

CLR PARSER

1. Introduction

Project Background and Relevance

Compiler Design is a key area in computer science that deals with translating high-level programming languages into low-level machine-executable code. This translation is done through multiple phases, one of which is **syntax analysis**. This phase, also known as **parsing**, checks whether the source code follows the correct grammatical structure defined by a context-free grammar (CFG).

Among the various parsing methods, **Canonical LR (CLR) parsing** is a powerful bottom-up technique that can handle a wide range of grammars. It uses **lookahead symbols** to reduce ambiguities and improve error detection during parsing. Unlike simpler parsers like LL(1) and SLR(1), CLR parsers offer greater precision, making them suitable for more complex programming language constructs.

This project focuses on implementing a CLR parser using Python, emphasizing the construction of **First and Follow sets**, **LR(1) items**, **closure**, **GOTO functions**, and the generation of **ACTION** and **GOTO tables**. The goal is to simulate how compilers perform syntax checking by accepting a CFG, generating its parsing tables, and using

Simple parsing methods fail to correctly process grammars that involve ambiguity, left recursion, or nested expressions. This creates a need for advanced parsing techniques that can handle such grammars with accuracy and clarity. To address this, the project aims to develop a parser that:

- Accepts a context-free grammar.
- Calculates First and Follow sets to support lookahead-based parsing.
- Builds the CLR(1) parsing table using closure and GOTO functions.
- Parses input strings and returns whether they are accepted or rejected according to the grammar.

Project Objectives

- To build a **CLR(1) parser** that accepts context-free grammars as input.
- To compute **First and Follow sets** essential for generating LR(1) items.
- To construct **closure** and **GOTO** functions for building parser states.
- To generate **ACTION** and **GOTO tables** used for table-driven parsing.
- To parse input strings using a simulated stack-based mechanism.
- To reinforce understanding of syntax analysis and bottom-up parsing.

2. Methodology

The Canonical LR (CLR) parser is a powerful bottom-up parsing technique that constructs a deterministic finite automaton (DFA) based on LR(1) items. CLR parsing involves several key steps, including augmenting the grammar, computing First and Follow sets, generating LR(1) item sets, building the parsing tables (ACTION and GOTO), and parsing input strings using a stack-based simulation. Each of these steps contributes to robust syntax analysis capable of handling complex grammars.

Step 1: Augmenting the Grammar

We begin by augmenting the given grammar with a new start symbol (S') and a production rule $S' \rightarrow S$, where S is the original start symbol. This ensures the parser has a unique accept state.

Step 2: Computing First and Follow Sets

First sets help determine the set of terminals that can begin strings derived from a non-terminal. Follow sets identify the set of terminals that can appear immediately to the right of a non-terminal. These are essential for generating LR(1) items that include lookahead symbols.

Step 3: Constructing LR(1) Items

An LR(1) item is a production with a dot (\bullet) marking the current position of the parser in the production rule and a lookahead symbol. For example, $[A \rightarrow \alpha \bullet \beta, a]$ indicates the parser expects to see β next and the lookahead terminal is 'a'.

Step 4: Closure Operation

The closure of a set of LR(1) items adds more items to the set by expanding non-terminals that immediately follow the dot. For each such non-terminal, we include productions for that symbol with lookahead symbols derived from First sets. This step ensures the parser is aware of all possible productions it may encounter next.

Step 5: GOTO Function

The GOTO function determines the transition between parser states. When the dot precedes a symbol X in an item, GOTO moves the dot past X and computes the closure of the resulting item set. This creates a DFA of parser states.

Step 6: Constructing ACTION and GOTO Tables

Each DFA state becomes a row in the parsing table. The **ACTION table** defines whether to shift, reduce, or accept based on terminal input and current state, while the **GOTO table** defines transitions for non-terminals.

- Shift: move to the next state
- Reduce: apply a grammar rule
- Accept: successful parse

- Error: invalid input

Step 7: Parsing the Input

The parser uses a stack to simulate the parsing process. It reads tokens from the input and consults the ACTION table for guidance. It shifts tokens and states onto the stack, and when a reduce action is encountered, it pops symbols and pushes the left-hand non-terminal and the corresponding GOTO state. If the ACTION is 'accept', the string is valid. Otherwise, an error is raised.

This end-to-end process emulates how real-world compilers perform syntax analysis using deterministic parsing logic with lookahead capabilities.

Specification Needed

- **Programming Language:** Python 3.x
- **Libraries Used:**
 - re for regular expressions
 - tabulate for displaying tables
 - collections for structured data handling
- **Tools:**
 - Any text/code editor (VS Code, PyCharm, Sublime)
 - Terminal or command line interface for running scripts
- **System Requirements:**
 - OS: Windows/Linux/macOS
 - RAM: Minimum 4GB
 - Python interpreter installed (version 3.6 or above)

3. Implementation

The CLR(1) parser project involves several critical modules working in harmony—`firstfollow.py`, `fixed_refactored_CLR.py`, and the Streamlit-based frontend in `app.py`. Below are the most significant code snippets that illustrate the core logic of the parser.

a. FIRST and FOLLOW Set Computation

Inside `firstfollow.py`, the `compute_first()` function handles both terminals and non-terminals. It recursively builds the FIRST set of each non-terminal by expanding productions and considering ϵ (epsilon, represented as `chr(1013)`) where applicable:

```
def compute_first(symbols):
    if isinstance(symbols, list):
        result = set()
```

```
for symbol in symbols:
    temp = compute_first(symbol)
    result |= temp - {chr(1013)}
    if chr(1013) not in temp:
        break
else:
    result.add(chr(1013))
return result
```

This logic supports computing the FIRST of a sequence (for predictive parsing), an essential step before building the CLR items.

b. Closure and GOTO Construction

In `fixed_refactored_CLR.py`, the heart of item generation lies in the `closure()` function, which constructs a set of LR(1) items. It ensures that for each non-terminal after a dot ('.'), all its productions are added recursively with appropriate lookahead symbols.

```
def closure(items):
    while True:
        flag = 0
        for i in items:
            if i.index('.') == len(i) - 1: continue
            body = i.split('->')[1]
            after_dot = body.split('.')[1]
            Y = after_dot[0]
            symbols = list(after_dot[1:])
            lastr = compute_first(symbols) - {chr(1013)} if symbols else
            i.lookahead
            for prod in production_list:
                head, body = prod.split('->')
                if head == Y:
                    newitem = Item(Y + '->' + body, list(lastr))
                    if not any(j == newitem and sorted(set(j.lookahead)) ==
sorted(set(newitem.lookahead)) for j in items):
                        items.append(newitem)
                    flag = 1
            if not flag: break
    return items
```

This snippet handles the propagation of lookaheads and recursive expansion of items—critical for CLR(1) correctness.

c. State Comparison for Uniqueness

To avoid duplicate states during automaton construction, the `states_equal()` function compares two item sets:

```
def states_equal(s1, s2):
    if len(s1) != len(s2): return False
    s1_sorted = sorted(s1, key=str)
    s2_sorted = sorted(s2, key=str)
    for i in range(len(s1_sorted)):
        if s1_sorted[i] != s2_sorted[i]: return False
        if sorted(s1_sorted[i].lookahead) !=
sorted(s2_sorted[i].lookahead): return False
    return True
```

This ensures each LR(1) state is uniquely represented before being added to the list of automaton states.

d. Streamlit Integration

In `app.py`, the uploaded grammar is passed to the parser backend via `run_clr()` after syncing it with `firstfollow.production_list`:

```
firstfollow.production_list.clear()
firstfollow.production_list.extend(productions)
result = run_clr(productions)
```

This enables real-time generation of CLR(1) states, parsing tables, and FIRST/FOLLOW sets through a web UI. The output is dynamically displayed using Streamlit widgets like `st.code()`, `st.dataframe()`, and `st.warning()`.

e. Parsing Table Generation

Finally, the parsing table is built using the `make_table()` function, where shift, reduce, and accept actions are mapped:

```
if getprodno(item) == 0:
    SLR_Table[s.no]['$'] = 'ac'
else:
    for term in item.lookahead:
        SLR_Table[s.no].setdefault(term, set()).add('r' +
str(getprodno(item)))
```


This logic maps terminal transitions and reduction actions using LR(1) lookahead semantics.

Results:

CLR(1) and LALR(1) Parser Generator


Upload a grammar file with productions (e.g., $E \rightarrow E+T$) one per line. Use ϵ to represent epsilon (ϵ).

Upload Grammar File


 Drag and drop file here
Limit 200MB per file • TXT

Browse files

Upload Grammar File

 Drag and drop file here
Limit 200MB per file • TXT

Browse files

 grammar.txt 24.0B ×

Grammar successfully loaded.

Productions:

$S \rightarrow AaAb$
 $S \rightarrow BbBa$
 $A \rightarrow$
 $B \rightarrow$

FIRST and FOLLOW Sets

S

- FIRST: {'b', 'a'}
- FOLLOW: {'\$'}

A

- FIRST: {'ε'}
- FOLLOW: {'b', 'a'}

B

- FIRST: {'ε'}
- FOLLOW: {'b', 'a'}

CLR(1) Items

Item 0

Z->.S, \$

S->.AaAb, \$

S->.BbBa, \$

A->., a

B->., b

Item 1

$Z \rightarrow S, \$$

Item 2

$S \rightarrow A.aAb, \$$

Item 3

$S \rightarrow B.bBa, \$$

Item 4 \Leftrightarrow

$S \rightarrow Aa.Ab, \$$

$A \rightarrow , b$

Item 5

$S \rightarrow Bb.Ba, \$$

$B \rightarrow , a$

Item 6

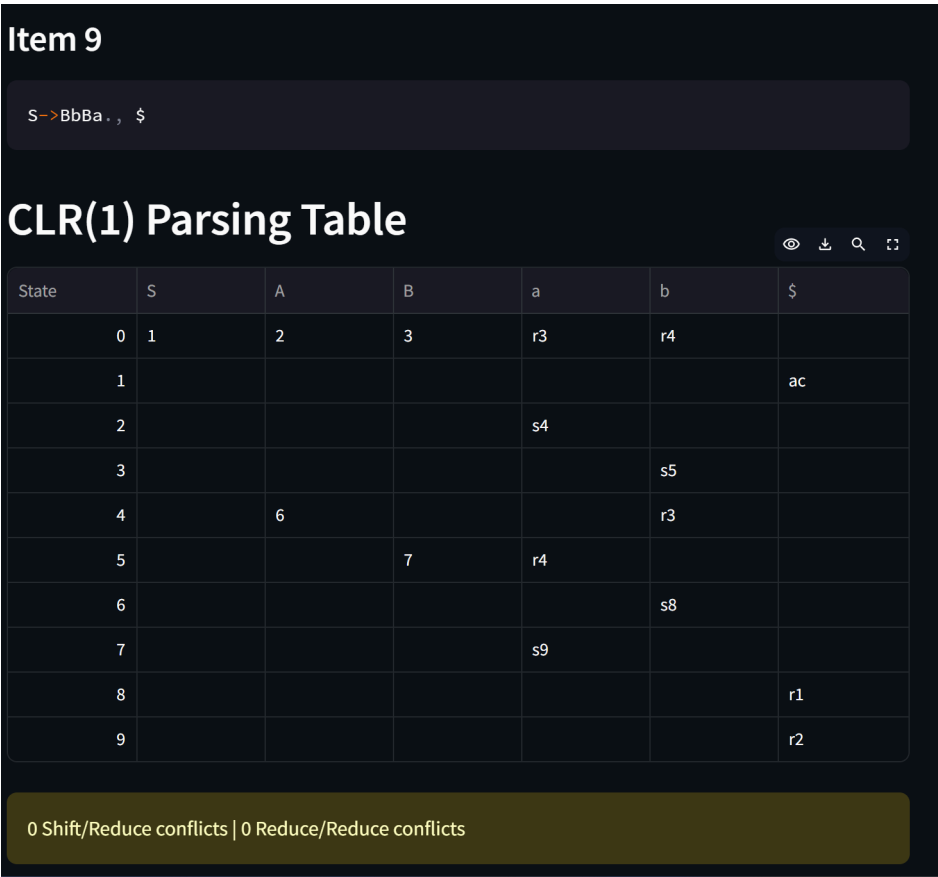
$S \rightarrow AaA.b, \$$

Item 7

$S \rightarrow BbB.a, \$$

Item 8 \Leftrightarrow

$S \rightarrow AaAb, \$$



4. Conclusion & Future Enhancements

Conclusion

This project successfully demonstrated the complete implementation of a **Canonical LR (CLR) parser** using Python. We began by studying the theory of bottom-up parsing and applied it practically through code to simulate how real compilers perform syntax analysis. The parser accepts a context-free grammar, constructs LR(1) item sets using **First and Follow sets**, generates parsing tables (ACTION and GOTO), and parses user-provided input strings using a table-driven mechanism.

Through this project, we developed a strong understanding of:

- The role of **First and Follow sets** in generating lookahead-based parsing items.
- How **closure** and **GOTO** functions build the DFA used for parsing.
- The internal structure of **LR parsing tables** and their practical usage.
- Simulating a real compiler’s parsing phase using **stack-based parsing** and **state transitions**.

The hands-on coding experience allowed us to see how theoretical grammar rules are applied in practical compiler construction. Debugging, testing edge cases, and verifying the correctness of parsing actions provided deep insights into error detection and grammar handling in modern compiler systems.

Overall, this project deepened our appreciation of compiler design and equipped us with the skills to implement parsers for more advanced language grammars.

Future Work / Enhancements

While the current version of the CLR parser achieves its basic objectives, several enhancements can be made to improve its functionality, usability, and learning value:

1. **Grammar Validation:**
Add automated checks to ensure the input grammar is suitable for CLR(1) parsing and provide helpful feedback if not. This can prevent runtime errors and improve usability.
2. **Parse Tree Generation:**
Visualizing the parse tree as the input string is processed would help learners understand how syntax trees are built step by step.
3. **Detailed Error Reporting:**
Currently, the parser simply accepts or rejects input. Adding descriptive error messages and identifying the location of syntax errors would enhance debugging and learning.
4. **Support for More Complex Grammars:**
Extend the system to support more advanced grammar rules (e.g., epsilon transitions, left-recursion removal, and operator precedence handling).
5. **Export Capabilities:**
Provide options to export parsing tables and logs as PDF or CSV for documentation and analysis purposes.