Assignment 2-Create an SLR parser for any given grammar

Name-Anvit Kumar

Roll no-102203161

Group-4C14

1. Objective

The goal of this assignment is to design and implement an SLR (Simple LR) parser for the following context-free grammar, which defines simple arithmetic expressions:

1. E -> E + T | T
2. T -> T * F | F
3. F -> (E) | id

The implementation will accomplish the two primary tasks outlined in the assignment:

- Generate the Canonical LR(0) Collection of Items: This involves creating the states of the parser's finite automaton.
- Construct the SLR ACTION and GOTO Parsing Tables: This involves populating the tables that will drive the parsing decisions.

Finally, the program will use these generated tables to parse a sample input string to demonstrate its correctness.

2. Methodology and Code Explanation

The parser is implemented in Python within a single class, SLRParser. This class encapsulates all the logic required for the entire process, from initial grammar analysis to the final parsing of a string. The process is broken down into several logical steps, each handled by specific methods within the class.

2.1. Grammar Processing

Before any states or tables can be built, the program must first understand the grammar.

- _parse_grammar(grammar_str): This method reads the grammar provided as a multi-line string. It performs two critical initial steps:

1. Augmentation: It augments the grammar by adding a new production, E' -> E (where E is the original start symbol). This new production provides a clear, unambiguous state for accepting the entire input.
2. Symbol Identification: It iterates through the productions to identify and store all terminal (e.g., id, +, *) and non-terminal (e.g., E, T, F) symbols.

- _compute_first_and_follow(): This method implements the standard algorithms to calculate the FIRST and FOLLOW sets for every non-terminal. These sets are essential for constructing the SLR parsing table. Specifically, the FOLLOW set of a non-terminal A is used to decide in which columns of the ACTION table to place the reduce action for a production where A is the head.

2.2. Task 1: Generating the Set of Items

This is the core of creating the parser's state machine. A "state" in an LR parser is a set of LR(0) items. An LR(0) item is a production with a dot (.) somewhere in its body, indicating how much of that production has been seen so far.

- _build_canonical_collection(): This method orchestrates the creation of all states. It relies on two fundamental helper functions:

1. _closure(items): Given a set of items, this function computes its "closure". If an item has a dot before a non-terminal (e.g., A -> α . B β), the closure operation adds all of B's productions to the set (with the dot at the beginning, e.g., B -> . γ). This process repeats until no new items can be added.
2. _goto(items, symbol): This function models a state transition. For a given state (a set of items) and a grammar symbol, it calculates the new state that the parser would transition to after seeing that symbol. It does this by moving the dot past the symbol in all applicable items and then computing the closure of the resulting set.

The _build_canonical_collection method starts with the closure of the very first item (E' -> . E) and systematically applies the goto function for all possible symbols, discovering all reachable states until the entire collection is built.

2.3. Task 2: Generating the Parsing Table

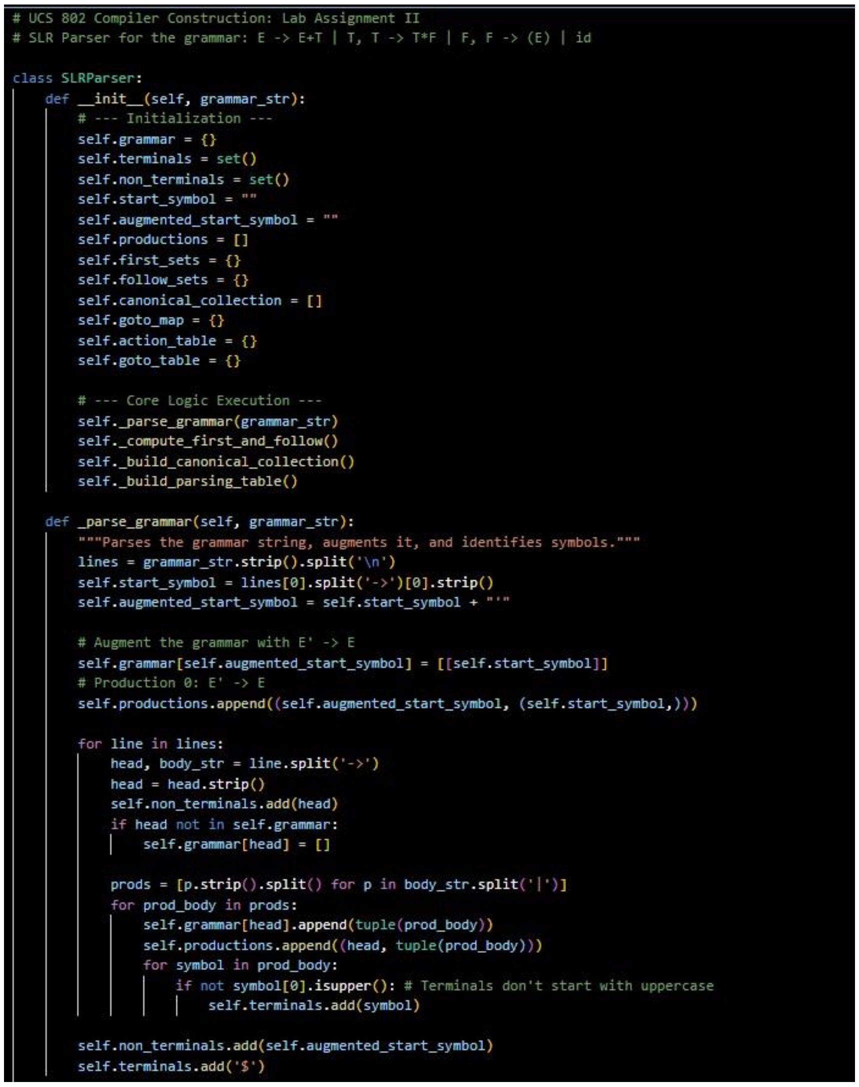
Once the canonical collection of states is created, the ACTION and GOTO tables can be populated.

- _build_parsing_table(): This method iterates through each state (I_i) in the canonical collection and fills the tables according to the SLR parsing rules:
  - SHIFT Action: If a state I_i contains an item A -> α . a β (where a is a terminal) and goto(I_i, a) = I_j, then the table entry ACTION[i, a] is set to Shift j.

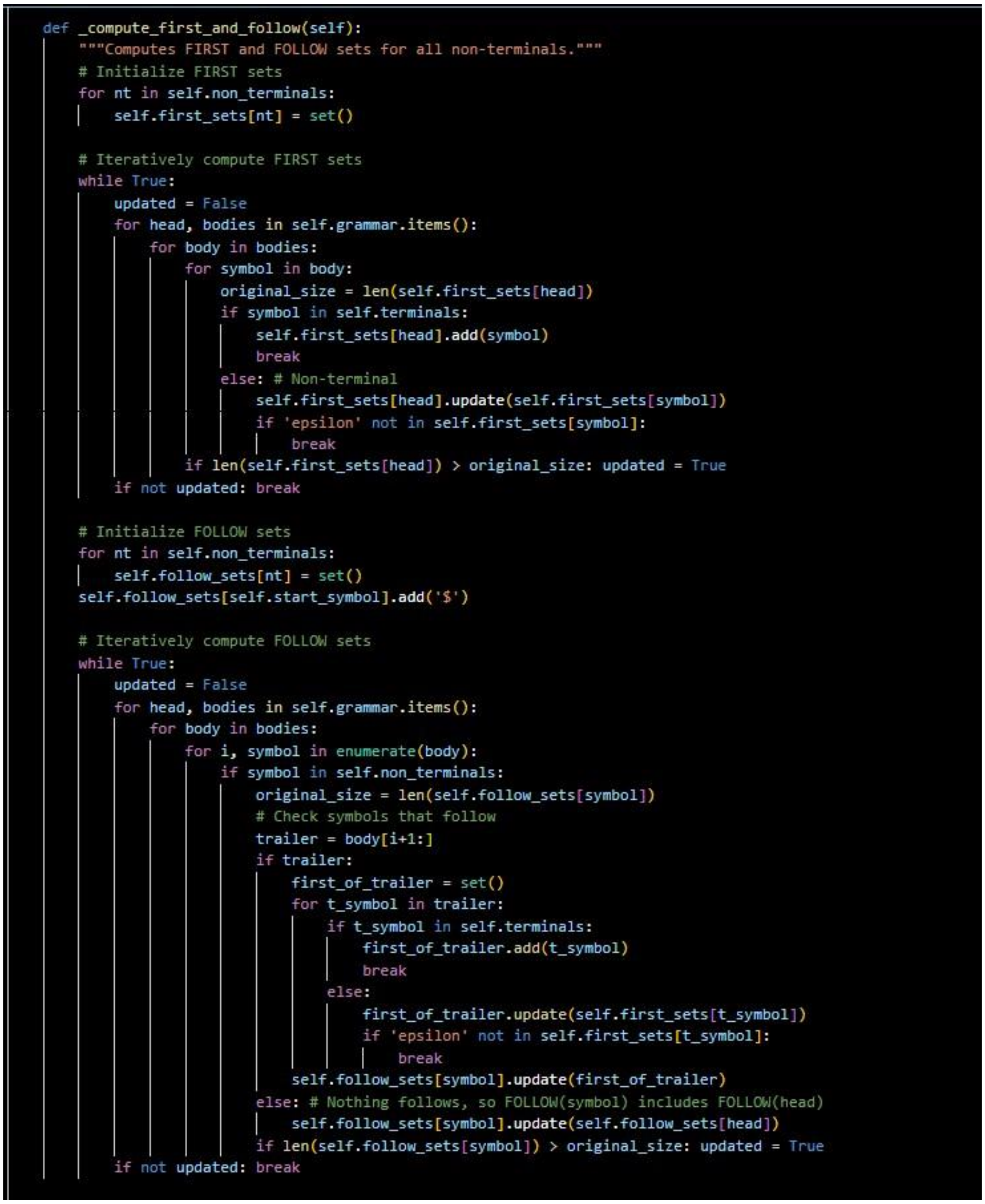
- REDUCE Action: If a state I_i contains a completed item A -> α ., then for every terminal b in FOLLOW(A), the table entry ACTION[i, b] is set to Reduce by A -> α.
- ACCEPT Action: If a state I_i contains the completed augmented item E' -> E ., then the table entry ACTION[i, $] is set to Accept.
- GOTO Entry: If goto(I_i, A) = I_j (where A is a non-terminal), then the table entry GOTO[i, A] is set to j.

2.4. Parsing a String

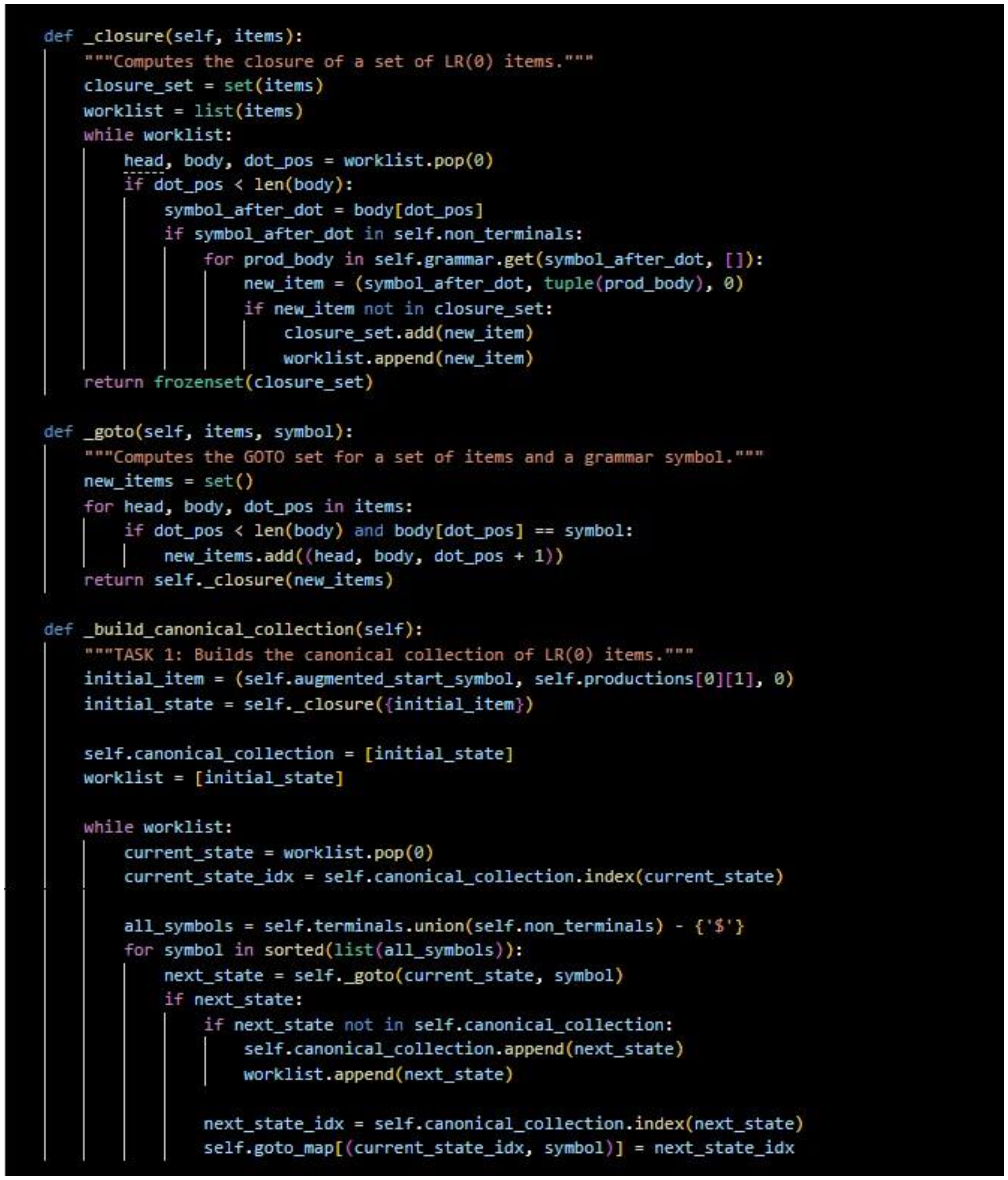
- parse(input_string): This method demonstrates the use of the generated tables. It implements a standard shift-reduce parsing algorithm using a stack. It reads the input string token by token and, at each step, uses the current state (from the top of the stack) and the current input token to look up the appropriate action in the parsing table. It continues until the input is either accepted or an error is found.



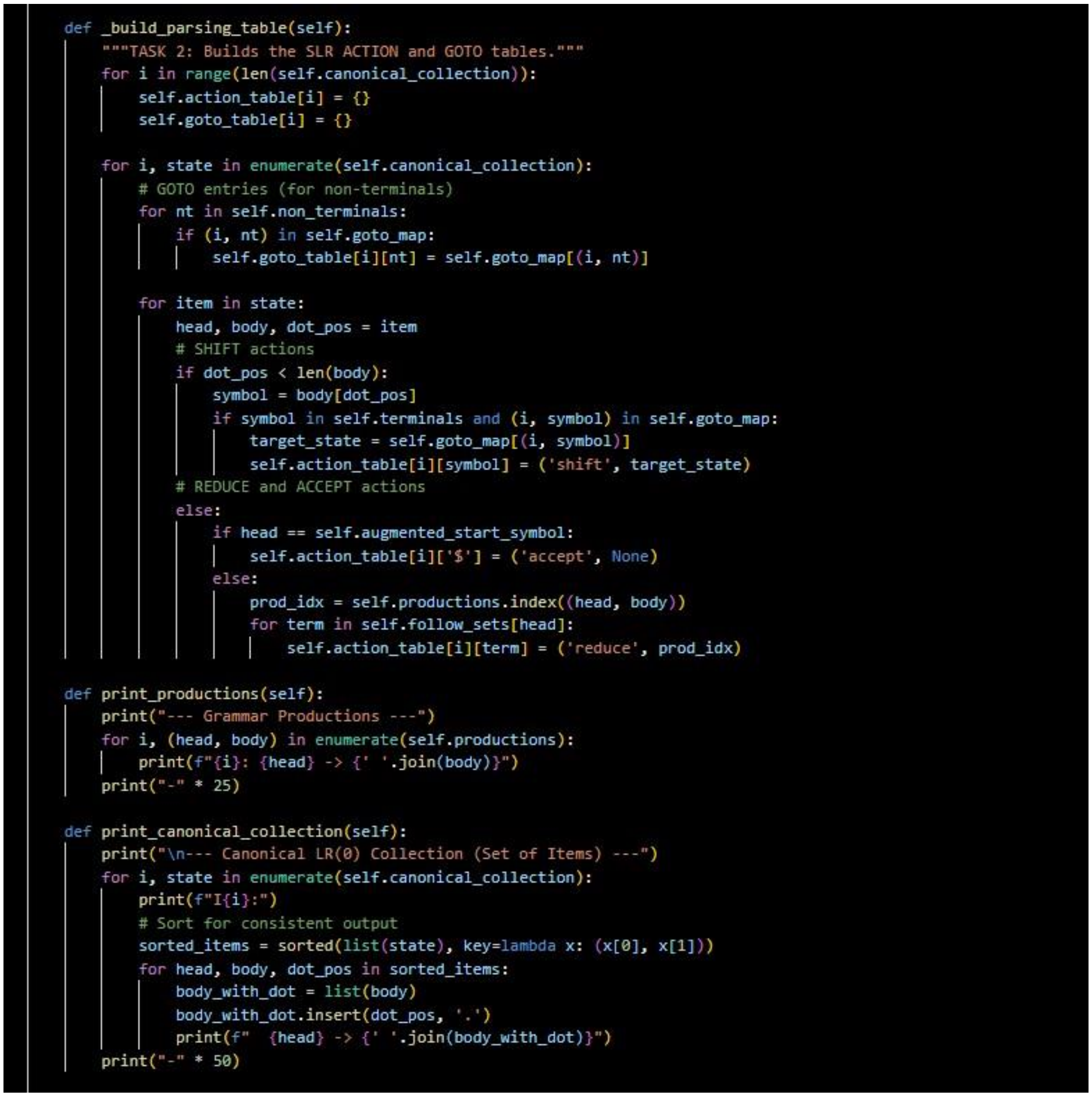
```python
# UCS 802 Compiler Construction: Lab Assignment II
# SLR Parser for the grammar: E -> E+T | T, T -> T*F | F, F -> (E) | id

class SLRParser:
    def __init__(self, grammar_str):
        # --- Initialization ---
        self.grammar = {}
        self.terminals = set()
        self.non_terminals = set()
        self.start_symbol = ""
        self.augmented_start_symbol = ""
        self.productions = []
        self.first_sets = {}
        self.follow_sets = {}
        self.canonical_collection = []
        self.goto_map = {}
        self.action_table = {}
        self.goto_table = {}

        # --- Core Logic Execution ---
        self._parse_grammar(grammar_str)
        self._compute_first_and_follow()
        self._build_canonical_collection()
        self._build_parsing_table()

    def _parse_grammar(self, grammar_str):
        """Parses the grammar string, augments it, and identifies symbols."""
        lines = grammar_str.strip().split('\n')
        self.start_symbol = lines[0].split('->')[0].strip()
        self.augmented_start_symbol = self.start_symbol + "'"

        # Augment the grammar with E' -> E
        self.grammar[self.augmented_start_symbol] = [[self.start_symbol]]
        # Production 0: E' -> E
        self.productions.append((self.augmented_start_symbol, (self.start_symbol,)))

        for line in lines:
            head, body_str = line.split('->')
            head = head.strip()
            self.non_terminals.add(head)
            if head not in self.grammar:
                self.grammar[head] = []

            prods = [p.strip().split() for p in body_str.split('|')]
            for prod_body in prods:
                self.grammar[head].append(tuple(prod_body))
                self.productions.append((head, tuple(prod_body)))
                for symbol in prod_body:
                    if not symbol[0].isupper(): # Terminals don't start with uppercase
                        self.terminals.add(symbol)

        self.non_terminals.add(self.augmented_start_symbol)
        self.terminals.add('$')
```



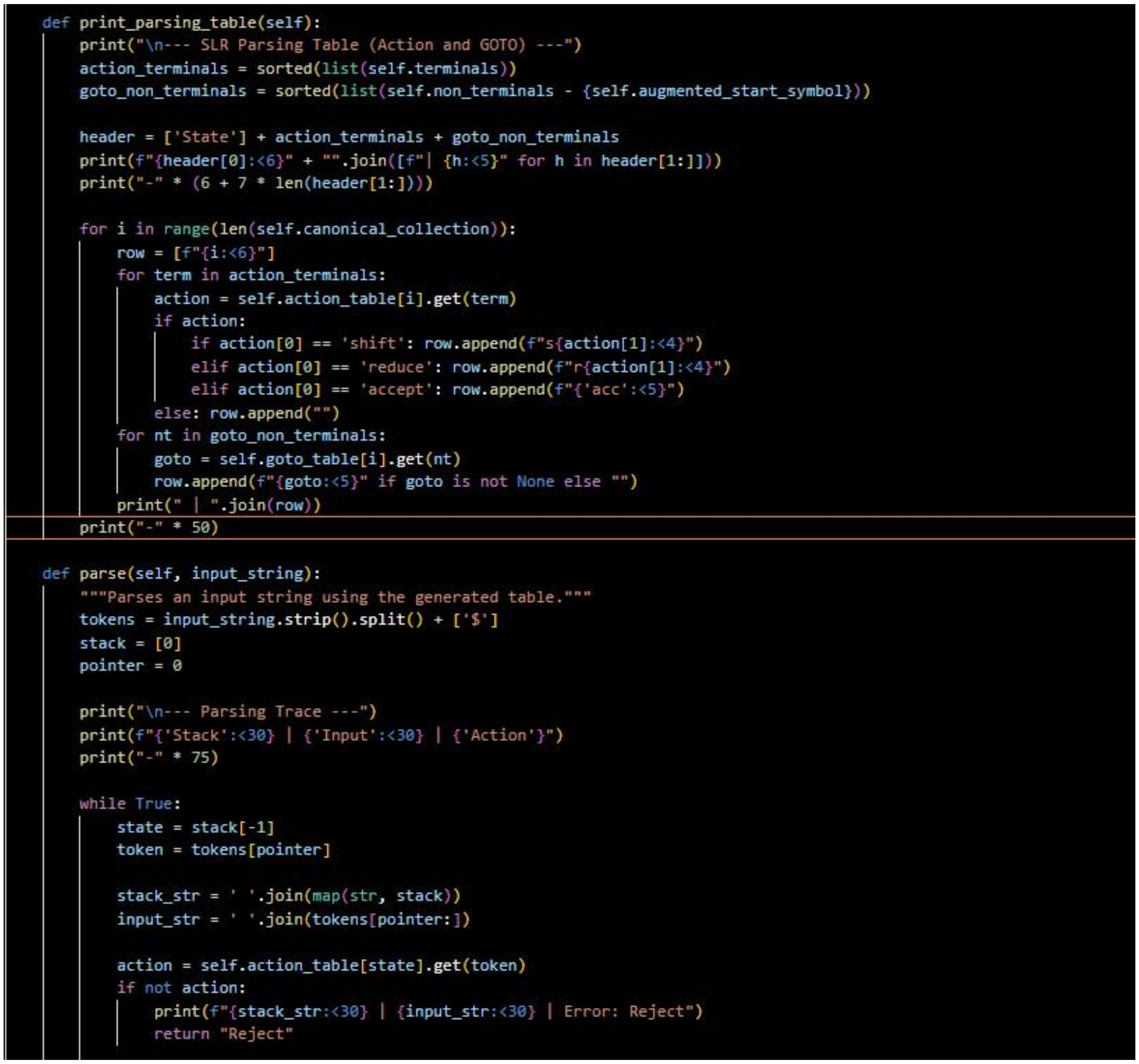
```python
def _compute_first_and_follow(self):
    """Computes FIRST and FOLLOW sets for all non-terminals."""
    # Initialize FIRST sets
    for nt in self.non_terminals:
        self.first_sets[nt] = set()

    # Iteratively compute FIRST sets
    while True:
        updated = False
        for head, bodies in self.grammar.items():
            for body in bodies:
                for symbol in body:
                    original_size = len(self.first_sets[head])
                    if symbol in self.terminals:
                        self.first_sets[head].add(symbol)
                        break
                    else: # Non-terminal
                        self.first_sets[head].update(self.first_sets[symbol])
                        if 'epsilon' not in self.first_sets[symbol]:
                            break
                if len(self.first_sets[head]) > original_size: updated = True
        if not updated: break

    # Initialize FOLLOW sets
    for nt in self.non_terminals:
        self.follow_sets[nt] = set()
    self.follow_sets[self.start_symbol].add('$')

    # Iteratively compute FOLLOW sets
    while True:
        updated = False
        for head, bodies in self.grammar.items():
            for body in bodies:
                for i, symbol in enumerate(body):
                    if symbol in self.non_terminals:
                        original_size = len(self.follow_sets[symbol])
                        # Check symbols that follow
                        trailer = body[i+1:]
                        if trailer:
                            first_of_trailer = set()
                            for t_symbol in trailer:
                                if t_symbol in self.terminals:
                                    first_of_trailer.add(t_symbol)
                                    break
                                else:
                                    first_of_trailer.update(self.first_sets[t_symbol])
                                    if 'epsilon' not in self.first_sets[t_symbol]:
                                        break
                            self.follow_sets[symbol].update(first_of_trailer)
                        else: # Nothing follows, so FOLLOW(symbol) includes FOLLOW(head)
                            self.follow_sets[symbol].update(self.follow_sets[head])
                        if len(self.follow_sets[symbol]) > original_size: updated = True
        if not updated: break
```



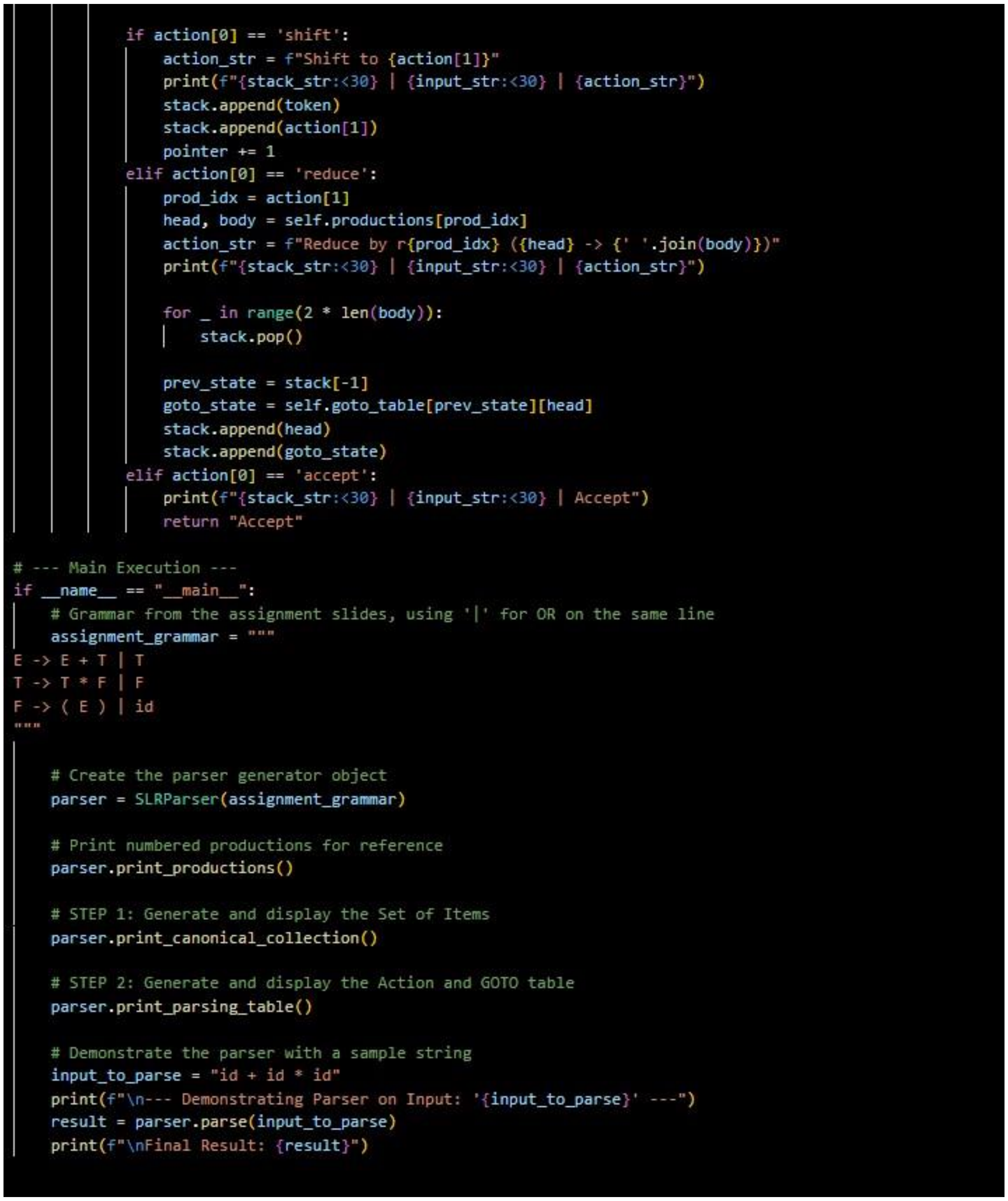
```python
def _closure(self, items):
    """Computes the closure of a set of LR(0) items."""
    closure_set = set(items)
    worklist = list(items)
    while worklist:
        head, body, dot_pos = worklist.pop(0)
        if dot_pos < len(body):
            symbol_after_dot = body[dot_pos]
            if symbol_after_dot in self.non_terminals:
                for prod_body in self.grammar.get(symbol_after_dot, []):
                    new_item = (symbol_after_dot, tuple(prod_body), 0)
                    if new_item not in closure_set:
                        closure_set.add(new_item)
                        worklist.append(new_item)
    return frozenset(closure_set)

def _goto(self, items, symbol):
    """Computes the GOTO set for a set of items and a grammar symbol."""
    new_items = set()
    for head, body, dot_pos in items:
        if dot_pos < len(body) and body[dot_pos] == symbol:
            new_items.add((head, body, dot_pos + 1))
    return self._closure(new_items)

def _build_canonical_collection(self):
    """TASK 1: Builds the canonical collection of LR(0) items."""
    initial_item = (self.augmented_start_symbol, self.productions[0][1], 0)
    initial_state = self._closure({initial_item})

    self.canonical_collection = [initial_state]
    worklist = [initial_state]

    while worklist:
        current_state = worklist.pop(0)
        current_state_idx = self.canonical_collection.index(current_state)

        all_symbols = self.terminals.union(self.non_terminals) - {'$'}
        for symbol in sorted(list(all_symbols)):
            next_state = self._goto(current_state, symbol)
            if next_state:
                if next_state not in self.canonical_collection:
                    self.canonical_collection.append(next_state)
                    worklist.append(next_state)

                next_state_idx = self.canonical_collection.index(next_state)
                self.goto_map[(current_state_idx, symbol)] = next_state_idx
```



```python
def _build_parsing_table(self):
    """TASK 2: Builds the SLR ACTION and GOTO tables."""
    for i in range(len(self.canonical_collection)):
        self.action_table[i] = {}
        self.goto_table[i] = {}

    for i, state in enumerate(self.canonical_collection):
        # GOTO entries (for non-terminals)
        for nt in self.non_terminals:
            if (i, nt) in self.goto_map:
                self.goto_table[i][nt] = self.goto_map[(i, nt)]

        for item in state:
            head, body, dot_pos = item
            # SHIFT actions
            if dot_pos < len(body):
                symbol = body[dot_pos]
                if symbol in self.terminals and (i, symbol) in self.goto_map:
                    target_state = self.goto_map[(i, symbol)]
                    self.action_table[i][symbol] = ('shift', target_state)
            # REDUCE and ACCEPT actions
            else:
                if head == self.augmented_start_symbol:
                    self.action_table[i]['$'] = ('accept', None)
                else:
                    prod_idx = self.productions.index((head, body))
                    for term in self.follow_sets[head]:
                        self.action_table[i][term] = ('reduce', prod_idx)

def print_productions(self):
    print("--- Grammar Productions ---")
    for i, (head, body) in enumerate(self.productions):
        print(f"{i}: {head} -> {' '.join(body)}")
    print("-" * 25)

def print_canonical_collection(self):
    print("\n--- Canonical LR(0) Collection (Set of Items) ---")
    for i, state in enumerate(self.canonical_collection):
        print(f"I{i}:")
        # Sort for consistent output
        sorted_items = sorted(list(state), key=lambda x: (x[0], x[1]))
        for head, body, dot_pos in sorted_items:
            body_with_dot = list(body)
            body_with_dot.insert(dot_pos, '.')
            print(f"  {head} -> {' '.join(body_with_dot)}")
    print("-" * 50)
```



```
def print_parsing_table(self):
    print("\n--- SLR Parsing Table (Action and GOTO) ---")
    action_terminals = sorted(list(self.terminals))
    goto_non_terminals = sorted(list(self.non_terminals - {self.augmented_start_symbol}))

    header = ['State'] + action_terminals + goto_non_terminals
    print(f"{header[0]:<6}" + "".join([f"| {h:<5}" for h in header[1:]]))
    print("-" * (6 + 7 * len(header[1:])))

    for i in range(len(self.canonical_collection)):
        row = [f"{i:<6}"]
        for term in action_terminals:
            action = self.action_table[i].get(term)
            if action:
                if action[0] == 'shift': row.append(f"s{action[1]:<4}")
                elif action[0] == 'reduce': row.append(f"r{action[1]:<4}")
                elif action[0] == 'accept': row.append(f"{'acc':<5}")
            else: row.append("")
        for nt in goto_non_terminals:
            goto = self.goto_table[i].get(nt)
            row.append(f"{goto:<5}" if goto is not None else "")
        print(" | ".join(row))
    print("-" * 50)

def parse(self, input_string):
    """Parses an input string using the generated table."""
    tokens = input_string.strip().split() + ['$']
    stack = [0]
    pointer = 0

    print("\n--- Parsing Trace ---")
    print(f"{'Stack':<30} | {'Input':<30} | {'Action'}")
    print("-" * 75)

    while True:
        state = stack[-1]
        token = tokens[pointer]

        stack_str = ' '.join(map(str, stack))
        input_str = ' '.join(tokens[pointer:])

        action = self.action_table[state].get(token)
        if not action:
            print(f"{stack_str:<30} | {input_str:<30} | Error: Reject")
            return "Reject"
```



```python
            if action[0] == 'shift':
                action_str = f"Shift to {action[1]}"
                print(f"{stack_str:<30} | {input_str:<30} | {action_str}")
                stack.append(token)
                stack.append(action[1])
                pointer += 1
            elif action[0] == 'reduce':
                prod_idx = action[1]
                head, body = self.productions[prod_idx]
                action_str = f"Reduce by r{prod_idx} ({head} -> {' '.join(body)})"
                print(f"{stack_str:<30} | {input_str:<30} | {action_str}")

                for _ in range(2 * len(body)):
                    stack.pop()

                prev_state = stack[-1]
                goto_state = self.goto_table[prev_state][head]
                stack.append(head)
                stack.append(goto_state)
            elif action[0] == 'accept':
                print(f"{stack_str:<30} | {input_str:<30} | Accept")
                return "Accept"

# --- Main Execution ---
if __name__ == "__main__":
    # Grammar from the assignment slides, using '|' for OR on the same line
    assignment_grammar = """
E -> E + T | T
T -> T * F | F
F -> ( E ) | id
"""

    # Create the parser generator object
    parser = SLRParser(assignment_grammar)

    # Print numbered productions for reference
    parser.print_productions()

    # STEP 1: Generate and display the Set of Items
    parser.print_canonical_collection()

    # STEP 2: Generate and display the Action and GOTO table
    parser.print_parsing_table()

    # Demonstrate the parser with a sample string
    input_to_parse = "id + id * id"
    print(f"\n--- Demonstrating Parser on Input: '{input_to_parse}' ---")
    result = parser.parse(input_to_parse)
    print(f"\nFinal Result: {result}")
```

Program Output and Results

```
--- Grammar Productions ---
0: E' -> E
1: E -> E + T
2: E -> T
3: T -> T * F
4: T -> F
5: F -> ( E )
6: F -> id
-------------------------

--- Canonical LR(0) Collection (Set of Items) ---
I0:
  E -> . E + T
  E -> . T
  E' -> . E
  F -> . ( E )
  F -> . id
  T -> . F
  T -> . T * F
I1:
  E -> . E + T
  E -> . T
  F -> ( . E )
  F -> . ( E )
  F -> . id
  T -> . F
  T -> . T * F
I2:
  E -> E . + T
  E' -> E .
I3:
  T -> F .
I4:
  E -> T .
  T -> T . * F
I5:
  F -> id .
I6:
  E -> E . + T
  F -> ( E . )
I7:
  E -> E + . T
  F -> . ( E )
  F -> . id
  T -> . F
  T -> . T * F
I8:
  F -> . ( E )
  F -> . id
  T -> T * . F
I9:
  F -> ( E ) .
I10:
  E -> E + T .
  T -> T . * F
I11:
  T -> T * F .
--------------------------------------------------

--- SLR Parsing Table (Action and GOTO) ---
State | $    | (    | )    | *    | +    | id   | E    | F    | T
-------------------------------------------------------------------
0     |  | s1   |  |  |  | s5   | 2    | 3    | 4
1     |  | s1   |  |  |  | s5   | 6    | 3    | 4
2     | acc   |  |  |  | s7   |  |  |  |
3     | r4   |  | r4   | r4   | r4   |  |  |  |
4     | r2   |  | r2   | s8   | r2   |  |  |  |
5     | r6   |  | r6   | r6   | r6   |  |  |  |
6     |  |  | s9   |  | s7   |  |  |  |
7     |  | s1   |  |  |  | s5   |  | 3    | 10
8     |  | s1   |  |  |  | s5   |  | 11   |
9     | r5   |  | r5   | r5   | r5   |  |  |  |
10    | r1   |  | r1   | s8   | r1   |  |  |  |
11    | r3   |  | r3   | r3   | r3   |  |  |  |
--------------------------------------------------

--- Demonstrating Parser on Input: 'id + id * id' ---

--- Parsing Trace ---
Stack                    | Input                    | Action
```

```
---------------------------------------------------------------------------
0                       | id + id * id $           | Shift to 5
0 id 5                  | + id * id $              | Reduce by r6 (F -> id)
0 F 3                   | + id * id $              | Reduce by r4 (T -> F)
0 T 4                   | + id * id $              | Reduce by r2 (E -> T)
0 E 2                   | + id * id $              | Shift to 7
0 E 2 + 7               | id * id $                | Shift to 5
0 E 2 + 7 id 5          | * id $                   | Reduce by r6 (F -> id)
0 E 2 + 7 F 3           | * id $                   | Reduce by r4 (T -> F)
0 E 2 + 7 T 10          | * id $                   | Shift to 8
0 E 2 + 7 T 10 * 8      | id $                     | Shift to 5
0 E 2 + 7 T 10 * 8 id 5 | $                        | Reduce by r6 (F -> id)
0 E 2 + 7 T 10 * 8 F 11 | $                        | Reduce by r3 (T -> T * F)
0 E 2 + 7 T 10          | $                        | Reduce by r1 (E -> E + T)
0 E 2                   | $                        | Accept

Final Result: Accept
```

Generic Program

1. Objective

The objective of this assignment is to implement the "OR" option: a generic SLR parser generator. This program is designed as a reusable tool that can:

1. Accept any valid SLR-compatible context-free grammar from the user.
2. Dynamically generate the Canonical LR(0) Collection of Items based on that grammar.
3. Construct the corresponding SLR ACTION and GOTO parsing tables.
4. Use the generated tables to parse an input string (also provided by the user) and determine if it is valid.

This approach demonstrates a more comprehensive understanding of the parsing process by creating a flexible tool rather than a single-purpose parser.

2. Methodology and Code Explanation

The parser generator is implemented in Python within a single class, SLRParser. The logic is designed to be independent of any specific grammar, allowing it to adapt to user input.

2.1. Dynamic Grammar Processing

The initial phase of the program is to understand the user-provided grammar.

- _parse_grammar(grammar_str): This method is built to handle a grammar format where productions are entered line by line. It automatically identifies the start symbol from the first production, augments the grammar (e.g., S' -> S), and dynamically identifies all terminal and non-terminal symbols. This flexibility is key to its generic nature.
- _compute_first_and_follow(): The algorithms to compute FIRST and FOLLOW sets are inherently generic and work for any context-free grammar. These are implemented as iterative algorithms that continue until no new symbols can be added to any set, ensuring they are correct for the specific grammar provided.

2.2. Task 1: Generating the Set of Items

The program dynamically generates the states of the parser's finite automaton from the user's grammar.

- _build_canonical_collection(): This core method is entirely data-driven. It begins with the closure of the augmented start symbol of the current grammar. Using the _closure and _goto helper functions, it explores all possible state transitions for all symbols found in the user's grammar, building the complete Canonical LR(0) Collection from scratch for each run.

2.3. Task 2: Generating the Parsing Table

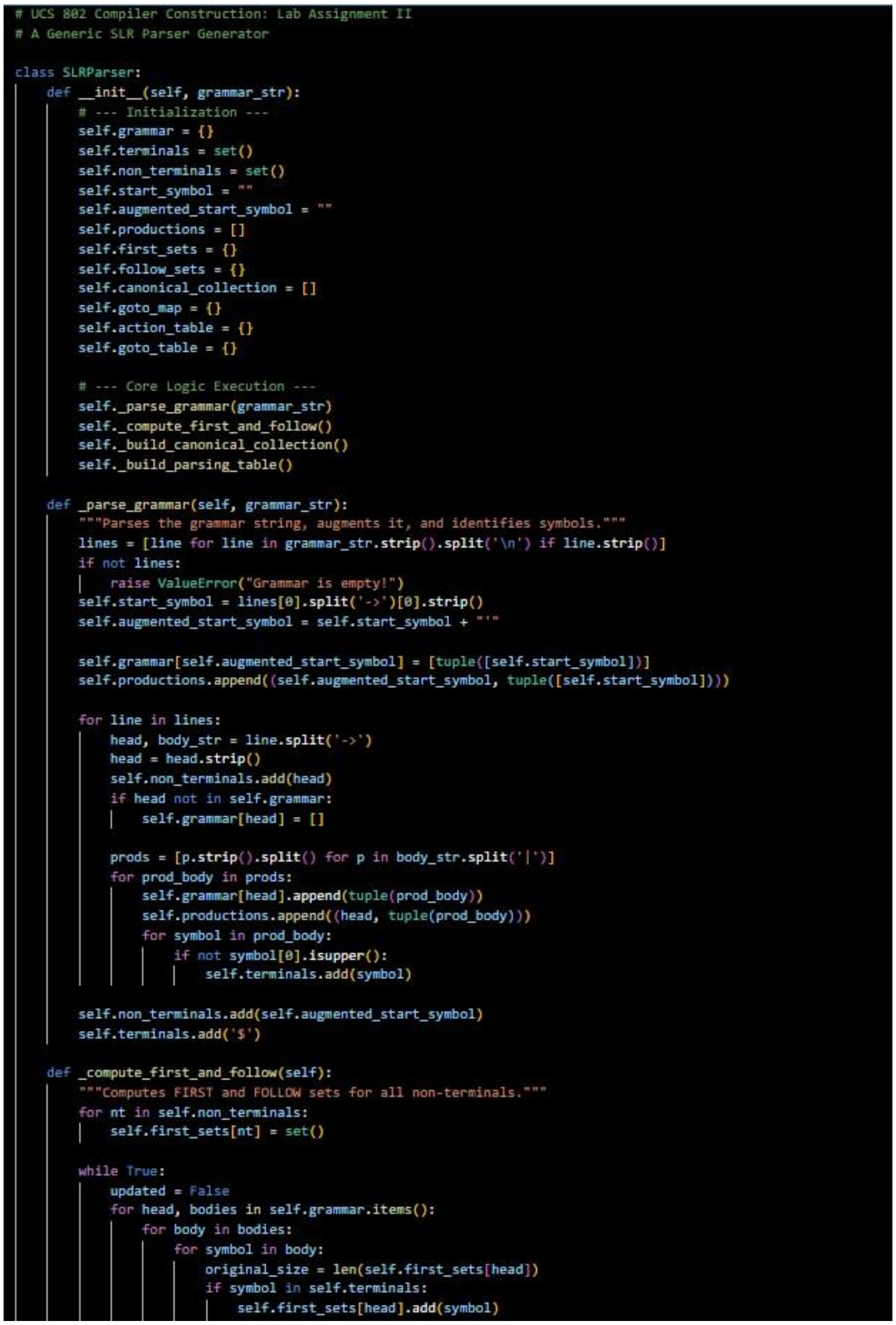
The construction of the ACTION and GOTO tables is also fully dynamic.

- _build_parsing_table(): This method populates the parsing tables based on the item sets generated in the previous step and the FOLLOW sets calculated earlier. It iterates through each generated state and applies the standard SLR table construction rules for Shift, Reduce, and Accept actions. Crucially, it includes logic to detect and report any shift-reduce or reduce-reduce conflicts that might arise, informing the user if their provided grammar is not SLR(1).

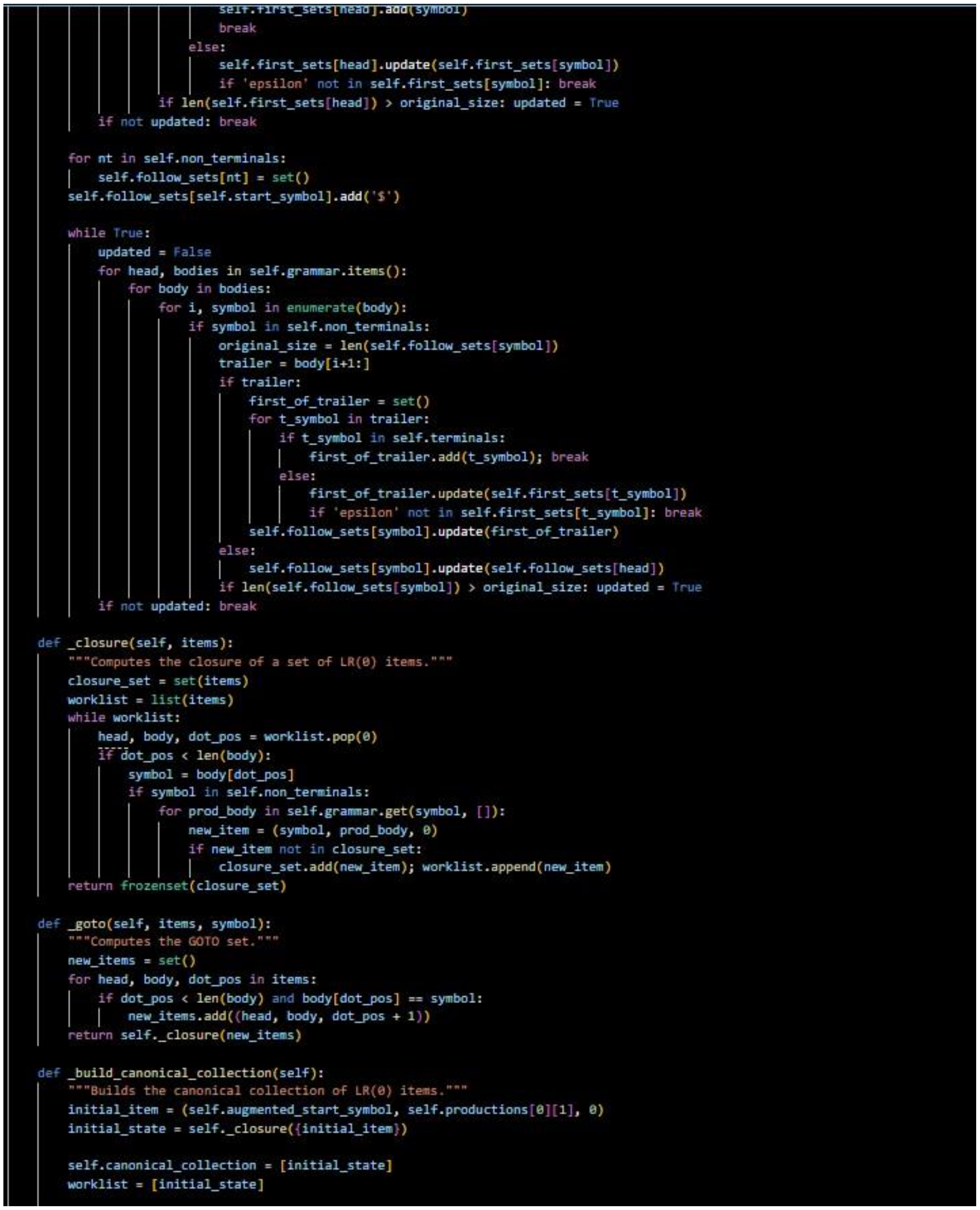
2.4. Interactive Parsing

The final part of the program demonstrates its utility as a tool.

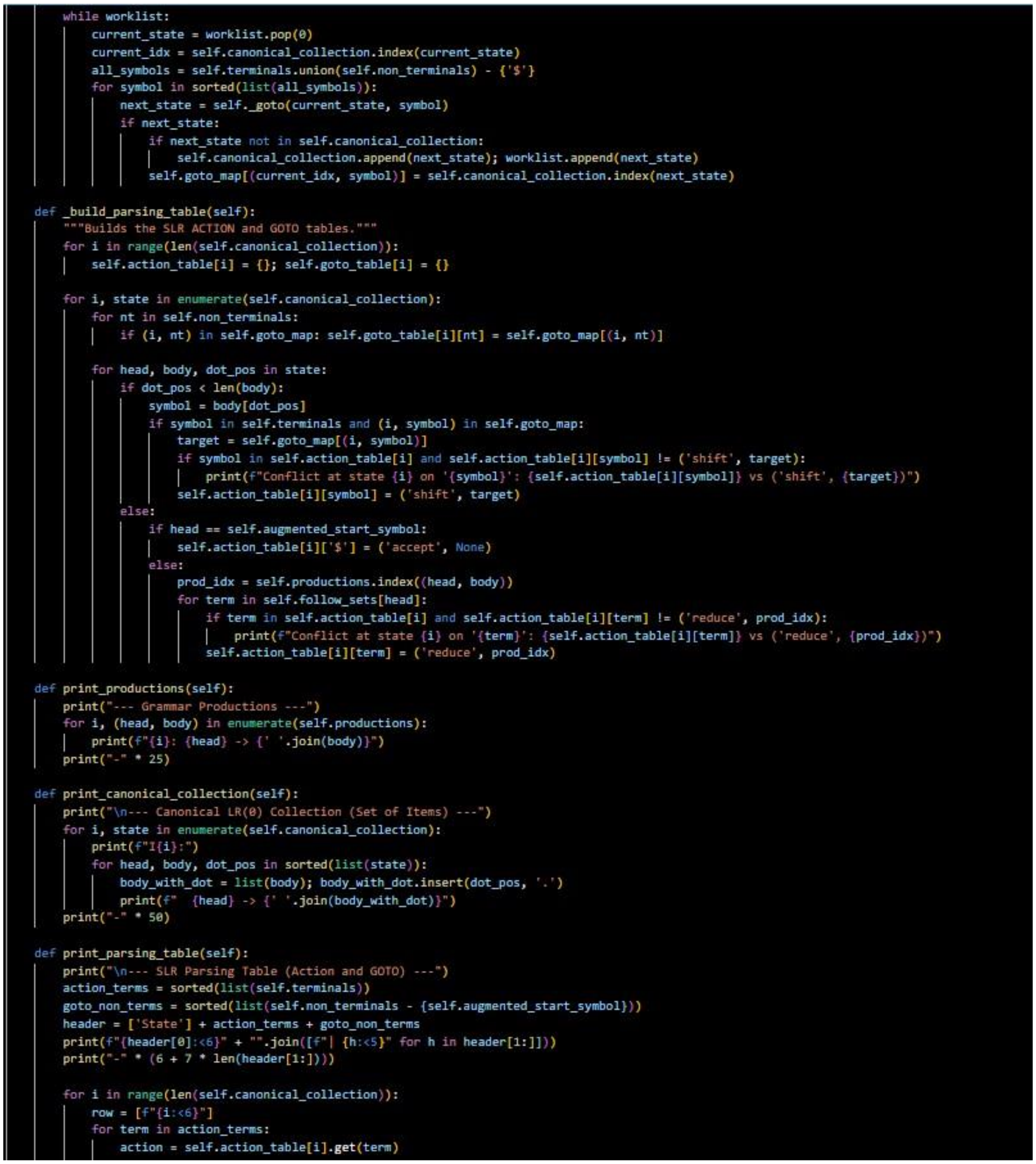
- parse(input_string): The shift-reduce parsing algorithm is generic. It simply needs a valid parsing table and an input string to function. This method provides a step-by-step trace of its operations for transparency.
- main Execution Block: The main part of the script is designed for user interaction. It prompts the user to enter their grammar, followed by a string to parse. It then instantiates the SLRParser class with the user's grammar, triggering the entire generation process, and finally calls the parse method with the user's string. This interactive loop makes it a practical tool for testing different grammars.



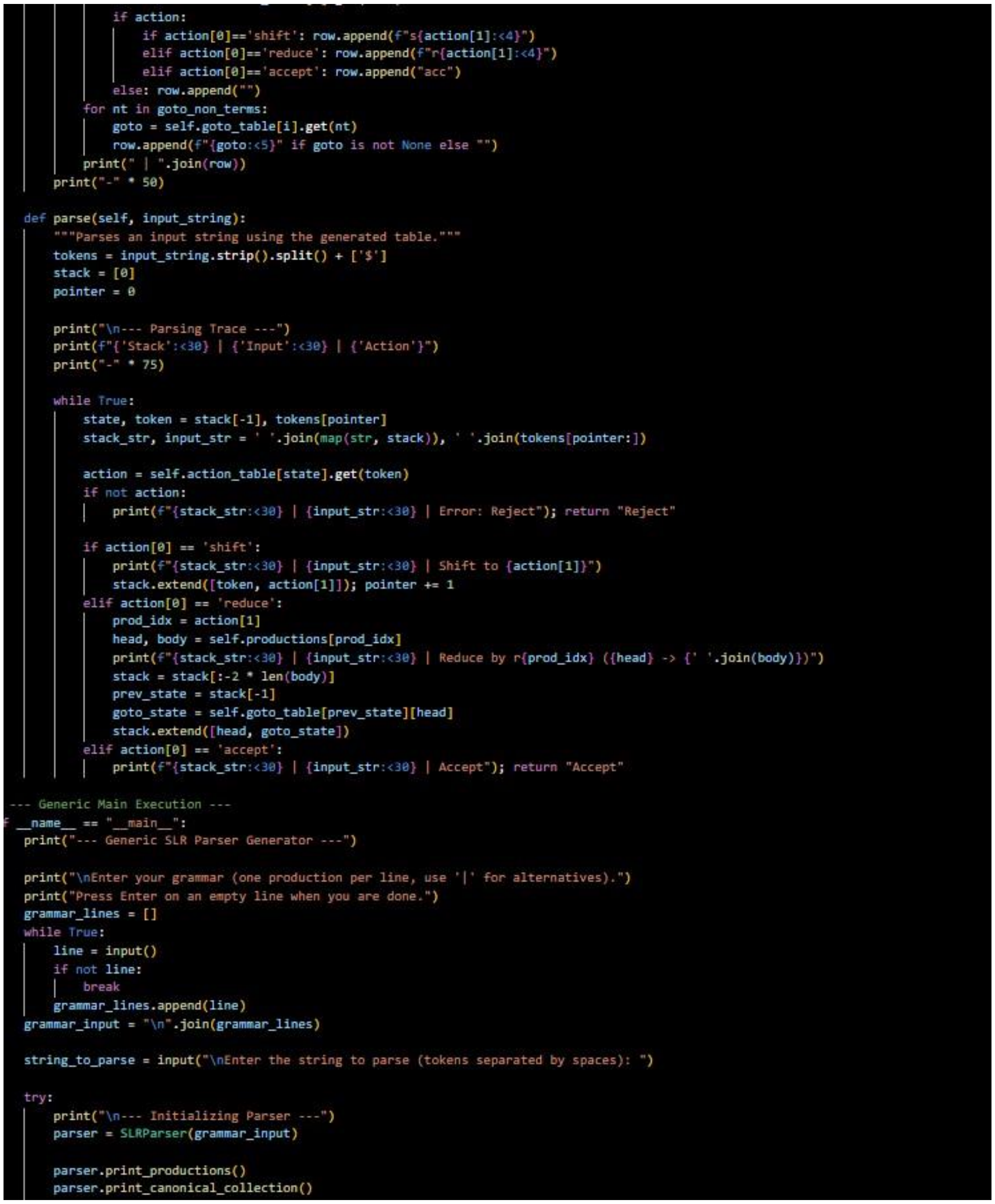
```python
# UCS 802 Compiler Construction: Lab Assignment II
# A Generic SLR Parser Generator

class SLRParser:
    def __init__(self, grammar_str):
        # --- Initialization ---
        self.grammar = {}
        self.terminals = set()
        self.non_terminals = set()
        self.start_symbol = ""
        self.augmented_start_symbol = ""
        self.productions = []
        self.first_sets = {}
        self.follow_sets = {}
        self.canonical_collection = []
        self.goto_map = {}
        self.action_table = {}
        self.goto_table = {}

        # --- Core Logic Execution ---
        self._parse_grammar(grammar_str)
        self._compute_first_and_follow()
        self._build_canonical_collection()
        self._build_parsing_table()

    def _parse_grammar(self, grammar_str):
        """Parses the grammar string, augments it, and identifies symbols."""
        lines = [line for line in grammar_str.strip().split('\n') if line.strip()]
        if not lines:
            raise ValueError("Grammar is empty!")
        self.start_symbol = lines[0].split('->')[0].strip()
        self.augmented_start_symbol = self.start_symbol + "'"

        self.grammar[self.augmented_start_symbol] = [tuple([self.start_symbol])]
        self.productions.append((self.augmented_start_symbol, tuple([self.start_symbol])))

        for line in lines:
            head, body_str = line.split('->')
            head = head.strip()
            self.non_terminals.add(head)
            if head not in self.grammar:
                self.grammar[head] = []

            prods = [p.strip().split() for p in body_str.split('|')]
            for prod_body in prods:
                self.grammar[head].append(tuple(prod_body))
                self.productions.append((head, tuple(prod_body)))
                for symbol in prod_body:
                    if not symbol[0].isupper():
                        self.terminals.add(symbol)

        self.non_terminals.add(self.augmented_start_symbol)
        self.terminals.add('$')

    def _compute_first_and_follow(self):
        """Computes FIRST and FOLLOW sets for all non-terminals."""
        for nt in self.non_terminals:
            self.first_sets[nt] = set()

        while True:
            updated = False
            for head, bodies in self.grammar.items():
                for body in bodies:
                    for symbol in body:
                        original_size = len(self.first_sets[head])
                        if symbol in self.terminals:
                            self.first_sets[head].add(symbol)
```



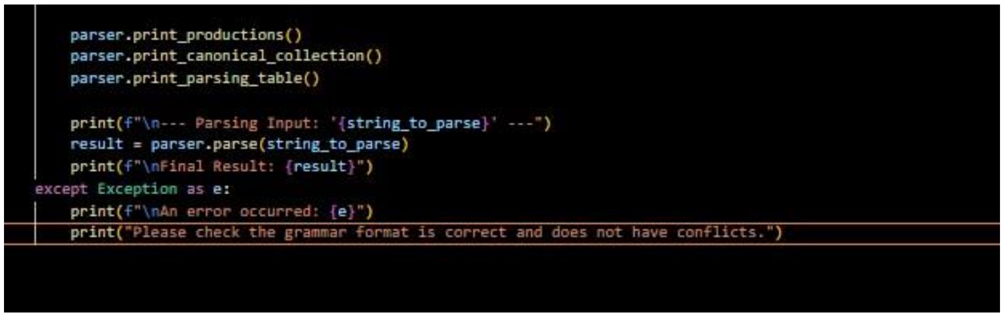
```
                            self.first_sets[head].add(symbol)
                            break
                        else:
                            self.first_sets[head].update(self.first_sets[symbol])
                            if 'epsilon' not in self.first_sets[symbol]: break
                    if len(self.first_sets[head]) > original_size: updated = True
            if not updated: break

        for nt in self.non_terminals:
            self.follow_sets[nt] = set()
        self.follow_sets[self.start_symbol].add('$')

        while True:
            updated = False
            for head, bodies in self.grammar.items():
                for body in bodies:
                    for i, symbol in enumerate(body):
                        if symbol in self.non_terminals:
                            original_size = len(self.follow_sets[symbol])
                            trailer = body[i+1:]
                            if trailer:
                                first_of_trailer = set()
                                for t_symbol in trailer:
                                    if t_symbol in self.terminals:
                                        first_of_trailer.add(t_symbol); break
                                    else:
                                        first_of_trailer.update(self.first_sets[t_symbol])
                                        if 'epsilon' not in self.first_sets[t_symbol]: break
                                self.follow_sets[symbol].update(first_of_trailer)
                            else:
                                self.follow_sets[symbol].update(self.follow_sets[head])
                            if len(self.follow_sets[symbol]) > original_size: updated = True
            if not updated: break

    def _closure(self, items):
        """Computes the closure of a set of LR(0) items."""
        closure_set = set(items)
        worklist = list(items)
        while worklist:
            head, body, dot_pos = worklist.pop(0)
            if dot_pos < len(body):
                symbol = body[dot_pos]
                if symbol in self.non_terminals:
                    for prod_body in self.grammar.get(symbol, []):
                        new_item = (symbol, prod_body, 0)
                        if new_item not in closure_set:
                            closure_set.add(new_item); worklist.append(new_item)
        return frozenset(closure_set)

    def _goto(self, items, symbol):
        """Computes the GOTO set."""
        new_items = set()
        for head, body, dot_pos in items:
            if dot_pos < len(body) and body[dot_pos] == symbol:
                new_items.add((head, body, dot_pos + 1))
        return self._closure(new_items)

    def _build_canonical_collection(self):
        """Builds the canonical collection of LR(0) items."""
        initial_item = (self.augmented_start_symbol, self.productions[0][1], 0)
        initial_state = self._closure({initial_item})

        self.canonical_collection = [initial_state]
        worklist = [initial_state]
```



```python
        while worklist:
            current_state = worklist.pop(0)
            current_idx = self.canonical_collection.index(current_state)
            all_symbols = self.terminals.union(self.non_terminals) - {'$'}
            for symbol in sorted(list(all_symbols)):
                next_state = self._goto(current_state, symbol)
                if next_state:
                    if next_state not in self.canonical_collection:
                        self.canonical_collection.append(next_state); worklist.append(next_state)
                    self.goto_map[(current_idx, symbol)] = self.canonical_collection.index(next_state)

    def _build_parsing_table(self):
        """Builds the SLR ACTION and GOTO tables."""
        for i in range(len(self.canonical_collection)):
            self.action_table[i] = {}; self.goto_table[i] = {}

        for i, state in enumerate(self.canonical_collection):
            for nt in self.non_terminals:
                if (i, nt) in self.goto_map: self.goto_table[i][nt] = self.goto_map[(i, nt)]

            for head, body, dot_pos in state:
                if dot_pos < len(body):
                    symbol = body[dot_pos]
                    if symbol in self.terminals and (i, symbol) in self.goto_map:
                        target = self.goto_map[(i, symbol)]
                        if symbol in self.action_table[i] and self.action_table[i][symbol] != ('shift', target):
                            print(f"Conflict at state {i} on '{symbol}': {self.action_table[i][symbol]} vs ('shift', {target})")
                        self.action_table[i][symbol] = ('shift', target)
                else:
                    if head == self.augmented_start_symbol:
                        self.action_table[i]['$'] = ('accept', None)
                    else:
                        prod_idx = self.productions.index((head, body))
                        for term in self.follow_sets[head]:
                            if term in self.action_table[i] and self.action_table[i][term] != ('reduce', prod_idx):
                                print(f"Conflict at state {i} on '{term}': {self.action_table[i][term]} vs ('reduce', {prod_idx})")
                            self.action_table[i][term] = ('reduce', prod_idx)

    def print_productions(self):
        print("--- Grammar Productions ---")
        for i, (head, body) in enumerate(self.productions):
            print(f"{i}: {head} -> {' '.join(body)}")
        print("-" * 25)

    def print_canonical_collection(self):
        print("\n--- Canonical LR(0) Collection (Set of Items) ---")
        for i, state in enumerate(self.canonical_collection):
            print(f"I{i}:")
            for head, body, dot_pos in sorted(list(state)):
                body_with_dot = list(body); body_with_dot.insert(dot_pos, '.')
                print(f"  {head} -> {' '.join(body_with_dot)}")
        print("-" * 50)

    def print_parsing_table(self):
        print("\n--- SLR Parsing Table (Action and GOTO) ---")
        action_terms = sorted(list(self.terminals))
        goto_non_terms = sorted(list(self.non_terminals - {self.augmented_start_symbol}))
        header = ['State'] + action_terms + goto_non_terms
        print(f"{header[0]:<6}" + "".join([f"| {h:<5}" for h in header[1:]]))
        print("-" * (6 + 7 * len(header[1:])))

        for i in range(len(self.canonical_collection)):
            row = [f"{i:<6}"]
            for term in action_terms:
                action = self.action_table[i].get(term)
```



```
                if action:
                    if action[0]=='shift': row.append(f"s{action[1]:<4}")
                    elif action[0]=='reduce': row.append(f"r{action[1]:<4}")
                    elif action[0]=='accept': row.append("acc")
                else: row.append("")
            for nt in goto_non_terms:
                goto = self.goto_table[i].get(nt)
                row.append(f"{goto:<5}" if goto is not None else "")
            print(" | ".join(row))
        print("-" * 50)

    def parse(self, input_string):
        """Parses an input string using the generated table."""
        tokens = input_string.strip().split() + ['$']
        stack = [0]
        pointer = 0

        print("\n--- Parsing Trace ---")
        print(f"{'Stack':<30} | {'Input':<30} | {'Action'}")
        print("-" * 75)

        while True:
            state, token = stack[-1], tokens[pointer]
            stack_str, input_str = ' '.join(map(str, stack)), ' '.join(tokens[pointer:])

            action = self.action_table[state].get(token)
            if not action:
                print(f"{stack_str:<30} | {input_str:<30} | Error: Reject"); return "Reject"

            if action[0] == 'shift':
                print(f"{stack_str:<30} | {input_str:<30} | Shift to {action[1]}")
                stack.extend([token, action[1]]); pointer += 1
            elif action[0] == 'reduce':
                prod_idx = action[1]
                head, body = self.productions[prod_idx]
                print(f"{stack_str:<30} | {input_str:<30} | Reduce by r{prod_idx} ({head} -> {' '.join(body)})")
                stack = stack[:-2 * len(body)]
                prev_state = stack[-1]
                goto_state = self.goto_table[prev_state][head]
                stack.extend([head, goto_state])
            elif action[0] == 'accept':
                print(f"{stack_str:<30} | {input_str:<30} | Accept"); return "Accept"

--- Generic Main Execution ---
f __name__ == "__main__":
    print("--- Generic SLR Parser Generator ---")

    print("\nEnter your grammar (one production per line, use '|' for alternatives).")
    print("Press Enter on an empty line when you are done.")
    grammar_lines = []
    while True:
        line = input()
        if not line:
            break
        grammar_lines.append(line)
    grammar_input = "\n".join(grammar_lines)

    string_to_parse = input("\nEnter the string to parse (tokens separated by spaces): ")

    try:
        print("\n--- Initializing Parser ---")
        parser = SLRParser(grammar_input)

        parser.print_productions()
        parser.print_canonical_collection()
```



```
        parser.print_productions()
        parser.print_canonical_collection()
        parser.print_parsing_table()

        print(f"\n--- Parsing Input: '{string_to_parse}' ---")
        result = parser.parse(string_to_parse)
        print(f"\nFinal Result: {result}")
    except Exception as e:
        print(f"\nAn error occurred: {e}")
        print("Please check the grammar format is correct and does not have conflicts.")
```

Program Output:

```
--- Generic SLR Parser Generator ---
Enter your grammar (one production per line, use '|' for alternatives).
Press Enter on an empty line when you are done.
S -> ( S ) | a
Enter the string to parse (tokens separated by spaces): ( a )
--- Initializing Parser ---
--- Grammar Productions ---
0: S' -> S
1: S -> ( S )
2: S -> a
-------------------------
--- Canonical LR(0) Collection (Set of Items) ---
I0:
  S -> . ( S )
  S -> . a
  S' -> . S
I1:
  S -> . ( S )
  S -> ( . S )
  S -> . a
I2:
  S' -> S .
I3:
  S -> a .
I4:
  S -> ( S . )
I5:
  S -> ( S ) .
--------------------------------------------------
--- SLR Parsing Table (Action and GOTO) ---
State | $    | (    | )    | a    | S
----------------------------------------
0     |  | s1    |  | s3    | 2
1     |  | s1    |  | s3    | 4
2     | acc |  |  |  |
3     | r2    |  | r2    |  |
4     |  |  | s5    |  |
4     |  |  | s5    |  |
5     | r1    |  | r1    |  |
--------------------------------------------------
--------------------------------------------------
--- Parsing Input: '( a )' ---
--- Parsing Trace ---
--- Parsing Input: '( a )' ---
--- Parsing Trace ---
Stack                    | Input                    | Action
--------------------------------------------------------------------------


--- Parsing Trace ---
Stack                    | Input                    | Action
--------------------------------------------------------------------------
0                        | ( a ) $                  | Shift to 1
```

```
0 ( 1                | a ) $              | Shift to 3
0 ( 1 a 3            | ) $                | Reduce by r2 (S -> a)
0 ( 1 S 4            | ) $                | Shift to 5
0 ( 1 S 4 ) 5        | $                  | Reduce by r1 (S -> ( S ))
Stack                | Input              | Action
-----------------------------------------------------------------------
0                    | ( a ) $            | Shift to 1
0 ( 1                | a ) $              | Shift to 3
0 ( 1 a 3            | ) $                | Reduce by r2 (S -> a)
0 ( 1 S 4            | ) $                | Shift to 5
0 ( 1 S 4 ) 5        | $                  | Reduce by r1 (S -> ( S ))
0                    | ( a ) $            | Shift to 1
0 ( 1                | a ) $              | Shift to 3
0 ( 1 a 3            | ) $                | Reduce by r2 (S -> a)
0 ( 1 S 4            | ) $                | Shift to 5
0 ( 1 S 4 ) 5        | $                  | Reduce by r1 (S -> ( S ))
0 ( 1 S 4            | ) $                | Shift to 5
0 ( 1 S 4 ) 5        | $                  | Reduce by r1 (S -> ( S ))
0 S 2                | $                  | Accept

0 S 2                | $                  | Accept
Final Result: Accept
```