Expression Notation Converter CLI

A Command-Line Utility for Mathematical Expression Notation Conversion

I.1.A) Introduction

The Expression Notation Converter CLI is a robust command-line utility designed to facilitate the conversion of mathematical expressions among infix, prefix (Polish), and postfix (Reverse Polish) notations. The language used for this project is C. Central to its operation are binary expression trees, which ensure accurate parsing, preservation of operator precedence, associativity, and correct grouping via parentheses. The program supports a comprehensive set of features, including variables (A-Z, a-z), multi-digit numbers, and standard arithmetic operators, notably handling right-associative exponentiation (^).

I.1.B) Features Overview

Notation Conversions Supported

- Infix ↔ Prefix
- Infix ↔ Postfix
- Prefix ↔ Postfix

Supported Syntax

- Operands: Single characters (e.g., A, x, 3) or multi-digit numbers
- Operators: +, -, *, /, ^
- Parentheses: Used in infix to denote precedence
- Associativity: Supports left and right associativity

Command-Line Features

- Command-line argument parsing
- Built-in guides/help system
- Displays both converted expression and structured output

II.1.A) Functions

Functions used in the program	
FUNCTION NAME	PURPOSE
`detectNotation`	This function attempts to determine the notation (infix, prefix, or postfix) of an expression based on the position of operators.
`createTreeNode`	Allocates memory for a new TreeNode and initializes its value and child pointers.
`isOperator`	Checks if a given character is one of the supported operators (+, -, *, /, ^).
`getPrecedence`	Returns the precedence level of an operator. Higher numbers indicate higher precedence.
`isRightAssociative`	Checks if a given operator is right-associative (currently only '^').
`notation`	Converts a Notation enum value into its corresponding string representation

	(e.g., INFIX to "Infix").
`printTreeInOrder`	Performs an in-order traversal of the expression tree and prints the expression in infix notation. It includes parentheses for correct precedence.
`printTreePreOrder`	Performs a pre-order traversal of the expression tree and prints the expression in prefix notation.
`printTreePostOrder`	Performs a post-order traversal of the expression tree and prints the expression in postfix notation.
`buildTreeFromInfix`	Constructs an expression tree from an infix expression string. It uses a shunting-yard like algorithm to convert infix to postfix, then builds the tree from the postfix form.
`buildPrefixHelper`	A recursive helper function used by buildTreeFromPrefix to build the tree.
`buildTreeFromPrefix`	Constructs an expression tree from a prefix expression string.
`buildTreeFromPostfix`	Constructs an expression tree from a postfix expression string.

II.1.B) Algorithm

```
BEGIN Program
   // Print Program Header
   Print "============================="
   Print " EXPRESSION NOTATION CONVERTER"
   Print "============================="

   // Get and Validate Input Arguments
   Get the input arguments from the user (expression string, notation flag).

   // Handle Help Argument
   IF user_input_is_help_only_argument() THEN
      Display program usage guidelines.
      EXIT Program.
   END IF

   // Validate Required Arguments
   IF required_arguments_are_missing() THEN
      Display "Error: Missing required arguments. Use --help for usage."
      EXIT Program.
   END IF

   // Determine Notation Type
   DECLARE notation_type AS ENUM {INFIX, PREFIX, POSTFIX, UNKNOWN}.
   SET notation_type = determine_notation_type(notation_flag).

   IF notation_type == UNKNOWN THEN
      Display "Error: Invalid notation flag. Use --infix, --prefix, or --postfix."
      EXIT Program.
   END IF

   // Build Expression Tree
   DECLARE root AS TreeNode.
   SWITCH notation_type:
```

```
        CASE INFIX:
            SET root = buildTreeFromInfix(expression).
        CASE PREFIX:
            SET root = buildTreeFromPrefix(expression).
        CASE POSTFIX:
            SET root = buildTreeFromPostfix(expression).

    // Print Conversion Results
    Print "============================="
    Print " CONVERSION RESULTS"
    Print "============================="

    Print "[ ORIGINAL EXPRESSION ]"
    Print notation_type AND expression.

    Print "[ CONVERTED EXPRESSIONS ]"
    SWITCH notation_type:
        CASE INFIX:
            Print "Prefix: " AND printTreePreOrder(root).
            Print "Postfix: " AND printTreePostOrder(root).
        CASE PREFIX:
            Print "Infix: " AND printTreeInOrder(root).
            Print "Postfix: " AND printTreePostOrder(root).
        CASE POSTFIX:
            Print "Prefix: " AND printTreePreOrder(root).
            Print "Infix: " AND printTreeInOrder(root).

END Program
```

III.) Running the Program

1. To run the program, the user can open the executable file of the program.

Step 1: Compile the Program

```
gcc main.c -o main
```

2. Run the program

For Help:

```
./main --help
```

For Converting an Expression:

```
./main "<expression>" --<notation>
```

Replace `<expression>` with your mathematical expression and `<notation>` with one of the following:

- `--infix` for infix notation
- `--prefix` for prefix notation
- `--postfix` for postfix notation

```
simonangelonarvaez@simons-MacBook-Air expression-notation-converter % ./main --help
Expression Notation Guidelines

  Usage: ./main "<expression>" --<notation>
    E.g: ./main "1 + 2 ^ (3 * 4) / 5" --infix

Valid operations: + | - | * | / | ^

Expression rules:
        - Must be enclosed in ""
        - Supports variables (A,B,C,x,y,z) and positive integers
                - A + B + C
                - 1 + 2 + 3
                - A + 1 + B
        - Integers may be multiple characters
                - A + BC + DEF
                - 1 + 23 + 456
        - Operands and operators must be separated by spaces
                - 1+2+3 (wrong)
                - 1 + 2 + 3 (correct)
        - Precedence is specified with '()'
                - (1 + 2) + 3 ^ (4 + 5)
simonangelonarvaez@simons-MacBook-Air expression-notation-converter %
```

Figure 1: Program usage guidelines and expression rules displayed when running `./main --help`. This output provides instructions on how to use the expression converter, including valid operations and formatting requirements.

```
================================================================================

 EXPRESSION NOTATION CONVERTER
 Discrete Structures 2
 BSCS 2A

 [ TEAM MEMBERS ]
        Michael Xavier E. Canonizado
        Simon Narvaez

================================================================================

 CONVERSION RESULTS

 [ ORIGINAL EXPRESSION ]
        Infix   : 1 + 2 * 3

 [ CONVERTED EXPRESSIONS ]
        Prefix  : + 1 * 2 3
        Postfix : 1 2 3 * +

================================================================================
```

*Figure 2: A sample output of the Expression Notation Converter, demonstrating the conversion of the infix expression "1 + 2 * 3" into its equivalent prefix ("+ 1 * 2 3") and postfix ("1 2 3 * +") notations.*

IV.) Conclusion

The Expression Notation Converter CLI successfully fulfills its objective as a robust command-line utility for converting mathematical expressions between infix, prefix, and postfix notations. The project effectively met all core requirements, including the implementation of six distinct conversion paths using expression trees, accurate handling of operator precedence and associativity, and robust command-line argument parsing with comprehensive help documentation.

Requirements Fulfillment:

1. **Core Functionality:**
    - Implemented all six conversion paths between notations
    - Developed expression tree construction algorithms for each notation type
    - Integrated proper operator precedence and associativity handling
2. **Technical Implementation:**
    - Utilized the shunting-yard algorithm for infix parsing
    - Implemented stack-based and recursive methods for postfix and prefix processing
    - Designed modular architecture with clear separation of concerns
3. **User Interface:**
    - Delivered both command-line argument and interactive menu interfaces
    - Included comprehensive help documentation (--help and --guide options)
    - Implemented clear error messaging and input validation

Exceeded Specifications:

- Added support for variables (A-Z, a-z) and multi-digit numbers
- Implemented exponentiation (^) with right-associativity
- Developed additional parenthesization logic for infix output

Deliverables Submitted:

1. Well-commented source code
2. Comprehensive README documentation
3. Example test cases

Attachment:

GITHUB Link: https://github.com/michaelcanonizado/expression-notation-converter

V.) Member Participation
CANONIZADO, Michael Xavier: Head Programmer
NARVAEZ, Simon Angelo Karlo: Assistant