Visualisation of Pratt Parsing

6CCS3PRJ Final Project Report

Author: Taherah Choudhury

Course: BSc Computer Science

Student ID: 20041339

Supervised by Dr Hubie Chen

April 2024

Originality avowal

I verify that I am the sole author of this report, except where explicitly stated to the contrary.

I grant the right to Kings College London to make paper and electronic copies of the submitted work for purposes of marking, plagiarism detection and archival, and to upload a copy of the work to Turnitin or another trusted plagiarism detection service.

I confirm this report does not exceed 25,000 words.

Taherah Choudhury

April 2024

Abstract

Parsing is a process that breaks down code (or sentences) into components based on the syntax (or grammatical) rules of the language.

This project aims to focus on a particular parser known as the Pratt Parser. Pratt parsing is a technique used to solve numeric infix expressions. Operators are assigned precedence values that determine which part of the expression will be calculated first.

The concept that pratt parsers use is known as Top-down Operator Precedence, an eloquent way of solving the Operator Precedence Problem. ‘Top-down’ describes how the parse tree is constructed, starting from the ‘root’ and working its way down to the ‘leaves’. In this report, we go into more detail and look at each the two kinds of parsing in the context of pratt parsing. We will study and explore the concept of parsers in general, focusing on precedence parsers and an existing software application.

Although the algorithm appears short and simple, grasping the intuition behind it can be difficult. The goal is to implement an interactive software which provides the user a visual way to analyse and understand the concept of the Pratt Parser through diagrammatic guides and examples.

Acknowledgement

I would like to express my gratitude towards my supervisor Dr Hubie Chen for his advice and guidance throughout the project. I am thankful to him for providing me information and help with all my queries. His knowledge has truly helped me with the completion of this project.

Contents

Chapter 1

Introduction

‘Parsers’ are a powerful tool used within the field of Computer Science and Linguistics. The process concerns the breaking down of some piece data into meaningful components based on certain grammatical rules making it easier to understand and process it. For the pratt parsing algorithm, certain elements could be tricky to grasp. Therefore, having the ability to visually investigate and explore the various intricacies can help an individual develop and improve their understanding.

1.1 Project Motivation

Pratt parsing is a fascinating parsing technique that is simple in its design but powerful in its capabilities. Therefore, building a software application that breaks down this parsing technique via examples in a visual manner poses as an interesting and useful challenge.

Ultimately, we want to implement a software application that thoroughly visits all the iterative steps when the pratt parser is executed on a given numeric expression. A key component of a pratt parser is the operator’s precedence levels. Assigning precedence values to each operator is essentially a way of implicitly parenthesising the expression so it can be parsed and calculated correctly. In Chapter 2, we look at Operator-Precedence and its significance to pratt parsing in more detail.

Enabling the user to visualise and explore how the pratt parser works can help with understanding how parsers handle and process expressions. Furthermore, providing a tool which produces a breakdown of a specific algorithm can serve as a way of comparing other parsing algorithms and how they differ from one another.

The software will be visualised in two different ways. The first way will be in a tree representation. The second way is as a data table. This table with be an alternative way to show the recursive steps carried out by the pratt parser. These two visual methods intend to be interchangeable whenever the user desires.

The functionality of this software aims for the user to study the algorithm in detail increasing or creating an in-depth understanding of how pratt parsing performs and behaves. Additionally, the user can select from a wide range of example expression as well as input their own expressions. However, the given example expressions intend to provide a wide range of inputs so the user may not need to build their own input expressions. Executing the parsing algorithm on various examples reinforces ones understanding better.

1.1 Aims & Objectives

When determining the software requirements at a later stage of development, it would be beneficial to lay down some core objectives. Also, it's important to identify research objectives before beginning the software implementation phase. It will provide us with a clearer understanding of the scope of the project and what essentials we must observe.

1.1.1 Core Objectives

These objectives are fundamental when building the desired software system:

1. Research and understand the role of a parser and the different classification parsers fall under.
2. Understand how the pratt parsing algorithm works with working examples.
3. Study what existing software is out there and the similarities with the desired software.
4. Agree on a list of requirements which the system must abide by.
5. Discuss the pros and cons of the different parsing algorithms and how they differ from the pratt parsing algorithm.
6. Build a software system that can execute the pratt parsing algorithm on a variety of numeric expressions.
7. Build an interactive user interface that allows users to navigate around the executed parsing algorithm.

Chapter 2

Background

This chapter involves studying and discussing the concept of parsing and understanding the different categories of parsers. We also look at how they link or differ to the pratt parsing technique.

2.1 Parsers

In linguistics, parsing is a process used in breaking down the meaning of sentences based on the language's specific grammatical rules. In Computer Science, parsing is the breaking down of source code based on syntax rules of the programming language. This kind of parsing is a significant process in compiler design. Here, parsers fall into two main categories, Top-down and Bottom-up parser. The main difference is the order in which they construct the parse tree, given a string. Depending on the grammar we are dealing with, we choose the parser type best suited for the language we are processing. Parsing used in Natural Language Processing (NLP) and is referred to as Syntactic analysis.

2.1.2 Syntax Analysis

Pratt parsing requires a general understanding of parsing and syntax analysis is one of the most known forms of parsing. Thus, understanding its operations and functions leads to a more in-depth comprehension of parsing.

The objective for syntax analysis is to analyse the given input program and verify whether it follows the grammar rules of the programming language. In the article "What is Parsing in NLP?" the author describes syntax analysis as "the process of analyzing the strings of symbols in natural language in accordance with the norms of formal grammar" [5]. Hence, the syntax analyser can catch syntax errors within the programming language during compile time. This is essential for compilers as without this process the compiler may suffer from unexpected errors later in the compiler stage.

2.2 Top-down Parsing

Top-down Parsing is a powerful parsing technique used in natural language processing and compiler design. The aim for this technique is to begin at the head of the given structure and recursively apply rules based on a particular grammar until it reaches the tail of the structure. The 'top-down' refers to the way it parses, constructing a parse tree working its way from the 'root' to the 'leaves'.

This algorithm is often too simple for parsing in practice and instead the concept is combined with extra steps for more complex problems. As said in Jeffrey Kegler's blog [6] 'Because a top-down parser is extremely simple, it is very easy to figure out what it is doing' and 'In its purest form, this idea is too simple for practical parsing'. On the contrary, the simplicity is a significant characteristic of this parser. This is due to the customizability when faced with problems that require 'special algorithms' [6] where we can easily adapt the parsing algorithm to fit our needs.

3.2.1 Recursive Decent Parsing

Recursive Decent parsing is a subcategory of top-down parsing. In terms of a parse tree, the algorithm starts at the 'root' and works its way down to the 'leaves. The basic idea is that non-terminals are associated with procedures or parsing functions. This is where the recursion element comes to play as the parsing functions call themselves when encountered by a non-terminal symbol that is associated with its own parsing function. Hence, the recursive call happens, although the calls are not always recursive.

In terms of pratt parsing, the algorithm processes each token of the expression and proceeds to make a recursive call that takes in the remaining tokens left to parse. This is where pratt parsing displays similar characteristics of recursive decent parsing.

3.3 Bottom-up Parsing

Bottom-up Parsing is another kind of parsing technique. Unlike, top-down parsing, the algorithm works ‘bottom-up’. Symbols are pushed into the stack leaving the symbol on the top being the last symbol in the syntactic structure. In terms of parse trees, the leaves of the tree are constructed first, and the root is constructed last. Although the pratt parser is a top-down parsing technique, it does possess certain characteristics of bottom-up parsing. More specifically, operator precedence parsing.

3.3.1 Operator Precedence Parsing

Operator precedence parsing is a shift-reduce parsing method. The technique revolves around solving numeric expressions based on their precedence levels. This allows any infix expression to be calculated without the need for brackets. Expressions are calculated differently based on the operator’s precedence. Take for example the following expression (directly taken from tutorialspoint [2]):

a + b * c

As multiplication has greater precedence than addition, we will then calculate b * c first. In a sense, it adds brackets implicitly as the expression,

a + (b * c)

will give us the same result. In cases where we have equal precedence association rules will be applied. In pratt parsing, this technique is used for calculating expressions. Hence the name top-down precedence parsing when we talk about pratt parsing.

3.4 Infix Expressions

The pratt parser we are studying works for only numeric infix expressions (the form x operator y). This is mainly why we need to observe the operator precedence table to determine what part of the expression must be calculated first. The algorithm places implicit brackets after looking at the operator precedence's. For example, the infix expression,

10+10-5/2

will be parsed based on the operator precedence level as,

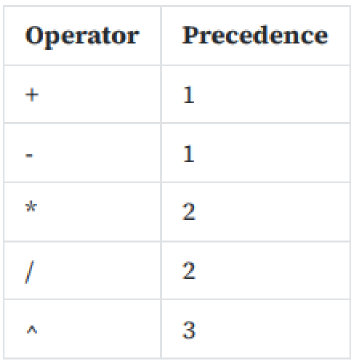
((10+10) -(5/2))

This is an example of how the algorithm places implicit brackets after observing the precedence levels. The pratt parser then calculates the parsed expression.

3.5 Visualising Pratt Parsing

To understand Pratt parsing, we must also understand precedence parsers. Pratt parsing algorithm uses the concept of 'Top-Down Operator Precedence,' developed originally by Vaughan Pratt. This parsing algorithm is an eloquent way to solve the operator precedence problem. Operators are associated with a number based on their precedence. The bigger the number the greater the precedence. Pratt parser carries a precedence limit. When an operator precedence value is less than the precedence limits the algorithm does not parse that operator. As Martin Janiczek describes in his paper 'Demystifying Pratt Parsers,' "You're only allowed to parse operators with precedence higher than this number." [3].

When understanding the way, the pratt parser looks at each operator we must know the operator precedence table that the pratt parser is using. Below is an example of a precedence table that a pratt parser may be using.



Operator	Precedence
+	1
-	1
*	2
/	2
^	3

Figure 1: From Martin Janiczek paper [4] an operator precedence table.

Pratt parsing uses the combination of top-down parsing and bottom-down parsing. More specifically, the use of operator-precedence parsing and recursive descent parsing. There are a variety of ways to visually depict such an algorithm. In Martin Janiczek's paper there is a flow chart that describes the process of pratt parsing.

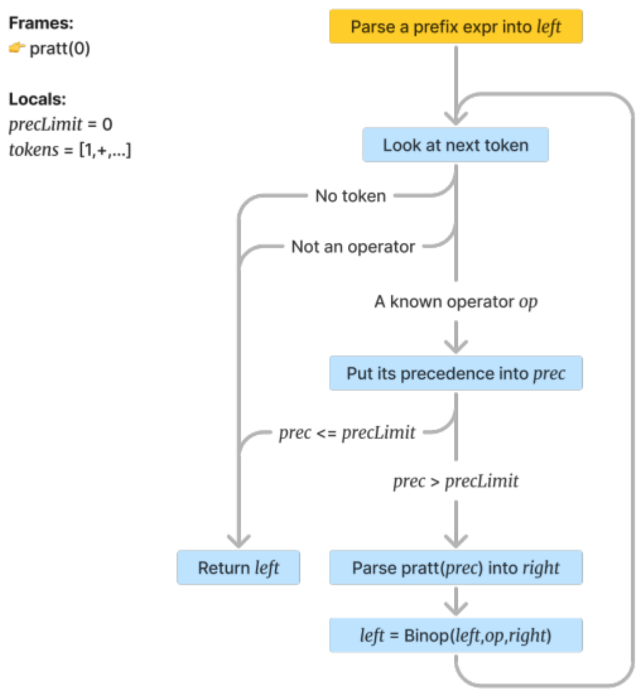
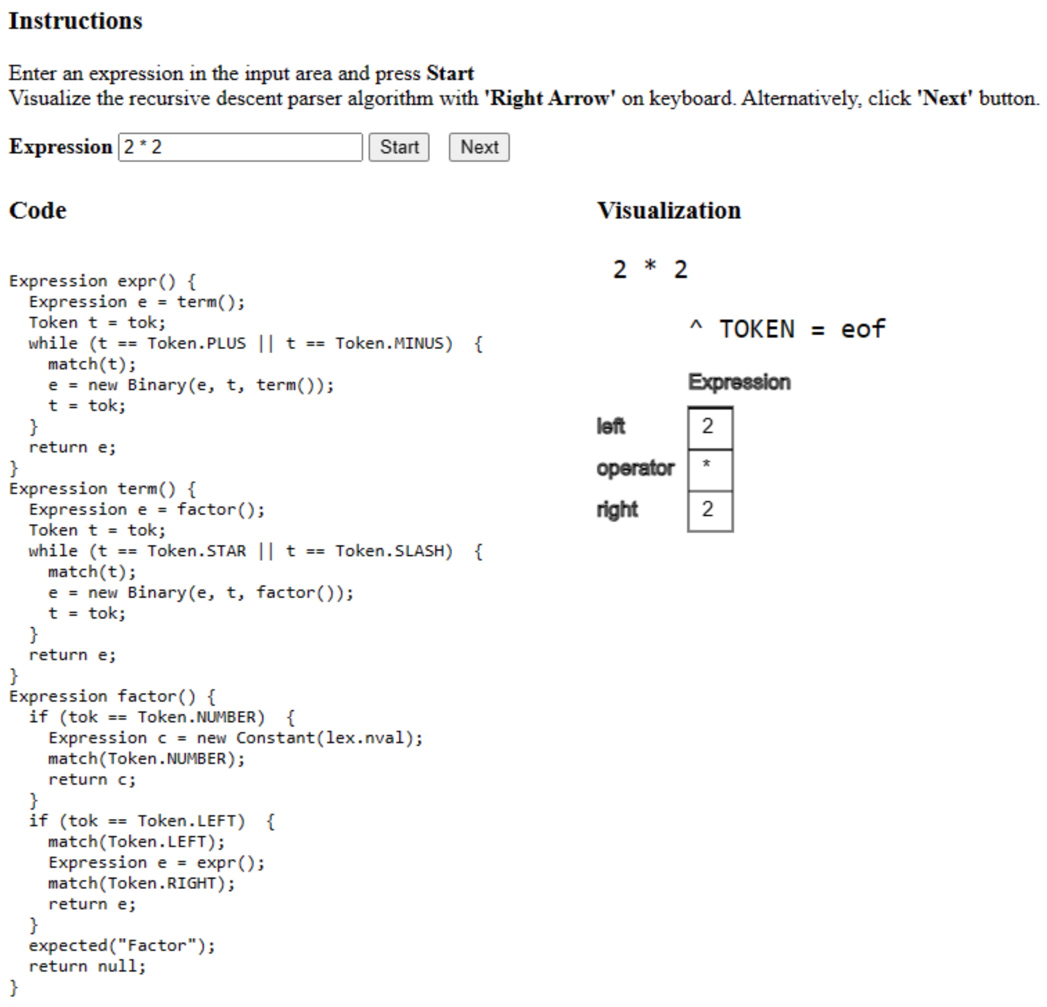


Figure 2: The flow chart from the paper 'Demystifying Pratt Parsers'[3]

The flow chart above shows the execution of the pratt parser. The flow chart organizes the order of the algorithm and the different branches for different scenarios. For learning the algorithm, this is an excellent way to understand and process what exactly a pratt parser is doing.

2.1 Example Software

Currently, there isn't any established software that visually depicts pratt parsing on numeric expressions. However, there exists a software that uses recursive-decent parsing to solve numeric expressions. The page made by Bilal Ekrem HARMANSA [5] has an example of a basic software that visually describes how a given numeric expression is parsed using recursive-decent parsing.



Instructions

Enter an expression in the input area and press **Start**
Visualize the recursive descent parser algorithm with **'Right Arrow'** on keyboard. Alternatively, click **'Next'** button.

Expression 2 * 2

Code

```
Expression expr() {
  Expression e = term();
  Token t = tok;
  while (t == Token.PLUS || t == Token.MINUS)  {
    match(t);
    e = new Binary(e, t, term());
    t = tok;
  }
  return e;
}
Expression term() {
  Expression e = factor();
  Token t = tok;
  while (t == Token.STAR || t == Token.SLASH)  {
    match(t);
    e = new Binary(e, t, factor());
    t = tok;
  }
  return e;
}
Expression factor() {
  if (tok == Token.NUMBER)  {
    Expression c = new Constant(lex.nval);
    match(Token.NUMBER);
    return c;
  }
  if (tok == Token.LEFT)  {
    match(Token.LEFT);
    Expression e = expr();
    match(Token.RIGHT);
    return e;
  }
  expected("Factor");
  return null;
}
```

Visualization

```
2 * 2

    ^ TOKEN = eof
```

	Expression
left	2
operator	*
right	2

Figure 3: Screenshot of webpage titled 'Visualization of Recursive Descent Parser'[5]

Provided above is a screenshot of the interface of the software at the final stage of the algorithm. When you enter a numeric expression, and press start, it displays the expression to be parsed. Then as you press the 'next' button it takes you through the steps highlighting each code that is being executed along with tables that are updated at each step. Although this is very simple it gives an idea of how you can display the various steps of an algorithm visually for explanation and learning purposes.

Chapter 3

Requirements

4.1 Brief

The goal of the project is to build a software that executes the pratt parsing algorithm on an expression and provides the user an interactive and visual representation of the execution and output. The system must provide users with features and functionalities with the purpose to assist in understanding pratt parsing.

4.2 System Requirements

The following subsections list the functional and non-functional requirements. It's essential to have a clear understanding of the scope of the project and being able to define and categorise the requirements.

4.2.1 Functional Requirements

The **functional requirements** here provide a clear and detailed description of what the software should do based on the scope of the project.

Core: these must be met at the end of the project.

- The algorithm must work for all expressions that use the multiply, divide, addition and subtract operator.

- Input string expressions must be tokenised into operands and operators.
- Appropriate error messages should be displayed when algorithm encounters an error for invalid input expressions.
- The user must be able to choose a table or tree representation to be displayed when they run the pratt parser.
- The user can go back and forth from each step of the pratt parser.

Desired: if possible, within the time constraint.

- The UI must produce moving visuals when the pratt parser is executed.
    1. These visuals can be paused reversed or fast forwarded.
- The user can interact with nodes in the tree representation.
    1. Be able to expand a node and get an explanation for that node.
    2. Be able to search for types of nodes using the search bar.

4.2.2 Non-Functional Requirements

Below is a list of the **non-functional requirements** where we look at how the software should perform and behave. It's important to identify such requirements so we can build a software that is well refined.

- The UI should be smooth and intuitive for the user to interact with.
- The algorithm should be able to take input string expressions of varied sizes.
    1. Longer input strings should be processed fast as shorter strings.
- The user should be able to get clear and correct visual depiction the pratt parsing algorithm consistently with little errors.
- Massive input expressions must be able to be processed withing an acceptable time.

Chapter 4

Design & Implementation

4.3 Explanation of the Pratt Parsing Algorithm

The pratt parsing algorithm used is based on the pratt parser written by Dr Hubie Chen in Lua code. The algorithm can process both expressions with brackets and without brackets.

The purpose of this algorithm is to parse a numeric expression and output the calculated result. For the algorithm to work, the expression must be broken down into tokens. Tokens represents either a number, operator, or bracket. Each token is then processed one at a time.

This algorithm centres around the two function ‘pratt’ and ‘ploop’. The expression first enters the ‘pratt’ function and begins by looking at the tokens. If there are no tokens, the function simply returns the empty tokens list. In the case where there are tokens, the function looks at the first element or head of the tokens list. If the expression begins with ‘(’ this is ignored and instead the rest or tail of the tokens are passed by the function when it makes a recursive call. For example, the expression:

(2*2)/2

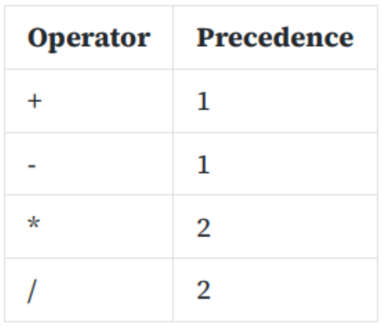
When this expression is tokenised, we get the tokens,

‘(’, ‘2’, ‘*’, ‘2’, ‘)’, ‘/’, ‘2’

The ‘pratt’ function with look at the first token ‘(’ and proceed to ignore it and continue the process on the remaining tokens, ‘2’, ‘*’, ‘2’, ‘)’, ‘/’, ‘2’.

When the first part of the expression is a number, the ‘pratt’ function executes the ‘ploop’ function. Function ‘ploop’ is where all the essential functionality of the parser takes place. In the instance where there are no tokens the function returns the current values the parser holds. In all other cases, the function looks at the first token. When an operator is identified, the precedence is compared to the current precedence or ‘limit’. If the precedence is less than or equal to the current precedence the current token is returned alongside the rest of the tokens. When the precedence is higher, the function ‘pratt’ is called passing in the new precedence and all the tokens except the operator token that was identified. Eventually, the ‘pratt’ function returns the final state where all the tokens have been parsed and there are no more tokens remaining. Finally, the values returned or parsed by the ‘pratt’ function are used to calculate the answer to the expression by the ‘ploop’ function making recursive calls. The final output is the answer and the empty list of tokens.

The precedence is used in a way that builds these invisible brackets around the expression before calculating the result. Ideally, we want to show the user this, to explain how the parser is processing the expression tokens in a visual manner. Below is the precedence table the algorithm will be using.



Operator	Precedence
+	1
-	1
*	2
/	2

Figure 4: Table from the paper written by Martin Janiczek[3]

4.4 Initial Class Diagram

The following class diagram (Figure 4) demonstrates the classes required and the relationships between them. This class diagram is designed based on the pratt parsing algorithm. Later, we introduce more classes and more relationships when looking at the GUI for the software.

The ‘PrattParser’ class is the main class where the pratt parsing algorithm exists. The ‘ParsingResult’ Class is the output results of the pratt method. The PrattParser has a composition relationship with the ‘Operator’ Class and ‘ParsingResult’ Class. This is because the PrattParser is a class that manages the creation of those classes. Without the PrattParser class the remaining classes become redundant. Moreover, a single instance of PrattParser contains more than one instance of Operator and ParsingResult.

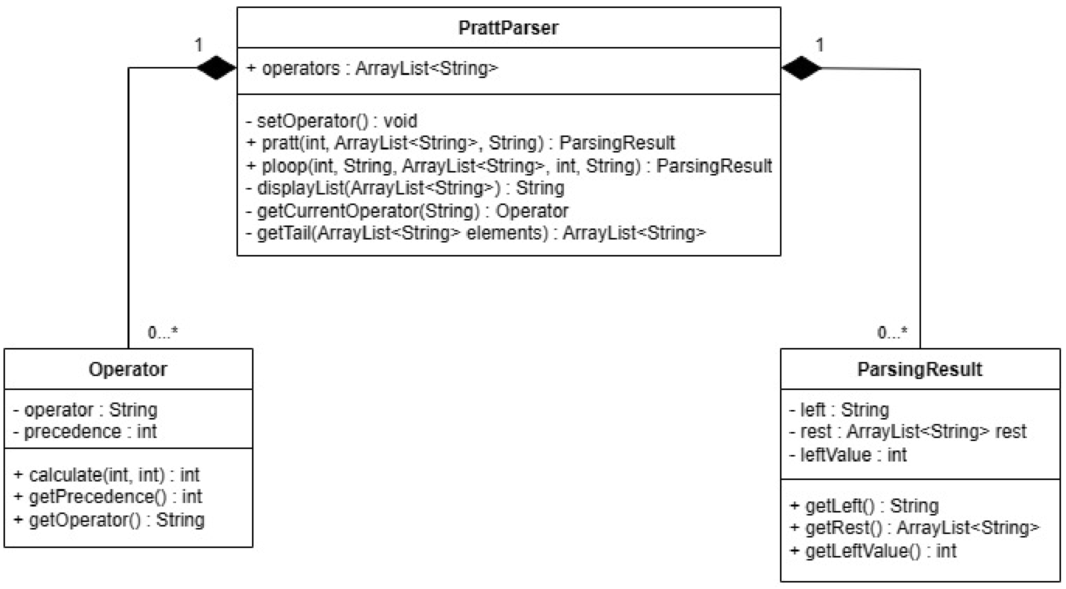

PrattParser
+ operators : ArrayList<String>
- setOperator() : void
+ pratt(int, ArrayList<String>, String) : ParsingResult
+ ploop(int, String, ArrayList<String>, int, String) : ParsingResult
- displayList(ArrayList<String>) : String
- getCurrentOperator(String) : Operator
- getTail(ArrayList<String> elements) : ArrayList<String>
1
1
0...*
0...*
Operator
- operator : String
- precedence : int
+ calculate(int, int) : int
+ getPrecedence() : int
+ getOperator() : String
ParsingResult
- left : String
- rest : ArrayList<String> rest
- leftValue : int
+ getLeft() : String
+ getRest() : ArrayList<String>
+ getLeftValue() : int

Figure 5: Class diagram for the system

4.5 Final Class Diagram

When designing the initial class diagram, it was not taken into consideration how the pratt parser will be executed on an expression. Thus, a new class was designed, responsible for the setting up and execution of the pratt parser on a given expression.

Figure 5 is the updated class diagram. It is same as the initial diagram but with a new class 'ParserExecute' where the numeric expression is tokenised and the pratt function is called on the tokenised expression. The 'ParserExecute' contains one instance of the 'PrattParser' and the 'ParsingResult' on object creation. Unlike the 'ParsingResult', the 'PrattParser' can exist without 'ParserExecute'. Hence the relationship is described as an aggregation.

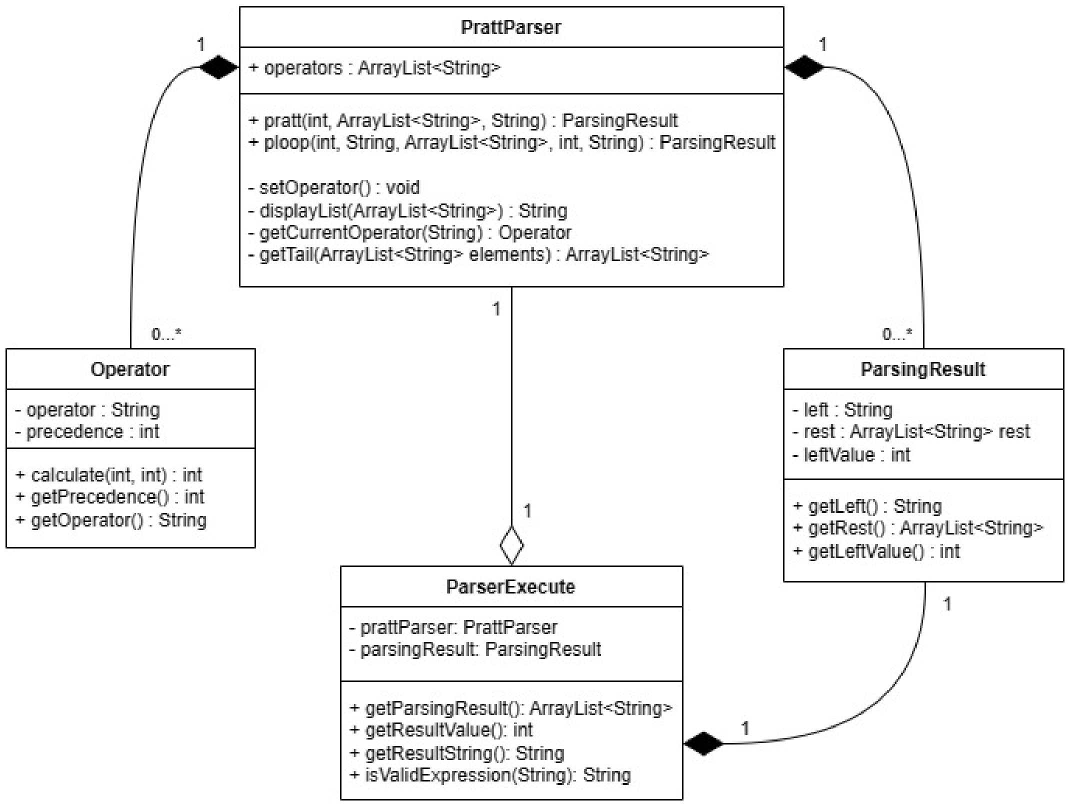

PrattParser
+ operators : ArrayList<String>
+ pratt(int, ArrayList<String>, String) : ParsingResult
+ ploop(int, String, ArrayList<String>, int, String) : ParsingResult
- setOperator() : void
- displayList(ArrayList<String>) : String
- getCurrentOperator(String) : Operator
- getTail(ArrayList<String> elements) : ArrayList<String>
1
1
1
0...*
0...*
Operator
- operator : String
- precedence : int
+ calculate(int, int) : int
+ getPrecedence() : int
+ getOperator() : String
ParsingResult
- left : String
- rest : ArrayList<String> rest
- leftValue : int
+ getLeft() : String
+ getRest() : ArrayList<String>
+ getLeftValue() : int
1
1
ParserExecute
- prattParser: PrattParser
- parsingResult: ParsingResult
+ getParsingResult(): ArrayList<String>
+ getResultValue(): int
+ getResultString(): String
+ isValidExpression(String): String
1
1

Figure 6: Updated Class diagram for the system

4.6 Design Choices

4.6.1 Algorithm Structure

The original Lua code (written by Dr Hubie Chen) does error checking of the expression in the main algorithm. Since, the software to be developed intends to provide the user with a selection of input expressions as well as the ability to input their own expression, it would be a desired choice to carry out error checking during this stage where the user is choosing their input. As a result, this will reduce the code required in the main algorithm and minimise the different tasks one method may carry out.

4.6.2 User Interface Appearance

The software is a learning tool that is informative and clear to understand. To achieve this, the user interface must be simple and easy to use with minimal distractive components. Text must clear for the user to read when explaining a component or giving instructions.

Figure 7 shows a screenshot of a potential design for the start screen when the application is launched. The screen was designed using SceneBuilder. The background is simple yet appealing with the central text precisely explaining what this application is for. As desired, and text can be read and understood easily. This design will be consistently repeated for all screens in the application.

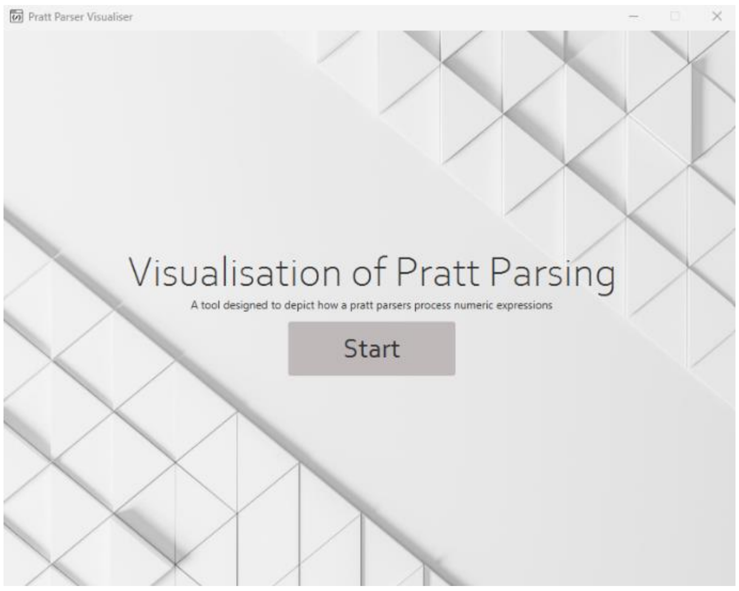

Pratt Parser Visualiser
Visualisation of Pratt Parsing
A tool designed to depict how a pratt parsers process numeric expressions
Start

Figure 7: Start screen from the software.

4.7 Object Orientated Solution

The system was implemented using JavaFX with SceneBuilder. This section will cover the main Java classes and briefly look at how they link with the GUI classes.

The software involves four main classes ‘ParserExecute’, ‘PrattParser’, ‘ParsingResult’, ‘Operator’. ‘ParsingStatus’ was later introduced for the GUI for the purpose of populating the table view. During the implementation, the ‘ParserExecute’ was implemented to carry out all the extra set up phases before the ‘pratt’ method is called. Initially, the validation code was written under the ‘PrattParser’ class. However, the class began to become congested with several methods reducing the level of modularity inside the class. It also made more sense for the validation to be carried out by that class, keeping the ’PrattParser’ code exclusive to the pratt parsing algorithm.

4.7.1 Description of Java Classes

Now let’s discuss each Java class in more detail, understanding their roles and significance to the overall software. The JavaFx classes are not discussed here as we are only looking at the backend implementation.

‘ParserExecute’

The main purpose of this class is to turn an expression into tokens then execute the pratt parser on that expression. It also handles verifying if given expression is a valid expression before the pratt parser is called.

In this implementation the controller class ‘PrattController’ uses the ‘ParserExecute’ This controller class calls the ‘ParserExecute’ to perform validation on expression before it executes the pratt parser on the input expression. Once the expression is validated and checked, ‘ParserExecute’ calls the ‘PrattParser’ to execute the ‘pratt’ function.

As previously stated, the class was introduced in the final class diagram design. This design choice has many benefits. Identifying certain functionalities in a class that can be placed in a new separate class is essential when refactoring and cleaning up code. By implementing ‘ParserExecute’, the code can now easily be adapted to write additional parsing algorithms in the future. The design reduces issues of dependencies when deciding to implement more parsing algorithms.

‘PrattParser’

This is where the core functionality of the pratt parser takes place. Contains a list of operators and the two main functions ‘pratt’ and ‘ploop’. As discussed before early in the chapter, these are the functions responsible for parsing a given expression and returning the result. At each iterative step of the algorithm, an informative string of text is added to a list and current values are added to another list. This is used by the class 'ParserExecute' as its output. This is then what is displayed to the user.

'ParsingResult'

This class stores the result of each step of the pratt parsing algorithm. Utilised by the class 'PrattParser'. The final output for the algorithm is the a 'ParsingResult' object containing the final parsed expression along with the answer to the expression.

'ParsingStatus'

This class is used to store the tokens and current token and operator being processed at each stage of the parsing algorithm. This class was created for the purpose to display to the user the different state at different steps of the parsing algorithm. The data within this class is displayed to the user in the form of a table.

'Operator'

This class stores the operator and its associated precedence. Furthermore, it calculates two operand numeric expressions based on the operator it contains.

4.7.2 Java Functions

While implementing the code for validating input expressions, it's worth to mention some interesting private methods that were coded to help with validation.

Class: 'ParserExecute' Function: 'isBracketsValid()'

This function checks for brackets found in the expression that are incorrect. For example, brackets which do not contain a closed bracket or an open bracket. The function uses a balanced bracket algorithm. First an empty stack is declared named "brackets". The algorithm works by looping through the input string of brackets. When encountered by an

“(” the bracket is pushed into the stack. When the “)” is found, the top element of the stack is popped indicating a successful bracket pair has been found. This works for any number of brackets and if at the end of the loop the stack is empty the input bracket string is valid.

Class: ‘ParserExecute’ Function: ‘getTokens()’

This function is an essential part of the core algorithm. This is because the ‘pratt’ function takes in the input expression in the form of tokens. So, this function is responsible for taking a string expression and splitting the expression into brackets, operators, or operands. In this implementation they are stored in a list. This function makes use of the function ‘isNumber()’to check whether or not each component of the string is an integer.

4.8 Graphical User Interface

When designing the GUI for the software prototype we need to decide the different components that are involved in producing a user interface. The system contains two main panels, the selection panel and the pratt parser panel. These panels interact closely with one another, where one determines the display of the other. These were designed and implemented based on the Model-View Controller (MVC) architectural design pattern.

4.8.1 Description of JavaFX Classes

The application was build using JavaFX and involves the use of the Model-View Controller (MVC) design pattern. The class ‘ParserExecute’ is this applications Model class. This subsection describes the Controller classes and the Views built using Scenebuilder.

‘StartController’

The ActionEvent method ‘onBtnStartClick’ is the only functionality the controller class has. It opens (or launches) a new scene when the user clicks the ‘btnStart’.

‘PrattController’

This is the main controller class is responsible for all the coordinating between the Model class ‘ParserExecute’ and the view ‘pratt-view’. All other Java classes interact with each other or with the ‘ParserExecute’ class. This class stores all the pratt parsing output results that is used by the ‘PrattController’ to update the view depending on the different way a user interacts with the view.

‘start-view.fxml’

The view is the first view the user encounters when the application is executed. It’s a simple view where there is one single interaction the user makes. This is the interaction with the ‘start’ button, and when clicks sends the user to the next view, the ‘pratt-view’.

‘pratt-view.fxml’

The ‘pratt-view’ is the view where most of the user interaction will take place. This is the view where the core functionalities of the prototype system are carried out. When a user selects or inputs an expression the ‘PrattController’ responds to this interaction by updating the ‘pratt-view’ to display the parser pane. Inside the parser pane, users can execute the pratt parsing algorithm on their expression. Following such interaction, the ‘PrattController’ works with the Model class ‘ParserExecute’ to update the FXML view file with the correct output of the pratt parsing algorithm. This is dependent on whether the user choses the branch display or the table display.

4.8.2 Description of Modular System Structure

The Start Pane

This is the first pane shown to the user when the application is executed. This pane contains the title of the application and a summary of the purpose of the application. When the user clicks start, they are sent to the corresponding pane.

The Selection Pane

After clicking start the user will be presented with this pane. The pane contains a list of expressions the user can select from. Additionally, there is a text field where the user can input their own expression. User must select an expression or input an expression that follows the accepted format before being sent to the next pane.

The Parser Pane

When user selects or inputs a valid expression, this pane is displayed to them. This pane is responsible for the central functionality of the software. The user will have their chosen expression displayed to them. User can now begin the pratt parser by clicking start. Along with the 'Start' button, the pane contains the buttons 'Next', 'Previous', 'Finish' and 'Reset'. These are the navigation buttons, designed for the user to incrementally go through the pratt parser. The user can go back and forth from the different stages of the algorithm as the wish as well as fast forward to the end by clicking the 'finish' button. Furthermore, the user can reset to clear the output and restart the algorithm.

Also include in this pane are the buttons 'Branch View' and 'Table View'. These buttons allow the user to switch from the two views available. The 'Branch View' can only be accessed once the pratt parser is executed. Finally, the user can click the 'back' button to return to the selection pane and choose a new expression to be parsed.

The Branch View

This is the view which the user first encounters when after an expression is chosen. This view displays each iteration of the parsing algorithm. Shows the user when important functions are executed and information on updated values. Displayed like a flow chart or like branches as the name suggests.

The Table View

This is the other view the user encounters and can be accessed after the pratt parser is executed. This view contains the tokens that are passed at each iteration as well as the current token and operator processes at each iteration of the algorithm for the chosen expression.

4.8.3 SceneBuilder and CSS Stylesheets

When designing the GUI for the application, SceneBuilder and CSS styling was utilised. SceneBuilder was used to develop the structural aspects of the GUI and CSS styling was employed to design the various aesthetics of the interface.

4.8.4 Some Additional Comments

The design process is fundamental for making sure the main functionalities will be correctly implemented. It was difficult to determine first, which classes were required. Even after the designing of the final class diagram, the completed software implemented a new class ‘ParsingStatus’. Certain aspects of the code cannot always be decided before hand until the implementation stage begins. Nonetheless, most of the design structure created beforehand remained the same with some minor additions of extra methods and variables. Whether the changes where ideal or if the design approach was ideal will be discussed in Chapter 5.

Chapter 5

Testing

This section looks at the various way the software was tested and whether the requirements have been met. We also look at example test cases based on our requirements.

5.1 Software Testing

Testing the software is essential when determining if the software system works correctly. To know if the software has met the discussed requirements, we discuss and look at the different kinds of tests that were carried out. The main functionality that was tested was the pratt parsing algorithm. The algorithm is central to the entire software prototype therefore requires to be tested thoroughly.

5.1.1 Unit Testing

The unit testing framework used to carry out these tests was JUnit. JUnit is an open-source framework, designed for Java programming. JUnit provides a developer with important features such as annotations and assertions. Annotations lets the developer write and identify the various test methods. For example, the annotations @Before, @After and @Test. These are useful when writing automated unit tests to know the purpose of each method. Assertions are methods used to test expected results. This can also be used to test expected failures, for example, when testing an input that is required to fail to know that the system is working correctly.

For this application, unit tests for the classes ‘ParserExecute’ and ‘PrattParser’ were written. These classes contain the fundamental functionalities for the software system to work correctly. ‘ParserExecute’ is the model class where all its data is retrieved by the ‘PrattController’ class to update the view. It’s also responsible for the validation checking for the user inputs. The class ‘PrattParser’ is where the pratt parsing algorithm is contained. When the user executes the parsing algorithm on their expression, this class is taking that expression and processes the expression using pratt parsing. The class works closely with the ‘ParserExecute’ class. The output data of ‘PrattParser’ is stored in the ‘ParserExecute’. Therefore, it is essential to test that both classes work as the software’s correctness is dependent on these classes. Now we look at the two test classes and some of their test cases.

‘ParserExecuteTest’

This class contains test methods that test the output strings for invalid expressions. Essentially, testing the method ‘isValidExpression’ found in the ‘ParserExecute’ class. These tests were written following the implementation of the ‘ParserExecute’ class. Specifically, when the discovered the class was functioning as intended, these tests were written to understand what kind of scenarios that the class did not function correctly for.

Below is a table of the outcome of the test execution of some input data. When any code was adjusted in this class, the tests were rerun to make sure nothing in the code had broken.

Test Case	**Expression**	**Expected Result**	**Pass/Fail**
testErrorMessageForMissingClosedBracket()	“(10+10”	"invalid expression, missing ')', try again"	PASS
testErrorMessageForMissingOpenBracket()	“10+10)”	"invalid expression, missing '(', try again"	PASS
testErrorMessageForLetters()	“hello world”	"invalid expression, must contain only numbers and valid operators, try again"	PASS
testInvalidExpressionForAddition()	“+2”	"invalid expression, expression cannot start with	PASS

		operator, try again"	
testInvalidExpressionForSubtraction()	“-2”	"invalid expression, expression cannot start with operator, try again"	PASS
testInvalidExpressionForMultiplication()	“*2”	"invalid expression, expression cannot start with operator, try again"	PASS
testInvalidExpressionForDivision()	“/2”	"invalid expression, expression cannot start with operator, try again"	PASS

‘PrattParserTest’

This test class tests the functionalities within the class 'PrattParser'. The test cases test for the correct output for a given valid expression. Essentially, it mimics what the 'ParserExecute' class does as the class is responsible for carrying out the execution of the pratt parsing algorithm and obtaining the output. To test this correctly, some non-test methods were implemented to make test cases work. This is because to retrieve the output of an expression the pratt parsing must take in an expression as tokens. Although, the test cases could create input that is form of tokens, for conciseness and readability of the test class, its desirable to have a method that does the tokenisation.

Test Case	Expression	Expected Result	Pass/Fail
testValidSingleBrackets()	"(10+10)"	20	PASS
testValidAddition()	"10+10"	20	PASS
testValidSubtraction()	"10-2"	8	PASS
testValidMultiplication()	"2*3"	6	PASS
testValidDivision()	"20/5"	4	PASS

5.1.2 Manual Testing

For frontend components of the application, manual testing was carried out. Manual testing is sufficient to test the GUI of this application. In a scenario where this was a web application or had components linked to the web, these functionalities could be unit tested. However, even in such cases, manual testing would still be needed. Here, we look at a few manual test cases to test the GUI was functioning correctly. Additionally, it determines the quality and performance of the system. The following test cases focus on the parser pane, making sure the pratt parser data is being outputted to the screen as desired.

Test Case/Steps	Expected Result	Pass/Fail
Choose an expression from the list then click ‘Select’	Goes to the Parser Pane	PASS
In the parser pane, click ‘Start’	Displays the first iteration of the pratt parser on the text area	PASS
In the parser pane, click ‘Next’	Displays the next iteration of the pratt parser on the text area	PASS
In the parser pane, click ‘Previous’	Displays the previous iteration of the pratt parser on the text area	PASS
In the parser pane, click ‘Reset’	Clears the text area, enables the start button, and disables the table view button.	PASS
In the parser pane, click ‘Finish’	Outputs all the iteration steps of the pratt parser in the text area and disables the ‘Next’ button.	PASS
In the parser pane, click ‘Table View’	Displays the table view of the pratt parser containing the updated values of each step of iteration.	PASS

In the parser pane, click 'Start', then 'Table View' and then click 'Branch View'	Displays the branch view of the pratt parser with the previous data loaded.	PASS
In the parser pane, after pressing 'Start' and clicking 'Table View', click 'Back'	Correctly returns to and displays the selection pane.	PASS
In the parser pane, after pressing 'Start' and clicking 'Table View', click 'Back' then choose an expression from list and click 'Select'	Erases previous expression, resets the branch view, and displays new selected expression.	PASS
Input the long expression '(2*6+(6+14)/2*2-(9*1)+7-5/2-1+(10*10))*2'	Correctly displays the entirety of the expression to screen.	PASS
Input the long expression '(2*6+(6+14)/2*2-(9*1)+7-5/2-1+(10*10))*2' and click 'Start'	Instantly displays the first line of iteration of the pratt parser to the branch view.	PASS

The process of carrying out these tests assists in determining whether the system requirements have been met. Thoroughly, testing the applications behaviour and performance allows evaluating whether we have met the established requirements. In Chapter 6 we evaluate the final software.

Chapter 6

Evaluation

6.1 Software Evaluation

To determine how successful the implemented software is, it is ideal to compare the software to the established requirements written in chapter 3. In this section, we evaluate the application based on whether each requirement has been met.

6.1.1 Functional

Most functional requirements were met for the software system. However, one of the requirements required the user to have a tree representation of the pratt parsing algorithm. In the finished software, the pratt parsing algorithm is displayed similarly to a tree but not with the visual depth that was desired. For example, there are no visual depiction of branches or nodes. This was mainly not met due to time constraints, but the current display does present the user with a thorough walkthrough of the individual iterative steps of the pratt parser.

When looking at the table view implementation, the user can switch to this type of view as desired by the requirements. It displays data on the iterative steps of the pratt parsing algorithm in a different way to the branch view. It's more of addition or helper to the branch view to help with the users understanding. This is certainly a good design choice when making sure that the user can use the tools the software provides to increase their understanding of the pratt parser.

6.1.2 Non-Functional

All the non-functional requirements were met as desired. During the manual testing phase these requirements were tested, and all successfully passed. As a result, the overall system works intuitively, essential for a learning tool. Additionally, software is responsive and does not suffer from any delays or lag issues. Therefore, the software performs well when user interacts with its variety of features. Input expressions that are significantly large are processed as quickly as the shorter expressions. They also are displayed correctly and clearly for the user to read.

6.1.3 Code Structure

The implementation process requires taking into consideration the maintainability and readability of the code being written. Throughout the coding process, classes were created and implemented when functionalities required so. Removing unnecessary dependencies is also carried out when coding and refactoring. This is also helped when writing code to test the functionality of the code. The approach to implement this system was overall excellent and allowed the refactoring process to be much smoother. The code also separated the pratt parsing algorithm to a separate class allowing in the future different parsing algorithms to be implemented much more easily without needing to restructure the entire code.

6.2 Project Evaluation

In chapter 1, project objectives were written out with the purpose to determine at the end if the methodology along with the implemented software was successful. The research objectives were mostly met however, having more examples of existing software, more specifically existing software for pratt parsing, would have deemed useful when analysing how other systems visualised the parsing algorithm.

Due to time constraints, the visual depiction of the parsing algorithm was not as detailed as desired. Indeed, there was a substantial amount of visualisation, however, animation that are colourful, would have been more engaging and interesting to the user. Perhaps, having done more research on visual animation on JavaFX, the final implementation may have had more aspects of animation when displaying the pratt parsing algorithm.

Chapter 7

Professional Issues

In terms of professional issues, the software does not inherently pose any concerns regarding professional issues. However, we can discuss some ethical and legal issues which were keep in mind when carrying out relevant research on the project area. During the research phase, it was made sure that sources and articles did not violate any legal or ethical principles. All information was retrieved from online pages that were safe and referenced the author of the contents of the page. Other than these considerations, no other professional concerns or issues were brought up due to the scope and nature of the project.

Chapter 8

Conclusion & Future Work

6.3 Conclusion

The projects main idea is to visualise the pratt parsing algorithm. For that reason, the implemented software has been successful in executing this concept. Furthermore, the code design structure suggests the potential the system has. This implies that the software can be extended to visualise more parsing algorithms. Since there isn’t any established work out there for pratt parsing visual tools, it can be seen as an excellent stepping stone for producing more software of this nature to help students and teachers have access to applications that assist in the learning process.

6.4 Future Work

As for most developed software, this system has much room for improvement and extension. When looking at the visual depth of the software, an obvious addition to the system would be animations depicting the pratt parsing algorithm. This can tie closely with the tree representation, by having progression of the branches and nodes of the tree.

Looking more further away from the scope of the project, it would be interesting to investigate extending the types of parsers that can be displayed by the software. Having a versatile system can be essential for a software tool used in learning environments. Some examples of parsers that could be implemented is a simpler operator-precedence parser.

Furthermore, the existing pratt parsing algorithm can be extended to parse expressions containing indices and square roots. This would allow a much larger variety of mathematical expressions to be parsed. In summary, this project has a huge space for expansion in terms of how parsing algorithms are displayed to the user and the types of parsers that can be chosen.

There are many more ways this software can be extended and adapted for future work. Moreover, it would certainly be interesting to observe the different approaches to implementation of such a software and how the existing software would have differed.

References

[1] abarker.github.io. (n.d.). *2. Introduction to Pratt parsing and its terminology — Typped documentation*. [online] Available at: https://abarker.github.io/typped/pratt_parsing_intro.html [Accessed 29 Feb. 2024].

[2] www.tutorialspoint.com. (n.d.). *What is Operator Precedence Parsing?* [online] Available at: https://www.tutorialspoint.com/what-is-operator-precedence-parsing [Accessed 31 Mar. 2024].

[3] COMPILER DESIGN -SYNTAX ANALYSIS COMPILER DESIGN -SYNTAX ANALYSIS. (n.d.). Available at: https://www.tutorialspoint.com/compiler_design/pdf/compiler_design_syntax_analysis.pdf.

[4] Janiczek, M. (2023). *Demystifying Pratt Parsers*. [online] Martin Janiczek. Available at: https://martin.janiczek.cz/2023/07/03/demystifying-pratt-parsers.html

[5] Pratt, V.R. (1973). Top-down operator precedence. *Proceedings of the 1st annual ACM SIGACT-SIGPLAN symposium on Principles of programming languages - POPL '73*. Available at: https://doi.org/10.1145/512927.512931

[6] maeyler.github.io. (n.d.). *Visualization of Recursive Descent Parser*. [online] Available at: https://maeyler.github.io/Automata-2018/cfg/Bilal_RecursiveDescentParser.html#form [Accessed 29 Mar. 2024].

[7] Parashar, N. (2022). *What is Parsing in NLP?* [online] Medium. Available at: https://medium.com/@niitwork0921/what-is-parsing-in-nlp-69f41b3e5620 [Accessed 20 Mar. 2024].

[8] jeffreykegler.github.io. (n.d.). *Parsing: Top-down versus bottom-up*. [online] Available at: https://jeffreykegler.github.io/Ocean-of-Awareness-blog/individual/2014/11/ll.html. [Accessed 29 Mar. 2024].

[9] www.oreilly.com. (n.d.). *9. Top Down Operator Precedence - Beautiful Code [Book]*. [online] Available at: https://www.oreilly.com/library/view/beautiful-code/9780596510046/ch09.html [Accessed 25 Mar. 2024].