Project 2 Report

The Python program analyzes symbol sequences according to grammatical rules to illustrate parsing, which is crucial in language processing. It guesses the next input symbol using a predictive parsing algorithm, with the help of a context-free grammar (CFG) that specifies appropriate language structures. In order to facilitate symbol replacement, production rules are linked to non-terminals (E, Q, T, R, and F) and terminal symbols (+, -, *, /, (,), and $). By associating input symbols with production rules, a parsing table expedites the procedure.

A stack containing the dollar sign ($) and the start symbol (E) is where the parsing process starts. After going over the input one more time, it modifies the stack in response to matches or parsing table consultation. Stack management helps with comprehension, gives immediate feedback, and guarantees compliance with CFG guidelines.

The program takes advantage of control flow techniques and built-in data structures to make it clear and succinct while utilizing the flexibility of Python. This demonstrates how Python may be used for a wide range of tasks, from simple scripting to intricate algorithms.

To put it simply, the software provides a solid basis for language processors by emphasizing CFGs and parsing algorithms. Because of its intuitive design, it is incredibly useful for both real-world software development and educational settings.

1. **Grammar Depiction:** A grammar dictionary is used to represent the grammar rules. A list of potential products for each non-terminal symbol that a key in this dictionary corresponds to is its value.
2. **Parsing Table:** The predictive parser uses a different dictionary called parsing_table as its parsing table. A tuple (non-terminal, terminal) is mapped to the associated production rule. Based on the next input terminal symbol and the current non-terminal symbol, this table is used to determine which production to apply.
3. The parsing function: It accepts an input string and tries to interpret it using the parsing table and grammar rules that are provided. It takes a stack-based methodology.
   - '$' and the start symbol "E" are used to initialize the stack.
   - Both the input string and the stack are iterated through concurrently.
   - It removes a symbol from the stack and compares it with the input symbol being used at each step.
   - It advances to the following symbol in the input if they match.
   - It refers to the parsing table to ascertain which production rule to apply if the symbol that was popped from the stack is non-terminal.
   - In the event that no production rule is discovered, the input string is deemed invalid.

The procedure carries on until an invalid state is detected or the stack is empty and the input string has been entirely processed, indicating that the string is accepted.

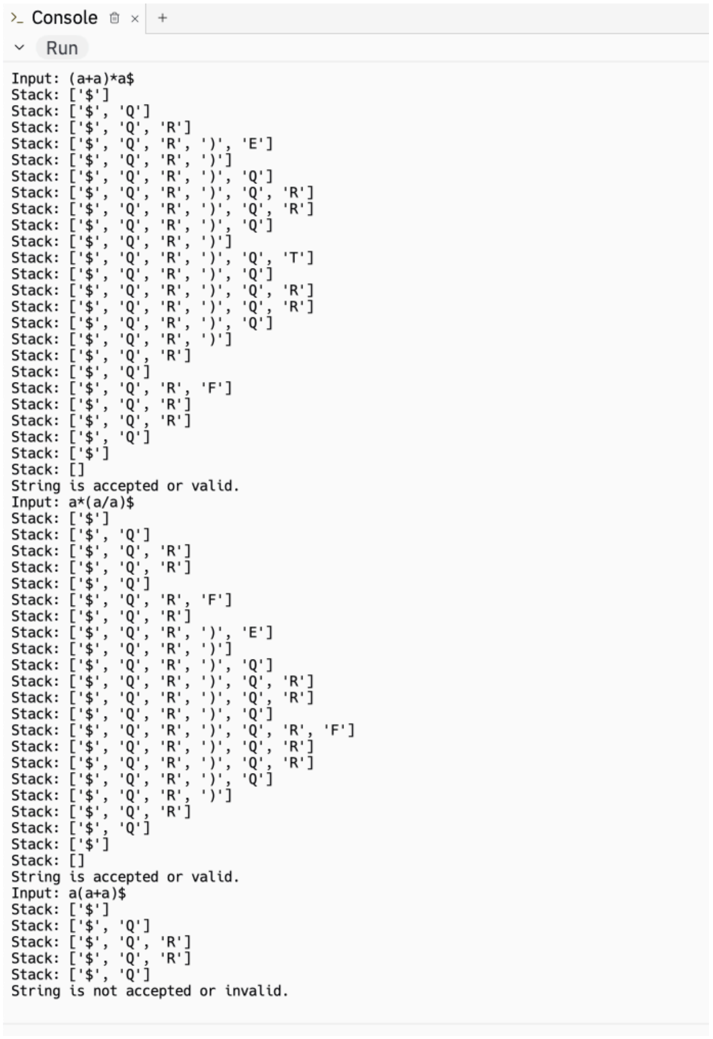
4. **Test Cases:** The test_cases list has three test cases, each of which represents a string that needs to be processed. The parse function receives these test cases and decides whether or not to accept them based on the grammatical rules.

5. **Integrated Features Used:**

- list(), which creates a list of symbols from the production string
- reversed**:** Before expanding the stack, this function flips the order of the symbols in the production list.

6.

7. **Checkpoints:** We've come across problems with the parse function when it checks if the top of the stack is a Non-Terminal Symbol. The problem was how it pushed epsilon, when in fact it is an empty string. We also came across problems in parsing_table where our format was recognizing it as a list of production rather than a single production rule.


Console

Run

```
Input: (a+a)*a$
Stack: ['$']
Stack: ['$', 'Q']
Stack: ['$', 'Q', 'R']
Stack: ['$', 'Q', 'R', ')', 'E']
Stack: ['$', 'Q', 'R', ')']
Stack: ['$', 'Q', 'R', ')', 'Q']
Stack: ['$', 'Q', 'R', ')', 'Q', 'R']
Stack: ['$', 'Q', 'R', ')', 'Q', 'R']
Stack: ['$', 'Q', 'R', ')', 'Q']
Stack: ['$', 'Q', 'R', ')']
Stack: ['$', 'Q', 'R', ')', 'Q', 'T']
Stack: ['$', 'Q', 'R', ')', 'Q']
Stack: ['$', 'Q', 'R', ')', 'Q', 'R']
Stack: ['$', 'Q', 'R', ')', 'Q', 'R']
Stack: ['$', 'Q', 'R', ')', 'Q']
Stack: ['$', 'Q', 'R', ')']
Stack: ['$', 'Q', 'R']
Stack: ['$', 'Q']
Stack: ['$', 'Q', 'R', 'F']
Stack: ['$', 'Q', 'R']
Stack: ['$', 'Q', 'R']
Stack: ['$', 'Q']
Stack: ['$']
Stack: []
String is accepted or valid.
Input: a*(a/a)$
Stack: ['$']
Stack: ['$', 'Q']
Stack: ['$', 'Q', 'R']
Stack: ['$', 'Q', 'R']
Stack: ['$', 'Q']
Stack: ['$', 'Q', 'R', 'F']
Stack: ['$', 'Q', 'R']
Stack: ['$', 'Q', 'R', ')', 'E']
Stack: ['$', 'Q', 'R', ')']
Stack: ['$', 'Q', 'R', ')', 'Q']
Stack: ['$', 'Q', 'R', ')', 'Q', 'R']
Stack: ['$', 'Q', 'R', ')', 'Q', 'R']
Stack: ['$', 'Q', 'R', ')', 'Q']
Stack: ['$', 'Q', 'R', ')', 'Q', 'R', 'F']
Stack: ['$', 'Q', 'R', ')', 'Q', 'R']
Stack: ['$', 'Q', 'R', ')', 'Q', 'R']
Stack: ['$', 'Q', 'R', ')', 'Q']
Stack: ['$', 'Q', 'R', ')']
Stack: ['$', 'Q', 'R']
Stack: ['$', 'Q']
Stack: ['$']
Stack: []
String is accepted or valid.
Input: a(a+a)$
Stack: ['$']
Stack: ['$', 'Q']
Stack: ['$', 'Q', 'R']
Stack: ['$', 'Q', 'R']
Stack: ['$', 'Q']
String is not accepted or invalid.
```