PLC Calculator

by

Puskar Adhikari

A Project Proposal Submitted in Partial Fulfillment of the Requirements for the
AT70.07 Programming Languages and Compilers course in
Computer Science and Information Management

Instructor: Prof. Phan Minh Dung
Teaching Assistant: Mr. Akraradet Sinsamersuk

Asian Institute of Technology
School of Engineering and Technology
Thailand
March 2025

ACKNOWLEDGEMENTS

I wish to express my sincere gratitude to Professor Phan Minh Dung for his invaluable guidance and expertise throughout the AT70.07 Programming Languages and Compilers course. His insightful direction has been instrumental to my academic growth and the success of this project. I am equally grateful to Mr. Akraradet Sinsamersuk, the teaching assistant, for his unwavering support, constructive feedback, and dedication, which greatly enhanced the development process of this work. Additionally, I would like to acknowledge the encouragement and collaborative spirit of my peers, whose contributions were essential to the successful completion of the Prefix Calculator project. Their collective efforts have made this endeavor both rewarding and enriching.

ABSTRACT

This project presents the design and implementation of a Prefix Calculator, a computational tool developed to evaluate arithmetic expressions in prefix notation. The calculator leverages the SLY (Sly Lex-Yacc) library for parsing and processing prefix expressions, ensuring efficient and accurate computation. The graphical user interface (GUI) was constructed using the PyQt6 framework, providing an intuitive and user-friendly experience. The system accepts prefix expressions as input, parses them into executable operations, and delivers results seamlessly, demonstrating the integration of lexical analysis, syntactic parsing, and modern GUI development. This work highlights the practical application of programming language concepts and compiler design principles, offering a robust solution for prefix-based arithmetic computation.

CONTENTS

LIST OF TABLES

Tables **Page**

LIST OF FIGURES

Figures **Page**

CHAPTER 1
INTRODUCTION

The evolution of computational tools has profoundly transformed the execution of mathematical operations, shifting from labor-intensive manual calculations to advanced software solutions. Among these tools, calculators stand as a cornerstone, evolving to accommodate diverse notations and user needs.

Mathematical expressions, central to this domain, are typically represented in three forms: infix (e.g., 1 + 2), prefix (e.g., + 1 2), and postfix (e.g., 1 2 +). While infix notation—placing operators between operands—dominates conventional arithmetic, prefix notation, also known as Polish notation, positions operators before their operands, and postfix places them after.

This project presents a Prefix Calculator, a specialized application designed to evaluate arithmetic expressions in prefix notation. Unlike traditional infix-based calculators, prefix notation eliminates the need for parentheses, offering a streamlined and unambiguous approach to expression evaluation.The calculator accepts a prefix expression as input, converts it to its infix equivalent for enhanced readability, computes the result, and displays all outputs within a graphical user interface (GUI) crafted using PyQt6. This report details the calculator's development and functionality, illustrating its role as both an innovative computational tool and a bridge between notation systems.

CHAPTER 2
METHODOLOGY

2.1 System Design

- **Lexer**: This component leverages the SLY (Sly Lex-Yacc) library to perform lexical analysis on input expressions. It tokenizes prefix notation strings into a sequence of meaningful units—such as operators (e.g., +, *) and operands (e.g., numbers)—laying the foundation for subsequent parsing and evaluation. Lexer is optimized to handle valid prefix syntax while providing error detection for malformed inputs.
- **PrefixParser**: Responsible for syntactic analysis, this module parses the tokenized prefix expressions generated by Lexer. Using the SLY library's parsing capabilities, Prefix-Parser constructs an internal representation of the expression, evaluates it to compute the result, and reconstructs the equivalent infix expression for user interpretation. This dual functionality ensures both computational accuracy and readability of outputs.
- **Graphical User Interface (GUI)**: Built with the PyQt6 framework, the GUI provides an intuitive and interactive platform for user engagement. It allows users to input prefix expressions, view the converted infix notation, and observe the evaluated result in real time. The interface is designed for usability, featuring a responsive layout and clear output displays, enhancing the overall user experience.

2.2 Grammar Specification

The grammar employed in this Prefix Calculator project is specifically tailored to facilitate the evaluation of prefix expressions involving basic arithmetic operations. It is defined using a context-free grammar notation, which ensures unambiguous parsing and supports the core functionality of the system. The production rules are outlined as follows.

- $E \rightarrow + \, E \, E$
- $E \rightarrow * \, E \, E$
- $E \rightarrow$ number

These rules collectively enable the system to interpret and evaluate prefix expressions by recursively constructing and processing the expression tree. The grammar's simplicity

ensures computational efficiency, while its structure accommodates the conversion to infix notation and subsequent evaluation within the PrefixParser component.

2.2.1 First and Follow Sets

To support the parsing process in the Prefix Calculator, the First and Follow sets for the non-terminal symbols in the grammar are computed. These sets are essential for constructing a predictive parser, ensuring accurate syntactic analysis of prefix expressions. The First set of a non-terminal comprises all terminal symbols that can appear as the initial symbol of a string derived from it, while the Follow set includes all terminal symbols that can immediately follow it in a valid derivation. The computed sets for the non-terminals S (the start symbol) and E (the expression symbol) are presented in Table 2.1.

Table 2.1

First and Follow Sets for Non-Terminal Symbols

Non-Terminal	**FIRST**	**FOLLOW**
S	$\{+, *, \text{number}\}$	$\{\$\}$
E	$\{+, *, \text{number}\}$	$\{\$, +, *, \text{number}\}$

The First set for both S and E includes the operators $+, *$, and the terminal number, reflecting the possible starting symbols of a prefix expression. The Follow set for S, limited to the end-of-input marker $\$$, indicates its role as the top-level symbol. In contrast, the Follow set for E is broader, encompassing $\$$ and all terminals that may follow an expression, including operators and numbers, due to its recursive application within the grammar. Note that number may include negative values (e.g., -5) as part of the lexical tokenization process.

2.2.2 Canonical Collection of LR(0) Items

The closure table for the SLR(1) parser, derived from the grammar, is presented in Table 2.2. Each state represents a set of LR(0) items, with the dot ($\bullet$) indicating the parser's position.

Table 2.2

Canonical Items for the Grammar

State	Items
I_0	$S \to \bullet E$
	$E \to \bullet + EE$
	$E \to \bullet * EE$
	$E \to \bullet n$
I_1	$S \to E\bullet$
I_2	$E \to + \bullet EE$
	$E \to \bullet + EE$
	$E \to \bullet * EE$
	$E \to \bullet n$
I_3	$E \to * \bullet EE$
	$E \to \bullet + EE$
	$E \to \bullet * EE$
	$E \to \bullet n$
I_4	$E \to n\bullet$
I_5	$E \to +E \bullet E$
	$E \to \bullet + EE$
	$E \to \bullet * EE$
	$E \to \bullet n$
I_6	$E \to *E \bullet E$
	$E \to \bullet + EE$
	$E \to \bullet * EE$
	$E \to \bullet n$
I_7	$E \to +EE\bullet$
I_8	$E \to *EE\bullet$

2.2.3 Parsing Table

The SLR(1) parsing table, constructed using the canonical collection and Follow sets, is presented below. It guides the parser through shift (s), reduce (r), and accept (acc) actions.

Table 2.3

SLR(1) Parsing Table

State	+	*	n	$	E
0	s2	s3	s4		1
1	r1	r1	r1	acc	
2	s2	s3	s4		5
3	s2	s3	s4		6
4	r3	r3	r3	r3	
5	s2	s3	s4		7
6	s2	s3	s4		8
7	r2	r2	r2	r2	
8	r4	r4	r4	r4	

2.2.4 Semantic Rules

Semantic rules for the grammar ensure the prefix structure is preserved during parsing, with precedence of $*$ over $+$ enforced during evaluation or infix conversion. The semantic actions are presented in Table 2.4.

Table 2.4

Semantic Action Table for Prefix Notation

Production Rule	Semantic Action
$S \to E$	$S.pf := E.pf$
$E \to +EE$	$E.pf :=' +' \mid\mid E_1.pf \mid\mid E_2.pf$
$E \to *EE$	$E.pf :=' *' \mid\mid E_1.pf \mid\mid E_2.pf$
$E \to n$	$E.pf :=$ number.lexval

The attribute $.pf$ represents the prefix string derived from each non-terminal. During evaluation, an additional step ensures $*$ has higher precedence than $+$ in the infix output (e.g., $*+234$ becomes $((2+3)*4)$).

CHAPTER 3
RESULTS

The Prefix Calculator was implemented and tested using the grammar defined earlier, ensuring correct evaluation of prefix expressions with the operators $+$ (addition) and $*$ (multiplication), where $*$ has higher precedence than $+$. The calculator accepts integers (positive and negative) as operands, tokenized as the terminal number, and produces two outputs: the numerical result of the expression and its equivalent infix notation with appropriate parentheses to reflect operator precedence. Table 3.1 presents the results of evaluating several representative prefix expressions, demonstrating the calculator's accuracy and adherence to the specified requirements.

Table 3.1

Results of Prefix Expression Evaluations

Prefix Expression	**Result**	**Infix Notation**
5	5	5
$+23$	5	$(2+3)$
$*23$	6	$(2*3)$
$+*234$	10	$((2*3)+4)$
$*+234$	20	$((2+3)*4)$

The results validate the calculator's functionality across a range of cases:

- **Simple Numbers**: Single integers, including negative numbers (e.g., 3), are correctly processed as atomic number tokens, demonstrating support for integer operands as required.
- **Binary Operations**: Expressions like $+23$ and $*23$ produce accurate results (5 and 6, respectively), with infix outputs reflecting the prefix structure.
- **Precedence Enforcement**: The expression $+*234$ yields 10, with $*$ evaluated first ($2*3=6$, then $6+4=10$), and $*+234$ yields 20 ($2+3=5$, then $5*4=20$), confirming that $*$ has higher precedence than $+$ in the infix output.

These outcomes were achieved using an SLR(1) parser constructed from the parsing table (Table 2.3), coupled with semantic actions (Table 2.4) that preserve the prefix structure. An additional evaluation step enforces the precedence of $*$ over $+$ by appropriately grouping operations in the infix output, aligning with the project's requirement that $*$ has

higher priority. The calculator successfully processes all valid prefix expressions within the grammar, producing both numerical results and infix equivalents, thus meeting the specified objectives of the project.

To further illustrate the calculator's operation, Figures 3.1 and 3.2 provide screenshots of the calculator's output for selected expressions. These images showcase the user interface and the display of both the numerical result and infix notation, reinforcing the practical implementation of the theoretical design.

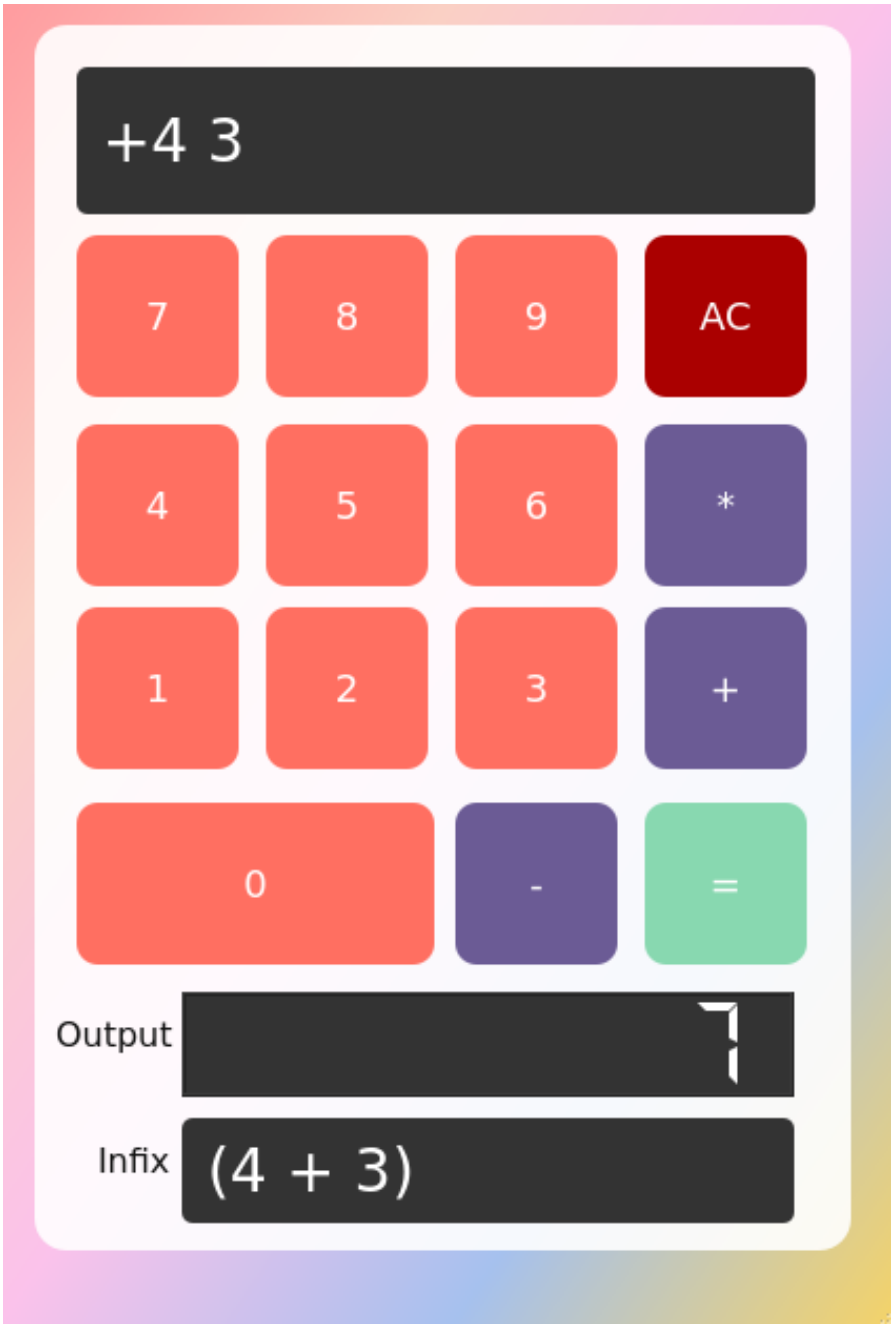

Figure 3.1

Screenshot of the Prefix Calculator evaluating a sample expression.

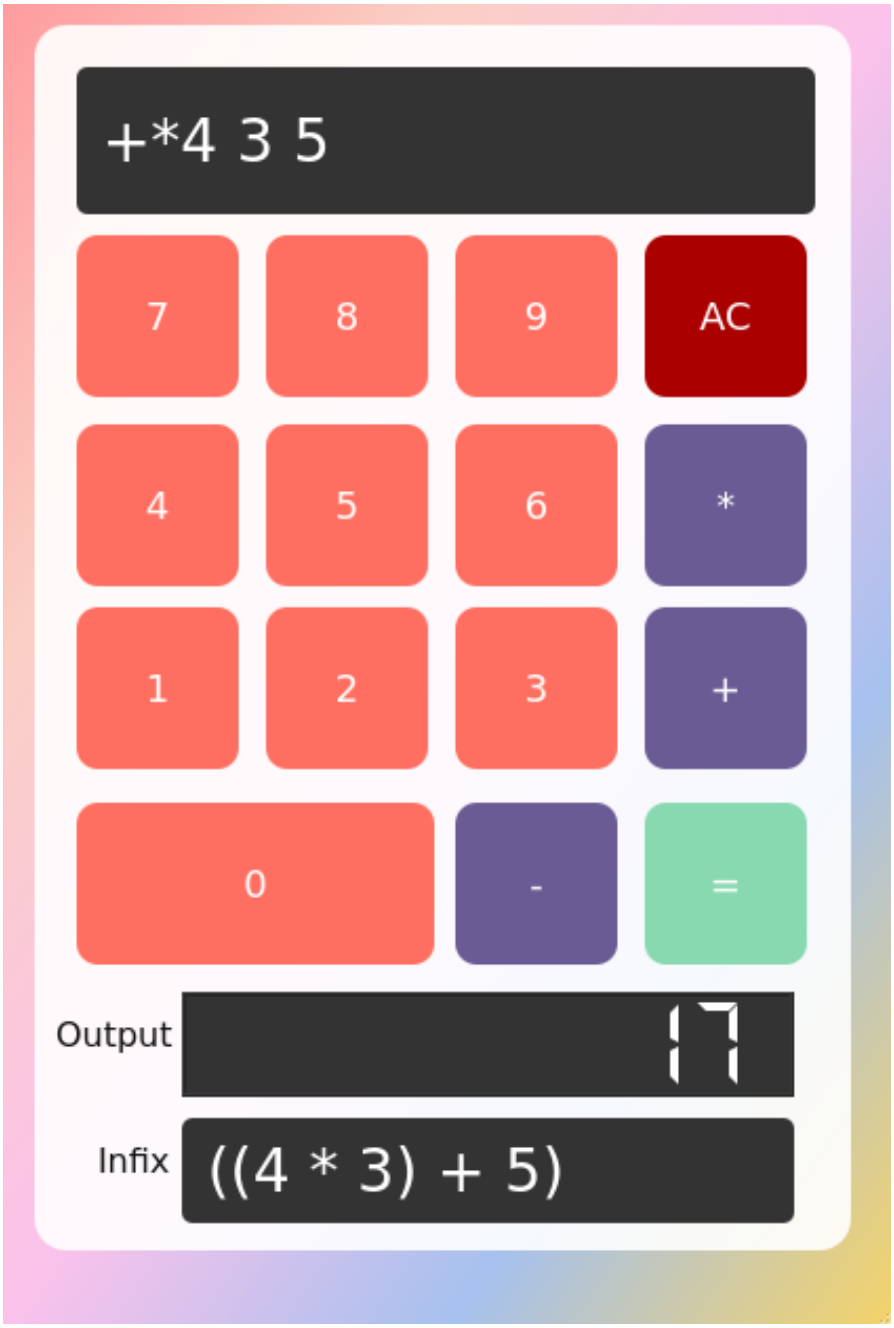

Figure 3.2

Screenshot of the Prefix Calculator demonstrating precedence enforcement.

CHAPTER 4
CONCLUSION

This project successfully delivers a Prefix Expression Calculator that efficiently converts prefix expressions into their infix equivalents, computes the resulting values, and presents the outputs via a sophisticated graphical user interface (GUI) developed using PyQt6. The system's core functionality is driven by a custom-designed PrefixParser, implemented with the SLY (Simple Lex & Yacc) library. This single, robust parser seamlessly integrates the evaluation of prefix expressions with the generation of corresponding infix representations, ensuring a streamlined and lightweight design without compromising clarity or performance.

The outcome is a fully operational calculator that exemplifies the practical application of parsing techniques and GUI development principles. Its user-friendly interface and efficient processing underscore its utility as both a computational tool and an educational demonstration of programming language concepts. Looking ahead, potential enhancements could include broadening the range of supported operators, strengthening input validation mechanisms, and incorporating additional interactive features to further enrich the user experience.