Implementing a Predictive Parser Interpreter Function

Objective

The goal of this assignment is to implement the `interpret` function for a predictive parser. This function will determine if a given sequence of input tokens matches a predefined grammar using a predictive parsing table. The function should manage a stack and use parsing rules to process the input, handling terminal and non-terminal symbols as needed.

Prerequisites

- Make sure **g++** is installed.
- atest version of the course repository.

```
$cd <path>/<to>/CSC355_Student
$git pull origin
```

- Lab5 directory in your repository (e.g., `$CSC355_telim/Labs/Lab5$`).

Details

Predictive parsing is a top-down parsing approach that does not require backtracking. It uses a parsing table to decide which production rules to apply based on the current input token and the top of the stack. In this assignment, you will implement a key part of this parser, the `interpret` function.

Grammar

The grammar we are using in this assignment is defined as follows:

$E \rightarrow TE'$
$E' \rightarrow +TE'|\epsilon$
$T \rightarrow FT'$
$T' \rightarrow *FT'|\epsilon$
$F \rightarrow (E)|id$

Parsing Table

You are provided with a parsing table (Table 1) that maps non-terminals and lookahead terminals to specific productions. The table looks like this in C++:

	id	+	*	(	)	**$**
E	$E \to TE'$			$E \to TE'$		
E'		$E' \to +TE'$			$E' \to \epsilon$	$E' \to \epsilon$
T	$T \to FT'$			$T \to FT'$		
T'		$T' \to \epsilon$	$T' \to *FT'$		$T' \to \epsilon$	$T' \to \epsilon$
F	$F \to id$			$F \to (E)$		

Table 1: Predictive Parsing Table

```
vector<vector<string>> TABLE = {
    {"Te", "error", "error", "Te", "error", "error"},      // E
    {"error", "+Te", "error", "error", "", ""},            // E'
    {"Ft", "error", "error", "Ft", "error", "error"},      // T
    {"error", "", "*Ft", "error", "", ""},                 // T'
    {"i", "error", "error", "(E)", "error", "error"}       // F
};
```

Note: In the parsing table, non-terminal symbols with a prime (e.g., `T'`) are represented in lowercase (e.g., `T'` is written as `t`) for simplicity.

Task Requirements

1. Implement the interpret Function: Your task is to complete the interpret function that takes a vector of tokens (input) and determines if it matches the grammar.

2. Handle Parsing Logic:

   - Use a stack to track the sequence of grammar symbols.
   - At each step, compare the top stack symbol and the current input token:
     - If they are equal, pop the stack and move to the next input token.
     - If they differ, refer to the parsing table to determine the next production.
   - If the parser encounters an invalid token or production, it should terminate and indicate a parsing error.

3. Terminal vs. Non-terminal Handling:

   - Terminal symbols are those that directly match the input tokens (e.g., +, *, etc.).

- Non-terminal symbols are those that need further expansion according to the parsing table (e.g., E, E', etc.).

4. End of Input: The parser should stop when both the stack and input contain only $.

Interpret() Function Algorithm

1. **Initialize** the stack with `$` as the bottom marker and `E` as the start symbol.
2. **Loop** until parsing is complete or an error is encountered.
3. **Check Stack**: If the top of the stack is a terminal or `$`, compare it to the current input symbol.
   - If they match, pop the stack and advance to the next input token.
   - If they do not match, return `false` to indicate a parsing error.
4. **Expand Non-Terminals**: If the top of the stack is a non-terminal:
   - Use the parsing table to get the corresponding production.
   - If a valid production exists, pop the non-terminal and push the production symbols onto the stack.
   - If no production exists, return `false` for a parsing error.
5. **End Condition**: If the top of the stack and current input token are both `$`, return `true` to indicate successful parsing.

Tips

- Remember to handle each symbol in the production right-to-left when pushing onto the stack.
- Be mindful of indexing when working with the parsing table and handling characters.
- Test your code with multiple input cases to verify accuracy.

How to Compile and Test Your Code

- You need to compile the code with C++ version ≥ 11 enabled. This may not be an issue in many Linux operating systems, but it may be in other operating systems, such as MacOS. Compile the code with the following command:

```
$g++ -std=c++14 -o top-down  top-down.cpp
```

- If compile was successful, the `top-down` executable binary file should've been generated.
- Run the executable with the following command: `$./top-down`

Expected Output

Check the `expected.out` under Lab5 directory for the expected output. All the test inputs can be found under the main function.

How to Submit Your Code

Add the `top-down.cpp` file to your repository.

```
$cd <path>/<to>/Lab5
$git add top-down.cpp
$git commit -m "your message, e.g., lab 5 - top-down.cpp"
$git push origin
```