Command Line Calculator

A COURSE PROJECT REPORT

By

Karan Sharma (RA2011027010077)
Dikcha Singh (RA2011027010096)
Santhanalakshmi K. (RA2011027010129)

Under the guidance of

Dr. S. Sharanya

In partial fulfilment for the Course

of

18CSC304J – COMPILER DESIGN

in

Data Science and Business Systems

FACULTY OF ENGINEERING AND TECHNOLOGY

SRM INSTITUTE OF SCIENCE AND TECHNOLOGY

Kattankulathur, Chenpalpattu District

APRIL 2022

SRM INSTITUTE OF SCIENCE AND TECHNOLOGY

(Under Section 3 of UGC Act, 1956)

BONAFIDE CERTIFICATE

Certified that this project report " **Command Line Calculator**" is the bonafide work of Karan Sharma (RA2011027010077), Dikcha Singh (RA2011027010096), Santhanalakshmi K. (RA2011027010129)

who carried out the projectwork under my supervision.

Dr. S. Sharanya,	**Dr. S. Sharanya,**
Subject Staff	**Course Cordinator**
Associate Professor,	**Associate Professor,**
Data Science and Business Systems	**Data Science and Business Systems**
SRM Institute of Science and Technology	SRM Institute of Science and Technology
Potheri, SRM Nagar, Kattankulathur,	Potheri, SRM Nagar, Kattankulathur,
Tamil Nadu 603203	Tamil Nadu 603203

TABLE OF CONTENTS

CHAPTER 1 – ABSTRACT

A command-line calculator which supports mathematical expressions with scientific functions is very useful for most developers. The calculator available with Windows does not support most scientific functions. Most of the time, I do not feel comfortable with the calculator available with Windows. I needed a calculator which will not restrict writing expressions. I use variables to store results. Every time I need a simple calculation, I have to face problems with the Windows calculator. To make such a calculator, I designed a complete Mathematics library with MFC. The most difficult part I found when designing such a calculator was the parsing logic. Later while working with .NET, the runtime source code compilation made the parsing logic easy and interesting. I read some articles on .NET CodeDOM compilation. And I decided to write a new command line calculator using CodeDOM. It uses runtime compilation and saves the variables by serializing in a file. Thus you can get the values of all the variables used in the previous calculation.

CHAPTER 2 – MOTIVATION

We have been studying compiler design, how it works and how the language is being compiled. This created a curiosity about how these compilers are being made and how they work. The compiler has features that we wanted to explore, such as-

- Compilers provide an essential interface between applications and architectures.
- Compilers embody a wide range of theoretical techniques.
- Compiler construction teaches programming and software engineering skills
- It teaches how real-world applications are designed.
- It brings us closer to the language to exploit it.
- Compiler bridges a gap between the language chosen & a computer architecture
- Compiler improves software productivity by hiding low-level details while delivering performance
- The compiler provides techniques for developing other programming tools, like error detection tools.
- Program translation can be used to solve other problems, like Binary translation.

In this command line calculator, the result is saved in a pre-defined variable called ans. The user can declare his/her own variables to store results and can use it later in different expressions. The validation of the variable name is the same as in C#. Similarly, expression support is the same as supported in C# .NET.

The calculate function calculates an expression. It uses the saved variables. I have generated code which has a declaration of the variables.

To Evaluate the given expressions.

To perform basic calculations

CHAPTER 3 – LIMITATIONS OF EXISTING METHODS

Limited User Interface: Command line calculators typically have minimal or no graphical user interface (GUI), which may make it harder for users who are not familiar with command line interfaces to use effectively. Users may need to input complex expressions manually, and the calculator may provide limited feedback or error handling.

Limited Functionality: Command line calculators may have limited built-in functions or operators compared to graphical calculators or dedicated math software. Advanced mathematical functions, unit conversions, or other specialized features may be missing or require additional implementation.

Lack of Flexibility: Command line calculators may have limited flexibility in handling different data types, such as fractions, matrices, or complex numbers. They may also have limitations in precision or range for handling large numbers or decimals with high precision.

Limited History and Memory: Command line calculators may not have a built-in history or memory feature, making it harder to recall previous calculations or store intermediate results for future use. Users may need to manually track their calculations or results outside of the calculator.

Limited Interactivity: Command line calculators may lack interactivity, such as the ability to edit or modify expressions or results directly on the screen, undo/redo functionality, or real-time feedback on syntax errors or mathematical inconsistencies.

Lack of Error Handling: Command line calculators may have limited error handling capabilities, and may not provide detailed error messages or prompts for incorrect inputs or invalid calculations, making it harder to identify and correct mistakes.

Platform Dependence: Command line calculators may be dependent on the specific operating system or command line environment where they are being used, which may limit their portability or compatibility across different platforms or environments.

CHAPTER 4 – PROPOSED METHOD WITH ARCHITECTURE

The rough workflow of our compiler as shown below in the diagram:

```
+--------------------------------------------------------+
|                User Interface                  |
| (Command Line Interface, Input Validation, Error Handling)|
+--------------------------------------------------------+
                    |
                    V
+--------------------------------------------------------+
|                  Parser Layer                  |
| (Lexical Analysis, Syntax Analysis, Abstract Syntax Tree)|
+--------------------------------------------------------+
                    |
                    V
+--------------------------------------------------------+
|               Semantic Analysis                 |
| (Type Checking, Variable Declarations, Function Signatures)|
+--------------------------------------------------------+
                    |
                    V
+--------------------------------------------------------+
|                Code Generation                  |
| (Intermediate Code or Machine Code Generation)         |
+--------------------------------------------------------+
                    |
                    V
+--------------------------------------------------------+
|                  Optimization                   |
| (Optional: Code Optimization Techniques)              |
+--------------------------------------------------------+
                    |
                    V
+--------------------------------------------------------+
|                   Execution                     |
| (Actual Calculation or Evaluation)                     |
+--------------------------------------------------------+
```

Customization and Extensibility: A compiler-based calculator can be designed to support custom functions, operators, or data types by defining them in the source code and compiling them into the calculator. This allows for greater flexibility and extensibility, allowing users to define their own mathematical functions or operators tailored to their specific needs.

High Performance: Compiled code can often be highly optimized for performance, allowing for faster and more efficient calculations compared to interpreted code used in some normal command line calculators. This can be especially beneficial for complex calculations or large datasets.

Portability: A compiled command line calculator can be compiled into machine code that is independent of the underlying operating system or environment, making it more portable and compatible across different platforms or systems.

Strong Typing and Error Checking: Compiler-based calculators typically enforce strong typing and perform thorough error checking during the compilation process, which can help catch potential errors or inconsistencies in expressions or inputs before the calculator is executed. This can help improve the reliability and accuracy of calculations.

Integration with Other Languages or Libraries: Compilers often provide integration with other programming languages or libraries, allowing for seamless integration with existing code or functionality. This can be advantageous for users who are already familiar with a particular programming language or have specific requirements that can be addressed using external libraries or APIs.

Debugging and Development Tools: Compiler-based calculators may provide debugging and development tools, such as step-by-step execution, variable inspection, or profiling, which can be helpful for developers or advanced users to diagnose and fix issues or optimize performance.

Enhanced Security: Compiler-based calculators can potentially offer enhanced security features, such as code obfuscation, access controls, or encryption, which can help protect sensitive or proprietary calculations or algorithms.

CHAPTER 5 – MODULES WITH DESCRIPTION

Lexical Analysis: This stage involves breaking down the input expression or code into a sequence of tokens, which are the basic units of syntax in the language of the calculator. Tokens can include numbers, operators, functions, parentheses, and other elements.

Syntax Analysis: This stage involves parsing the sequence of tokens and building an abstract syntax tree (AST) that represents the syntactic structure of the input expression or code. The AST serves as an intermediate representation that can be used for further processing and evaluation.

Semantic Analysis: This stage involves checking the semantic correctness of the input expression or code, including checking for correct data types, variable declarations, function signatures, and other semantic rules. This stage may also involve type inference, where the data types of expressions are deduced based on context.

Code Generation: This stage involves generating machine code or intermediate code from the AST or the semantic analysis results. The generated code is typically in a format that can be executed directly by the computer's processor or by an interpreter.

Optimization: This optional stage involves applying various optimizations to the generated code to improve its performance, such as constant folding, common subexpression elimination, or loop unrolling. Optimization techniques can vary depending on the specific requirements and characteristics of the calculator.

Execution: Once the code is generated and optimized (if applicable), it can be executed to perform the actual calculations or evaluations. The result of the calculation is typically displayed as output to the user in the command line interface.

Error Handling: Throughout the various stages of the compiler, error handling mechanisms are implemented to detect and report any syntax errors, semantic errors, or other issues in the input expression or code. Error messages or prompts may be displayed to the user to indicate the location and nature of the errors.

User Interface: Although a command line calculator may not have a graphical user interface (GUI) like a typical calculator with buttons and displays, it may still provide a text-based interface for users to input expressions, view results, and interact with the calculator. The user interface may include features such as command line prompts, input validation, and output formatting.

ALGORITHM

```
Step 1 — START
Step 2 — input
Step 3 — parse_expr()
step 3.1 parse_term()
step 3.1.1 parse_factor()
step 3.2 parse_num_op()
step 3.2.1 parse_rest_term()
step 3.3 parse_factor()
step 3.3.1 parse_num_op()
step 3.4 parse_rest_term()
step 3.4.1 parse_rest_expr()
Step 4 — STOP
```

CHAPTER 6 – CODE

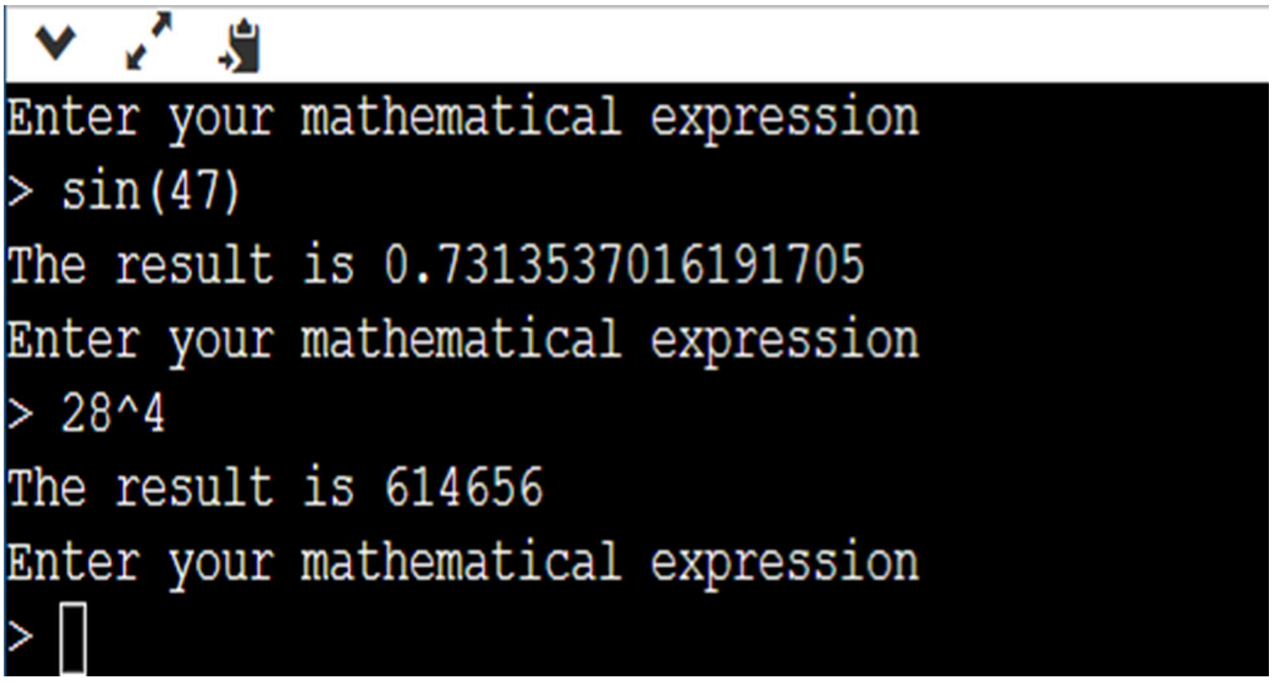
```
from typing import Any, Callable, Dict, List, Tuple, Union
from enum import Enum
import math
from sys import argv

class InvalidExpressionError(Exception):
    pass

class TokenType(Enum):

    NUMBER = 'NUMBER'
    NEGATIVE_NUMBER = 'NEGATIVE_NUMBER'
    SUM = 'SUM'
    SUBTRACTION = 'SUBTRACTION'
    MULTIPLICATION = 'MULTIPLICATION'
    DIVISION = 'DIVISION'
    POWER = 'POWER'
    OPEN_PARENTHESIS = 'OPEN_PARENTHESIS'
    CLOSE_PARENTHESIS = 'CLOSE_PARENTHESIS'
    FUNCTION = 'FUNCTION'

priority_map: Dict[TokenType, int] = {
    TokenType.NUMBER: 0,
    TokenType.NEGATIVE_NUMBER: 0,
    TokenType.CLOSE_PARENTHESIS: 0,
    TokenType.SUM: 1,
    TokenType.SUBTRACTION: 1,
    TokenType.MULTIPLICATION: 2,
    TokenType.DIVISION: 2,
    TokenType.POWER: 3,
    TokenType.FUNCTION: 4,
    TokenType.OPEN_PARENTHESIS: 5
}

MAX_PRIORITY =
priority_map[TokenType.OPEN_PARENTHESIS]

def get_token_type_priority(type: TokenType) -> int:
    return priority_map[type]

class Token:

    def __init__(self, type: TokenType, base_priority: int, value: Any =
None) -> None:
        self.type = type

        self.priority = get_token_type_priority(self.type)
        if (self.priority != 0):
            self.priority += base_priority

        self.value = value

    def __repr__(self) -> str:
        return f'<{self.type}: {self.value} ({self.priority})>'

    def __str__(self) -> str:
        return self.__repr__()

Number = Union[int, float]
Function = Callable[[List[Token]], Number]

def sqrt(args: Tuple[Token]) -> Number:
    radicand = args[0].value
    if radicand < 0:
        raise InvalidExpressionError(f'Invalid expression: radicand of
root with even index cannot be negative')

    return math.sqrt(radicand)


def nroot(args: Tuple[Token, Token]) -> Number:
    radicand = args[0].value
    index = args[1].value

    if index % 2 == 0:

        if radicand < 0:
            raise InvalidExpressionError(f'Invalid expression: radicand of
root with even index cannot be negative')
    return radicand ** (1 / index)


def sin(args: Tuple[Token]) -> Number:
    radians = math.radians(args[0].value)
    return math.sin(radians)


def cos(args: Tuple[Token]) -> Number:
    radians = math.radians(args[0].value)
    return math.cos(radians)


def tan(args: Tuple[Token]) -> Number:
    radians = math.radians(args[0].value)
    return math.tan(radians)


function_map: Dict[str, Function] = \
{
    'sqrt': sqrt,
    'nroot': nroot,
    'sin': sin,
    'cos': cos,
    'tan': tan
}


def get_expression() -> str:
    while True:
        try:
            expression = input('Enter your mathematical expression\n>
').replace(' ', '')

            if expression == 'exit':
                exit(0)

            if len(expression) > 0:
                break

        except KeyboardInterrupt:
            continue

        except EOFError:
            exit(0)

    return expression


def is_function_name(char: str) -> bool:
    ascii_value = ord(char)
    return 64 < ascii_value < 91 or 96 < ascii_value < 123


def tokenize_expression(expression: str) -> List[Token]:
    token_list: List[Token] = []

    token: Union[Token, None] = None
    base_priority = 0
    for char in expression:

        if token is not None:
            if token.type == TokenType.NUMBER:
                if char.isdigit():
                    token.value = token.value * 10 + int(char)
                    continue
                token_list.append(token)
                token = None

            elif token.type == TokenType.NEGATIVE_NUMBER:
                if char.isdigit():
                    token.value = token.value * 10 - int(char)
                    continue
                if token.value == 0:
                    raise InvalidExpressionError(f'Invalid expression
"{expression}": subtraction operator \'-\' not allowed in this context')
                token_list.append(token)
                token = None

            elif token.type == TokenType.FUNCTION:
                if is_function_name(char):
                    token.value += char
                    continue
                token_list.append(token)
                token = None

        if char.isdigit():
            token = Token(TokenType.NUMBER, base_priority,
int(char))
            continue

        if char == '+':
            token_list.append(Token(TokenType.SUM, base_priority))
            continue
        if char == '-':

            if len(token_list) != 0 and (token_list[-1].type ==
TokenType.NUMBER or token_list[-1].type ==
TokenType.CLOSE_PARENTHESIS):
                token_list.append(Token(TokenType.SUBTRACTION,
base_priority))
            else:
                token = Token(TokenType.NEGATIVE_NUMBER,
base_priority, 0)
            continue

        if char == '*':
            token_list.append(Token(TokenType.MULTIPLICATION,
base_priority))
            continue

        if char == '/':
            token_list.append(Token(TokenType.DIVISION,
base_priority))
            continue

        if char == '^':
            token_list.append(Token(TokenType.POWER, base_priority))
            continue

        if char == '(':
            base_priority += MAX_PRIORITY
            token_list.append(Token(TokenType.OPEN_PARENTHESIS,
base_priority))
            continue

        if char == ')':
            base_priority -= MAX_PRIORITY
            token_list.append(Token(TokenType.CLOSE_PARENTHESI
S, base_priority))
            continue

        if char == ',':
            continue

        # Check if char can be part of a function name.
        if is_function_name(char):
            token = Token(TokenType.FUNCTION, base_priority, char)
            continue

        raise InvalidExpressionError(f"The expression {expression}
contains an error: invalid character '{char}'")

    if token is not None:
        token_list.append(token)

    return token_list


def get_highest_priority_token_index(token_list: List[Token]) -> int:
    highest_priority_index = 0
    highest_priority = 0
    index = 0
    for token in token_list:
        if token.priority > highest_priority:
            highest_priority = token.priority
            highest_priority_index = index
        index += 1

    return highest_priority_index
```

```
def get_binary_operands(token_list: List[Token], index: int) ->
Tuple[Token, Token]:
    return (
        token_list[index - 1],
        token_list[index + 1]
    )


def remove_binary_operands(token_list: List[Token], index: int) ->
None:

    token_list.pop(index + 1)
    token_list.pop(index - 1)


def find_closing_parenthesis_index(token_list: List[Token], index:
int) -> int:

    parenthesis_depth = 1

    for token in token_list[index + 1:]:
        index += 1
        if token.type == TokenType.OPEN_PARENTHESIS:
            parenthesis_depth += 1
        elif token.type == TokenType.CLOSE_PARENTHESIS:
            parenthesis_depth -= 1

            if parenthesis_depth == 0:
                return index


def evaluate_expression(token_list: List[Token]) -> Number:
    while True:
        token_index = get_highest_priority_token_index(token_list)
        token = token_list[token_index]

        if token.priority == 0:
            return token.value
        token.priority = 0

        if token.type == TokenType.SUM:
            left_operand, right_operand = get_binary_operands(token_list,
token_index)
            result = left_operand.value + right_operand.value
            token.type = TokenType.NUMBER
            token.value = result
            remove_binary_operands(token_list, token_index)

        elif token.type == TokenType.SUBTRACTION:
            left_operand, right_operand = get_binary_operands(token_list,
token_index)
            result = left_operand.value - right_operand.value
            token.type = TokenType.NUMBER
            token.value = result
            remove_binary_operands(token_list, token_index)

        elif token.type == TokenType.MULTIPLICATION:
            left_operand, right_operand = get_binary_operands(token_list,
token_index)
            result = left_operand.value * right_operand.value
            token.type = TokenType.NUMBER
            token.value = result
            remove_binary_operands(token_list, token_index)

        elif token.type == TokenType.DIVISION:
            left_operand, right_operand = get_binary_operands(token_list,
token_index)
            if right_operand.value == 0:
                raise InvalidExpressionError('Invalid expression: cannot
divide by zero')

            result = left_operand.value / right_operand.value
```

```
            token.type = TokenType.NUMBER
            token.value = result
            remove_binary_operands(token_list, token_index)

        elif token.type == TokenType.POWER:
            left_operand, right_operand = get_binary_operands(token_list,
token_index)

            result = left_operand.value ** right_operand.value
            token.type = TokenType.NUMBER
            token.value = result
            remove_binary_operands(token_list, token_index)

        elif token.type == TokenType.OPEN_PARENTHESIS:
            closing_parenthesis_index =
find_closing_parenthesis_index(token_list, token_index)

            if token_list[token_index - 1].type ==
TokenType.FUNCTION:
                token.value = []
                for children in token_list[token_index + 1 :
closing_parenthesis_index]:
                    token.value.append(children)
                for index in range(closing_parenthesis_index, token_index,
-1):
                    token_list.pop(index)

            else:
                token_list.pop(closing_parenthesis_index)
                token_list.pop(token_index)

        elif token.type == TokenType.FUNCTION:
            parenthesis = token_list[token_index + 1]

            function = function_map.get(token.value)
            if function is None:
                raise InvalidExpressionError(f'Invalid expression: undefined
function "{token.value}"')
            token.value = function(parenthesis.value)

            token.type = TokenType.NUMBER
            token_list.pop(token_index + 1)


def print_token_list(token_list: List[Token]) -> None:
    for token in token_list:
        print(token)


def main() -> None:
    if len(argv) == 2:
        expression = argv[1].replace(' ', '')
        token_list = tokenize_expression(expression)
        result = evaluate_expression(token_list)
        print(result)
        exit(0)

    while True:
        try:
            expression = get_expression()
            token_list = tokenize_expression(expression)
            result = evaluate_expression(token_list)
            print(f'The result is {result}')

        except KeyboardInterrupt:
            break

        except InvalidExpressionError as exc:

            print(exc)
            continue
```

```
if __name__ == '__main__':
    main()
```

OUTPUT:



```
Enter your mathematical expression
> sin(47)
The result is 0.7313537016191705
Enter your mathematical expression
> 28^4
The result is 614656
Enter your mathematical expression
> 
```

CHAPTER 7 – CONCLUSION

This is a powerful and versatile command-line calculator that really lives up to your expectations. Preloaded on all modern Linux distributions, this can make your number-crunching tasks much easier to handle without leaving your terminals. Besides, if your shell script requires floating point calculation, it can easily be invoked by the script to get the job done. All in all, CLC should definitely be in your productivity tools.

CHAPTER 8 – REFERENCES

https://en.wikipedia.org/wiki/PL/0

https://www.cs.cmu.edu/~aplatzer/course/Compilers/waitegoos.pdf

https://www.webopedia.com/definitions/high-level-language/

https://en.wikibooks.org/wiki/Introduction_to_Programming_Languages/Grammars?fbclid=IwAR0nLkq2rIAyA5DbDRHBXYpHWsNo21XYas-7GjeUe82G-DWtdAydk8oeBys

https://softwareengineering.stackexchange.com/questions/165543/how-to-write-a-very-basic-compiler

https://visualstudiomagazine.com/articles/2014/05/01/how-to-write-your-own-compiler-part-1.aspx