DSAA CA2 Final Report

School Of Computing (SOC)
Diploma in Information Technology
ST1507 Data Structures And Algorithms (AI)

Done by:
Lim Chuan Hao (1922264)
Tan Jing Wen Sherisse (1935967)
Class: DIT/FT/2B/11

Contents

1 Introduction

This is the final report for the team's Data Structures And Algorithms(DSAA) CA2 Assignment. The team members are Chuan Hao and Sherisse of class DIT/FT/2B/11. The appendix will contain any additional information, as well as source code and references.

2 About the Application

The application the team made has a Command Line Interface(CLI) that allows the user to input mathematical expressions for evaluation as well as sort mathematical expression(s) in files. This is done by using the team's crafted lexical grammar which follows a regular language to tokenize the given mathematical expression. The application's parser is a recursive descent parser that has a Lookahead Left-to-Right(LALR) implementation. It uses a context-free grammar — a modified Backus–Naur Form(BNF) notation that is different from the lexer — to parse the series of tokens from the lexer into an abstract-syntax tree (AST). Lastly, the interpreter uses a visitor pattern, with a Python Object Oriented Programming (OOP) approach to evaluate the AST for the value of a given expression.

The overall application can be divided into 2 sections: the `basic` application, which fulfills all the basic requirements as specified in the brief, and the `advance` application, which is an extension of the `basic` application with additional advanced features.

The application contains two main features: `expression_evaluator` and `expression_sorter`. The `expression_sorter` is common across both the `basic` and `advance` applications, whereas the `expression_evaluator` is implemented differently in both applications.

2.1 Features Implemented

The application basic features are:

- Supports fully parenthesized expressions
- Supports integer and float operands
- Support positive and negative operands
- Application supports OOP
- Made our own Python classes
- Placed classes in separate files
- Supports input validation

The application advanced features are:

- Custom error messages
- Support more mathematical functions and expressions
- Non fully parenthesized expressions
- Curses CLI
- Unary operators
- Choose between sorting in ascending or descending order
- Alternative ways of printing the parse tree (Post and In order)

3 Application Guide

This section focuses on how to use the team's application.

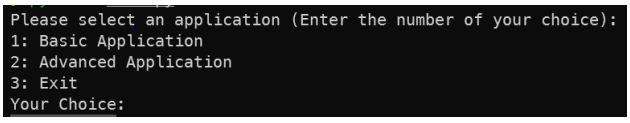

Figure 1: A screenshot of how the main application screen when launched

The user can launch the main application by doing `python main.py` in the root directory of the project. The user can then make a selection by entering the number of his choice and pressing enter. An example of how the screen looks like can be seen in figure 1 above.

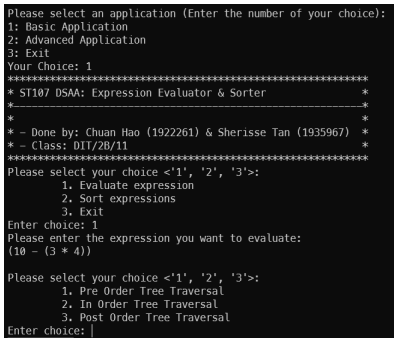

Figure 2: A screenshot of the basic application when user is prompted for an input

Figure 3: A screenshot of the advance application when user is prompted for an expression

3.1 Basic Application

The user is expected to and will be prompted to provide a numerical input to choose between what they would like to do, and to provide expressions as well as files. The display is printed out on the terminal using `print()` statements that will also provide the user with instructions as to what to input.

3.2 Advance Application

For the advanced application, the user is expected to use the arrow keys (such as *UP*) to navigate the menu, and the *ENTER* key to make a selection. The user can also use the *ESCAPE* key to navigate back to a previous section. Lastly, the user is expected to follow the instructions on screen when prompted. The user can refer to the figure 3 above for a reference.

4 Application Overview

This sections focuses on what the team has done in terms of the data structures used in the application and its architecture.

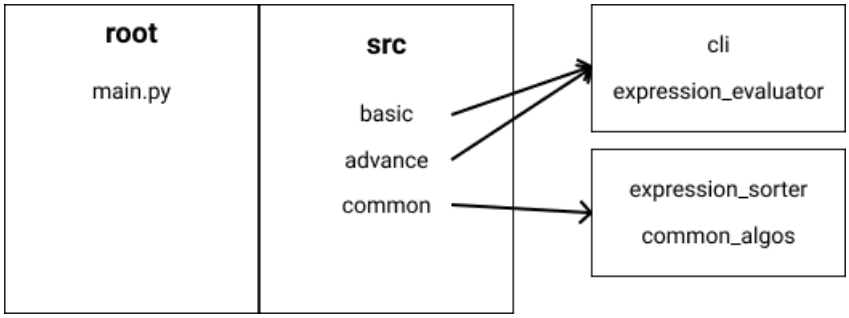

Figure 4: The diagram shows how the modules are related to each other, with the bolded text generally showing the directory of the modules. The arrows represent which modules are contained in another.

Based on the diagram above, figure 4, we can see that the application has a root `main.py` python file that starts the entire application. The `basic, advance and common` modules have their own directory respectively and are called when needed.

4.1 Data Structures Overview

The team has split the application modules into 3 distinct groups as seen above: `basic, advance and common` modules. The `basic and advance` modules mainly handle the CLI and logic behind the respective applications, while the `common` module handles common logic between the 2 applications, like reading and writing to files and the sorting logic behind expressions. In addition, python files and `Strings` were used as configs and constants for the application itself.

As for an overview of the classes used in these modules, refer to the figure 5 below

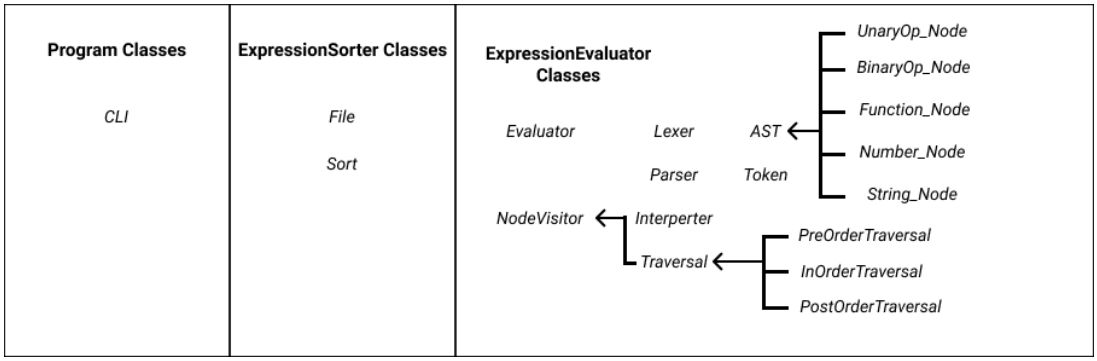

Figure 5: The diagram shows an overview of which classes belongs to which module. The arrows represent inheritance among the classes, with the child pointing back to the parent

4.1.1 Program Classes used

For the program classes, the team is referring to the `CLI` classes implemented for the `basic and advance` applications as modules. The team plans to use 2 different `CLI` classes for each application with the `CLI` classes controlling the user's input and the flow of the whole application.

4.1.2 Expression Sorter Classes

The classes and descriptions are given below:

- `File` ⇒ Handles any interactions like Input/Output(I/O) with files
- `Sort` ⇒ Handles the sorting of the evaluated expression by the given standards

4.1.3 Expression Evaluator Classes

The classes and descriptions are given below:

- `Token` ⇒ A class used to store the information of a token. (Token Types are predetermined by the team)
- `NodeVisitor` ⇒ An abstract class inherited by the `Interpreter` and `Traversal` class. Implements the base visitor pattern

- `AST` ⇒ An abstract base class for the nodes of the AST
- `UnaryOp_Node` ⇒ The node representation of a unary operation (example, making a number negative *-3*)
- `BinaryOp_Node` ⇒ The node representation of a binary (*+-*/*) operation
- `Function_Node` ⇒ The node representation of a Function (example, *PI()*, *cos(20)*)
- `String_Node` ⇒ The node representation of a String (*'ab'*, *''*, *'110101101'*, *'AF723B.7124B3FA'* etc.)
- `Number_Node` ⇒ The node representation of a Number (*3*, *2.1*, *100* etc.)
- `Lexer` ⇒ Handles tokenizing a given expression into predetermined tokens
- `Parser` ⇒ Handles parsing the tokens from the `Lexer` into an AST with nodes
- `Interpreter` ⇒ Handles going through the AST, to interpret and calculate the final evaluation of the in initial given expression
- `Traversal` ⇒ Acts as a abstract class for the other traversal classes to inherit from
- `PreOrderTraversal` ⇒ Traverses a given AST to get the pre-order traversal graph
- `InOrderTraversal` ⇒ Traverses a given AST to get the in-order traversal graph
- `PostOrderTraversal` ⇒ Traverses a given AST to get the post-order traversal graph
- `Evaluator` ⇒ Acts as the main class, containing the `Lexer, Parser` and `Interpreter`

Below is a description of how the classes are inherited:

- `NodeVisitor` ⇐ Inherited by `Interpreter and Traversal`. The base class implements the basic visitor pattern.
- `Traversal` ⇐ Inherited by `PreOrderTraversal, InOrderTraversal and PostOrderTraversal`. Traversal has an adjusted visitor pattern implementation allowing for traversal
- `AST` ⇐ Inherited by `UnaryOp_Node, BinaryOp_Node, Function_Node Number_Node, String_Node`. This allows all the nodes to share the same parent class and properties of `AST`.

4.1.4 Constants and Configs

For the application, the team will use python `Strings` in CAPITAL LETTERS to denote when the variable is used as a constant or config. The `token_type.py` file specifically holds all of the predetermined token types for the application.

4.2 Application Architecture

4.2.1 Common

Expression Sorter

The `expression_sorter` is a common module that is utilised by both the Basic and the Advance Application. It consists of two classes: `File` and `Sort`.

`File` contains 2 `staticmethods` that reads and writes from and to files respectively.

`Sort` deals with the sorting of expressions in a list by using the `.sort()` method that will handle all other necessary logic. `Sort.sort()` returns a nested list of expressions that are already sorted by value, and length if there are multiple expressions with the same value, using the `mergeSort` algorithm, an ordered sort algorithm with a time complexity of `O(n log n)`.

Common Algorithms

The `common_algos` module mainly contains simple algorithms that could be used for both the `basic and advance` applications. As of now, the module contains 2 simple algorithms, to convert a binary and hexadecimal string back to a decimal number.

4.2.2 Basic Application

The display on the terminal is controlled by the `CLI` class using a variety of `staticmethods` for each identified section of the application. It also consolidates the 2 main features of the basic application: `expression_evaluator` and `expression_sorter`.

The tree traversal module is called within `cli.py`. The main compiler comprises of 3 main components: lexer, parser and interpreter and is run from within the `Evaluator.evalute() staticmethod` that is also called in `cli.py`.

4.2.3 Advance Application

Similar to the `basic` application, the `CLI` class controls the application logic and the user's controls. The `CLI` does this by using the inbuilt `curses` library, allowing much finer control over the entire terminal application screen. The `advance` application also uses an advanced `expression_evaluator` module with more features. These include features like a more robust lexer and parser, more implemented mathematical functions and strings. The logic for evaluator is quite similar to the `basic` application's `expression_evaluator` modules and as such it will be discussed more in the section below.

5 How the expression evaluator works

This section will cover the more technical aspects of how the `expression_evaluator` module works to evaluate a given expression. In the module, `Evaluator` is a wrapper class that contains the `Lexer, Parser` and `Interpreter`. The general order is that the expression is passed through the `Lexer` to generate tokens that is passed to the `Parser` to generate an AST that is interpreted and evaluated by the `Interpreter`.

5.1 Lexer

For the `Lexer`, the team crafted their own lexical grammar, which can be found in appendix A, in order to tokenize the given expression into a series of tokens. This has the advantage of fully expressing our tokenizing logic in a simple way, allowing for easier collaboration,

implementation and error checking. This also ensured edge cases such as differentiating between the tokens `Token(MUL, '*')` and `Token (POWER, '**')` were accounted for. Thus the `Lexer` was able to robustly and consistently tokenize the given expression into a series of tokens.

5.2 Parser

As for the parser, the team used a context-free grammar, in conjunction with the recursive descent parser that has a Lookahead Left-to-Right(LALR)implementation. This allowed for the sequence of tokens from the `Lexer` to be parsed into as AST easily, while at the same time checking for logical errors, such as non-fully parenthesised expressions, or invalid expressions. The built AST is then passed on to the `Interpreter`

5.3 Interpreter

Lastly, the interpreter uses a visitor pattern, where the methods acting on the AST nodes are implemented seperately from the data structure allowing for the team to evaluate and traverse the AST depending on the implementation. This also allows the team to check for runtime errors during interpretation of the AST as well. Thus, after the AST has been interpretered and evaluated, the value of the expression would have been calculated.

6 Challenges faced by the team

The main challenges that the team faced after the time of the iterim report were in error handling and utilising the curses library.

Error Handling

The team encountered some difficulties in deciding how to catch errors and display them to the user. This is because there are a variety of ways that an error could have occurred, especially within the compiler, and the team did not want to return a generic error message for every error encountered. As such the team decided to break down the logic of the application into modules, and implementing the error by modules, making the errors more specific and testable.

CLI with the curses library

The team also faced another challenge when dealing with the curses library, as it is a much lower level library, requiring math for manipulating the terminal screen. As such it took much longer than expected to get the application working, but the team managed to get the CLI working in the end.

Implementing an interpreter for evaluating expression

The team faced a similar challenge as before, implementing the interpreter, as the team had to integrate the interpreter with the rest of the application this time, while also implementing it with an OOP. As such, the team had to spend more time, structuring and planning on how to implement the interpreter. Fortunately, the planning and module based approach to the application made it much easier to port the interpreter code in the end and the team was able to successful port both the basic and advance interpreter.

7 Roles and contribution

Chuan Hao

Chuan Hao is the team lead and in charge of the advance application which is an extension of the basic application that includes additional features such as, support for non-fully paranthesised expressions and mathematical functions like trigonometric functions, exponential and logarithmic functions, etc. The advance application also uses the curses library in order to build the CLI. Chuan Hao worked on the code in the `./src/advance` folder and `./main.py`.

Sherisse

Sherisse is in charge of the basic application which is the application that fulfills all the basic requirements as specified in the brief as well as the common components, the File I/O and the Sorter, for the `expression_sorter` feature that is utilised in both the basic and advance application. As such, Sherisse mainly wrote the code in the `./src/basic` and `./src/common` folders.

Combined

Together, the team came up with the application architecture and file structure that they would be using for the assignment. Additionally, the team also worked on the interim and final reports together.

8 Key Takeaways

Through this assignment, the team learned about the concept of a compiler, as well as AST, and the theory behind it, as well as how to actually implement them in Python. Additionally, the team was also able to learn more about the 4 main aspects of OOP: Abstraction, Encapsulation, Inheritance and Polymorphism.

Lastly, the team also learned more about how to work together on a single application and utilising GIT version control in order to facilitate this collaboration.

Appendices

A Expression Evaluator

This appendix section will contain the additional information pertaining to the expression evaluator.

A.1 Basic Evaluator

A.1.1 Token Types

```
--- Utilities ---
INIT: 'INIT'
EOF: 'EOF'
WHITESPACE: 'WHITESPACE'
RPARAN: 'RPARAN'
LPARAN: 'LPARAN'

    --- Numbers ---
NUMBER: 'NUMBER'
DOT: 'DOT'

    --- Operators ---
OPERATOR: 'OPERATOR'
PLUS: 'PLUS'
MINUS: 'MINUS'
MUL: 'MUL'
DIV: 'DIV'
POWER: 'POWER'
```

A.1.2 Lexical Grammar

```
expr: LPARAN term ( ( "+" | "-" | "*" | "/" | "**" ) term )* RPARAN
term: expr | factor
factor: MINUS factor | NUMBER
```

A.2 Advance Evaluator

A.2.1 Token Types

```
--- Special Tokens ---
PLUS: '+'
MINUS: '-'
MODULUS: '%'
MUL: '*'
DIV: '/'
INT_DIV: '//'
POWER: '**'
LPARAM: '('
RPARAM: ')'
```

```
COMMA: ','

EOF: 'EOF'

--- Normal Tokens ---
NUMBER, IDENTIFIER, STRING
```

A.2.2 Reserved Keywords

```
--- Reserved Keywords ---
Arity 0: E, PI
Arity 1: sin, cos, tan, floor, ceil, ln, lg, factorial, sqrt, bin, hex
Arity 2: log, pow, mod, perm, comb
```

A.2.3 Lexical Grammar

```
--- Lexical Grammar ---
NUMBER: DIGIT* ('.' DIGIT+)?
IDENTIFIER: ALPHA+
STRING: '\'' <any char except '\''>* '\''
DIGIT: '0' ... '9'
ALPHA: 'a' ... 'z' | 'A' ... 'Z'
```

A.2.4 Syntax Grammar

```
--- Syntax Grammar ---
expression: term (('+' | '-' | '%') term)*
term: factor (('*' | '/' | '//') factor)*
factor: unary (('**') unary)*
unary: ('+' | '-') unary
 | call
call: IDENTIFIER ('(' arguments? ')')
 | primary
arguments: expression (',' expression)*
primary: NUMBER
 | STRING
 | '(' expression ')'
```

A.3 All supported features

```
The full list of supported advanced mathematical features are:
**Single Number expression**

**Math Functions**
- Arity 0
    - E
    - PI
- Arity 1
```

```
    - sin
    - cos
    - tan
    - floor
    - ceil
    - ln
    - lg
    - factorial
    - sqrt
    - bin
    - hex
- Arity 2
    - log
    - pow
    - remainder
    - perm
    - comb

**Unary Operators**

- Postive +
- Negative -

**Binary Operators**

Basic
- +
- -
- *
- /
- %
- //
    - Integer Division
- **
    - Exponential
```

B Project/Application File Structure

```
.
|-- main.py
`-- src
    |-- advance
    |   |-- cli.py
    |   |-- expression_evaluator
    |   |   |-- compiler
    |   |   |   |-- __init__.py
    |   |   |   |-- interpreter.py
    |   |   |   |-- lexer.py
    |   |   |   |-- node_traversal
    |   |   |   |   |-- __init__.py
    |   |   |   |   |-- post_order.py
    |   |   |   |   |-- pre_order.py
    |   |   |   |   `-- traversal.py
    |   |   |   `-- parser.py
    |   |   |-- evaluator.py
    |   |   |-- __init__.py
    |   |   |-- nodes
    |   |   |   |-- ast.py
    |   |   |   |-- binary_op_node.py
    |   |   |   |-- function_node.py
    |   |   |   |-- __init__.py
    |   |   |   |-- node_visitor.py
    |   |   |   |-- number_node.py
    |   |   |   |-- string_node.py
    |   |   |   `-- unary_op_node.py
    |   |   |-- README.md
    |   |   `-- tokens
    |   |       |-- __init__.py
    |   |       |-- token.py
    |   |       `-- token_type.py
    |   `-- __init__.py
    |-- basic
    |   |-- cli.py
    |   |-- expression_evaluator
    |   |   |-- compiler
    |   |   |   |-- __init__.py
    |   |   |   |-- interpreter.py
    |   |   |   |-- lexer.py
    |   |   |   |-- node_traversal
    |   |   |   |   |-- __init__.py
    |   |   |   |   |-- in_order.py
    |   |   |   |   |-- post_order.py
    |   |   |   |   `-- pre_order.py
    |   |   |   `-- parser.py
    |   |   |-- evaluator.py
    |   |   |-- __init__.py
    |   |   |-- nodes
    |   |   |   |-- ast.py
    |   |   |   |-- binary_op_node.py
    |   |   |   |-- __init__.py
    |   |   |   |-- node_visitor.py
    |   |   |   `-- number_node.py
    |   |   `-- tokens
    |   |       |-- __init__.py
```

```
    |   |       |-- token.py
    |   |       `-- token_type.py
    |   `-- __init__.py
    |-- common
    |   |-- common_algos
    |   |   |-- common_algos.py
    |   |   `-- __init__.py
    |   |-- expression_sorter
    |   |   |-- file.py
    |   |   |-- __init__.py
    |   |   `-- sort.py
    |   `-- README.md
    `-- __init__.py
```

Listing 1: Folder structure for the whole project/application

```
.
|-- main.py
`-- src
    |-- advance
    |   |-- cli.py
    |   |   `-- The advance application CLI class
    |   `-- expression_evaluator
    |       `-- The advance expression evaluator (Details above in \ref{appendix:
    expression_evaluator})
    |-- basic
    |   |-- cli.py
    |   |   `-- The basic application CLI class
    |   `-- expression_evaluator
    |       `-- The basic expression evaluator (Details above in \ref{appendix:
    expression_evaluator})
    |-- common
        |-- common_algos
        |   `-- The common algorithm modules code
        `-- expression_sorter
            `-- Contains the File and Sort class modules
```

Listing 2: Description of the folder structure for the project/application

C Source Code

C.1 ./

C.1.1 ./main.py

```
from src.basic.cli import CLI as BasicCLI
from src.advance import CLI as AdvanceCLI

if __name__ == '__main__':
    application_choice = None
    while True:
        print("Please select an application (Enter the number of your choice):")
        print("1: Basic Application")
        print("2: Advanced Application")
        print('3: Exit')
        application_choice = input('Your Choice: ')
        if application_choice in set(['1', '2', '3']):
            cli = None
            if application_choice == '1':
                cli = BasicCLI()
                cli.run()
            elif application_choice == '2':
                cli = AdvanceCLI()
            elif application_choice == '3':
                print('Thanks for using our application, bye :D')
                break
```

Listing 3: Source Code for: ./main.py

C.2 ./src/advance

C.2.1 ./src/advance/__init__.py

```
from .cli import CLI
```

Listing 4: Source Code for: ./src/advance/__init__.py

C.2.2 ./src/advance/cli.py

```
'''Advance Application CLI file

The Advance Applicaton CLI class, meant to be imported by the main program to run
    the advance application
'''
import curses
import time
import curses.panel as panel

# Importing evaluator
from .expression_evaluator import Evaluator

# Importing file and sort
from ..common.expression_sorter import File, Sort

class CLI(object):
    def __init__(self):
```

```
        # Curses objs
        self.__stdscr = None
        self.__height, self.__width = None, None
        self.__history_length = 300

        self.__header_window = None
        self.__header_height, self.__header_width = None, None
        self.__header_y, self.__header_x = None, None

        # Application window
        self.__application_window = None
        self.__application_height, self.__application_width = None, None

        # Selection
        self.__selection_panel = None
        self.__selection_window = None
        self.__selection_y, self.__selection_x = None, None

        # Expression
        self.__application_terminal_panel = None
        self.__application_terminal_window = None
        self.__application_terminal_y, self.__application_terminal_x = None, None
        # Expression visual pad
        self.__application_history_pad = None
        self.__application_history_pad_pos = None

        # Exit panel
        self.__exit_panel = None
        self.__exit_window = None
        self.__exit_y, self.__exit_x = None, None

        # Configs
        self.__application_title = ['ST107 DSAA: Expression Evaluator & Sorter', '
    Advance Application']
        self.__application_instructions = [
            "For the selection menu: Use 'UP' and 'DOWN' arrow keys to navigate the
    menu. Press 'ENTER' to select and option",
            "For the application history: Use the 'UP' and 'DOWN' arrow keys to move
     through the history. Press 'ESC' to leave the history"
        ]
        self.__creator_names = ['Chuan Hao(1922264)', 'Sherisse(1935967)']
        self.__creator_class = 'DIT/FT/2B/11'
        self.__selection_options = [
            {
                'str': 'Evaluate an Expression',
                'method_name': '__update_expression_evaluator'
            },
            {
                'str': 'Sort the Expression in a file',
                'method_name': '__update_file_sorter'
            },
            {
                'str': "Look at the application's history",
                'method_name': '__update_application_history_pad'
            },
            {
                'str': 'Exit',
                'method_name': '__update_exit'
```

```
            }
        ]

        self.__current_application = None
        self.__current_application_attributes = None

        curses.wrapper(self.__main)

    def __error(self):
        raise Exception('Unexpected CLI error has occurred')

    def __set_up(self):
        '''
        Main helper function to set up everything
        Set up only includes things that will run only once
        '''
        self.__set_up_config()
        self.__set_up_windows()
        self.__set_up_windows_configs()
        self.__set_up_panels()

    def __set_up_config(self):
        '''
        Set up curses configs
        '''
        # curses.curs_set(1)
        # Setting up color pairs
        curses.init_pair(1, curses.COLOR_BLACK, curses.COLOR_WHITE) # Selection highlight
        curses.init_pair(2, curses.COLOR_BLUE, curses.COLOR_BLACK) # Expression Evaluator
        curses.init_pair(3, curses.COLOR_GREEN, curses.COLOR_BLACK) # Expression File Sorter
        curses.init_pair(4, curses.COLOR_CYAN, curses.COLOR_BLACK) # Application History
        curses.init_pair(5, curses.COLOR_RED, curses.COLOR_BLACK) # Exit application

    def __set_up_windows(self):
        '''
        Set up other windows
        '''
        self.__height, self.__width = self.__stdscr.getmaxyx()

        # Header window
        self.__header_height, self.__header_width = int(self.__height * 0.3), self.__width
        self.__header_window = self.__stdscr.derwin(self.__header_height, self.__header_width, 0, 0)
        self.__header_y, self.__header_x = (1, 1)
        self.__header_window.move(self.__header_y, self.__header_x)

        # Application window
        self.__application_height, self.__application_width = self.__height - self.__header_height, self.__width
        self.__application_window = self.__stdscr.derwin(self.__application_height, self.__application_width, self.__header_height, 0)
        self.__application_window.box()

```

```
        # Selection window
        self.__selection_window = self.__application_window.derwin(self.__application_height - 2, self.__application_width - 2, 1, 1)
        self.__selection_y, self.__selection_x = (0, 0)
        self.__selection_window.move(self.__selection_y, self.__selection_x)

        # Expression window
        self.__application_terminal_window = self.__application_window.derwin(self.__application_height - 2, self.__application_width - 2, 1, 1)
        self.__application_terminal_y, self.__application_terminal_x = (0, 0)
        self.__application_terminal_window.move(self.__application_terminal_y, self.__application_terminal_x)

        # Expression visual pad
        self.__application_history_pad = curses.newpad(self.__history_length, self.__application_width - 2)
        self.__application_history_pad_pos = self.__history_length - 1

        # Exit window
        self.__exit_window = self.__application_window.derwin(self.__application_height - 2, self.__application_width - 2, 1, 1)
        self.__exit_y, self.__exit_x = (0, 0)
        self.__exit_window.move(self.__selection_y, self.__selection_x)

    def __set_up_windows_configs(self):
        # Standard Keypads
        self.__stdscr.keypad(1)
        self.__header_window.keypad(1)
        self.__application_window.keypad(1)
        self.__selection_window.keypad(1)
        self.__application_terminal_window.keypad(1)
        self.__application_history_pad.keypad(1)

        # Expression
        self.__application_terminal_window.scrollok(True)
        self.__application_history_pad.scrollok(True)

    def __set_up_panels(self):
        self.__selection_panel = panel.new_panel(self.__selection_window)
        self.__application_terminal_panel = panel.new_panel(self.__application_terminal_window)
        self.__exit_panel = panel.new_panel(self.__exit_window)

    def __refresh(self):
        self.__stdscr.noutrefresh()
        self.__header_window.noutrefresh()
        self.__application_window.noutrefresh()
        self.__selection_window.noutrefresh()
        self.__application_terminal_window.noutrefresh()
        self.__exit_window.noutrefresh()
        curses.doupdate()

    def __update_application_panel(self, top_panel=None):
        '''
        Private helper function to erase the application, set the given panel to the top and update the panels
        '''
        self.__application_window.erase()
```

```
        self.__header_window.erase()
        self.__load_header()
        self.__load_application()
        if top_panel is not None:
            top_panel.top()
            panel.update_panels()

    def __load_header(self):
        '''
        Private helper function for loading the header
        '''
        # Setup
        self.__header_y, self.__header_x = (1, 1)
        self.__header_window.box()
        # Calculate width for later on
        width = self.__header_width - 2
        width = width//2

        # Adding title
        for title in self.__application_title:
            x = width - len(title)//2
            self.__header_window.addstr(self.__header_y, x, title)
            self.__header_y += 1

        # Adding break
        self.__header_window.addstr(self.__header_y, self.__header_x, '-'*(self.
    __header_width - 2))
        self.__header_y += 1

        # Adding names
        creator_names_str = f"- Done by: {' & '.join(self.__creator_names)}"
        x = width - len(creator_names_str)//2
        self.__header_window.addstr(self.__header_y, x, creator_names_str)
        self.__header_y += 1

        # Adding class
        creator_class_str = f"- Class: {self.__creator_class}"
        x = width - len(creator_class_str)//2
        self.__header_window.addstr(self.__header_y, x, creator_class_str)
        self.__header_y += 1

        # Newline
        self.__header_y += 1

        # Adding instructions
        for instruction in self.__application_instructions:
            x = width - len(instruction)//2
            self.__header_window.addstr(self.__header_y, x, instruction)
            self.__header_y += 1

        # Adding current application
        self.__header_y += 1
        x = width - len(self.__current_application)//2
        if self.__current_application_attributes is None:
            self.__header_window.addstr(self.__header_y, x, self.
    __current_application)
        else:
            self.__header_window.addstr(self.__header_y, x, self.
```

```
    __current_application, self.__current_application_attributes)

    def __load_application(self):
        '''
        Helper private function to load the default state of the application
        '''
        self.__application_window.box()

    def __load_selection(self):
        '''
        Helper private function to load the default state of the selection
        '''
        # Curses config
        curses.cbreak()
        curses.noecho()
        curses.curs_set(0)

        # Set application config
        self.__current_application = 'Selection Menu'
        self.__current_application_attributes = curses.color_pair(1)

        # Update panel
        self.__update_application_panel(self.__selection_panel)

    def __update_selection(self):
        '''
        Update selection
        '''
        # Set up selection
        self.__load_selection()
        self.__refresh()
        current_index = 0
        while True:
            width = self.__application_width // 2
            height = self.__application_height // 2
            y = height - len(self.__selection_options)//2
            for index, option in enumerate(self.__selection_options):
                # Get string and find x, y
                option_str = f"{index + 1}. {option['str']}"
                x = width - len(option_str)//2
                if current_index == index:
                    self.__selection_window.addstr(y, x, option_str, curses.
A_STANDOUT)
                else:
                    self.__selection_window.addstr(y, x, option_str)
                # Update y
                y += 1
            self.__refresh()
            user_key = self.__selection_window.getch()
            if user_key in set([curses.KEY_UP, ord('w'), ord('k')]):
                # For key up keypress
                if current_index == 0:
                    # If we are looping back
                    current_index = len(self.__selection_options) - 1
                else:
                    # If normal key press
                    current_index -= 1
            elif user_key in set([curses.KEY_DOWN, ord('s'), ord('j')]):
```

```
                if current_index == len(self.__selection_options) - 1:
                    # If we are looping back
                    current_index = 0
                else:
                    # If normal key press
                    current_index += 1
            elif user_key in set([curses.KEY_ENTER, 10, 13]):
                # When enter is pressed
                option = self.__selection_options[current_index]
                method_name = f"_{self.__class__.__name__}{option['method_name']}"
                method = getattr(self, method_name, self.__error)

                # Call the method selected
                return_code = method()

                if return_code == 1:
                    # Application should exit, exit code 1
                    return

                # Re-load the selection
                self.__load_selection()
                self.__refresh()

    def __load_exit(self):
        '''
        Helper private function to load the default state of the exit
        '''
        curses.noecho()
        curses.curs_set(0)

        self.__current_application = 'Exiting Application'
        self.__current_application_attributes = curses.color_pair(5)

        self.__update_application_panel(self.__exit_panel)

    def __update_exit(self):
        # Set up exit
        self.__load_exit()
        self.__refresh()

        # Final exit draw
        width = self.__application_width // 2
        height = self.__application_height // 2
        y = height - 2
        s = f"Thank you for using the advance application"
        x = width - len(s) // 2
        self.__exit_window.addstr(y, x, s)
        y += 1
        self.__refresh()

        # Countdown to leave
        for count in range(3, -1, -1):
            # Get string for countdown
            s = f"The application will close in {count} seconds"
            # Cal
            x = width - len(s) // 2
            # Show string
            self.__exit_window.addstr(y, x, s)
```

```
            self.__refresh()
            time.sleep(1)

        y += 1

        # Bye message
        s = f"Bye. :D"
        x = width - len(s) // 2
        self.__exit_window.addstr(y, x, s)
        self.__refresh()

        time.sleep(1)

        return 1 # Returns exit code for the program to end

    def __load_application_terminal_window(self, current_application,
    current_application_attribute):
        curses.noecho()
        curses.curs_set(0)
        curses.cbreak()

        self.__current_application = current_application
        self.__current_application_attributes = current_application_attribute
        self.__application_terminal_y = self.__application_terminal_window.getmaxyx
    ()[0]
        self.__application_terminal_y -= 1
        self.__application_terminal_window.move(self.__application_terminal_y, self.
    __application_terminal_x)

        self.__update_application_panel(self.__application_terminal_panel)

    def __update_expression_evaluator(self):
        self.__load_application_terminal_window('Expression Evaluator', curses.
    color_pair(2))
        self.__refresh()

        expression_evaluator_prompt_str = "Press 'i' to start writing your
    expression, Press 'v' to look at the history of the application, Press 'ESC' to
    go back to the selection menu"
        self.__write_expression_visual_pad(expression_evaluator_prompt_str)
        self.__application_terminal_window.addstr(self.__application_terminal_y,
    self.__application_terminal_x, expression_evaluator_prompt_str)
        self.__application_terminal_window.scroll()

        while True:
            user_key = self.__application_terminal_window.getch()
            if user_key in set([27]):
                # Esc key, Return to selection
                return
            elif user_key in set([ord('v')]):
                # v key, visual mode, Switch to looking at the history
                self.__update_application_history_pad()
                # Load expression again
                self.__load_application_terminal_window('Expression Evaluator',
    curses.color_pair(2))
                self.__refresh()
            elif user_key in set([ord('i')]):
                # Insert mode, user writes their expression
```

```
393                 expression_prompt = "Your Expression: "
394                 self.__application_terminal_window.addstr(self.
    __application_terminal_y, self.__application_terminal_x, expression_prompt)
395                 self.__refresh()
396 
397                 # Setting for user input to show up
398                 curses.echo()
399                 curses.curs_set(1)
400                 try:
401                     # Get the expression typed in
402                     expression_raw_input = self.__application_terminal_window.getstr
    () # Read as bytes
403                     expression_input = str(expression_raw_input, "utf-8")
404                     self.__application_terminal_window.scroll()
405 
406                     # Write the expression line to history
407                     self.__write_expression_visual_pad(f"{expression_prompt}{
    expression_input}")
408 
409                     # Evaluator logic
410                     evaluator = Evaluator()
411                     evaluation = evaluator.evaluate(expression_input)
412 
413                     # Get order of traversal
414                     order_chosen = None
415                     while order_chosen not in set(['1', '2']):
416                         self.__application_terminal_window.addstr(self.
    __application_terminal_y, self.__application_terminal_x, 'Please select an order
     of traversal (Enter the number):')
417                         self.__write_expression_visual_pad('Please select an order
    of traversal (Enter the number):')
418                         self.__application_terminal_window.scroll()
419 
420                         self.__application_terminal_window.addstr(self.
    __application_terminal_y, self.__application_terminal_x, '1. Pre-Order Traversal
    ')
421                         self.__write_expression_visual_pad('1. Pre-Order Traversal')
422                         self.__application_terminal_window.scroll()
423 
424                         self.__application_terminal_window.addstr(self.
    __application_terminal_y, self.__application_terminal_x, '2. Post-Order
    Traversal')
425                         self.__write_expression_visual_pad('2. Post-Order Traversal'
    )
426                         self.__application_terminal_window.scroll()
427 
428                         self.__refresh()
429 
430                         order_prompt = "Your selection: "
431                         self.__application_terminal_window.addstr(self.
    __application_terminal_y, self.__application_terminal_x, order_prompt)
432                         order_raw_input = self.__application_terminal_window.getstr
    () # Read as bytes
433                         order_input = str(order_raw_input, "utf-8")
434                         order_chosen = order_input
435 
436                         self.__write_expression_visual_pad(f"{order_prompt}{
    order_input}")
```

```
437
438                     # Get traversal based on selection
439                     traversal = None
440                     if order_chosen == '1':
441                         traversal = evaluator.get_traversal('pre_order')
442                     elif order_chosen == '2':
443                         traversal = evaluator.get_traversal('post_order')
444
445                     # Show evaluatiopn and traversal tree
446                     evaluation_str = f"Evaluation: {expression_input} = {evaluation}
    "
447                     self.__application_terminal_window.addstr(self.
    __application_terminal_y, self.__application_terminal_x, evaluation_str)
448                     self.__write_expression_visual_pad(evaluation_str)
449                     self.__application_terminal_window.scroll()
450
451                     self.__application_terminal_window.addstr(self.
    __application_terminal_y, self.__application_terminal_x, 'Traversal: ')
452                     self.__write_expression_visual_pad('Traversal: ')
453                     self.__application_terminal_window.scroll()
454                     for traverse in traversal:
455                         self.__application_terminal_window.addstr(self.
    __application_terminal_y, self.__application_terminal_x, traverse)
456                         self.__write_expression_visual_pad(traverse)
457                         self.__application_terminal_window.scroll()
458
459                     self.__refresh()
460
461                 except Exception as e:
462                     error_msg = f"Error occured: {str(e)}"
463                     self.__application_terminal_window.addstr(self.
    __application_terminal_y, self.__application_terminal_x, error_msg, curses.
    color_pair(5))
464                     self.__write_expression_visual_pad(error_msg, curses.color_pair
    (5))
465                     self.__application_terminal_window.scroll()
466
467                 # Process the expression and do something
468                 curses.noecho()
469                 curses.curs_set(0)
470             else:
471                 continue
472
473             self.__write_expression_visual_pad(expression_evaluator_prompt_str)
474             self.__application_terminal_window.addstr(self.__application_terminal_y,
     self.__application_terminal_x, expression_evaluator_prompt_str)
475             self.__application_terminal_window.scroll()
476
477             self.__refresh()
478
479     def __update_file_sorter(self):
480         self.__load_application_terminal_window('File Sorter', curses.color_pair(3))
481         self.__refresh()
482
483         expression_evaluator_prompt_str = "Press 'i' to start writing the file
    locations, Press 'v' to look at the history of the application, Press 'ESC' to
    go back to the selection menu"
484         self.__write_expression_visual_pad(expression_evaluator_prompt_str)
```

```
485         self.__application_terminal_window.addstr(self.__application_terminal_y,
    self.__application_terminal_x, expression_evaluator_prompt_str)
486         self.__application_terminal_window.scroll()
487 
488         while True:
489             user_key = self.__application_terminal_window.getch()
490             if user_key in set([27]):
491                 # Esc key, Return to selection
492                 return
493             elif user_key in set([ord('v')]):
494                 # v key, visual mode, Switch to looking at the history
495                 self.__update_application_history_pad()
496                 # Load expression again
497                 self.__load_application_terminal_window('File Sorter', curses.
    color_pair(2))
498                 self.__refresh()
499             elif user_key in set([ord('i')]):
500                 # Insert mode, user writes their expression
501                 # Setting for user input to show up
502                 curses.echo()
503                 curses.curs_set(1)
504                 # Get the input folder location
505                 # Prompt
506                 input_file_location_prompt = "Input file location: "
507                 self.__application_terminal_window.addstr(self.
    __application_terminal_y, self.__application_terminal_x,
    input_file_location_prompt)
508                 # User input
509                 input_file_location_raw_input = self.__application_terminal_window.
    getstr() # Read as bytes
510                 input_file_location_input = str(input_file_location_raw_input, "utf
    -8")
511                 # Write the expression line to history
512                 self.__write_expression_visual_pad(f"{input_file_location_prompt}{
    input_file_location_input}")
513 
514                 # Get the output folder location
515                 # Prompt
516                 output_file_location_prompt = "Output file location: "
517                 self.__application_terminal_window.addstr(self.
    __application_terminal_y, self.__application_terminal_x,
    output_file_location_prompt)
518                 # User input
519                 output_file_location_raw_input = self.__application_terminal_window.
    getstr() # Read as bytes
520                 output_file_location_input = str(output_file_location_raw_input, "
    utf-8")
521                 # Write the expression line to history
522                 self.__write_expression_visual_pad(f"{output_file_location_prompt}{
    output_file_location_input}")
523 
524                 # Get sort order
525                 order_chosen = None
526                 while order_chosen not in set(['1', '2']):
527                     self.__application_terminal_window.addstr(self.
    __application_terminal_y, self.__application_terminal_x, 'Please select an order
     of sorting (Enter the number):')
528                     self.__write_expression_visual_pad('Please select an order of
```

```
    sorting (Enter the number):')
529                 self.__application_terminal_window.scroll()
530
531                 self.__application_terminal_window.addstr(self.
    __application_terminal_y, self.__application_terminal_x, '1. Ascending')
532                 self.__write_expression_visual_pad('1. Ascending')
533                 self.__application_terminal_window.scroll()
534
535                 self.__application_terminal_window.addstr(self.
    __application_terminal_y, self.__application_terminal_x, '2. Descending')
536                 self.__write_expression_visual_pad('2. Descending')
537                 self.__application_terminal_window.scroll()
538
539                 self.__refresh()
540
541                 order_prompt = "Your selection: "
542                 self.__application_terminal_window.addstr(self.
    __application_terminal_y, self.__application_terminal_x, order_prompt)
543                 order_raw_input = self.__application_terminal_window.getstr() #
    Read as bytes
544                 order_input = str(order_raw_input, "utf-8")
545                 order_chosen = order_input
546
547                 self.__write_expression_visual_pad(f"{order_prompt}{order_input}
    ")
548
549             sort_order = None
550             if order_chosen == '1':
551                 sort_order = 'ascending'
552             elif order_chosen == '2':
553                 sort_order = 'descending'
554
555             f = File()
556             try:
557                 expressions = f.read(input_file_location_input)
558                 evaluated_expressions = []
559                 for expression in expressions:
560                     expression = expression[0]
561                     evaluator = Evaluator()
562                     evaluation = evaluator.evaluate(expression)
563                     evaluated_expressions.append([expression, evaluation])
564                 sorter = Sort(evaluated_expressions, sort_order=sort_order)
565                 sorted_expressions = sorter.sort()
566                 f.write(output_file_location_input, sorted_expressions)
567
568                 self.__write_expression_visual_pad('')
569                 self.__application_terminal_window.scroll()
570
571                 self.__application_terminal_window.addstr(self.
    __application_terminal_y, self.__application_terminal_x, '>>>>Evaluation and
    sorted started:')
572                 self.__write_expression_visual_pad('>>>>Evaluation and sorted
    started:')
573                 self.__application_terminal_window.scroll()
574
575                 self.__write_expression_visual_pad('')
576                 self.__application_terminal_window.scroll()
577
```

```
                    for sorted_expression in sorted_expressions:
                        evaluation = sorted_expression[0]
                        expressions = sorted_expression[1]
                        self.__application_terminal_window.addstr(self.__application_terminal_y, self.__application_terminal_x, f"*** Expressions with value = {evaluation}")
                        self.__write_expression_visual_pad(f"*** Expressions with value = {evaluation}")
                        self.__application_terminal_window.scroll()
                        for expression in expressions:
                            self.__application_terminal_window.addstr(self.__application_terminal_y, self.__application_terminal_x, f"{expression} ==> {evaluation}")
                            self.__write_expression_visual_pad(f"*** Expressions with value = {evaluation}")
                            self.__application_terminal_window.scroll()
                        self.__write_expression_visual_pad('')
                        self.__application_terminal_window.scroll()

                    self.__application_terminal_window.addstr(self.__application_terminal_y, self.__application_terminal_x, '>>>Evaluation and sorting completed!')
                    self.__write_expression_visual_pad('>>>Evaluation and sorting completed!')
                    self.__application_terminal_window.scroll()

                    self.__write_expression_visual_pad('')
                    self.__application_terminal_window.scroll()

                except Exception as e:
                    error_msg = f"Error occured: {str(e)}"
                    self.__application_terminal_window.addstr(self.__application_terminal_y, self.__application_terminal_x, error_msg, curses.color_pair(5))
                    self.__write_expression_visual_pad(error_msg, curses.color_pair(5))
                    self.__application_terminal_window.scroll()

                # Process the expression and do something
                curses.noecho()
                curses.curs_set(0)
            else:
                continue

            self.__application_terminal_window.addstr(self.__application_terminal_y, self.__application_terminal_x, expression_evaluator_prompt_str)
            self.__write_expression_visual_pad(expression_evaluator_prompt_str)
            self.__application_terminal_window.scroll()

            self.__refresh()

    def __write_expression_visual_pad(self, history_str, attribute=None):
        '''
        Writes the given string to the expression pad
        '''
        if attribute is None:
            self.__application_history_pad.addstr(self.__history_length - 1, 0, history_str)
```

```
622         else:
623             self.__application_history_pad.addstr(self.__history_length - 1, 0,
    history_str, attribute)
624         self.__application_history_pad.scroll()
625 
626     def __load_application_hisotry_pad(self):
627         # Mainly for setting curses settings
628         curses.curs_set(0)
629         curses.noecho()
630         curses.cbreak()
631 
632         self.__current_application = f"Application History"
633         self.__current_application_attributes = curses.color_pair(3)
634 
635         # Load header to update it
636         self.__update_application_panel()
637 
638     def __update_application_history_pad(self):
639         self.__load_application_hisotry_pad()
640         self.__refresh()
641         while True:
642             self.__application_history_pad.refresh(self.
    __application_history_pad_pos - (self.__application_height - 2), 0, self.
    __header_height + 1, 1, self.__height - 2, self.__width - 1)
643             user_key = self.__application_history_pad.getch()
644             if user_key in set([curses.KEY_UP, ord('w'), ord('k')]):
645                 # Move down
646                 self.__application_history_pad_pos -= 1
647             elif user_key in set([curses.KEY_DOWN, ord('s'), ord('j')]):
648                 # Move up
649                 if self.__application_history_pad_pos < self.__history_length:
650                     self.__application_history_pad_pos += 1
651             elif user_key in set([27, ord('i')]):
652                 return
653 
654     def __main(self, stdscr):
655         self.__stdscr = stdscr
656         self.__set_up()
657 
658         # Start the application
659         # Application loop is -> Update -> Load -> Refresh -> loop
660         # Refresh is handled by the Update function
661         # Start the main application here
662         self.__update_selection()
663 
664 if __name__ == '__main__':
665     cli = CLI()
```

Listing 5: Source Code for: `./src/advance/cli.py`

C.3 `./src/advance/expression_evaluator`

C.3.1 `./src/advance/expression_evaluator/__init__.py`

```
1 from .evaluator import Evaluator
```

Listing 6: Source Code for: `./src/advance/expression_evaluator/__init__.py`

C.3.2 ./src/advance/expression_evaluator/evaluator.py

```
'''Python file for the expression evaluataor

The expression evaluator is a wrapper class for the compiler, wrapping all the
    functionality into a single class and method
'''
from .compiler import Lexer, Parser, Interpreter
from .compiler.node_traversal import PreOrderTraversal, PostOrderTraversal

class Evaluator(object):
    def __init__(self):
        self.__tokens = None
        self.__ast = None

    # Main public methods
    def evaluate(self, input_expression):
        '''
        Public method used to call an evaluation for an expression
        Populates the instances __tokens and __ast
        '''
        lexer = Lexer(input_expression)
        self.__tokens = lexer.get_tokens()
        parser = Parser(self.__tokens)
        self.__ast = parser.get_ast()
        interpreter = Interpreter(self.__ast)
        evaluation = interpreter.get_interpretation()
        return evaluation

    def get_traversal(self, type):
        if type == 'pre_order':
            traversal = PreOrderTraversal()
            return traversal.traverse(self.__ast)
        if type == 'post_order':
            traversal = PostOrderTraversal()
            return traversal.traverse(self.__ast)
```

Listing 7: Source Code for: ./src/advance/expression_evaluator/evaluator.py

C.4 ./src/advance/expression_evaluator/compiler

C.4.1 ./src/advance/expression_evaluator/compiler/parser.py

```
'''Python file for compiler's parser

This contains the parser, which creates the AST from a series of tokens
'''
# Importing token types
from ..tokens import Token
from ..tokens.token_type import PLUS, MINUS, MUL, DIV, MODULUS, INT_DIV, POWER,
    LPARAM, RPARAM, COMMA, EOF
from ..tokens.token_type import NUMBER, IDENTIFIER, STRING

# Importing nodes
from ..nodes import Number_Node, String_Node, UnaryOp_Node, BinaryOp_Node,
    Function_Node

class Parser(object):
```

```
14      def __init__(self, tokens):
15          self.__tokens = tokens
16          self.__cur_pos = 0
17          # When the ast is fully built
18          self.__ast = None
19
20      # Class helper methods
21      def __check_end(self, pos: int):
22          '''
23          Helper private method to check if the position given has already reached the
        end of the given tokens
24          '''
25          return self.__cur_pos >= len(self.__tokens)
26
27      def __get_token(self, pos: int):
28          '''
29          Helper private method to get the token at a given pos
30          If the token has reached the end, return None
31          '''
32          if not self.__check_end(pos):
33              return self.__tokens[pos]
34          return None
35
36      def __check_match_token_type(self, token: Token, token_types: list):
37          '''
38          Helper private method to see if given token matches any of the given types
39          '''
40          if token is None:
41              return False
42          return token.type in set(token_types)
43
44      def __error(self):
45          # Placeholder error method
46          # TODO: Make proper one
47          raise Exception('Parser error')
48
49      # Class auxiliary methods
50      @property
51      def __is_end(self):
52          '''
53          Auxiliary private method to check if the parser has reached the end of the
        tokens given
54          '''
55          return self.__check_end(self.__cur_pos)
56
57      @property
58      def __cur_token(self) -> Token:
59          '''
60          Auxiliary private method used to get the current token the parser is on
61          '''
62          return self.__get_token(self.__cur_pos)
63
64      @property
65      def __peek(self):
66          '''
67          Auxiliary private method peek at the next token
68          '''
69          return self.__get_token(self.__cur_pos + 1)
```

```
70
71      def __advance(self):
72          '''
73          Auxiliary private method to advance the parser
74          '''
75          self.__cur_pos += 1
76
77      def __match_token(self, token_types: list):
78          '''
79          Auxiliary private method to check if current token match given token_types
80          '''
81          return self.__check_match_token_type(self.__cur_token, token_types)
82
83      def __consume(self, token_types: list):
84          '''
85          Auxiliary private method to check if the current token's type matches the
    given token_types
86          Consumes the token if it is
87          '''
88          if self.__match_token(token_types):
89              self.__advance()
90          else:
91              # TODO: Err shld be specific to token types given
92              self.__error()
93
94      # Parser Grammar implementation
95      def __expression(self):
96          node = self.__term()
97          token_types = [PLUS, MINUS, MODULUS]
98          while self.__match_token(token_types):
99              token = self.__cur_token
100             self.__consume(token_types)
101             node = BinaryOp_Node(token, node, self.__term())
102         return node
103
104     def __term(self):
105         node = self.__factor()
106         token_types= [MUL, DIV, INT_DIV]
107         while self.__match_token(token_types):
108             token = self.__cur_token
109             self.__consume(token_types)
110             node = BinaryOp_Node(token, node, self.__factor())
111         return node
112
113     def __factor(self):
114         node = self.__unary()
115         token_types = [POWER]
116         while self.__match_token(token_types):
117             token = self.__cur_token
118             self.__consume(token_types)
119             node = BinaryOp_Node(token, node, self.__unary())
120         return node
121
122     # TODO: Chuan Hao
123     # Think of a better way to check and continue to store token types to check
124     def __unary(self):
125         if self.__match_token([PLUS, MINUS]):
126             token = self.__cur_token
```

```
            self.__consume([PLUS, MINUS])
            return UnaryOp_Node(token, self.__unary())
        else:
            return self.__call()

    def __call(self):
        if self.__match_token([IDENTIFIER]):
            token = self.__cur_token
            self.__consume([IDENTIFIER])
            self.__consume([LPARAM])
            # Else we have arguments to parse
            arguments = self.__arguments()
            self.__consume([RPARAM])
            return Function_Node(token, arguments)

        else:
            return self.__primary()

    def __arguments(self) -> list:
        nodes = []
        # To tell if there are more expression, check for ending of function
        if not self.__match_token([RPARAM]):
            nodes.append(self.__expression())
            while self.__match_token([COMMA]):
                self.__consume([COMMA])
                nodes.append(self.__expression())
        return nodes

    def __primary(self):
        if self.__match_token([NUMBER]):
            token = self.__cur_token
            self.__consume([NUMBER])
            return Number_Node(token)
        elif self.__match_token([STRING]):
            token = self.__cur_token
            self.__consume([STRING])
            return String_Node(token)
        elif self.__match_token([LPARAM]):
            self.__consume([LPARAM])
            expression = self.__expression()
            self.__consume([RPARAM])
            return expression
        else:
            # If when parsing, primary grammar fails, there is an error
            self.__error()

    # Main Parser logic
    def __parse(self):
        '''
        Private method for the parser to parse the given tokens into the ast
        '''
        # TODO: Chuan Hao
        # What if there is nothing
        ast = self.__expression()
        if not self.__match_token([EOF]):
            # TODO: Chuan Hao
            # Should have parsed all tokens and left EOF
            self.__error()
```

```
        return ast

    # Public methods
    def get_ast(self):
        '''
        Public method to get the ast of the parser
        '''
        # Check if ast has already been parsed
        if self.__ast is not None:
            return self.__ast
        self.__ast = self.__parse()
        return self.__ast
```

Listing 8: Source Code for: `./src/advance/expression_evaluator/compiler/parser.py`

C.4.2 `./src/advance/expression_evaluator/compiler/__init__.py`

```
from .lexer import Lexer
from .parser import Parser
from .interpreter import Interpreter
```

Listing 9: Source Code for: `./src/advance/expression_evaluator/compiler/__init__.py`

C.4.3 `./src/advance/expression_evaluator/compiler/lexer.py`

```
'''Python file for compiler's lexer

This contains the Lexer class which handles turning the input text(raw expression)
    into a series of tokens
'''

# Importing token types
from ..tokens import Token
from ..tokens.token_type import PLUS, MINUS, MUL, DIV, MODULUS, INT_DIV, POWER,
    LPARAM, RPARAM, COMMA, EOF
from ..tokens.token_type import NUMBER, IDENTIFIER, STRING
from ..tokens.token_type import RESERVED_KEYWORDS

# Building the Lexer class
class Lexer(object):
    def __init__(self, text):
        self.__text = text
        self.__tokens = []
        self.__cur_pos = 0

    # Class helper methods
    def __check_end(self, pos: int):
        '''
        Helper private method to check if the position given has already reached the
     end of the text
        '''
        return pos >= len(self.__text)

    def __get_char(self, pos: int):
        '''
        Helper private method to get the char at a given pos
        If the pos has reached the end, return None
        '''
```

```
        if not self.__check_end(pos):
            return self.__text[pos]
        return None

    # TODO: Chuan Hao, implement the syntax error
    def __syntax_error(self):
        pass

    def __error(self):
        # Placeholder error method
        # TODO: Make proper one
        raise Exception('Lexer error')

    # Class auxiliary methods
    @property
    def __cur_char(self):
        '''
        Auxiliary private method to return the current char of the lexer
        '''
        return self.__get_char(self.__cur_pos)

    @property
    def __peek(self):
        '''
        Auxiliary private method to peek at the next char without moving the lexer
        '''
        return self.__get_char(self.__cur_pos + 1)

    @property
    def __is_end(self):
        '''
        Auxiliary private method to check if the lexer has reached the end
        '''
        return self.__check_end(self.__cur_pos)

    def __advance(self):
        '''
        Auxiliary private method to advance the lexer to the next character
        '''
        self.__cur_pos += 1

    # Lexer, utility methods
    def __is_whitespace(self, c: str):
        if c is None:
            return False
        return c.isspace()

    def __is_quote(self, c: str):
        if c is None:
            return False
        return c == "\'"

    def __is_digit(self, c: str):
        if c is None:
            return False
        return c.isdigit()

    def __is_alpha(self, c: str):
```

```
89          if c is None:
90              return False
91          return c.isalpha()
92
93      # Lexer logic methods
94      def __skip_whitespace(self):
95          '''
96          Private method used to skip whitespace as it is ignored
97          '''
98          while self.__is_whitespace(self.__cur_char):
99              self.__advance()
100         return
101
102     def __number(self):
103         '''
104         Private method used to tokenize number tokens
105         '''
106         pos = self.__cur_pos
107         number_value = ''
108         # For any digits before, DIGIT*
109         while self.__is_digit(self.__cur_char):
110             number_value += self.__cur_char
111             self.__advance()
112         # If there is a '.'
113         if self.__cur_char == '.':
114             number_value += '.'
115             self.__advance()
116             # Rest of DIGIT+
117             while self.__is_digit(self.__cur_char):
118                 number_value += self.__cur_char
119                 self.__advance()
120         # Error checking if it was only a .
121         if number_value == '.':
122             self.__error()
123
124         number_value = float(number_value)
125         return Token(NUMBER, number_value, pos)
126
127     def __identifier(self):
128         '''
129         Private method used to tokenize identifiers
130
131         Differentiated with no quotes and only alphas
132         '''
133         pos = self.__cur_pos
134         identifier_value = ''
135         while self.__is_alpha(self.__cur_char):
136             identifier_value += self.__cur_char
137             self.__advance()
138
139         # Invalid identifier
140         # Only accepts reserved keywords
141         if identifier_value not in RESERVED_KEYWORDS:
142             self.__error()
143         return Token(IDENTIFIER, identifier_value, pos)
144
145     def __string(self):
146         '''
```

```
        Private method used to tokenize strings
        '''
        pos = self.__cur_pos
        string_value = ''
        # Need first quote
        if self.__is_quote(self.__cur_char):
            self.__advance()
        else:
            self.__error()

        # Consume all chars in the middle
        while not self.__is_quote(self.__cur_char) and self.__cur_char is not None:
            string_value += self.__cur_char
            self.__advance()

        # Need ending quote
        if self.__is_quote(self.__cur_char):
            self.__advance()
        else:
            self.__error()

        return Token(STRING, string_value, pos)

    # Main lexer methods
    def __get_next_token(self):
        '''
        Private method to get the next token from the given text
        '''
        if not self.__is_end:
            # Skipping all the ignored chars
            if self.__is_whitespace(self.__cur_char):
                self.__skip_whitespace()
                return self.__get_next_token()

            # Checking single char tokens
            if self.__cur_char == '+':
                token = Token(PLUS, '+', self.__cur_pos)
                self.__advance()
                return token
            elif self.__cur_char == '-':
                token = Token(MINUS, '-', self.__cur_pos)
                self.__advance()
                return token
            elif self.__cur_char == '%':
                token = Token(MODULUS, '%', self.__cur_pos)
                self.__advance()
                return token
            elif self.__cur_char == '(':
                token = Token(LPARAM, '(', self.__cur_pos)
                self.__advance()
                return token
            elif self.__cur_char == ')':
                token = Token(RPARAM, ')', self.__cur_pos)
                self.__advance()
                return token
            elif self.__cur_char == ',':
                token = Token(COMMA, ',', self.__cur_pos)
                self.__advance()
```

```
                return token

            # Checking double char tokens
            if self.__cur_char == '*':
                # If POWER
                if self.__peek == '*':
                    token = Token(POWER, '**', self.__cur_pos)
                    self.__advance()
                    self.__advance()
                    return token
                # else MUL
                token = Token(MUL, '*', self.__cur_pos)
                self.__advance()
                return token
            elif self.__cur_char == '/':
                # If INT_DIV
                if self.__peek == '/':
                    token = Token(INT_DIV, '//', self.__cur_pos)
                    self.__advance()
                    self.__advance()
                    return token
                # else DIV
                token = Token(DIV, '/', self.__cur_pos)
                self.__advance()
                return token

            # Check multi-strings
            # Check NUMBER, then IDENTIFIER, then STRING
            if self.__is_digit(self.__cur_char) or self.__cur_char == '.':
                return self.__number()
            elif self.__is_alpha(self.__cur_char):
                return self.__identifier()
            elif self.__is_quote(self.__cur_char):
                return self.__string()

            # If not able to tokenize, error
            self.__error()

        # If we have reached the end, EOF
        return Token(EOF, None, self.__cur_pos)

    # Public methods
    def get_tokens(self):
        '''
        Public method to get the tokens tokenized by the lexer
        '''
        # If lexer has already tokenized the text
        if len(self.__tokens) > 0:
            return self.__tokens

        tokens = []
        while True:
            cur_token = self.__get_next_token()
            tokens.append(cur_token)

            if cur_token.type == EOF:
                break
        self.__tokens = tokens
```

```
263         return self.__tokens
```

Listing 10: Source Code for: `./src/advance/expression_evaluator/compiler/lexer.py`

C.4.4 `./src/advance/expression_evaluator/compiler/interpreter.py`

```
'''Python file for the compiler's Interpreter class

The interpreter class is meant to go through and interpret(evaluate) the AST
'''
# Python libs
import math

# Importing token types
from ..tokens import Token
from ..tokens.token_type import PLUS, MINUS, MUL, DIV, MODULUS, INT_DIV, POWER,
    LPARAM, RPARAM, COMMA, EOF
from ..tokens.token_type import E, PI, SIN, COS, TAN, FLOOR, CEIL, LN, LG, FACTORIAL
    , SQRT, BIN, HEX, LOG, POW, MOD, PERM, COMB

# Import AST and NodeVisitor
from ..nodes import AST
from ..nodes import Number_Node, String_Node, UnaryOp_Node, BinaryOp_Node,
    Function_Node
from ..nodes import NodeVisitor

# Common algos
from ....common.common_algos import bin_to_decimal, hex_to_decimal

class Interpreter(NodeVisitor):
    def __init__(self, ast: AST):
        self.__ast = ast

    # Class helper methods
    def __error(self):
        # Placeholder error method
        # TODO: Make proper one
        raise Exception('Interpreter error')

    # Errors
    # Type error
    # Mixing type op

    # Node Type Visitor Implementation
    def visit_BinaryOp_Node(self, node: BinaryOp_Node):
        token_type = node.token.type
        left = self.visit(node.left)
        right = self.visit(node.right)
        # print(node.token)
        # print(left, right)
        # Check if binary op is done on same type of variable
        if not (type(left) == type(right) or ((isinstance(left, int) or isinstance(
    left, float)) == (isinstance(right, int) or isinstance(right, float)))):
            self.__error()

        # For both str or int
        if token_type in set([PLUS]):
            # Supports only number or str
```

```
            if not (isinstance(left, int) or isinstance(left, float) or isinstance(left, str)):
                # Type for bin op is not supported
                self.__error()
            if token_type == PLUS:
                return left + right

        # For int only
        if token_type in set([MINUS, MODULUS, MUL, DIV, POWER, INT_DIV]):
            # Supports only number
            if not (isinstance(left, int) or isinstance(left, float)):
                # Type for bin op is not supported
                self.__error()
            if token_type == MINUS:
                return left - right
            elif token_type == MODULUS:
                return left % right
            elif token_type == MUL:
                return left * right
            elif token_type == DIV:
                return left / right
            elif token_type == POWER:
                return left ** right
            elif token_type == INT_DIV:
                return left // right

        # There is an error?
        self.__error()

    def visit_UnaryOp_Node(self, node: UnaryOp_Node):
        token_type = node.token.type
        child = self.visit(node.child)

        # Node only supports number
        if not (isinstance(child, int) or isinstance(child, float)):
            self.__error()

        if token_type == PLUS:
            return child
        elif token_type == MINUS:
            return -child

    def visit_Function_Node(self, node: Function_Node):
        node_token = node.token
        function_name = node_token.value
        arguments = [self.visit(argument) for argument in node.arguments]
        if function_name in set([E, PI]):
            # Arity 0
            if not node.check_arity(0):
                self.__error()
            # Return based on function call
            if function_name == E:
                return math.e
            elif function_name == PI:
                return math.pi
        elif function_name in set([SIN, COS, TAN, FLOOR, CEIL, LN, LG, FACTORIAL, SQRT, BIN, HEX]):
            # Arity 1
```

```
105             if not node.check_arity(1):
106                 self.__error()
107             argument = arguments[0]
108             # Return based on function call
109             if function_name in set([SIN, COS, TAN, FLOOR, CEIL, LN, LG, FACTORIAL,
    SQRT]):
110                 # Only takes in number
111                 if not (isinstance(argument, int) or isinstance(argument, float)):
112                     # Type error
113                     self.__error()
114                 # Eval
115                 if function_name == SIN:
116                     return math.sin(argument)
117                 elif function_name == COS:
118                     return math.cos(argument)
119                 elif function_name == TAN:
120                     return math.tan(argument)
121                 elif function_name == FLOOR:
122                     return math.floor(argument)
123                 elif function_name == CEIL:
124                     return math.ceil(argument)
125                 elif function_name == LN:
126                     return math.log(argument)
127                 elif function_name == LG:
128                     return math.log10(argument)
129                 elif function_name == FACTORIAL:
130                     return math.factorial(argument)
131                 elif function_name == SQRT:
132                     return math.sqrt(argument)
133             if function_name in set([BIN, HEX]):
134                 # Only takes in strings
135                 if not (isinstance(argument, str)):
136                     self.__error()
137                 # Eval
138                 if function_name == BIN:
139                     return bin_to_decimal(argument)
140                 elif function_name == HEX:
141                     return hex_to_decimal(argument)
142         elif function_name in set([LOG, POW, MOD, PERM, COMB]):
143             # Arity 2
144             if not node.check_arity(2):
145                 self.__error()
146             # Making sure all are numbers
147             for argument in arguments:
148                 if not (isinstance(argument, int) or isinstance(argument, float)):
149                     self.__error()
150 
151             if function_name == LOG:
152                 return math.log(arguments[0], arguments[1])
153             elif function_name == POW:
154                 return math.pow(arguments[0], arguments[1])
155             elif function_name == MOD:
156                 return arguments[0]%arguments[1]
157             elif function_name == PERM:
158                 # TODO: Check n >= r
159                 # Also try catch for no float value stuff
160                 n = arguments[0]
161                 r = arguments[1]
```

```
                return ((math.factorial(n)) / (math.factorial(n - r)))
            elif function_name == COMB:
                # TODO: Check n >= r
                n = arguments[0]
                r = arguments[1]
                return ((math.factorial(n)) / (math.factorial(r) * math.factorial(n
    - r)))

    def visit_Number_Node(self, node: Number_Node):
        return node.value

    def visit_String_Node(self, node: String_Node):
        return node.value

    # Public methods
    def get_interpretation(self):
        return self.visit(self.__ast)
```

Listing 11: Source Code for: ./src/advance/expression_evaluator/compiler/interpreter.py

C.5 ./src/advance/expression_evaluator/compiler/node_traversal

C.5.1 ./src/advance/expression_evaluator/compiler/node_traversal/traversal.py

```
'''Python file for the traversal abstract class

This is an abstract class inherited from the NodeVisitor class to allow to kwargs
'''
from ...nodes import AST
from ...nodes import NodeVisitor

class Traversal(NodeVisitor):
    def visit(self, node: AST, **kwargs):
        '''
        Public visit method
        The visit method implemented is used to call the respective visit method
    based on the node type
        It then passes the return value back
        '''
        node_name = type(node).__name__
        method_name = f"visit_{node_name}"
        vist_method = getattr(self, method_name, self.__visit_method_error)
        return vist_method(node, **kwargs)

    def __visit_method_error(self, node: AST, **kwargs):
        '''
        Private helper method
        Used to raise a NotImplementedError when the node type visit method is not
    implemented
        '''
        node_name = type(node).__name__
        error_msg = f"Visit method for {node_name} not implemented"
        error_msg += '\n'
        error_msg += f"Please implement the method visit_{node_name}"
        error_msg += '\n'
        error_msg += f"Did not expect kwargs, {','.join(['(' + str(k) + ',' + str(v)
     + ')' for k, v in kwargs.items()])}"
```

```
31          raise NotImplementedError(error_msg)
```

Listing 12: Source Code for: `./src/advance/expression_evaluator/compiler/node_traversal/traversal.py`

C.5.2 `./src/advance/expression_evaluator/compiler/node_traversal/__init__.py`

```
1 from .pre_order import PreOrderTraversal
2 from .post_order import PostOrderTraversal
```

Listing 13: Source Code for: `./src/advance/expression_evaluator/compiler/node_traversal/__init__.py`

C.5.3 `./src/advance/expression_evaluator/compiler/node_traversal/post_order.py`

```
1  '''Python file for the post_order traversal
2
3  This is a helper class used to generate the post-order traversal of a given ast
4  '''
5  from ...nodes import AST
6  from ...nodes import Number_Node, String_Node, UnaryOp_Node, BinaryOp_Node,
       Function_Node
7
8  from .traversal import Traversal
9
10 class PostOrderTraversal(Traversal):
11     def __init__(self):
12         self.__traversal = []
13
14     # Public methods
15     def traverse(self, ast: AST):
16         '''
17         Public method called to get the traversal graph
18         '''
19         self.visit(ast)
20         return self.__traversal
21
22     # Node Type Visitor Implementation
23     def visit_BinaryOp_Node(self, node: BinaryOp_Node, level: int=1):
24         token = node.token
25         self.visit(node.left, level=level+1)
26         self.visit(node.right, level=level+1)
27         self.__traversal.append(f"{'>'*level}: {str(token)}")
28     def visit_UnaryOp_Node(self, node: UnaryOp_Node, level: int=1):
29         token = node.token
30         self.visit(node.child, level=level+1)
31         self.__traversal.append(f"{'>'*level}: {str(token)}")
32     def visit_Function_Node(self, node: Function_Node, level: int=1):
33         token = node.token
34         for argument in node.arguments:
35             self.visit(argument, level=level+1)
36         self.__traversal.append(f"{'>'*level}: {str(token)}")
37     def visit_Number_Node(self, node: Number_Node, level: int=1):
38         token = node.token
39         self.__traversal.append(f"{'>'*level}: {str(token)}")
40     def visit_String_Node(self, node: String_Node, level: int=1):
41         token = node.token
42         self.__traversal.append(f"{'>'*level}: {str(token)}")
```

Listing 14: Source Code for: `./src/advance/expression_evaluator/compiler/node_traversal/post_order.py`

C.5.4 ./src/advance/expression_evaluator/compiler/node_traversal/pre_order.py

```python
'''Python file for the pre_order traversal

This is a helper class used to generate the pre-order traversal of a given ast
'''
from ...nodes import AST
from ...nodes import Number_Node, String_Node, UnaryOp_Node, BinaryOp_Node,
    Function_Node

from .traversal import Traversal

class PreOrderTraversal(Traversal):
    def __init__(self):
        self.__traversal = []

    # Public methods
    def traverse(self, ast: AST):
        '''
        Public method called to get the traversal graph
        '''
        self.visit(ast)
        return self.__traversal

    # Node Type Visitor Implementation
    def visit_BinaryOp_Node(self, node: BinaryOp_Node, level: int=1):
        token = node.token
        self.__traversal.append(f"{'>'*level}: {str(token)}")
        self.visit(node.left, level=level+1)
        self.visit(node.right, level=level+1)
    def visit_UnaryOp_Node(self, node: UnaryOp_Node, level: int=1):
        token = node.token
        self.__traversal.append(f"{'>'*level}: {str(token)}")
        self.visit(node.child, level=level+1)
    def visit_Function_Node(self, node: Function_Node, level: int=1):
        token = node.token
        self.__traversal.append(f"{'>'*level}: {str(token)}")
        for argument in node.arguments:
            self.visit(argument, level=level+1)
    def visit_Number_Node(self, node: Number_Node, level: int=1):
        token = node.token
        self.__traversal.append(f"{'>'*level}: {str(token)}")
    def visit_String_Node(self, node: String_Node, level: int=1):
        token = node.token
        self.__traversal.append(f"{'>'*level}: {str(token)}")
```

Listing 15: Source Code for: ./src/advance/expression_evaluator/compiler/node_traversal/pre_order.py

C.6 ./src/advance/expression_evaluator/nodes

C.6.1 ./src/advance/expression_evaluator/nodes/string_node.py

```python
'''Python Class String_Node

String_Node used to represent a string
Only stores the token and value
'''
# Import Token and AST
```

```python
from ..tokens import Token
from .ast import AST

class String_Node(AST):
    def __init__(self, token: Token):
        self.token = token
        self.value = self.token.value
```

Listing 16: Source Code for: `./src/advance/expression_evaluator/nodes/string_node.py`

C.6.2 `./src/advance/expression_evaluator/nodes/__init__.py`

```python
# AST nodes
from .ast import AST
from .binary_op_node import BinaryOp_Node
from .function_node import Function_Node
from .number_node import Number_Node
from .string_node import String_Node
from .unary_op_node import UnaryOp_Node

# Node Visitor
from .node_visitor import NodeVisitor
```

Listing 17: Source Code for: `./src/advance/expression_evaluator/nodes/__init__.py`

C.6.3 `./src/advance/expression_evaluator/nodes/number_node.py`

```python
'''Python Class Number_Node

Number_Node used to represent a number
Only stores the token and value
'''
# Import Token and AST
from ..tokens import Token
from .ast import AST

class Number_Node(AST):
    def __init__(self, token: Token):
        self.token = token
        self.value = self.token.value
```

Listing 18: Source Code for: `./src/advance/expression_evaluator/nodes/number_node.py`

C.6.4 `./src/advance/expression_evaluator/nodes/binary_op_node.py`

```python
'''Python Class BinaryOp_Node

BinaryOp Node represents a binary operation with a left and right expression
'''
# Import Token and AST
from ..tokens import Token
from .ast import AST

class BinaryOp_Node(AST):
    def __init__(self, token: Token, left: AST, right: AST):
        self.token = token
        self.left = left
```

```
13          self.right = right
```

Listing 19: Source Code for: ./src/advance/expression_evaluator/nodes/binary_op_node.py

C.6.5 ./src/advance/expression_evaluator/nodes/ast.py

```
1  '''Python file for the AST class
2
3  The Abstract Syntax Tree(AST) class, is meant to be inherited by all the other nodes
       classes
4  '''
5  class AST(object):
6      pass
```

Listing 20: Source Code for: ./src/advance/expression_evaluator/nodes/ast.py

C.6.6 ./src/advance/expression_evaluator/nodes/node_visitor.py

```
1  '''Python file for NodeVisitor Class
2
3  The NodeVisitor Class is an abstract class meant to be inherited by the compiler's
       interpreter
4  It is also used to create the traversal classes
5  '''
6  # Import AST
7  from .ast import AST
8
9  class NodeVisitor(object):
10     def visit(self, node: AST):
11         '''
12         Public visit method
13         The visit method implemented is used to call the respective visit method
       based on the node type
14         It then passes the return value back
15         '''
16         node_name = type(node).__name__
17         method_name = f"visit_{node_name}"
18         vist_method = getattr(self, method_name, self.__visit_method_error)
19         return vist_method(node)
20
21     def __visit_method_error(self, node: AST):
22         '''
23         Private helper method
24         Used to raise a NotImplementedError when the node type visit method is not
       implemented
25         '''
26         node_name = type(node).__name__
27         error_msg = f"Visit method for {node_name} not implemented"
28         error_msg += '\n'
29         error_msg += f"Please implement the method visit_{node_name}"
30         raise NotImplementedError(error_msg)
```

Listing 21: Source Code for: ./src/advance/expression_evaluator/nodes/node_visitor.py

C.6.7 ./src/advance/expression_evaluator/nodes/function_node.py

```
1  '''Python Class Function_Node
2
```

```
Function_Node represents a function call node
Used by the reserved keywords and takes in arguments with arity
'''
# Import Token and AST
from ..tokens import Token
from .ast import AST

class Function_Node(AST):
    def __init__(self, token: Token, arguments: list):
        self.token = token
        self.arguments = arguments
        self.arity = len(self.arguments)

    def check_arity(self, actual_arity):
        '''
        Helper method to check if the given arity matches the arity of the function
        Used to check runtime(interpretation) errors
        '''
        return self.arity == actual_arity
```

Listing 22: Source Code for: ./src/advance/expression_evaluator/nodes/function_node.py

C.6.8 ./src/advance/expression_evaluator/nodes/unary_op_node.py

```
'''Python Class Unary_Node

Unary_Node used to represent unary operations like negative
Has a child, which represents the node to do the operation on
'''
# Import Token and AST
from ..tokens import Token
from .ast import AST

class UnaryOp_Node(AST):
    def __init__(self, token: Token, child: AST):
        self.token = token
        self.child = child
```

Listing 23: Source Code for: ./src/advance/expression_evaluator/nodes/unary_op_node.py

C.7 ./src/advance/expression_evaluator/tokens

C.7.1 ./src/advance/expression_evaluator/tokens/__init__.py

```
from .token import Token
```

Listing 24: Source Code for: ./src/advance/expression_evaluator/tokens/__init__.py

C.7.2 ./src/advance/expression_evaluator/tokens/token.py

```
'''Python file contain the Token Class

The Token Class is used to create tokens by the lexer
These token are later parsed by the parser
'''

class Token(object):
```

```
    def __init__(self, _type: str, value, pos: int):
        self.type = _type
        self.value = value
        self.pos = pos

    def __str__(self):
        s = f"Token({self.type}, {self.value})"
        return s
```

Listing 25: Source Code for: ./src/advance/expression_evaluator/tokens/token.py

C.7.3 ./src/advance/expression_evaluator/tokens/token_type.py

```
'''Python file for all the token types

This contains all the static token types used in the evaluator
'''
# Special Tokens
# +, -, *, /, %
PLUS, MINUS, MUL, DIV, MODULUS = ['PLUS', 'MINUS', 'MUL', 'DIV', 'MODULUS']
# **, //
INT_DIV, POWER = ['INT_DIV', 'POWER']
# (, )
LPARAM, RPARAM = ['LPARAM', 'RPARAM']
# ,
COMMA = 'COMMA'
EOF = 'EOF'

# Normal Token Types
NUMBER = 'NUMBER'
IDENTIFIER = 'IDENTIFIER'
STRING = 'STRING'

# Reserved Keywords
E, PI = ['E', 'PI']
SIN, COS, TAN, FLOOR, CEIL, LN, LG, FACTORIAL, SQRT, BIN, HEX = ['sin', 'cos', 'tan'
    , 'floor', 'ceil', 'ln', 'lg', 'factorial', 'sqrt', 'bin', 'hex']
LOG, POW, MOD, PERM, COMB = ['log', 'pow', 'mod', 'perm', 'comb']
RESERVED_KEYWORDS = set([
    # Arity 0
    E, PI,
    # Arity 1
    SIN, COS, TAN, FLOOR, CEIL, LN, LG, FACTORIAL, SQRT, BIN, HEX,
    # Arity 2
    LOG, POW, MOD, PERM, COMB
])
```

Listing 26: Source Code for: ./src/advance/expression_evaluator/tokens/token_type.py

C.8 ./src/basic

C.8.1 ./src/basic/__init__.py

```
\subsubsection{\lstinline[language=Bash]{./src/basic/cli.py}}

\begin{lstlisting}[language=Python, caption={Source Code for: \lstinline{./src/basic
    /cli.py}}]
'''
```

```
    This Python File deals with the CLI of the application

    It also contains a CLI class that will have the different necessary print
    statements for each section identified
'''

#* Importing Modules
import os.path
from .expression_evaluator.evaluator import Evaluator
from .expression_evaluator.compiler.node_traversal import PreOrder, InOrder,
    PostOrder
from ..common.expression_sorter import File, Sort


class CLI:
    #* General

    def __init__(self):
        pass

    @staticmethod
    def print_header():
        print("*" * 60)
        print(f"* ST107 DSAA: Expression Evaluator & Sorter                *")
        print(f"*{'-' * 58}*")
        print(f"*{' ' * 58}*")
        print("* - Done by: Chuan Hao (1922261) & Sherisse Tan (1935967)  *")
        print("* - Class: DIT/2B/11                                       *")
        print("*" * 60)

    @staticmethod
    def print_selectionScreen():
        print("Please select your choice <'1', '2', '3'>:")
        print("\t 1. Evaluate expression")
        print("\t 2. Sort expressions")
        print("\t 3. Exit")

        return input("Enter choice: ")

    @staticmethod
    def print_exit():
        print("\nBye, thanks for using ST107 DSAA: Expression Evaluator & Sorter")

    @staticmethod
    def print_continue():
        input("\nPress any key to continue....\n")


    #* Expression Evaluator

    @staticmethod
    def print_inputExpression():
        return input("Please enter the expression you want to evaluate: \n")

    @staticmethod
    def print_traversalSelection():
        print("\nPlease select your choice <'1', '2', '3'>:")
        print("\t 1. Pre Order Tree Traversal")
```

```
61          print("\t 2. In Order Tree Traversal")
62          print("\t 3. Post Order Tree Traversal")
63
64          choice = -1
65          while choice not in ["1", "2", "3"]:
66              choice = input("Enter choice: ")
67
68          return choice
69
70      @staticmethod
71      def print_parseTree(traversalChoice, ast):
72          print("\nExpression Tree:")
73
74          if traversalChoice == "1":
75              preorder = PreOrder()
76              preorder.traverse(node = ast)
77
78          elif traversalChoice == "2":
79              inorder = InOrder()
80              inorder.traverse(node = ast)
81
82          elif traversalChoice == "3":
83              postorder = PostOrder()
84              postorder.traverse(node = ast)
85
86      @staticmethod
87      def print_evaluateResult(result):
88          print("\nExpression evaluates to:")
89          print(result)
90
91
92      #* Expression Sorter
93
94      @staticmethod
95      def get_files():
96          input_file = ""
97          output_file = ""
98
99          print("\nPlease enter your input and output files below..")
100         while not os.path.exists(input_file):
101             input_file = input("Please enter input file: ")
102
103         while not os.path.exists(output_file):
104             output_file = input("Please enter output file: ")
105
106         return (input_file, output_file)
107
108     @staticmethod
109     def get_sortSettings():
110         choice = -1
111
112         while choice not in ['1', '2']:
113             print("\nPlease enter your choice <'1', '2'>:")
114             print("\t 1. Sort by Ascending")
115             print("\t 2. Sort by Descending")
116
117             choice = input("Enter choice: ")
118
```

```
        if choice == '1':
            sort_order = "ascending"

        else:
            sort_order = "descending"

        return sort_order

    @staticmethod
    def print_sortResult(sortedList):
        print(">>> Evaluating and Sorting started:")

        for sublist in sortedList:
            value = sublist[0]
            print(f"\n*** Expressions with value = {value}")

            for expression in sublist[1]:
                print(f"{expression[0]} ==> {value}")

        print("\n>>> Evaluating and Sorting completed!")


    def run(self):
        CLI.print_header()
        done = False

        while not done:
            choice = CLI.print_selectionScreen()

            if choice == '1':
                expression_evaluated = False

                # Continue trying to get a valid expression input from the user as long as there was an error raised
                while not expression_evaluated:
                    try:
                        expression = CLI.print_inputExpression()
                        traversalChoice = CLI.print_traversalSelection()
                        ast, result = Evaluator.evaluate(expression)

                        expression_evaluated = True
                    except Exception as error:
                        print(error)
                        continue

                CLI.print_parseTree(traversalChoice, ast)
                CLI.print_evaluateResult(result)

                CLI.print_continue()

            elif choice == '2':
                valid_expressions = True

                input_file, output_file = CLI.get_files()
                sort_order = CLI.get_sortSettings()

                allExpressions = File.read(input_file)

```

```
                # Obtain the evaluated value for each expression in the list provided
                for expression in allExpressions:
                    try:
                        expression.append(Evaluator.evaluate(expression[0])[1])
                    except Exception as error:
                        print(f"There was an invalid expression in {input_file}.. The specific error is as follows:")
                        print(error, "\n")

                        valid_expressions = False
                        break

                if valid_expressions:
                    # Sort the expressions according to value
                    # sort = Sort(all_expr_list = allExpressions, sort_type = sort_type, sort_order = sort_order)
                    sort = Sort(all_expr_list = allExpressions, sort_order = sort_order)
                    sortedList = sort.sort()

                    CLI.print_sortResult(sortedList)
                    File.write(output_file, sortedList)

                    CLI.print_continue()

            elif choice == '3':
                CLI.print_exit()
                done = True
```

Listing 27: Source Code for: `./src/basic/__init__.py`

C.9 ./src/basic/expression_evaluator

C.9.1 ./src/basic/expression_evaluator/__init__.py

```
\subsubsection{\lstinline[language=Bash]{./src/basic/expression_evaluator/evaluator.py}}

\begin{lstlisting}[language=Python, caption={Source Code for: \lstinline{./src/basic/expression_evaluator/evaluator.py}}]
'''
This Python file deals with the evaluation of expressions and,
integrates the lexer, parser and interpreter together
'''

from .compiler import Lexer, Parser, Interpreter

class Evaluator:
    @staticmethod
    def evaluate(expression):
        if len(expression) <= 0:
            raise ValueError("\nInput Expression Length must be greater than 0\n")

        # Initialising the different components of the basic compiler
        lexer = Lexer(expression)
        parser = Parser(lexer)
        interpreter = Interpreter(parser)
```

```
21
22          # Obtaining the result
23          ast, result = interpreter.interpret()
24
25          if ast == None:
26              raise Exception("Error obtaining parse tree.. Please try again")
27
28          elif result == None:
29              raise Exception("Error obtaining evaluated expression value.. Please try
        again")
30
31          else:
32              return ast, result
```

Listing 28: Source Code for: `./src/basic/expression_evaluator/__init__.py`

C.10 ./src/basic/expression_evaluator/compiler

C.10.1 ./src/basic/expression_evaluator/compiler/parser.py

```
1  '''
2  This Python File contains the main 'Parser' class
3
4  #* Its main purpose is to parse the sequence of tokens from the lexer and check the
       grammer
5  #! At this stage, it is assumed that all characters in the input __expression is a
       valid token
6  '''
7
8  from ..tokens import Token
9  from ..tokens import INIT, EOF, WHITESPACE, OPERATOR, NUMBER, PLUS, MINUS, MUL, DIV,
       POWER, LPARAN, RPARAN, DOT
10 from ..nodes import Number_Node, BinaryOp_Node
11 from .lexer import Lexer
12
13
14 class Parser(object):
15     def __init__(self, lexer):
16         self.lexer = lexer
17         self.__all_tokens = lexer.get_all_tokens()
18
19         self.__token_index = 0
20         self.__current_token = self.__all_tokens[self.__token_index]
21
22     #* Getters ( No need for Setters because these attributes should only be altered
        within the class )
23     def get_all_tokens(self):
24         return self.__all_tokens
25
26     def get_token_index(self):
27         return self.__token_index
28
29     def get_current_token(self):
30         return self.__current_token
31
32
33     #* Utilities
34
```

```
    def __error(self, error_type):
        if error_type == "non-matching_token_types" or error_type == "internal_error
    ":
            raise SystemError("An internal __error has occurred in the parser..
    Please try again\n")

        elif error_type == "multiple_consecutive_operators":
            raise SyntaxError("There are multiple consecutive operators in your
    expression..\n")

        elif error_type == "multiple_consecutive_numbers":
            raise SyntaxError("There are multiple consecutive numbers in your
    expression with no operators between..\n")

        elif error_type == "term_error":
            raise SyntaxError("Multiple expressions detected.. Please try again\n")

        elif error_type == "factor_error":
            raise SystemError("An unexpected error has occurred in the Parser..
    Please check your NUMBER inputs\n")

        elif error_type == "incorrect_paranthesis" or error_type == "!EOF":
            raise SyntaxError("The expression provided is not a legal fully
    paranthesised expression\n")

        else:
            raise SystemError("An unexpected error has occurred in the Parser..
    Please try again\n")


    def __advance(self):
        #* Advance to the next character of the input __expression and,
        #* Update self.__current_token if not reached the end of the stream of
    tokens

        if self.__token_index < len(self.__all_tokens):
            self.__token_index += 1
            self.__current_token = self.__all_tokens[self.__token_index]

    def __peek(self):
        #* "Peek" into the next token of the stream of tokens and,
        #* return this token if it is not the end of the stream of tokens

        pos = self.__token_index + 1

        if pos < len(self.__all_tokens):
            return self.__all_tokens[pos]

        return None

    def __eat(self, token_type):
        #* Compare the current token type with the passed token type
        #* If they match, "__eat" the current token and __advance to next token
        #* Else, raise an Exception __error
        if self.__current_token.token_type == token_type:
            self.__advance()

        else:
```

```
            error_type = "non-matching_token_types"
            self.__error(error_type)


    #* Grammer

    def __expr(self):
        ''' LPARAN TERM ( (OPERATOR) TERM )* RPARAN '''

        # Performing a check for INIT token
        # If INIT Token is found, raise an __error
        # This should only occur if this is called separately from self.parse()
        if self.__current_token.token_type == INIT:
            error_type = "internal_error"
            self.__error(error_type)

        node = None
        left___term = self.__term()

        if self.__current_token.token_type in [PLUS, MINUS, MUL, DIV, POWER]:
            # Peek and make sure that the next token is not an OPERATOR
            # (with the exception of MINUS due to MINUS FACTOR)
            # If it is an operator, raise and __error
            if self.__peek().token_type in [PLUS, MUL, DIV, POWER]:
                error_type = "multiple_consecutive_operators"
                self.__error(error_type)

            node = self.__current_token

            if self.__current_token.token_type == PLUS:
                self.__eat(PLUS)
            elif self.__current_token.token_type == MINUS:
                self.__eat(MINUS)
            elif self.__current_token.token_type == MUL:
                self.__eat(MUL)
            elif self.__current_token.token_type == DIV:
                self.__eat(DIV)
            elif self.__current_token.token_type == POWER:
                self.__eat(POWER)

            right___term = self.__term()
            node = BinaryOp_Node(left___term, node, right___term)

            if self.__current_token.token_type != RPARAN:
                error_type = "incorrect_paranthesis"
                self.__error(error_type)

        elif self.__current_token.token_type != RPARAN:
            error_type = "incorrect_paranthesis"
            self.__error(error_type)

        if node == None:
            return left___term

        return node

    def __term(self):
        ''' FACTOR | EXPR '''
```

```
143
144         # EXPR
145         if self.__current_token.token_type == LPARAN:
146             self.__eat(LPARAN)
147 
148             node = self.__expr()
149             return node
150 
151         # FACTOR
152         elif self.__current_token.token_type == NUMBER or (self.__peek().token_type
    == NUMBER and self.__current_token.token_type == MINUS):
153             node = self.__factor()
154             return node
155 
156         else:
157             error_type = "term_error"
158             self.__error(error_type)
159 
160     def __factor(self):
161         ''' MINUS FACTOR | NUMBER '''
162         node = self.__current_token
163 
164         # NUMBER
165         if node.token_type == NUMBER:
166             #* Peek and make sure that the next token is not a NUMBER
167             #* If it is, raise an __error
168             if self.__peek().token_type == NUMBER:
169                 error_type = "multiple_consecutive_numbers"
170                 self.__error(error_type)
171 
172             self.__eat(NUMBER)
173             return Number_Node(node)
174 
175         # MINUS FACTOR
176         if node.token_type == MINUS:
177             self.__eat(MINUS)
178 
179             self.__current_token.token_value *= -1
180             node = self.__current_token
181 
182             self.__eat(NUMBER)
183             return Number_Node(node)
184 
185         error_type = "factor_error"
186         self.__error(error_type)
187 
188 
189     #* Main
190 
191     def parse(self):
192         #* Parse the stream of tokens, checking the grammer
193 
194         # Perform a check of the Paranthesis count before starting any parsing
195         left_paran_count = right_paran_count = 0
196         for token in self.__all_tokens:
197             if token.token_type == LPARAN:
198                 left_paran_count += 1
199             if token.token_type == RPARAN:
```

```
                right_paran_count += 1

        # Basics of fully-paranthesised __expressions -> same count of left and right paranthesis && even number of paranthesis overall
        if left_paran_count != right_paran_count or (left_paran_count + right_paran_count) % 2 != 0 or left_paran_count < 1 or right_paran_count < 1:
            error_type = "incorrect_paranthesis"
            self.__error(error_type)


        # If this point is reached, the paranthesis check has passed
        # As such, "eat" the INIT token and start parsing
        self.__eat(INIT)
        ast = self.__expr()

        return ast
```

Listing 29: Source Code for: `./src/basic/expression_evaluator/compiler/parser.py`

C.10.2 `./src/basic/expression_evaluator/compiler/__init__.py`

```

from .lexer import Lexer
from .parser import Parser
from .interpreter import Interpreter
```

Listing 30: Source Code for: `./src/basic/expression_evaluator/compiler/__init__.py`

C.10.3 `./src/basic/expression_evaluator/compiler/lexer.py`

```

'''
## This is the Python File containing the main 'Lexer' class

#* The main purpose is to tokenize all the characters in the input expression provided by the user
#! Take note that there will be no checking of the syntax of the input expression in this class
'''

from ..tokens import Token
from ..tokens import INIT, EOF, WHITESPACE, OPERATOR, NUMBER, PLUS, MINUS, MUL, DIV, POWER, LPARAN, RPARAN, DOT

class Lexer(object):
    def __init__(self, text):
        self.__text = text.strip()

        # Indexing input text
        self.__pos = 0
        self.__current_char = self.__text[self.__pos]

        # Tokenizing
        self.__current_token_type = INIT
        self.__current_token_value = None

    #* Getters ( No need for Setters because these attributes should only be altered within the class )
```

```
    def get_text(self):
        return self.__text

    def get_pos(self):
        return self.__pos

    def get_current_char(self):
        return self.__current_char

    def get_current_token_type(self):
        return self.__current_token_type

    def get_current_token_value(self):
        return self.__current_token_value


    def __error(self, error_type, character = None):
        if error_type == "unrecognised_token_type":
            raise Exception(f"Lexical Error: Invalid character(s) detected\n")

        elif error_type == "unrecognised_operator":
            raise Exception(f"Lexical Error: Support for the {character} operator has not yet been implemented\n")

        elif error_type == "invalid_float":
            raise Exception(f"Lexical Error: Invalid float / integer value(s)\n")

        else:
            raise SystemError("An unexpected error has occurred in the Lexer.. Please try again\n")


    def __check_token_type(self, char):
        #* Checks and returns the token_type of the character passed in

        if char == None:
            return EOF

        if char.isspace():
            return WHITESPACE

        if char.isdigit():
            return NUMBER

        if char == "(":
            return LPARAN

        if char == ")":
            return RPARAN

        if char in ["+", "-", "*", "/"]:
            return OPERATOR

        if char == ".":
            return DOT

        #! The character passed in does not match any token type, raise an __error
        error_type = "unrecognised_token_type"
```

```
81          self.__error(error_type)
82
83      def __peek(self):
84          #* "Peek" into the next character of the input expression and,
85          #* return this character if it is not the end of the input expression
86
87          pos = self.__pos + 1
88
89          if pos >= len(self.__text):
90              return None
91          return self.__text[pos]
92
93      def __advance(self):
94          #* Advance to the next character of the input expression and,
95          #* Update self.__current_char if not reached the end of the input expression
96
97          self.__pos += 1
98
99          if self.__pos >= len(self.__text):
100             self.__current_char = None
101         else:
102             self.__current_char = self.__text[self.__pos]
103
104     def __differentiate_between_operators(self):
105         #* Check and differentiate between the different types of accepted operators
    :
106         #* PLUS, MINUS, MUL, DIV, POWER
107         #* Then, update the current_token_type to represent the actual operator
108
109         if self.__current_token_value == "*":
110             if self.__peek() == "*":
111                 self.__advance()
112                 self.__current_token_value += self.__current_char
113                 self.__current_token_type = POWER
114             else:
115                 self.__current_token_type = MUL
116
117         elif self.__current_token_value == "+":
118             self.__current_token_type = PLUS
119
120         elif self.__current_token_value == "-":
121             self.__current_token_type = MINUS
122
123         elif self.__current_token_value == "/":
124             self.__current_token_type = DIV
125
126         #! Theorectically, this should never occur as a check is made previously to
127         #! determine if self.current_token is an OPERATOR token type.
128         #! However, we are still checking just in case
129         else:
130             error_type = "unrecognised_operator"
131             self.__error(error_type, self.__current_token_value)
132
133     def __get_number(self):
134         #* Get the entirety of the NUMBER value, whether it is a Float or an Integer
135
136         number_value = self.__current_char
137
```

```
        # Normal Integer
        while self.__check_token_type(self.__peek()) == NUMBER:
            self.__advance()
            number_value += self.__current_char

        # Float Number
        if self.__check_token_type(self.__peek()) == DOT:
            number_value += "."
            self.__advance()

            #! If the next character after "." is not a number,
            #! Raise an __error
            if self.__check_token_type(self.__peek()) != NUMBER:
                error_type = "invalid_float"
                self.__error(error_type)

            while self.__check_token_type(self.__peek()) == NUMBER:
                self.__advance()
                number_value += self.__current_char

                if self.__check_token_type(self.__peek()) == DOT:
                    error_type = "invalid_float"
                    self.__error(error_type)

        self.__current_token_value = float(number_value)
        self.__advance()

    def __get_next_token(self):
        #* Gets and returns the next token of the input string
        #* If the next token is a WHITESPACE, continue advancing to the next non-
    WHITESPACE token

        # EOF
        if self.__current_char == None:
            return Token(EOF, None)

        self.__current_token_value = ""
        self.__current_token_value += self.__current_char
        self.__current_token_type = self.__check_token_type(self.__current_char)

        # WHITESPACE
        if self.__current_token_type == WHITESPACE:
            while self.__check_token_type(self.__peek()) == WHITESPACE:
                self.__advance()

            self.__advance()
            return self.__get_next_token()

        # OPERATORS
        if self.__current_token_type == OPERATOR:
            self.__differentiate_between_operators()
            self.__advance()
            return Token(self.__current_token_type, self.__current_token_value)

        # PARANTHESIS
        elif self.__current_token_type == LPARAN or self.__current_token_type ==
    RPARAN:
            self.__advance()
```

```
            return Token(self.__current_token_type, self.__current_token_value)

        # NUMBER
        elif self.__current_token_type == DOT or self.__current_token_type == NUMBER
    :
            self.__get_number()
            return Token(NUMBER, self.__current_token_value)

        #! If this point is reached,
        #! The next char does not have a recognised token type
        error_type = "unrecognised_token_type"
        self.__error(error_type)

    def get_all_tokens(self):
        #* Continuously calls get_next_token() until the entire input expression has
     been transformed into tokens

        # The starting token is always an INIT
        init_token = Token(INIT, None)
        tokens = [init_token]

        while True:
            current_token = self.__get_next_token()
            tokens.append(current_token)

            # The last token is always an EOF
            if current_token.token_type == EOF:
                break

        return tokens
```

Listing 31: Source Code for: ./src/basic/expression_evaluator/compiler/lexer.py

C.10.4 ./src/basic/expression_evaluator/compiler/interpreter.py

```

from ..tokens import PLUS, MINUS, MUL, DIV, POWER
from ..nodes import NodeVisitor
from .lexer import Lexer
from .parser import Parser

class Interpreter(NodeVisitor):
    def __error(self):
        raise Exception("Error interpreting expression.. Please try again")

    def visit_Number_Node(self, node):
        return node.token_value

    def visit_BinaryOp_Node(self, node):
        token_type = node.operator.token_type

        if token_type == PLUS:
            left_term = self.visit(node.left_term)
            right_term = self.visit(node.right_term)

            return left_term + right_term

        if token_type == MINUS:
```

```
            left_term = self.visit(node.left_term)
            right_term = self.visit(node.right_term)

            return left_term - right_term

        if token_type == MUL:
            left_term = self.visit(node.left_term)
            right_term = self.visit(node.right_term)

            return left_term * right_term

        if token_type == DIV:
            left_term = self.visit(node.left_term)
            right_term = self.visit(node.right_term)

            return left_term / right_term

        if token_type == POWER:
            left_term = self.visit(node.left_term)
            right_term = self.visit(node.right_term)

            return left_term ** right_term

        self.__error()

    def interpret(self):
        ast = self.parser.parse()

        if ast == None:
            return (None, None)

        return (ast, self.visit(ast))
```

Listing 32: Source Code for: `./src/basic/expression_evaluator/compiler/interpreter.py`

C.11 ./src/basic/expression_evaluator/compiler/node_traversal

C.11.1 ./src/basic/expression_evaluator/compiler/node_traversal/__init__.py

```

from .pre_order import PreOrder
from .in_order import InOrder
from .post_order import PostOrder
```

Listing 33: Source Code for: `./src/basic/expression_evaluator/compiler/node_traversal/__init__.py`

C.11.2 ./src/basic/expression_evaluator/compiler/node_traversal/in_order.py

```
'''
    This Python File deals with printing the expression in InOrder

    InOrder:
        left - root - right
'''

from ...nodes import NodeVisitor

class InOrder(NodeVisitor):
```

```
    def __init__(self, level = 0):
        self.__level = level

    #* Getter and Setter
    def get_level(self):
        return self.__level

    def set_level(self, level):
        self.__level = level


    def traverse(self, node):
        if type(node).__name__ == "BinaryOp_Node":
            # Left
            self.__level += 1
            self.traverse(node.left_term)

            # Root
            self.__level -= 1
            print(str("~" * self.__level) + " " + node.operator.token_value)

            # Right
            self.__level += 1
            self.traverse(node.right_term)

        elif type(node).__name__ == "Number_Node":
            print(str("~" * self.__level) + " " + str(node.token_value))
            return

        else:
            self.visit_error(node)
```

Listing 34: Source Code for: ./src/basic/expression_evaluator/compiler/node_traversal/in_order.py

C.11.3 ./src/basic/expression_evaluator/compiler/node_traversal/post_order.py

```
'''
    This Python File deals with printing the expression in InOrder

    PostOrder:
        left - right - root
'''

from ...nodes import NodeVisitor

class PostOrder(NodeVisitor):
    def __init__(self, level = 0):
        self.__level = level

    #* Getter and Setter
    def get_level(self):
        return self.__level

    def set_level(self, level):
        self.__level = level


    def traverse(self, node):
```

```
        if type(node).__name__ == "BinaryOp_Node":
            self.__level += 1

            # Left
            self.traverse(node.left_term)

            # Right
            self.traverse(node.right_term)
            self.__level -= 1

            # Root
            print(str("~" * self.__level) + " " + node.operator.token_value)

        elif type(node).__name__ == "Number_Node":
            print(str("~" * self.__level) + " " + str(node.token_value))
            return

        else:
            self.visit_error(node)
```

Listing 35: Source Code for: ./src/basic/expression_evaluator/compiler/node_traversal/post_order.py

C.11.4 ./src/basic/expression_evaluator/compiler/node_traversal/pre_order.py

```
'''
    This Python File deals with printing the expression in PreOrder

    PreOrder:
        root - left - right
'''

from ...nodes import NodeVisitor

class PreOrder(NodeVisitor):
    def __init__(self, level = 0):
        self.__level = level

    #* Getter and Setter
    def get_level(self):
        return self.__level

    def set_level(self, level):
        self.__level = level


    def traverse(self, node):
        if type(node).__name__ == "BinaryOp_Node":
            # Root
            print(str("~" * self.__level) + " " + node.operator.token_value)
            self.__level += 1

            # Left
            self.traverse(node.left_term)

            # Right
            self.traverse(node.right_term)
            self.__level -= 1

```

```
        elif type(node).__name__ == "Number_Node":
            print(str("~" * self.__level) + " " + str(node.token_value))
            return

        else:
            self.visit_error(node)
```

Listing 36: Source Code for: ./src/basic/expression_evaluator/compiler/node_traversal/pre_order.py

C.12 ./src/basic/expression_evaluator/nodes

C.12.1 ./src/basic/expression_evaluator/nodes/__init__.py

```

from .ast import AST
from .binary_op_node import BinaryOp_Node
from .number_node import Number_Node
from .node_visitor import NodeVisitor
```

Listing 37: Source Code for: ./src/basic/expression_evaluator/nodes/__init__.py

C.12.2 ./src/basic/expression_evaluator/nodes/number_node.py

```

'''
This is the Python File containing the Child class, 'Number_Node' that inherits from
    AST
'''
from .ast import AST

class Number_Node(AST):
    def __init__(self, token):
        self.token = token
        self.token_value = token.token_value

    def __str__(self):
        return f"{self.token}"
```

Listing 38: Source Code for: ./src/basic/expression_evaluator/nodes/number_node.py

C.12.3 ./src/basic/expression_evaluator/nodes/binary_op_node.py

```

'''
This is the Python File containing the Child class, 'BinaryOp_Node' that inherits
    from AST
'''
from .ast import AST

class BinaryOp_Node(AST):
    def __init__(self, left_term, operator, right_term):
        self.left_term = left_term
        self.operator = operator
        self.right_term = right_term

    def __str__(self):
        return f"{self.left_term}, {self.operator}, {self.right_term}"
```

Listing 39: Source Code for: ./src/basic/expression_evaluator/nodes/binary_op_node.py

C.12.4 ./src/basic/expression_evaluator/nodes/ast.py

```
'''
This is the Python file containing the Parent Class 'AST',
which will be inherited by the binary_op_node and the number_node
'''

class AST:
    def __init__(self):
        pass
```

Listing 40: Source Code for: ./src/basic/expression_evaluator/nodes/ast.py

C.12.5 ./src/basic/expression_evaluator/nodes/node_visitor.py

```
'''
This is the Python File containing the NodeVisitor class used for the Interpreter
'''

class NodeVisitor(object):
    def __init__(self, parser):
        self.parser = parser

    def visit_error(self, node):
        node_name = type(node).__name__
        return f"Interpreter Error: visit_{node_name} has not been implemented yet..
    "

    def visit(self, node):
        node_name = type(node).__name__
        method_name = f"visit_{node_name}"

        visit_method = getattr(self, method_name, self.visit_error)
        return visit_method(node)
```

Listing 41: Source Code for: ./src/basic/expression_evaluator/nodes/node_visitor.py

C.13 ./src/basic/expression_evaluator/tokens

C.13.1 ./src/basic/expression_evaluator/tokens/__init__.py

```
from .token import Token
from .token_type import INIT, EOF, WHITESPACE, OPERATOR, NUMBER, PLUS, MINUS, MUL,
    DIV, POWER, LPARAN, RPARAN, DOT
```

Listing 42: Source Code for: ./src/basic/expression_evaluator/tokens/__init__.py

C.13.2 ./src/basic/expression_evaluator/tokens/token.py

```
'''
This Python file contains the 'Token' class

Each individual token should have a token_type and a token_value
'''
```

```

class Token:
    def __init__(self, token_type, token_value):
        self.token_type = token_type
        self.token_value = token_value

    def __str__(self):
        return f"TOKEN({self.token_type}, {self.token_value})"
```

Listing 43: Source Code for: ./src/basic/expression_evaluator/tokens/token.py

C.13.3 ./src/basic/expression_evaluator/tokens/token_type.py

```
'''
This Python file contains the various token_types implemented for the basic feature

Tokens implemented:
    INIT, EOF, WHITESPACE
    OPERATOR, NUMBER
    PLUS, MINUS, MUL, DIV, POWER
    LPARAN, RPARAN, DOT
'''

INIT, EOF, WHITESPACE = "INIT", "EOF", "WHITESPACE"
OPERATOR, NUMBER = "OPERATOR", "NUMBER"
PLUS, MINUS, MUL, DIV, POWER = "PLUS", "MINUS", "MUL", "DIV", "POWER"
LPARAN, RPARAN, DOT = "LPARAN", "RPARAN", "DOT"
```

Listing 44: Source Code for: ./src/basic/expression_evaluator/tokens/token_type.py

C.14 ./src/common/common_algos

C.14.1 ./src/common/common_algos/__init__.py

```
from .common_algos import bin_to_decimal, hex_to_decimal
```

Listing 45: Source Code for: ./src/common/common_algos/__init__.py

C.14.2 ./src/common/common_algos/common_algos.py

```
'''Python file for common algorithms

Mainly used to store algorithms written to make usage more convenient
'''

def bin_to_decimal(s: str):
    '''
    Helper function that takes in a binary string and returns the converted decimal
    number
    '''
    for c in s:
        assert c in set(['.', '1', '0'])
    decimal_value = 0
    frac = False
    frac_power = 1
    for c in s:
```

```
        if c == '.':
            frac = True
            continue
        else:
            parsed_c = int(c)
        if not frac:
            decimal_value *= 2
            decimal_value += parsed_c
        else:
            decimal_value += parsed_c * (2**(-frac_power))
            frac_power += 1
    return decimal_value

def hex_to_decimal(s: str):
    '''
    Helper function that takes in a hexadecimal string and returns the converted decimal number
    '''
    a_ord = ord('A')
    for c in s:
        assert c in set(['.'] + [str(num) for num in range(10)] + [chr(a_ord + offset) for offset in range(6)])
    decimal_value = 0
    frac = False
    frac_power = 1
    for c in s:
        if c == '.':
            frac = True
            continue
        else:
            if c in set([str(num) for num in range(10)]):
                parsed_c = int(c)
            elif c in set([chr(a_ord + offset) for offset in range(6)]):
                parsed_c = 10 + ord(c) - a_ord
        if not frac:
            decimal_value *= 16
            decimal_value += parsed_c
        else:
            decimal_value += parsed_c * (16**(-frac_power))
            frac_power += 1
    return decimal_value


if __name__ == '__main__':
    print(bin_to_decimal('1101'))
    print(bin_to_decimal('1101.1101'))

    print(hex_to_decimal('91ABF.FFF'))
```

Listing 46: Source Code for: `./src/common/common_algos/common_algos.py`

C.15 `./src/common/expression_sorter`

C.15.1 `./src/common/expression_sorter/__init__.py`

```

from .file import File
```

```
from .sort import Sort
```

Listing 47: Source Code for: ./src/common/expression_sorter/__init__.py

C.15.2 ./src/common/expression_sorter/file.py

```

'''
This Python File will deal with the File I/O required for the
Expression Sorting Section of the application.
'''
from pathlib import Path
class File:
    @staticmethod
    def read(filename):
        file_input = []

        # Reading file input
        f = open(filename, 'r')
        for line in f:
            file_input.append([line.strip()])
        f.close()

        return file_input

    @staticmethod
    def write(filename, sortedList):
        f = open(filename, 'w+')
        for sublist in sortedList:
            value = sublist[0]
            f.write(f"*** Expressions with value = {value}\n")

            for expression in sublist[1]:
                #print(type(sublist[i]))
                f.write(f"{expression[0]} ==> {value}\n")

            f.write("\n")

        f.close()
```

Listing 48: Source Code for: ./src/common/expression_sorter/file.py

C.15.3 ./src/common/expression_sorter/sort.py

```
'''
This is the Python File containing the Sort class that
deals with sorting the various expressions from the input file

#* Features: Sort by Value, Length, Digit Order   ( Type )
#* Features: Sort by Ascending / Descending       ( Order )
'''

from .file import File

class Sort:
    def __init__(self, all_expr_list = None, sort_type = "value", sort_order = "
    ascending"):
        self.__all_expr_list = all_expr_list
```

```
        self.__sort_type = sort_type
        self.__sort_order = sort_order

    def error(self, error_type = None):
        if error_type == "invalid_sort_type":
            raise ValueError("Invalid Sorting Type.. Only 'Value' or 'Length' accepted")
        elif error_type == "invalid_sort_order":
            raise ValueError("Invalid Sorting Order.. Only 'ascending' or 'descending' accepted")
        else:
            raise Exception("Error occurred while sorting..")


    #* Getter and Setter

    def get_all_expr_list(self):
        return self.__all_expr_list

    def set_all_expr_list(self, expr_list):
        self.__all_expr_list = expr_list

    def get_sort_type(self):
        return self.__sort_type

    def set_sort_type(self, sort_type):
        if sort_type not in ["value", "length"]:
            error_type = "invalid_sort_type"
            self.error(error_type)

        self.__sort_type = sort_type

    def get_sort_order(self):
        return self.__sort_order

    def set_sort_order(self, sort_order):
        if sort_order not in ["ascending", "descending"]:
            error_type = "invalid_sort_order"
            self.error(error_type)

        self.__sort_order = sort_order

    #* 'Preprocessing' expression - get length of expression, remove whitespaces
    def preprocess_expr(self):
        all_expressions = self.get_all_expr_list()

        for expression in all_expressions:
            # Removing whitespaces
            expression[0] = expression[0].replace(" ", "")

            # Appending length of expression
            expression.append(len(str(expression[0])))

        return all_expressions

    #* Compile the sorted list into sublists based on value
    def compile_list_by_value(self, sorted_exprList):
        value_list = [expression[1] for expression in sorted_exprList]
```

```
70          expression_list = [expression for expression in sorted_exprList]
71
72          unique_value_list = []
73          for value in value_list:
74              if value not in unique_value_list:
75                  unique_value_list.append(value)
76
77          compiledList = []
78
79          for i in range(0, len(unique_value_list)):
80              value = unique_value_list[i]
81
82              compiledList.append([value])
83              compiledList[i].append([expression for expression in expression_list if
    expression[1] == value])
84
85          return compiledList
86
87
88      #* Sorting
89
90      # 'Middleman' for mergeSort() method
91      def sort(self):
92          all_expressions = self.preprocess_expr()
93
94          sortedList = self.mergeSort(all_expressions)
95          sortedList = self.compile_list_by_value(sortedList)
96
97          for sublist in sortedList:
98              if len(sublist[1]) > 1:
99                  self.set_sort_type("length")
100                 sublist = self.mergeSort(sublist[1])
101
102         return sortedList
103
104
105     # Merge Sort WHEEEEEEEEE
106     def mergeSort(self, expr_list):
107         sort_order = self.get_sort_order()
108         sort_type = self.get_sort_type()
109
110         if sort_type == "value":
111             list_index = 1
112         elif sort_type == "length":
113             list_index = 2
114         else:
115             error_type = "invalid_sort_type"
116             self.error(error_type)
117
118         if len(expr_list) > 1:
119             #* Dividing the expr_list
120
121             middleIndex = int(len(expr_list) / 2)
122
123             # Splitting into left and right halves
124             left_half = expr_list[:middleIndex]
125             right_half = expr_list[middleIndex:]
126
```

```
            # Recursive call to continuously split the list into two halves
            self.mergeSort(left_half)
            self.mergeSort(right_half)

            left_index = right_index = merge_index = 0
            merge_list = expr_list


            #* Sorting && Merging

            while left_index < len(left_half) and right_index < len(right_half):
                if sort_order == "ascending":
                    if left_half[left_index][list_index] < right_half[right_index][
list_index]:
                        merge_list[merge_index] = left_half[left_index]
                        left_index += 1

                    else:
                        merge_list[merge_index] = right_half[right_index]
                        right_index += 1

                elif sort_order == "descending":
                    if left_half[left_index][list_index] > right_half[right_index][
list_index]:
                        merge_list[merge_index] = left_half[left_index]
                        left_index += 1

                    else:
                        merge_list[merge_index] = right_half[right_index]
                        right_index += 1

                else:
                    error_type = "invalid_sort_order"
                    self.error(error_type)

                merge_index += 1

            # Handling any items still left in the left half of the list
            while left_index < len(left_half):
                merge_list[merge_index] = left_half[left_index]

                left_index += 1
                merge_index += 1

            # Handling any items still left in the right half of the list
            while right_index < len(right_half):
                merge_list[merge_index] = right_half[right_index]

                right_index += 1
                merge_index += 1

        return expr_list
```

Listing 49: Source Code for: `./src/common/expression_sorter/sort.py`