Prefix Calculator

by

Shreeyukta Pradhanang(st125168)

A Project Report 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 a GUI-based Prefix Calculator developed using Python 3.9 and PyQt6, designed for the Programming Language and Compiler course at the Asian Institute of Technology (AIT). The application evaluates arithmetic expressions written in prefix notation, supporting the + and * operators with correct operator precedence. It computes the result and simultaneously converts the input to its infix equivalent, enhancing understanding of expression parsing. The backend leverages lexical analysis and parsing techniques using the sly library to tokenize and interpret expressions. The user interface is built with Qt Designer, offering a clean and interactive way for users to input and visualize expressions. This tool serves as both a functional calculator and an educational aid in understanding prefix expression parsing and compiler principles.

CONTENTS

LIST OF TABLES

CHAPTER 1
INTRODUCTION

In programming language theory and compiler design, understanding how expressions are parsed and evaluated is fundamental. Arithmetic expressions can be written in various notations, with prefix notation (also known as Polish notation) being one of the most significant due to its elimination of parentheses and its recursive structure. This project introduces a Prefix Calculator — a graphical application that interprets, evaluates, and visualizes prefix expressions.

The calculator is developed using Python 3.9 and PyQt6, integrating compiler principles such as lexical analysis, parsing, and syntax tree construction. The application supports basic arithmetic operations (+, *), applies operator precedence rules, and converts prefix expressions into their equivalent infix form, making it easier for users to understand the underlying structure of the expression.

Beyond simple evaluation, this tool provides an educational platform for students and learners to explore how tokenization, parsing rules, and semantic actions work together in a compiler's front-end. By offering both the computed result and the infix conversion, the Prefix Calculator bridges theoretical concepts with practical implementation.

CHAPTER 2
METHODOLOGY

2.1 System Design

The Prefix Calculator is designed as a modular application reflecting a simplified compiler pipeline. It comprises the following components:

- **User Interface (UI)**: Developed using PyQt6 and Qt Designer, allowing users to input expressions and view results interactively.
- **Lexer**: Tokenizes the input expression into recognizable symbols (numbers and operators).
- **Parser**: Implements recursive descent parsing to evaluate prefix expressions based on defined grammar rules.
- **Memory Manager**: A simple data store for variable assignments (extendable for future use).
- **Display System**: Outputs both the evaluated result and equivalent infix expression.

2.2 Lexical Analysis

The lexical analyzer (`MyLexer`) uses the `sly` library to tokenize input strings. It identifies the following tokens:

- `NUMBER`: Integer values
- `PLUS`, `TIMES`: Arithmetic operators
- (Optional) `NAME`, `ASSIGN`: For future variable support

Whitespace is ignored, and regular expressions are used to define token patterns.

2.3 Grammar and Parsing

The calculator uses prefix notation and follows this grammar:

- $S \to E$
- $E \to +E\,E$
- $E \to *E\,E$
- $E \to$ number

Parsing is conducted using a recursive descent parser (`PrefixParser`), which:

- Recognizes the prefix structure

- Recursively evaluates the expression
- Simultaneously constructs an infix representation

2.4 Expression Evaluation

Evaluation is integrated within the parser. For each valid expression:

- Arithmetic operations are computed recursively.
- A stack is used to construct the infix string in parallel.

Example: For the input `+ * 2 3 4`

1. Parse `* 2 3` into `6`
2. Evaluate `+ 6 4` into `10`
3. Construct infix: `((2 * 3) + 4)`

2.4.1 First and Follow Sets

In context-free grammar analysis, the **FIRST** set of a non-terminal symbol represents all the terminal symbols that can appear at the beginning of some string derived from that symbol. For the prefix calculator, since `S → E`, we have `FIRST(S) = FIRST(E) = { +, *, number }`, as any valid expression begins with an operator or a number.

The **FOLLOW** set contains all terminals that can legally appear immediately after a non-terminal in some derivation. `FOLLOW(S)` includes only { $ } since it is the start symbol and is followed by end-of-input. `FOLLOW(E)` includes { +, *, number, $ } because in rules like `+ E E`, the second `E` could appear in the middle or at the end of an expression. These sets are essential for guiding top-down parsers like recursive descent to resolve which rule to apply next.

FIRST Sets

- FIRST(S) = { +, *, number }
- FIRST(E) = { +, *, number }

FOLLOW Sets

- FOLLOW(S) = { $ }
- FOLLOW(E) = { +, *, number, $ }

2.4.2 Canonical Collection of LR(0) Items

Table 2.1 illustrates the closure table constructed for the SLR(1) parser based on the given grammar. Each state in the table corresponds to a collection of LR(0) items, where the position of the dot (●) indicates the current point of parsing within a production rule.

Table 2.1

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	$E \to *EE\bullet$

2.4.3 Parsing Table

The SLR(1) parsing table, derived from the canonical collection of LR(0) items and the FOLLOW sets, is shown below. It directs the parser's behavior using actions such as shift (s), reduce (r), and accept (acc).

Table 2.2

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	

2.4.4 Semantic Rules

The grammar's semantic rules are designed to retain the prefix format throughout parsing, while also enforcing the correct precedence of * over + during both computation and conversion to infix notation. These semantic actions are detailed in Table 2.3.

Table 2.3

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

To represent prefix expressions during parsing, each non-terminal is associated with an attribute that constructs the corresponding prefix form. When converting to infix notation, operator precedence is explicitly handled to ensure accurate translation. For example, the prefix expression $* +\ 2\ 3\ 4$ is correctly transformed into its infix equivalent $((2 + 3) * 4)$, maintaining the intended precedence of multiplication over addition.

2.4.5 GUI Implementation

The graphical user interface (GUI) for the prefix calculator was developed using **PyQt6**, providing an intuitive and visually appealing layout for user interaction. The interface includes an input field for entering prefix expressions, a button grid for digit and operator inputs, and output fields to display the evaluated result in both infix notation and numeric form. QPushButtons were dynamically connected to logic using Python slots and signals, enabling real-time interaction and evaluation. Styles were applied using QSS (Qt Style Sheets) to enhance the visual experience and usability of the calculator.

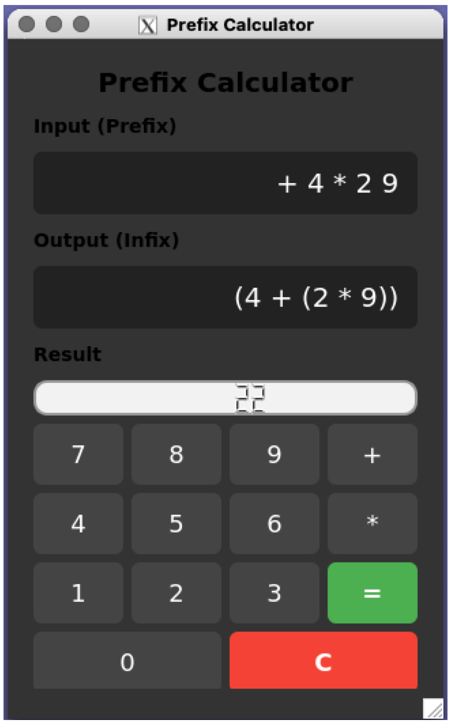

Figure 2.1

Graphical User Interface of the Prefix Calculator

2.4.6 Evaluation

The core logic of the application is built around a recursive parser that evaluates prefix expressions. The process begins with lexical analysis using a custom lexer implemented with the `sly` library, which tokenizes the input string into operators and numbers. The parser then interprets these tokens based on a set of defined grammar rules:

$$E \rightarrow +\ E\ E$$

$$E \rightarrow *\ E\ E$$

$$E \rightarrow \text{number}$$

As each subexpression is parsed, the parser recursively computes the result of arithmetic operations while simultaneously constructing the corresponding infix expression. This dual-pass approach enables the calculator to show both the numeric answer and a human-readable version of the original input.

CHAPTER 3
CONCLUSION

This project successfully demonstrates the development of a functional prefix calculator with a graphical user interface. It allows users to input arithmetic expressions in prefix notation and view both the evaluated result and the corresponding infix form. The system integrates a custom lexer and a recursive parser, showcasing fundamental concepts of compiler construction, such as lexical analysis, syntax analysis, and expression evaluation.

The use of PyQt6 provides an interactive and user-friendly interface, making the calculator accessible and visually engaging. The architecture of the application separates concerns cleanly between the GUI and core logic, ensuring maintainability and modularity. Overall, the project bridges theoretical compiler concepts with practical application development, serving as both an educational tool and a working software solution.