Vanilla)
- **Backend**: Python Flask, RESTful API
- **Lexer**: C with Flex(LEX) / Bison(YACC)
- **Parser**: Pure Python CLR implementation
- **Visualization**: HTML tables, DOT/Graphviz

Lexical Analysis - What We Implemented

Core Concept

Convert regular expressions to DFA for token recognition

Our Implementation Approach:

1. **Thompson's Construction**: Regex $\rightarrow$ NFA
2. **Subset Construction**: NFA $\rightarrow$ DFA
3. **DFA Minimization**: Optimize state count

Example Flow:

$$\texttt{(a|b)*} \rightarrow \text{NFA} \rightarrow \text{DFA} \rightarrow \text{Graph}$$

Key Innovation

We built this from scratch in C, then wrapped it as an executable that our Python backend calls via subprocess.

Parsing - What We Implemented

Core Concept

CLR(1) parsing with automatic parse table generation

Our Implementation Approach:

1. **Grammar Processing**: Parse CFG input format
2. **CLR Item Sets**: Build canonical collection
3. **Parse Table**: Generate ACTION/GOTO tables
4. **Parser Engine**: Execute parsing with trace

Example Grammar:

```
E -> E '+' T | T;
T -> T '*' F | F;
F -> ( E ) | 'id';
```

Development Workflow

Phase 1: Core Algorithms

- Implemented Thompson's construction in C
- Built CLR parser engine in Python
- Created basic visualization functions

Phase 2: Integration

- Designed REST API for component communication
- Built Flask server with error handling
- Created responsive web interface

Phase 3: Optimization

- Implemented flyweight pattern for memory efficiency
- Added lazy evaluation for large grammars
- Enhanced visualization with better formatting

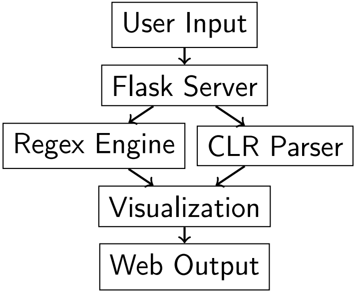
Phase 4: Frontend Development

- User interface improvements
- Performance optimization

Essential Files:

- `server.py` - Flask backend
- `index.html` - Web interface
- `part1_regex/` - source codes for DFA convertion
- `cfg_parser.py` - CLR parser
- `visualization.py` - HTML/DOT generators
- `regex_visualizer.exe` - Compiled lexer

Data Flow:



Regex Engine Implementation

C-based Regex to DFA Converter:

```c
// Main regex processing function
int process_regex(char* regex_input) {
    // 1. Parse regex into AST
    RegexAST* ast = parse_regex(regex_input);

    // 2. Thompson's construction: AST -> NFA
    NFA* nfa = thompson_construction(ast);

    // 3. Subset construction: NFA -> DFA
    DFA* dfa = subset_construction(nfa);

    // 4. DFA minimization
    DFA* minimized = minimize_dfa(dfa);

    // 5. Output JSON for web interface
    output_dfa_json(minimized);

    return 0;
}
```

Integration with Python:

```python
# Flask endpoint calls C executable
process = subprocess.run([REGEX_EXE_PATH],
                        input=regex_input,
                        capture_output=True, text=True)
result = json.loads(process.stdout)
```

CLR Parser Implementation

Grammar Processing:

```python
class GrammarProcessor:
    def parse_grammar(self, cfg_text: str) -> Grammar:
        # Clean and normalize input
        cfg_text = self._clean_input(cfg_text)

        # Extract productions: A -> alpha | beta
        raw_productions = self._extract_raw_productions(cfg_text)

        # Convert to Production objects
        self.productions = self._normalize_productions(raw_productions)

        # Classify symbols as terminals/non-terminals
        self.terminals, self.non_terminals = self._extract_symbols()

        return Grammar(self.productions, self.terminals,
                       self.non_terminals, start_symbol)
```

CLR Item Set Construction

Core CLR Algorithm:

```python
class CLRItemSetBuilder:
    def build_clr_automaton(self, grammar: Grammar) -> CLRAutomaton:
        # Create augmented grammar: S' -> S
        augmented_grammar = self._create_augmented_grammar(grammar)

        # Start with initial item set
        start_items = self._create_start_items(augmented_grammar)
        start_state = CLRState(start_items, 0)

        states = [start_state]
        transitions = {}

        # Build all states using CLOSURE and GOTO
        self._build_states_optimized(states, transitions,
                                     state_map, augmented_grammar)

        return CLRAutomaton(states, transitions, 0)
```

Parse Table Generation

ACTION and GOTO Table Construction:

```
def build_parse_table(self, automaton: CLRAutomaton) -> ParseTable:
    action_table = {}
    goto_table = {}

    for state in automaton.states:
        for item in state.items:
            if not item.is_complete():
                # Shift action: [A ->    a    , b]
                next_symbol = item.next_symbol()
                if next_symbol in self.grammar.terminals:
                    target_state = automaton.transitions.get(
                        (state.state_id, next_symbol))
                    if target_state:
                        action_table[(state.state_id, next_symbol)] = \
                            f"shift {target_state}"
            else:
                # Reduce action: [A ->     , a]
                if item.production.lhs != self.augmented_start:
                    action_table[(state.state_id, item.lookahead)] = \
                        f"reduce {item.production}"

    return ParseTable(action_table, goto_table)
```

Web Interface Integration

Flask API Endpoints:

```
@app.route('/visualize-dfa', methods=['POST'])
def visualize_dfa():
    data = request.json
    regex_input = data.get('regex')

    # Call C executable for DFA generation
    process = subprocess.run([REGEX_EXE_PATH],
                             input=regex_input,
                             capture_output=True, text=True)

    if process.returncode == 0:
        output_data = json.loads(process.stdout)
        return jsonify(output_data)
    else:
        return jsonify({"error": process.stderr}), 500

@app.route('/visualize-parser', methods=['POST'])
def visualize_parser():
    data = request.json
    cfg_input = data.get('cfg')
    string_input = data.get('input')

    # Use Python CLR parser
    parser_visualizer = CFGParserVisualizer()
    grammar_result = parser_visualizer.process_grammar(cfg_input)
    parse_result = parser_visualizer.parse_input(string_input)

    return jsonify({
        "parseTreeDot": parse_result['tree_dot'],
        "parseTableHtml": grammar_result['tables_html'],
        "parseTraceHtml": parse_result['trace_html']
    })
```

Visualization Engine

HTML Table Generation:

```python
class HTMLTableGenerator:
    def generate_action_goto_tables_html(self, action_table, goto_table,
                                         terminals, non_terminals):
        html_lines = []

        # Create responsive table with accessibility
        html_lines.append('<table class="grammar-table" role="table">')

        # Generate headers with ACTION/GOTO sections
        html_lines.append(self._generate_table_header(
            terminals, non_terminals))

        # Generate table body with color-coded actions
        for state in sorted(all_states):
            html_lines.append(self._generate_table_row(
                state, terminals, non_terminals,
                action_table, goto_table))

        html_lines.append('</table>')
        return '\n'.join(html_lines)

    def _format_action(self, action: str) -> str:
        # Color-code different action types
        if action.startswith('shift'):
            return f'<span class="shift-action">{action}</span>'
        elif action.startswith('reduce'):
            return f'<span class="reduce-action">{action}</span>'
        elif action == 'accept':
            return f'<span class="accept-action">{action}</span>'
```

Lexical Analysis: C-Backend Architecture

Python to C Integration

The Python Flask server calls a pre-compiled C executable `regex_visualizer.exe` using `subprocess.run()`.

- The regex string is piped to the C program's **stdin**.
- The C program pipes its final JSON output to **stdout**.
- Python captures and parses this JSON to send to the frontend.

Core C Components:

- **Lexer** (`regex_lexer.l`)
  A Flex file that tokenizes the input regex string (e.g., | $\rightarrow$ `PIPE`).
- **Parser** (`regex_parser.y`)
  A Bison file that defines the regex grammar and builds an Abstract Syntax Tree (AST).

Data Structures & Engine:

- **AST** (`ast.h`)
  Defines the `ASTNode` struct for tree representation (e.g., `NODE_UNION`, `NODE_CONCAT`).
- **Engine** (`engine.h/.c`)
  Implements core data structures (`List`, `Set`, `NFAState`, `DFAState`) and all conversion logic.

Lexical Analysis: C-Backend Implementation

Core Algorithmic Pipeline

The `main()` function in `main_regex.c` orchestrates the entire conversion process, managed by `process_regex_to_dfa_dot()`.

Step-by-Step Conversion:

1. **AST Generation**
   `regexparse()` is called to build the AST from `stdin`.
2. **Thompson's Construction (`ast_to_nfa()`)**
   Recursively walks the AST to construct an equivalent NFA.
3. **Subset Construction (`nfa_to_dfa()`)**
   Converts the NFA to a DFA using `epsilon_closure()` and `nfa_move()` helper functions.
4. **Serialization (Output Generation)**
   The final DFA state list is converted into two formats:
   - `dfa_to_dot_string()`: For Graphviz visualization.
   - `dfa_to_json()`: For the web interface's state table.

Final Output

The C program combines the DOT string and state table JSON into a single JSON object and prints it to `stdout` to be captured by Python.

Parsing: Python CLR Engine Logic

1. Grammar Processing (GrammarProcessor)

- The `parse_grammar()` method receives the raw CFG text.
- It cleans the input, extracts all productions, and classifies symbols into **terminals** and **non-terminals**.
- The grammar is augmented with a new start symbol (e.g., `S' -> S`) to begin the parsing process.

2. CLR(1) Automaton Construction (CLRItemSetBuilder)

- This is the core of the parser generator.
- It builds the **canonical collection of CLR(1) item sets**.
- It repeatedly uses the **CLOSURE** operation (to find all items in a state) and the **GOTO** operation (to find transitions between states).

3. Parse Table Generation

- The `build_parse_table()` function iterates over the completed automaton's states and items.
- It populates the **ACTION** table (with `shift`, `reduce`, `accept` actions) and the **GOTO** table (for non-terminal transitions).

Parsing: Python Backend & Visualization

Flask API Endpoints (server.py)

The Flask server acts as the central controller:

- `/visualize-dfa`: Receives a regex string. It acts as a bridge to the C-backend (Part 1) by calling `regex_visualizer.exe` using `subprocess.run()`.
- `/visualize-parser`: Receives the CFG and input string. It calls the internal Python CLR parser engine to get the results.

Dynamic Visualization Engine (visualization.py)

This component translates the parser's output into clean HTML:

- **Parse Table**: Generates a responsive HTML table for the `ACTION` and `GOTO` data.
- **Color-Coded Actions**: Uses CSS classes (e.g., `.shift-action`, `.reduce-action`) to make the table easy to read.
- **Parse Trace**: Creates a step-by-step HTML table showing the parser's stack, input, and actions for a given string.

Conceptual Design & AI Implementation

AI as an Implementation Engine

We used AI to accelerate the coding of well-defined tasks and components.

- **Algorithm Implementation:**
  Wrote the C code for known algorithms, such as **Thompson's Construction** (AST to NFA) and **Subset Construction** (NFA to DFA).
- **Boilerplate Generation:**
  Generated the Python class structures for the **Flask server** and the **HTML/DOT visualizers** based on our specifications.
- **Frontend Scripting:**
  Assisted with client-side JavaScript for DOM manipulation and **AJAX (‘fetch‘) calls** to the backend.

Human as the Conceptual Architect

We were responsible for the core compiler design, logic, and system architecture.

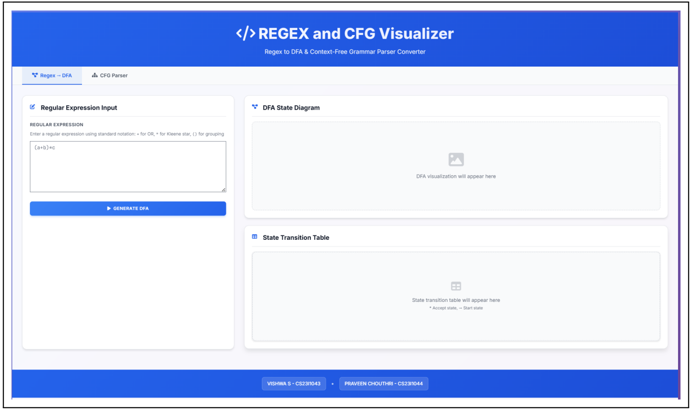
- **Grammar AST Design:**
  Conceptually designed the **regex grammar for Lex/Yacc**, defining operator precedence and the C **Abstract Syntax Tree (AST)** structure.
- **Data Contract Definition:**
  Designed the specific **JSON data contract** that acts as the "glue" between the C engine, Python backend, and JavaScript frontend.
- **Logical Workflow:**
  Designed the entire **multi-step workflow** for the CFG parser (Parse Grammar → Select Symbol → Build Table → Parse String).
- **Visualization Strategy:**
  Determined ***what*** to visualize and ***how*** (e.g., color-coded tables, DOT graphs, and a step-by-step trace).

System Features

What Our System Can Do:

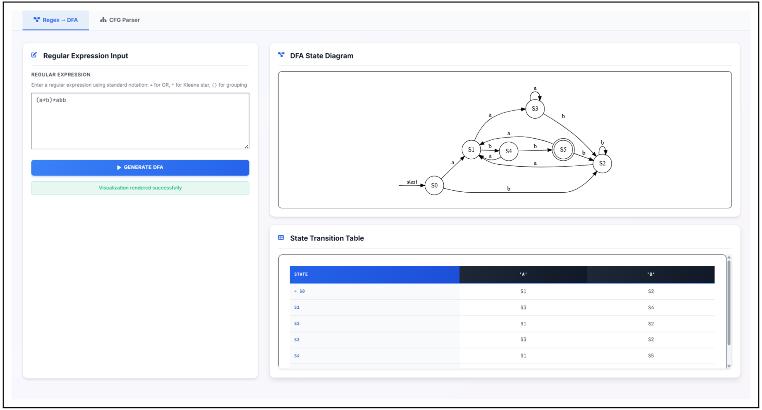
1. **Regex to DFA Conversion**
   - Input any regular expression
   - See step-by-step NFA construction
   - View final DFA with state transitions
   - Interactive state diagram visualization
2. **Grammar Processing**
   - Parse context-free grammar input
   - Generate complete CLR parse tables
   - Color-coded ACTION/GOTO tables
   - Conflict detection and reporting
3. **String Parsing**
   - Parse input strings with step-by-step trace
   - Show stack operations and parsing actions
   - Generate parse trees in DOT format
   - Real-time error detection and reporting

Main Interface Overview



Web interface showing grammar input panel and visualization output

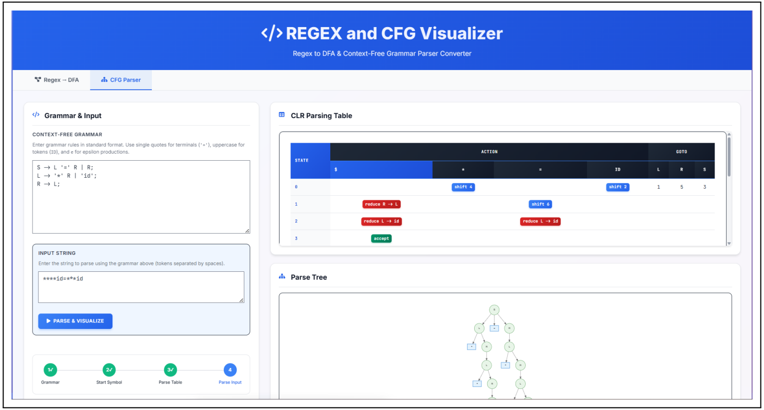
DFA Visualization



STATE	'A'	'B'
→ S0	S1	S2
S1	S3	S4
S2	S1	S2
S3	S3	S2
S4	S1	S5

Deterministic Finite Automaton generated from regular expression

CLR - Visualization



CFG Parser

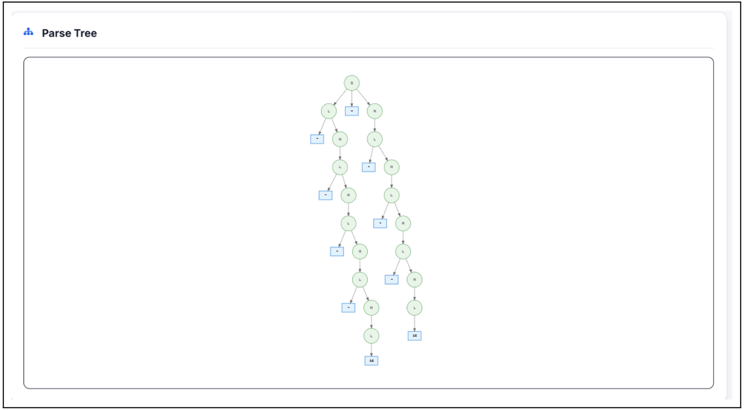
CLR Parse Table

CLR Parsing Table

STATE	ACTION				GOTO		
	$	*	=	ID	L	R	S
0		shift 4		shift 2	1	5	3
1	reduce R → L		shift 6				
2	reduce L → id		reduce L → id				
3	accept						

ACTION and GOTO table with color-coded parsing actions

Parse Tree Visualization



Parse tree showing hierarchical grammar derivation

Step-by-Step Parsing Trace

Parser Action Trace

Parsing Trace

STEP	STACK	INPUT	ACTION	PRODUCTION
1	0	* * * * id = * * * id ...	shift 4	
2	0*4	* * * id = * * * id $	shift 4	
3	0*4*4	* * id = * * * id $	shift 4	
4	0*4*4*4	* id = * * * id $	shift 4	
5	[illegible]	[illegible]	[illegible]	

Detailed parsing trace with stack operations and actions

Thank You!

Team Members:
CS23I1043 & CS23I1044

Course: Compiler Design

GitHub Repository: https://github.com/Vishwa1805/CD_Project