Criteria A – Computational Thinking

**Success Criteria**

The design specifications and success criteria for a successful Reverse Polish Notation calculator project are as follows.

1. Provide accurate and reliable mathematical calculations using reverse polish notation (RPN).
2. Complex formulas should be supported, i.e., formulas that have multiple operands and operators should be executed in correct order of operations accurately.
3. Allow users to enter RPN expressions using an intuitive and easy-to-use interface. For example, an ‘enter’ should suffice for the calculator to evaluate the expression.
4. Support a fundamental range of mathematical operators, including addition, subtraction, multiplication, exponents, modulo and division.
5. Provide user-friendly feedback, including error messages if the expression is invalid.
6. The utility should be fast and responsive for users, as measured through Google’s Lighthouse performance metric.
7. The calculator should work on different web browsers and operating systems without any issues.
8. The calculator must be able to export its answers and evaluations to the screen to be interpreted by the user.
9. The RPN must utilize the stack data structure, leveraging its LIFO nature for greater efficiency.
10. The tool’s codebase must be built with further potential development in mind and have support for expandability.

**Table of Functions**

|  |  |  |  |
| --- | --- | --- | --- |
| Planned Function | Purpose and Details | Processes / Arguments | Input / Output |
| reversePolish() | The main container for invoking the calculator through the frontend. Will contain all code required to parse and evaluate reverse polish notation formulas. | String manipulation in order to parse formulas as well as iterative behaviours to evaluate each ‘block’ of the formula and arrive at an answer. | Input: Formula in Reverse Polish Notation format.  (5 2 + 6 -)  Output: Outcome of the formula, whether it was invalid or not.  (1) |
| errorMsg() | To inform the user of an input error, the function will return a string to display in the frontend rendering mechanism. | Constant array in which multiple potential strings are stored. Function returns a random user-friendly error message through the Math.floor/random library functions, prompting them to review their formula and/or check console for more information. | Input: n/a.  Output: A pseudo-random string from the array of random error messages, directing users to check console if further information is desired regarding wrong formula syntax. |
| cls() | Helper/utility function in order to reset the calculator. | Extremely linear process, function simply resets input, visual cues, and stored values from previous code runs and calculations. | Input: n/a.  Output: n/a.  Pure code execution. |
| set() | Helper/utility function for user experience to provide an example of how to use the calculator. | Also, a non-input-based function that sets the current formula to an example. | Input: n/a  Output: Sets the current formula within the calculator to an example value.  Pure code execution. |

**Justification of Data Structures**

A Reverse Polish Notation (RPN) calculator is a type of calculator that evaluates expressions using a last-in-first-out (LIFO) behaviours. In RPN notation, operands are entered first, followed by the operator. This means that the operator acts on the most recently entered operands, rather than the operands entered first. For example, to calculate "1 + 2", you would enter "1", followed by "2", and then "+" (the operator). The calculator would then evaluate the expression by adding the top two operands of the stack, which are "3" and "4". The result, "7", would be pushed back onto the stack.

There are several data structures that could theoretically be used to implement a RPN calculator, including arrays, linked lists, queues, and stacks. However, stacks are the most appropriate data structure for a RPN calculator (aligning with Success Criteria #9) for several reasons, including the fact that stacks fundamentally follow LIFO behaviour principles. Stacks are designed to allow elements to be added and removed from the top only, with either a **push** or **pop** operation. In contrast, arrays and linked lists allow elements to be added and removed from any position. While arrays could be used to simulate the behaviour of a stack, this requires additional logic to ensure that LIFO integrity is maintained, which has the potential to make code more complex and error prone, not to mention requiring more time to accomplish the same outcome.

Additionally, stacks provide constant time complexity for push and pop operations (therefore fulfilling Success Criteria #6), unlike arrays. Pushing an element onto the stack and popping an element off the stack take the same amount of time, regardless of the size of the stack. This makes stacks more efficient than arrays within this context. Arrays also require resizing when they reach their capacity, which can be an expensive operation, especially if the array is large (and when performing a complex calculation). Linked lists, while also having constant-time complexity for push and pop operations, require additional memory allocation overhead and are generally less efficient than stacks.

Stacks are also inherently well-suited to handle nested expressions, which are common (if not necessary) in RPN notation. Nested expressions occur when an operand is itself an expression that needs to be evaluated first. For example, to evaluate the expression "3 \* (4 + 5)", you would enter "3", followed by "4", "+", "5", and then "\*". With a stack, nested expressions can be evaluated easily by pushing the result of the inner expression back onto the stack before continuing with the outer expression (satisfying Success Criteria #2 and #1, to some extent for the latter).

Stacks are the most appropriate data structure for implementing a RPN calculator because they are designed for LIFO behaviour, have constant-time complexity for push and pop operations, and can handle nested expressions easily. While other data structures like arrays and linked lists could be used to implement a RPN calculator, they require additional time and logic or have overhead that make them less efficient and more complex than stacks.

**Pseudocode**

The following pseudocode algorithm parses the raw formula given by the user. The formula is trimmed before being split at each space character for further evaluation.

// splits formula and ensures formula is not empty

// assumes that NEWEXPRESSION is the formula passed in

TRIMMED = NEWEXPRESSION.trim()

EXPRESSION = TRIMMED.split(" ")

STACK = []

If EXPRESSION = "" Then

output ["", ""]

Endif

//end

The following pseudocode algorithm outlines the logic that is used in determining which operator to apply to any two given numbers popped from the stack and pushes the result back to be further processed.

// evaluating an operator and its operands

// assumes that EXPRESSION is the split + trimmed formula

// also assumes that the current index (i) in the loop is the operator

A = STACK.pop()

B = STACK.pop()

If EXPRESSION[i] = "+" Then

STACK.push(B + A)

ElseIf EXPRESSION[i] = "-" Then

STACK.push(B - A)

ElseIf EXPRESSION[i] = "\*" OR EXPRESSION[i] = "x" Then

STACK.push(B \* A)

ElseIf EXPRESSION[i] = "/" Then

STACK.push(B / A)

ElseIf EXPRESSION[i] = "^" Then

STACK.push(B ^ A)

ElseIf EXPRESSION[i] = "%" Then

STACK.push(B % A)

Else

output "Invalid operator."

Endif

// end

**Test Plan**

|  |  |  |  |
| --- | --- | --- | --- |
| Test Number | Criteria for Success | Test Data | Expected Outcome |
| 1 | Provide accurate and reliable mathematical calculations using reverse polish notation. | Input: 2 2 +  Input: 3 2 - 4 \* | Outcome: 4  Outcome: 4 |
| 2 | Complex formulas should be supported, i.e., formulas that have multiple operands and operators should be executed in correct order of operations accurately. | Input: 5 7 + 2 \* 8 4 / - 3 \* 9 + 6 2 - / 1 - 4 \* 2 / 7 3 \* -  Input: 2.23 334 + 23 \* 554 / 6.1 - 7 ^ 8.4 % 1 + | Outcome: 14.5  Outcome: 1.7918907326794304  (Validated and corroborated by [WisdomSky](https://wisdomsky.github.io/reverse-polish-notation-js-parser/)’s RPN calculator) |
| 3 | Allow users to enter RPN expressions using an intuitive and easy-to-use interface. For example, an ‘enter’ should suffice for the calculator to evaluate the expression. | A deployed instance of the website and external feedback from parties uninvolved with the development of the product. | Outcome: Worded feedback about each design specifications and whether they are met, partially met, or not met from the audience’s perspective. |
| 4 | Support a fundamental range of mathematical operators, including addition, subtraction, multiplication, exponents, modulo and division. | Input: 2 3 + 4 \* 5 / 6 - 7 ^ 8 %  Input: 10 5 %  Input: 2 3 ^ | Outcome: 0  Outcome: 0  Outcome: 8  (See above for validation) |
| 5 | Provide user-friendly feedback, including error messages if the expression is invalid. | Usage of the website by those who are not directly or indirectly involved in the product’s development. | Outcome: Feedback about each of the previously defined success criteria and the audience’s personal opinions on whether these criteria are met, partially met, or not met within the presented solution. |
| 6 | The utility should be fast and responsive for users, as measured through Google’s Lighthouse performance metric. | Google’s Lighthouse performance utility available in the Inspector window on Chromium based browsers. | Outcome: The performance metric must be above 80%. |
| 7 | The calculator should work on different web browsers and operating systems without any issues. | Manually ensure that all functions and frontend experiences are identical across the modern versions of Chromium, Edge, and Firefox. | Outcome: must have a success rate across browsers at 100%. |
| 8 | The calculator must be able to export its answers and evaluations to the screen to be interpreted by the user. | Input: 5 6 \* 7 + 8 9 + \* 4 / 2 ^ 10 - 3 \* 2 \* 1 / | Outcome: The correct answer should always display in the frontend.  (See above for validation) |
| 9 | The RPN must utilize the stack data structure, leveraging its LIFO nature for greater efficiency. | Code within the program that is used for storing ‘blocks’ of the inputted RPN formula. | Outcome: Should only utilize stack.push() and stack.pop() as JavaScript data structures are dynamic and the compiler changes them without notice at times. |
| 10 | The tool’s codebase must be built with further potential development in mind and have support for expandability. | Code within the tool must be designed with the addition of more operators, error identification, etc. | Outcome: Adding an additional operator such as square root should not require readjusting or refactoring previously written code. |

**Record of Tasks (Logical Plan)**

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| Task Number | Planned Action | Planned Outcome | Time Estimated | Target Completion Date | Target Criterion |
| 1 | Begin constructing the plan for the product with definitions of Success Criteria in a document named Criteria A. | The design specifications for the product will have been defined and a clear understanding of the requirements for the solution will have been gained. | 30 minutes. | 27/04 | A |
| 2 | The table of functions, justification of data structures, pseudocode and test plan will be completed. | The same document will be used to plan the rest of the product’s development phase, including pseudocode for essential algorithms and test plans for evaluating the success of the product. | 1 hour. | 27/04 | A |
| 3 | Set up a development environment within Visual Studio Code, with a clean folder structure. | The VSC coding environment will be successfully set up and completely functional. | 5 minutes. | 27/04 | B |
| 4 | Set up Vue Instance boilerplate for the frontend of the app. | The Vue app script will be set up, within a file named script.js. | 15 minutes. | 27/04 | B |
| 5 | Create a main function for RPN functionality that takes in an expression and outputs an answer. | The function will be able to take in an expression from the frontend and output it with a temporary block of console.log(). | 15 minutes. | 27/04 | B |
| 6 | Create a process that takes the expression inputted, trims the string for whitespaces, splits string at the space. | The process will take in an input from the previously made function called from the frontend using Vue’s method/event handler and output the split string. | 5 minutes. | 27/04 | B |
| 7 | Create an iteration and selection block(s) that checks whether the expression is a numerical operand or and operator. Must ensure that only numbers are added to the stack. | The program will be able to differentiate between operands and operators through selection blocks. | 15 minutes. | 27/04 | B |
| 8 | Create a switch statement that operates between the two top operands of the stack based on the selected operator. | The program outputs the correct answer when a formula is inputted.  Input: 7 8 +  Output: 15 | 10 minutes. | 27/04 | B |
| 9 | Add error handling to ensure that only one item is remaining in the stack when calculation is complete. If not, the formula was invalid. | The program understands invalid formulas and does not return system error.  Input: 4 5 6 +  Output: ‘Invalid formula’. | 5 minutes. | 27/04 | B |
| 10 | Add further error specification and print the error into the console. This is to be done through a switch statement. | The program can gracefully recover from error and specify what went wrong.  Input 1 0 /  Output: ‘Infinity’.  Input: hef42irge  Output: ‘Non-mathematical formula’. | 10 minutes. | 27/04 | B |
| 11 | Call the function from a Vue event handler method. | The program calls the function with the current formula within the box using the Vue viewmodel. | 1 minute. | 27/04 | B |
| 12 | Utility functions for clearing the calculator and helping the user understand the calculator’s functions. | Random frontend-user-friendly error messages prompting to check for console for further information if desired. Also, a utility for setting the box to a simpler formula from a button click. | 5 minutes. | 27/04 | B |
| 13 | Final UI improvements. | The interface is fast and responsive, and keyboard shortcuts (enter) can be used. | 15 minutes. | 27/04 | B |
| 14 | Obtain audience feedback and complete individual product evaluation. | Adequate amounts of audience feedback for each success criteria must be recorded within a separate document in Criteria C, and the individual product evaluation will be completed. | 2 hours. | 01/05 | C |
| 15 | Consider possible improvements of the product. | The Criteria C document will have been completed with high detail discussing the possible aesthetic and functional improvements to the product. | 1 hour. | 01/05 | C |