<AE C++ Interpreter>

Version <1.0>

[Note: The following template is provided for use with the Unified Process for EDUcation. Text enclosed in square brackets and displayed in blue italics (style=InfoBlue) is included to provide guidance to the author and should be deleted before publishing the document. A paragraph entered following this style will automatically be set to normal (style=Body Text).]

[To customize automatic fields (which display a gray background when selected), select File>Properties and replace the Title, Subject and Company fields with the appropriate information for this document. After closing the dialog, automatic fields may be updated throughout the document by selecting Edit>Select All (or Ctrl-A) and pressing F9, or simply click on the field and press F9. This must be done separately for Headers and Footers. Alt-F9 will toggle between displaying the field names and the field contents. See Word help for more information on working with fields.]

Revision History

|  |  |  |  |
| --- | --- | --- | --- |
| **Date** | **Version** | **Description** | **Author** |
| 12/03/23 | 1.0 | Listing out the test cases for our project | Zutshi |
|  |  |  |  |
|  |  |  |  |
|  |  |  |  |

Table of Contents

1. Purpose 4

2. Test case identifier 4

3. Test item 4

4. Input specifications 4

5. Output specifications 4

6. Environmental needs 4

6.1.1 Hardware 4

6.1.2 Software 4

6.1.3 Other 4

7. Special procedural requirements 5

8. Intercase dependencies 5

# Purpose

This Test Case Specification document for the "Arithmetic Expression Evaluator" defines a set of test cases for evaluating the functionality of an arithmetic expression processing program developed in C++. The program is designed to handle a wide range of arithmetic expressions, including those with nested parentheses, various operators (addition, subtraction, multiplication, division, and exponentiation), and follows the mathematical rules such as order of operations and handling edge cases like division by zero.

# Test case identifier

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| Test Case ID | Expression | Expected Result | Actual Result | Pass/fail |
| TC-V001 | 3+4 | 7 | 7 | Pass |
| TC-V002 | 8-(5-2) | 5 | 5 | Pass |
| TC-V003 | 10\*2/5 | 4 | 4 | Pass |
| TC-V004 | 2^3 | 8 | 8 | Pass |
| TC-V005 | 4 \* (3 + 2) % 7 - 1 | 5 | 5 | Pass |
| TC-V006 | (((2 + 3))) + (((1 + 2))) | 8 | 8 | Pass |
| TC-V007 | ((5 \* 2) - ((3 / 1) + ((4 % 3)))) | 6 | 6 | Pass |
| TC-V008 | (((2 ^ (1 + 1)) + ((3 - 1) ^ 2)) / ((4 / 2) % 3)) | 4 | 4 | Pass |
| TC-V009 | ((((5 - 3))) \* (((2 + 1))) + ((2 \* 3)))) | 12 | 12 | Pass |
| TC-V0010 | ((9 + 6)) / ((3 \* 1) / (((2 + 2))) - 1) | -60 | -60 | Pass |
| TC-V0011 | +(-2) \* (-3) – ((-4) / (+5)) | 6.8 | 6.8 | Pass |
| TC-V0012 | -(+1) + (+2) | 1 | 1 | Pass |
| TC-V0013 | -(-(-3)) + (-4) + (+5) | -2 | -2 | Pass |
| TC-V0014 | +2 ^ (-3) | 0.125 | 0.125 | Pass |
| TC-V0015 | -(+2) \* (+3) - (-4) / (-5) | -6.8 | -6.8 | Pass |
| TC-V0016 | 2 \* (4 + 3 - 1 | Error/unbalanced operators | Error/ invalid operators | pass |
| TC-V0017 | \*5+2 | Error/invalid operators | Error/invalid operators | pass |
| TC-V0018 | 3/0 | Error/zero division | Error/zero division | pass |
| TC-V0019 | (((9+3)))\* | Error/invalid operators | Error/invalid operators | pass |
| TC-V0020 | 4.4+7.3 | 11.7 | 11.7 | pass |

# Test item

**Feature: Arithmetic Expression Evaluator**

The Arithmetic Expression Evaluator feature is capable of parsing string inputs that represent arithmetic expressions, correctly applying operator precedence, and evaluating the result. The code supports operations including addition (**+**), subtraction (**-**), multiplication (**\***), division (**/**), modulo (**%**), and exponentiation (**^**). Parentheses for indicating operation order are also handled, and the program includes error handling for invalid inputs such as division by zero.

**Specific examples from the code include:**

a) **Requirements Specification:**

* The **getPrecedence** function determines the priority of mathematical operations, essential for respecting the mathematical order of operations.
* Division and modulo by zero are explicitly checked in the **applyOperator** function, which ensures compliance with mathematical rules and returns an error if such a case occurs.
* The **evaluateExpression** function orchestrates the evaluation process, utilizing stacks to manage operators and operands, emphasizing the requirement for handling nested operations.

b) **Design Specification:**

* A stack-based approach is used for both operands (**stack<double> values**) and operators (**stack<char> operators**), allowing for the correct application of operator precedence and parenthesis grouping.
* The **applyOperator** function demonstrates polymorphism by applying different operations based on the operator character, reflecting the design's modular approach.
* Error flags are communicated through a boolean reference (**bool& error**), showing a design choice for error propagation without exception handling.

c) **User Guide:**

* Usage of the program is initiated by prompting the user for an input (**getline(cin, expression)**), illustrating the interface for expression entry.
* Error messages are descriptive and output to standard error (**cerr**), guiding the user in understanding the nature of the error, as seen in the error checks within the **applyOperator** function.
* The main function's flow from input prompt to result display (**cout << "Result: " << result << endl;**) provides a clear usage example for the end-user.

# Input specifications

Each test case to be executed by the Arithmetic Expression Evaluator requires a string input that represents an arithmetic expression. The input specification includes:

* **Numeric Values:** These should be in decimal or integer format. Example inputs include single digits (**'3'**, **'9'**), multiple digits (**'10'**, **'200'**), and decimal numbers (**'3.14'**, **'0.001'**).
* **Arithmetic Operators:** Valid operators specified by single characters such as **'+', '-', '\*', '/', '%', '^'** for addition, subtraction, multiplication, division, modulo, and exponentiation respectively.
* **Parentheses:** Open **'('** and close **')'** parentheses to specify the order of operations within the expression.
* **Whitespace:** Spaces may be included in expressions and should be ignored by the evaluator.
* **Error Flag:** A boolean value passed by reference (**bool& error**) within functions to signal if an error has occurred during evaluation.

Inputs will be specified by value within the test cases. For example, the test case for addition might specify **'3 + 4'** as an input value.

**Required Relationships Between Inputs:**

* **Operator Precedence:** The precedence of operators must reflect standard arithmetic rules; the **getPrecedence** function defines this relationship.
* **Operand-Operator Pairing:** Each operator must be paired with appropriate operands as per arithmetic rules. For instance, binary operators require two operands, one before and one after the operator.
* **Parentheses Grouping:** Parentheses should be used to override default precedence rules and group expressions to be evaluated first.
* **Error Handling:** The program must handle errors such as division by zero, modulo by zero, and syntax errors such as unmatched parentheses or invalid characters. These should result in appropriate error messages and an error flag being set.

There are no external databases, files, constant tables, or transaction files required for the basic operation of the test cases as the program is designed to operate from terminal input and output. Memory resident areas are the stacks used for values and operators (**stack<double> values**, **stack<char> operators**), and the program does not require any special values passed by the operating system for its operation.

# Output specifications

For each input expression provided to the Arithmetic Expression Evaluator, the output will be either a numeric value reflecting the evaluated result of the expression or an error message indicating the type of error encountered. The outputs for the test items are as follows:

**Numeric Output:**

* The result of correctly evaluated expressions, output as a double-precision floating-point number. Examples include **7** for the input **3 + 4**, **5** for **8 - (5 - 2)**, and **-6.8** for **-(+2) \* (+3) - (-4) / (-5)**.
* For expressions involving division, the output must be the quotient of the division, unless the divisor is zero, which should trigger an error.
* For expressions with exponentiation, the output must accurately reflect the power operation, including handling of negative exponents.

**Error Output:**

* For invalid expressions, such as those containing a division by zero or syntax errors (e.g., unmatched parentheses, invalid characters), the program must output an appropriate error message to the standard error stream (**std::cerr**).
* Examples of error messages include "Error: Division by zero." for an input like **4 / 0**, "Error: Modulo by zero." for an input like **4 % 0**, and "Error: Unmatched closing parenthesis." for an input like **2 \* (4 + 3 - 1**.
* The error flag, passed by reference in functions, will be set to **true** in case of any errors, signaling the calling function of the error state.

**Required Features:**

* The program must handle both integer and floating-point numbers, allowing for a wide range of arithmetic calculations.
* The output must reflect the correct order of operations as dictated by conventional arithmetic precedence rules.
* Error messages must be clear, descriptive, and indicate the exact nature of the error to guide the user in correcting the input expression.

All outputs must be exact, and in the case of floating-point numbers, they should be accurate up to a reasonable number of decimal places as determined by the precision of double in C++.

# Environmental needs

### Hardware

No specialized hardware is required to execute tests for the arithmetic expression project beyond a standard computer system capable of running a C++ development environment. Minimum hardware specifications include:

* **Processor:** Any modern CPU capable of running the chosen operating system and compiler.
* **Memory:** At least 512MB of RAM to ensure smooth compilation and execution.
* **Storage:** Minimal, only enough to store the source code and compiled binaries.

### Software

The software environment required to execute this test case includes:

* **Operating System:** Any operating system that can run a C++ compiler, such as Windows, macOS, Linux, or Unix-based systems.
* **Compiler:** A C++ compiler such as GCC, Clang, or MSVC++ is required. The C++ standard should be C++11 or later to ensure compatibility with the code features.
* **Simulators/Test Tools:** No specific simulators or test tools are required, as the program can be run directly in a terminal or command-line interface.
* **Text Editor/IDE:** A text editor or Integrated Development Environment (IDE) like Visual Studio Code, Eclipse, or Xcode to write and manage code.

### Other

Additional requirements include:

* **Command Line Interface/Terminal:** To enter input expressions and view the output or error messages.
* **Standard Library Support:** The program utilizes the C++ standard library, including **<iostream>**, **<string>**, **<stack>**, **<cmath>**, and **<cctype>**, so the environment must have these available.
* **Error Handling Mechanism:** As the program prints error messages to standard error (**std::cerr**), the execution environment must support standard I/O streams.

# Special procedural requirements

The test process for the Arithmetic Expression Evaluator is straightforward, given the program's operation via a command-line interface. Nonetheless, certain procedural stipulations are necessary:

* **Code Compilation:** Prior to executing any test cases, compile the C++ source code with an appropriate compiler. It is crucial to apply compiler settings that align with the C++11 standard or newer, as the code demands.
* **Test Execution:** Implement the test cases in a defined sequence. This is particularly pertinent if you are not employing automated scripts for testing. Manual entry of the test expressions into the command line is required.
* **Consistency of Input:** Care must be taken to input test expressions accurately, avoiding any extraneous spaces or characters that could potentially alter the test results.
* **Error Logging:** Keep a vigilant eye on the standard error stream for potential error messages. Any such messages should be documented and assessed against the predicted test outcomes.
* **Verification of Output:** Each test case output should be scrutinized, especially when dealing with floating-point numbers, to confirm that the precision and accuracy are within the expected range.
* **Closing Procedures:** Post the completion of test cases, verify the proper termination of the program and the absence of lingering processes. Should there be a protocol for logging results, promptly record the findings to avoid misplacement.
* **Environment Preparation:** Before starting each new test case, it may be necessary to refresh the test environment, ensuring no previous test states or data impact subsequent results.

# Intercase dependencies

* **Code Compilation:** Prior to executing any test cases, compile the C++ source code with an appropriate compiler. It is crucial to apply compiler settings that align with the C++11 standard or newer, as the code demands.
* **Basic Operations Test Cases:** Test cases evaluating basic operations such as addition, subtraction, multiplication, and division should be executed before those involving complex expressions with multiple operators and parentheses. This is to ensure that foundational operations work correctly before testing more intricate expressions that build upon these basics
* **Parentheses Handling Test Cases:** Test cases for expressions that involve parentheses should follow the basic operations. This ensures that the program's handling of operator precedence and the correct evaluation of expressions within parentheses is verified after basic arithmetic operations are confirmed to be functioning.
* **Error Handling Test Cases:** It is advisable to execute test cases for normal operations before proceeding to error handling test cases, such as division by zero or invalid syntax. This ensures that the program’s basic functionality is intact before testing its robustness against incorrect inputs.
* **Performance Test Cases:** If there are any performance-related test cases, such as evaluating very long expressions or stress testing the evaluator with a high volume of operations, these should be executed after the functional test cases have been verified to prevent performance testing from obscuring functional issues