**School of Computing (SoC)**

**Diploma in Applied AI and Analytics**

**AY 2021/2022 Semester 2**

**ST1507 Data Structures and Algorithms**

**Assignment Two**

**Expression Evaluator and Sorter (using Parse Trees)**

|  |  |
| --- | --- |
| Class: | DAAA/FT/2B/03 |
| Lecturer: | Ms Hwee Shan Tay |
| Members: | Ethan Tan (P2012085)  Reshma (P2011972) |

|  |  |
| --- | --- |
|  |  |
|  |  |
|  |  |
|  |

### TABLE OF CONTENTS

[TABLE OF CONTENTS 2](#_Toc94794218)

[Description 5](#_Toc94794219)

[User Guidelines 5](#_Toc94794220)

[Start up 5](#_Toc94794221)

[Option 1: Evaluate a Fully Parenthesised Expression 6](#_Toc94794222)

[Option 3: Evaluate and Sort Expressions 6](#_Toc94794223)

[Option 7: Exit / Quit 6](#_Toc94794224)

[Implementation: Overview of Classes 7](#_Toc94794225)

[Node (inherits ABC) 7](#_Toc94794226)

[TempNode (inherits Node) 7](#_Toc94794227)

[MathNode (inherits Node) 7](#_Toc94794228)

[Operand (inherits MathNode) 7](#_Toc94794229)

[Operator (inherits MathNode) 7](#_Toc94794230)

[PrintOrientation (inherits Enum) 7](#_Toc94794231)

[TreeTraversalOrder (inherits Enum) 7](#_Toc94794232)

[Tokenizer 8](#_Toc94794233)

[Lexer 8](#_Toc94794234)

[Tree 8](#_Toc94794235)

[ParseTree (inherits Tree) 8](#_Toc94794236)

[Expression 8](#_Toc94794237)

[InvalidOptionError (inherits Exception) 8](#_Toc94794238)

[InvalidExpressionError (inherits SyntaxError) 8](#_Toc94794239)

[Implementation: Complexity Analysis 9](#_Toc94794240)

[Tokenization 9](#_Toc94794241)

[Lexical Analysis 9](#_Toc94794242)

[Parse Tree Insertion 9](#_Toc94794243)

[Parse Tree Construction 9](#_Toc94794244)

[Parse Tree Evaluation 9](#_Toc94794245)

[Parse Tree Printing (DFS) 9](#_Toc94794246)

[Parse Tree Printing (BFS) 9](#_Toc94794247)

[Merge Sort 9](#_Toc94794248)

[Implementation: Design 10](#_Toc94794249)

[Merge Sort 10](#_Toc94794250)

[Expression 10](#_Toc94794251)

[Operator 10](#_Toc94794252)

[PrintOrientation / TreeTraversalOrder 10](#_Toc94794253)

[Tree / Node 10](#_Toc94794254)

[Parse Tree 10](#_Toc94794255)

[Implementation: OOP Concepts 11](#_Toc94794256)

[Abstraction 11](#_Toc94794257)

[Inheritance 11](#_Toc94794258)

[Polymorphism 11](#_Toc94794259)

[Overloading 11](#_Toc94794260)

[Overriding 11](#_Toc94794261)

[Encapsulation 11](#_Toc94794262)

[Implementation: Inheritance Scheme 12](#_Toc94794263)

[Implementation: Summary of Data Structures Used 12](#_Toc94794264)

[Challenges faced 12](#_Toc94794265)

[Technical 12](#_Toc94794266)

[Group Work 12](#_Toc94794267)

[Key takeaways / Learning achievements 13](#_Toc94794268)

[Roles & Contributions 13](#_Toc94794269)

[Roles 13](#_Toc94794270)

[Contributions 13](#_Toc94794271)

[Appendix 14](#_Toc94794272)

[src/utils/io\_utils.py 14](#_Toc94794273)

[src/utils/mergesort.py 15](#_Toc94794274)

[src/\_\_init\_\_.py 16](#_Toc94794275)

[src/exceptions.py 16](#_Toc94794276)

[src/expression.py 17](#_Toc94794277)

[src/lexer.py 18](#_Toc94794278)

[src/math\_node.py 19](#_Toc94794279)

[src/node.py 20](#_Toc94794280)

[src/operand\_.py 22](#_Toc94794281)

[src/operator\_.py 23](#_Toc94794282)

[src/parse\_tree.py 24](#_Toc94794283)

[src/print\_orientation.py 31](#_Toc94794284)

[src/temp\_node.py 32](#_Toc94794285)

[src/tokenizer.py 32](#_Toc94794286)

[src/tree\_traversal\_order.py 33](#_Toc94794287)

[src/tree.py 34](#_Toc94794288)

[main.py 38](#_Toc94794289)

[References 41](#_Toc94794290)

### Description

This report entails the design, implementation, analysis and use of a program intended to evaluate simple arithmetic expressions.

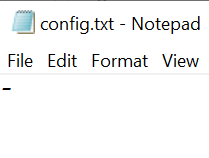
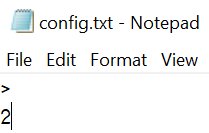
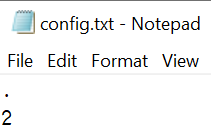
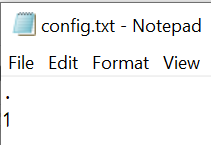
The program supports the following functions:

1. Evaluate and display the parse tree of a fully parenthesised valid arithmetic expression
2. Evaluate and display the parse tree of any valid arithmetic expression
3. Read, evaluate and sort arithmetic expressions from a specified text file and save the report to a specified output file
4. Fully parenthesize a valid arithmetic expression
5. Change the display mode for options 1 and 2
6. Register new operators to be used
7. Exit the program

### User Guidelines

**Configuration:**

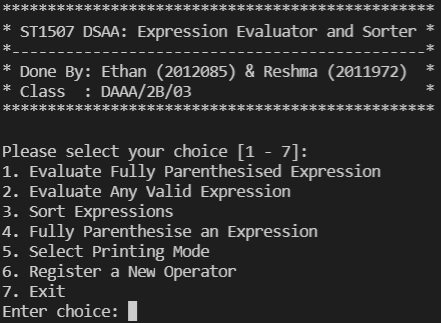
Users can select which separator and operator implementation group they want to use, through a configuration file, *config.txt*.



An alternative path can be specified through the command line.



The user can choose any valid single character for the separator (including space character) and can only choose from option 1 (Operator Group One) or 2 (Operator Group Two) for the operator implementation group.

The default options are ‘.’ and 1 for the separator and operator implementation group respectively. These defaults apply if the configuration file cannot be found or if any option is omitted. The application loads the configuration(s) specified, and it cannot be changed afterwards.

#### Start up

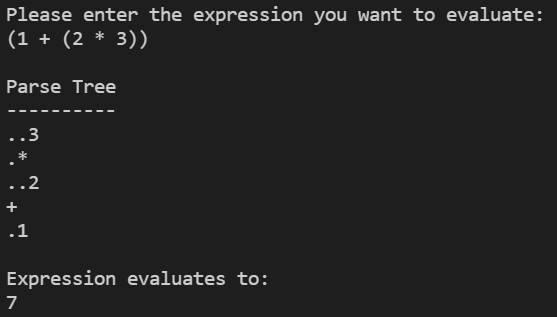
Upon start up, the program greets the user with a menu, displaying all the functions it supports.

The program will ignore invalid selections.

#### Option 1: Evaluate a Fully Parenthesised Expression

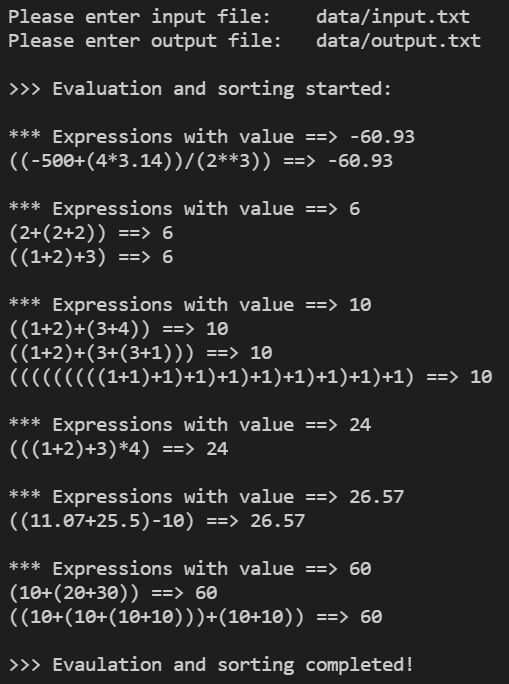
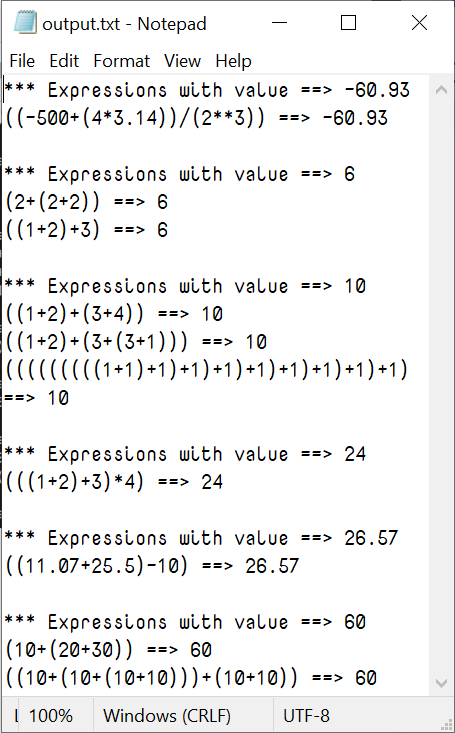
This option accepts a fully parenthesized valid arithmetic expression. It displays the parse tree used to evaluate the expression, along with the result the expression evaluates to.

The separator and base operator implementation group are the ones specified during configuration.



The orientation and traversal order used for printing can be configured in Option 5.

#### Option 3: Evaluate and Sort Expressions

In this option, the user will be prompted to enter an input and output file. Subsequently, the program will read the input file, evaluate and sort the expressions in the input file.

The sorting sorts the value of the expressions in ascending order, followed by their lengths in ascending order. If expressions have the same value and length, they will be sorted by the number of brackets in descending order.

The output will be displayed on the screen as well as written to the output file specified by the user.

The report will not be saved if no output file is specified.

#### Option 7: Exit / Quit

This option terminates the program and exits, while bidding a friendly farewell to the user :D



### Implementation: Overview of Classes

#### Node (inherits ABC)

Description: Represents a single node data structure, with pointers to its parent and children

Purpose: Abstract base class for nodes

Implements: display, update\_widths, \_\_str\_\_ and \_\_repr\_\_ methods, used for vertical printing

Challenges: Pointers are meant to be easily manipulated by Tree and ParseTree classes

Properties are indicated to be for internal use through single underscores

#### TempNode (inherits Node)

Description: Represents empty space useful for padding

Purpose: Provides padding for vertical printing

Implements: display (polymorphism) method, returns whitespace characters

#### MathNode (inherits Node)

Description: Represents a node in a parse tree, either an operand or operator

Purpose: Abstract base class for Operand and Operator subclasses

Implements: \_\_call\_\_ and \_\_str\_\_ (polymorphism) methods, used for evaluation and printing

Asserts: \_\_lt\_\_ and \_\_gt\_\_ (polymorphism) methods, used for parse tree construction

#### Operand (inherits MathNode)

Description: Represents a node in a parse tree, whose value is an operand

Purpose: Wrapper class for operands

Implements: \_\_lt\_\_, \_\_gt\_\_, get\_priority, augment\_priority and copy methods,

useful for parse tree construction

#### Operator (inherits MathNode)

Description: Represents a node in a parse tree, whose value is an operator

Purpose: Wrapper class for operators

Implements: \_\_lt\_\_, \_\_gt\_\_, get\_priority, augment\_priority and copy methods,

useful for parse tree construction

#### PrintOrientation (inherits Enum)

Description: Represents the collection of all valid print orientation options

Purpose: Ensures user-selected print orientations are valid

Members: HORIZONTAL, VERTICAL

#### TreeTraversalOrder (inherits Enum)

Description: Represents the collection of all valid tree traversal options

Purpose: Ensures user-selected tree traversals are valid

Members: IN\_ORDER, PRE\_ORDER, POST\_ORDER

#### Tokenizer

Description: Represents a tokenizer object useful for tokenizing an expression

Purpose: Extract tokens from a string of characters

Implements: \_\_combine and tokenize methods

#### Lexer

Description: Represents a lexer object useful for performing lexical analysis

Purpose: Creates a list of token objects from a list of string tokens

Implements: \_\_lex\_token and lex methods

#### Tree

Description: Represents a tree data structure

Purpose: Base class for a tree

Implements: print\_tree and change\_print\_mode methods, with other utility methods, for

different ways of printing the tree

Encapsulates: \_root and \_currentPointer, protected properties

\_\_depth\_symbol, \_\_print\_orientation & \_\_print\_traversal\_order, private properties

#### ParseTree (inherits Tree)

Description: Represents a parse tree data structure

Purpose: Implement useful methods for parsing and evaluating arithmetic expressions

Implements: read, parse, build and \_\_prepare methods for tree construction

\_\_validate\_parse\_tree and \_\_validate\_fully\_parenthesised methods for validation

print\_tree, evaluate, evaluate\_and\_sort, reconstruct\_expression and

register\_new\_operator methods

#### Expression

Description: Represents a single arithmetic expression

Purpose: Wrapper class for an expression and its result

Implements: \_\_lt\_\_, \_\_le\_\_, \_\_gt\_\_, \_\_ge\_\_ and comparison\_tuple methods for comparisons

get\_result, \_\_str\_\_ and \_\_repr\_\_ methods for displaying

#### InvalidOptionError (inherits Exception)

Description: Represents an error whenever a user selects an invalid option

Purpose: Debugging

#### InvalidExpressionError (inherits SyntaxError)

Description: Represents an error whenever a user enters an invalid expression

Purpose: Debugging

Default: ‘Invalid Expression’

### Implementation: Complexity Analysis

#### Tokenization

Space Complexity: O(t) (space required to store tokens is proportional to number of tokens)

Time Complexity: O(t) [Ideal] (ideally, the regex matching process should be linear)

Legend: t = total number of tokens

#### Lexical Analysis

Space Complexity: O(t) (space required to store objects is proportional to number of tokens)

Time Complexity: O(t) (iterates over the list of tokens once)

Legend: t = total number of tokens

#### Parse Tree Insertion

Space Complexity: O(n) (extra space required is proportional to the number of nodes added)

Time Complexity: O(n) [Amortised worst case] (amortised constant time to add each node)

Legend: n = total number of nodes inserted

#### Parse Tree Construction

Space Complexity: O(t) (all subprocesses have linear space complexity)

Time Complexity: O(t) (all subprocesses have linear time complexity)

Legend: t = total number of tokens

#### Parse Tree Evaluation

Space Complexity: O(1) (evaluated in-place)

Time Complexity: O(n) (node values are updated recursively)

Legend: n = total number of nodes

#### Parse Tree Printing (DFS)

Space Complexity: O(1) (no extra space required)

Time Complexity: O(n) (visits every node once)

Legend: n = total number of nodes

#### Parse Tree Printing (BFS)

Space Complexity: O(m) (stores child nodes in a list)

Time Complexity: O(n) (visits every node twice)

Legend: m = total number of operands in the expression

n = total number of nodes

#### Merge Sort

Space Complexity: O(n log n) (creates sub lists with each partition)

Time Complexity: O(n log n) (linear sorting with logarithmic partitions)

Legend: n = total number of items

### Implementation: Design

#### Merge Sort

Merge sort was used instead of other sorting algorithms like Bubble sort and Insertion sort due to efficiency. Merge sort has a time complexity of O(n log n), which is a vast improvement from O(n2) (Bubble and Insertion sort). It does have a drawback of increased space complexity as our implementation does not perform sorting in-place.

#### Expression

The Expression class was introduced to facilitate the sorting of expressions. It acts as a wrapper around a single arithmetic expression and provides automatic predefined comparison methods which are utilised in sorting functions (in this case, merge sort).

#### Operator

The Operator class provides a simple interface for new operators to be registered, hence making the program extensible.

#### PrintOrientation / TreeTraversalOrder

Enums were used to contain all the supported options for printing orientation (horizontal/vertical) and tree traversal orders (pre-order/in-order/post-order) as comparing enum members is more robust and scalable as compared to comparing raw strings.

Custom values can also be specified for each member, which were used in displaying the active configuration.

#### Tree / Node

The edges (implemented as pointers) were delegated to the Node class as it seemed logical.

#### Parse Tree

The parsing and construction used a pointer (labelled currentPointer) to track the position of the last token object inserted. This is an efficient method of handling insertion and enables parsing non-fully parenthesised expressions as an added bonus.

Subtrees can be easily pruned as well by removing or re-assigning the pointer to the root of that sub-tree.

### 

### Implementation: OOP Concepts

#### Abstraction

Node and MathNode are abstract classes which are not meant to be instantiated from directly. Instead, they enforce a template for subclasses to follow and define what methods must be overridden in them.

#### Inheritance

The Parse Tree class inherits the Tree class

The Operand and Operator classes inherit the MathNode class, which inherits the generic Node class

Custom error classes inherit the Exception and SyntaxError classes

Methods and properties were grouped together in the highest logical common superclass

#### Polymorphism

##### Overloading

\_\_lt\_\_, \_\_gt\_\_methods were overloaded in Expression, Operator and Operand classes

get\_priority, augment\_priority and copy methods were defined differently in Operator and Operand classes

##### Overriding

\_\_str\_\_ and \_\_repr\_\_ methods were overridden in Expression, Node and MathNode classes

#### Encapsulation

Private properties and methods in Expression, Lexer, Tokenizer, Operator, Tree and ParseTree classes were prefixed with double underscores

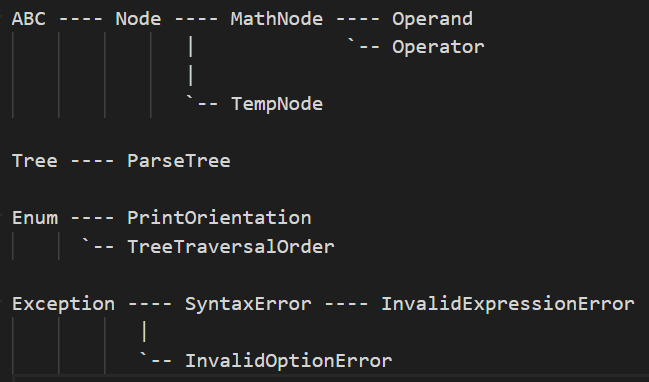
Protected (private within inheritance) properties and methods in Tree and ParseTree classes were prefixed with single underscores

Internal properties of Node and MathNode classes were prefixed with single underscores to denote internal use (i.e. use with care). They were not fully privatised to enable ease of manipulation within the Tree and ParseTree classes

Python’s property decorator was used in the encapsulation of the private \_\_print\_orientation and \_\_print\_traversal\_order attributes of the Tree class. Custom validation was performed within the corresponding setter functions

### 

### Implementation: Inheritance Scheme



### Implementation: Summary of Data Structures Used

|  |  |
| --- | --- |
| **Name** | **Type** |
| dict | Built-in |
| list | Built-in |
| set | Built-in |
| tuple | Built-in |
| Node | Custom |
| Tree | Custom |

### Challenges faced

#### Technical

It was slightly difficult enabling the tokenizer to recognize negative numbers. In the end, the solution was to split up the tokenization process into 2 parts: extracting plain tokens and combining the negative prefix with their corresponding operands.

A considerable amount of debugging was required to make the parse tree’s insert method work as expected. The implementation makes use of a pointer and recursive node floating to place nodes in their appropriate positions.

#### Group Work

There were no issues working together. All was well and there were no conflicts.

### Key takeaways / Learning achievements

I learnt the significance of Binary Trees in computing. Binary Trees are used for searching and sorting as they provide a means of storing data in a hierarchical way. Through this project, I was also able to comprehend why we made use of a Binary Tree over other types of trees. Initially, I was not able to grasp the concept of the different tree traversal modes that we were taught in class as it didn’t seem useful at first. However, incorporating the different tree traversal modes into this project aided me in understanding that the different ways of traversing trees are just as similar to how we access Linear Data Structures like arrays, stacks and lists. In a nutshell, this project has helped to deepen my knowledge of Trees in Data Structure.

### Roles & Contributions

#### Roles

Ethan : Designer

Reshma : Scribe, Tester, Validator

#### Contributions

Ethan:

* io\_utils.py
* exceptions.py
* lexer.py
* math\_node.py
* node.py
* operand\_.py
* operator\_.py
* temp\_node.py
* tokenizer.py

Reshma:

* mergesort.py
* expression.py

Together:

* print\_orientation.py
* tree\_traversal.py
* tree.py
* parse\_tree.py
* main.py
* Report

### 

### Appendix

#### src/utils/io\_utils.py

|  |
| --- |
| '''  Class   : DAAA/FT/2B03  Member 1: Ethan Tan (P2012085)  Member 2: Reshma    (P2011972)  '''  import os  # Clears the terminal output (emulates refreshing a window)  def clear\_console():      os.system('cls' if os.name == 'nt' else 'clear')  # Retrieves input from the user which spans multiple lines  def multiline\_input(prompt: str):      lines = []      print(prompt)      while True:          line = input()          if line:              lines.append(line)          else:              break      return '\n'.join(lines)  # Loads an arbitrary configuration file  def load\_config(config\_file: str, default\_config):      if os.path.exists(config\_file):          with open(file=config\_file, mode='r') as f:              config = f.read().rstrip().splitlines()              config = list(filter(lambda s: s != '',config))              return (\*config, \*default\_config[len(config):])      return default\_config |

#### src/utils/mergesort.py

|  |
| --- |
| **'''**  **Class : DAAA/FT/2B03**  **Member 1: Ethan Tan (P2012085)**  **Member 2: Reshma (P2011972)**  **'''**  **# Sorts an arbitrary list using the merge sort algorithm**  **def mergeSort(l):**  **# Termination clause**  **if len(l) > 1:**  **# Calculate split indices**  **mid = int(len(l)/2)**  **leftHalf = l[:mid]**  **rightHalf = l[mid:]**  **# Sorts sub-lists recursively**  **mergeSort(leftHalf)**  **mergeSort(rightHalf)**  **# Reset merge indices**  **leftIndex, rightIndex, mergeIndex = 0, 0, 0**  **mergeList = l**  **# Merges sub-lists**  **while leftIndex < len(leftHalf) and rightIndex < len(rightHalf):**  **if leftHalf[leftIndex] < rightHalf[rightIndex]:**  **mergeList[mergeIndex] = leftHalf[leftIndex]**  **leftIndex += 1**  **else:**  **mergeList[mergeIndex] = rightHalf[rightIndex]**  **rightIndex += 1**  **mergeIndex += 1**  **# Copies remaining items in left sub-list**  **while leftIndex < len(leftHalf):**  **mergeList[mergeIndex] = leftHalf[leftIndex]**  **leftIndex += 1**  **mergeIndex += 1**  **# Copies remaining items in right sub-list**  **while rightIndex < len(rightHalf):**  **mergeList[mergeIndex] = rightHalf[rightIndex]**  **rightIndex += 1**  **mergeIndex += 1** |

#### src/\_\_init\_\_.py

|  |
| --- |
| **'''**  **Class : DAAA/FT/2B03**  **Member 1: Ethan Tan (P2012085)**  **Member 2: Reshma (P2011972)**  **'''**  **# Centralized imports**  **from .parse\_tree import ParseTree**  **from .utils.io\_utils import \*** |

#### src/exceptions.py

|  |
| --- |
| '''  Class : DAAA/FT/2B03  Member 1: Ethan Tan (P2012085)  Member 2: Reshma (P2011972)  '''  # Exception class for invalid options  class InvalidOptionError(Exception):  def \_\_init\_\_(self, \*args: object) -> None:  super().\_\_init\_\_(\*args)  # Exception class for invalid expressions  class InvalidExpressionError(SyntaxError):  def \_\_init\_\_(self, message='Invalid Expression', \*args: object) -> None:  super().\_\_init\_\_(message, \*args) |

#### src/expression.py

|  |
| --- |
| '''  Class   : DAAA/FT/2B03  Member 1: Ethan Tan (P2012085)  Member 2: Reshma    (P2011972)  '''  from typing import Tuple, Union  # Wrapper class for sorting expressions based on custom function  class Expression:      def \_\_init\_\_(self, expr: str, result: Union[float, int]) -> None:          self.\_\_expr = expr.replace(' ', '')          self.\_\_result = result      # Comparison function      def comparison\_tuple(self) -> Tuple[Union[float, int], int, int]:          return self.\_\_result, len(self.\_\_expr), -self.\_\_expr.count('(')      # Comparison wrappers      def \_\_lt\_\_(self, otherExpression: 'Expression') -> bool:          return self.comparison\_tuple() < otherExpression.comparison\_tuple()      def \_\_le\_\_(self, otherExpression: 'Expression') -> bool:          return self.comparison\_tuple() <= otherExpression.comparison\_tuple()      def \_\_gt\_\_(self, otherExpression: 'Expression') -> bool:          return self.comparison\_tuple() > otherExpression.comparison\_tuple()      def \_\_ge\_\_(self, otherExpression: 'Expression') -> bool:          return self.comparison\_tuple() >= otherExpression.comparison\_tuple()      # Getter for expression result      def get\_result(self) -> Union[float, int]:          return self.\_\_result      # To be displayed as part of one of the options (evaluate and sort)      def \_\_str\_\_(self) -> str:          return self.\_\_expr + ' ==> ' + str(self.\_\_result)      # Same as str      def \_\_repr\_\_(self) -> str:          return self.\_\_str\_\_() |

#### src/lexer.py

|  |
| --- |
| '''  Class : DAAA/FT/2B03  Member 1: Ethan Tan (P2012085)  Member 2: Reshma (P2011972)  '''  from typing import Dict  from .operator\_ import Operator  from .operand\_ import Operand  # Performs lexical analysis on extracted tokens,  # converting them into their actual objects  class Lexer:  def \_\_init\_\_(self,  token\_lookup: Dict[str, Operator]) -> None:  self.\_\_token\_lookup = token\_lookup  # Returns a copy of the object of the associated operand or operator  def \_\_lex\_token(self, token):  return self.\_\_token\_lookup[token].copy()  # Iterates through a given list of tokens, and  # conducts lexical analysis  def lex(self, tokens):  lexed\_token\_list = []  for token in tokens:  if token in '()':  lexed\_token\_list.append(token)  elif token in self.\_\_token\_lookup.keys():  lexed\_token\_list.append(self.\_\_lex\_token(token=token))  else:  lexed\_token\_list.append(Operand(value=float(token) if '.' in token else int(token)))  return lexed\_token\_list |

#### src/math\_node.py

|  |
| --- |
| '''  Class : DAAA/FT/2B03  Member 1: Ethan Tan (P2012085)  Member 2: Reshma (P2011972)  '''  from typing import Any  from abc import abstractmethod  from .node import Node  # Abstract superclass for operands and operators  class MathNode(Node):  def \_\_init\_\_(self, value, symbol, func) -> None:  super().\_\_init\_\_(value=value, width=len(str(value)))  self.\_symbol = symbol  self.\_func = func  # Calls sub-trees recursively before updating its value  def \_\_call\_\_(self) -> Any:  if self.\_left is not None and self.\_right is not None:  self.\_left()  self.\_right()  self.\_value = self.\_func(self.\_left.\_value, self.\_right.\_value)  # Returns symbol  def \_\_str\_\_(self) -> str:  return str(self.\_symbol)  # Abstract comparison methods  @abstractmethod  def \_\_lt\_\_(self, otherNode) -> bool:  pass  @abstractmethod  def \_\_gt\_\_(self, otherNode) -> bool:  pass  # Abstract methods  @abstractmethod  def get\_priority(self) -> int:  pass  @abstractmethod  def augment\_priority(self) -> None:  pass  @abstractmethod  def copy(self):  pass |

#### src/node.py

|  |
| --- |
| '''  Class : DAAA/FT/2B03  Member 1: Ethan Tan (P2012085)  Member 2: Reshma (P2011972)  '''  from abc import ABC  # Abstract class for a node  class Node(ABC):  def \_\_init\_\_(self,  value,  width) -> None:  self.\_value = value  self.\_parent = None  self.\_left = None  self.\_right = None  self.\_width = width  # Returns string representation of internal value  def \_\_str\_\_(self) -> str:  return str(self.\_value)  # Same as str  def \_\_repr\_\_(self) -> str:  return self.\_\_str\_\_()  # Returns display for vertical tree printing  def display(self):  if self.\_left is not None and self.\_right is not None:  diff = self.\_right.\_width - self.\_left.\_width  return ' ' \* abs(min(diff, 0)) + f'{self.\_\_str\_\_():^{self.\_width - abs(diff)}}' + ' ' \* max(diff, 0)  return self.\_\_str\_\_()  # Update widths recursively for vertical tree printing  def update\_widths(self):  if self.\_left is not None:  self.\_left.update\_widths()  if self.\_right is not None:  self.\_right.update\_widths()  if self.\_left is not None and self.\_right is not None:  self.\_width = self.\_left.\_width + 1 + self.\_right.\_width |

#### src/operand\_.py

|  |
| --- |
| **'''**  **Class : DAAA/FT/2B03**  **Member 1: Ethan Tan (P2012085)**  **Member 2: Reshma (P2011972)**  **'''**  **from .math\_node import MathNode**  **from .exceptions import InvalidExpressionError**  **# Class for operands (numbers)**  **class Operand(MathNode):**  **def \_\_init\_\_(self, value) -> None:**  **super().\_\_init\_\_(value, value, lambda \_, \_\_: self.\_value)**  **# Operands have the highest priority**  **def \_\_lt\_\_(self, \_) -> bool:**  **return False**  **def \_\_gt\_\_(self, \_) -> bool:**  **return True**  **# Method should not be called if expression is valid**  **def get\_priority(self):**  **raise InvalidExpressionError()**  **# Increase the priority of the root of an expression within parentheses**  **def augment\_priority(self):**  **pass**  **# Returns a copy of self**  **def copy(self) -> 'Operand':**  **return Operand(value=self.\_value)** |

#### src/operator\_.py

|  |
| --- |
| '''  Class : DAAA/FT/2B03  Member 1: Ethan Tan (P2012085)  Member 2: Reshma (P2011972)  '''  from .math\_node import MathNode  class Operator(MathNode):  def \_\_init\_\_(self, symbol, func, priority) -> None:  super().\_\_init\_\_(None, symbol, func)  self.\_\_priority = priority  # Operators are compared based on their priority  def \_\_lt\_\_(self, otherNode) -> bool:  return self.\_\_priority < otherNode.get\_priority()  def \_\_gt\_\_(self, otherNode) -> bool:  return self.\_\_priority > otherNode.get\_priority()  # Getter for priority  def get\_priority(self) -> int:  return self.\_\_priority  # Increase the priority of the root of an expression within parentheses  def augment\_priority(self):  self.\_\_priority += 3  # Returns a copy of self  def copy(self) -> 'Operator':  return Operator(symbol=self.\_symbol, func=self.\_func, priority=self.\_\_priority) |

#### src/parse\_tree.py

|  |
| --- |
| # type: ignore  '''  Class : DAAA/FT/2B03  Member 1: Ethan Tan (P2012085)  Member 2: Reshma (P2011972)  '''  from typing import List, Literal  import re  from .exceptions import InvalidExpressionError, InvalidOptionError  from .math\_node import MathNode  from .operator\_ import Operator  from .operand\_ import Operand  from .tokenizer import Tokenizer  from .lexer import Lexer  from .tree import Tree  from .expression import Expression  from .utils.mergesort import mergeSort  # For registering custom operators  import math  from math import \*  class ParseTree(Tree):  def \_\_init\_\_(self,  depth\_symbol: str = '.',  mode: int = 1,  precision: int = 5) -> None:  super().\_\_init\_\_(depth\_symbol=depth\_symbol)  self.\_\_precision = precision  self.\_\_token\_lookup = [None, {  '+': Operator('+', lambda a, b: a + b, priority=1),  '-': Operator('-', lambda a, b: a - b, priority=1),  '\*': Operator('\*', lambda a, b: a \* b, priority=2),  '/': Operator('/', lambda a, b: a / b, priority=2),  '\*\*': Operator('\*\*', lambda a, b: a \*\* b, priority=3)  }, {  '+': Operator('+', lambda a, b: max(a, b), priority=1),  '-': Operator('-', lambda a, b: min(a, b), priority=1),  '\*': Operator('\*', lambda a, b: round(a \* b), priority=2),  '/': Operator('/', lambda a, b: round(a / b), priority=2),  '\*\*': Operator('\*\*', lambda a, b: a % b, priority=3)  }][mode]  self.\_\_tokenizer = Tokenizer(self.\_\_token\_lookup.keys())  self.\_\_lexer = Lexer(self.\_\_token\_lookup)  self.\_\_expression = ''  self.\_\_prev\_build = ''  # General parse tree utilities  # Inserts a new node  # Time complexity: O(h), h=tree height  def insert(self, node: MathNode):  if self.\_root is None:  self.\_\_assign\_root(node=node)  elif self.currentPointer.\_left is None:  self.currentPointer = self.\_\_assign\_child(parent=node, child=self.currentPointer, pos='left')  self.\_root = self.currentPointer  elif self.currentPointer.\_right is None:  self.currentPointer = self.\_\_assign\_child(parent=self.currentPointer, child=node, pos='right')  else:  self.currentPointer = self.\_\_float\_child(currentPointer=self.currentPointer, node=node)  # Assigns a new node to become the root node  def \_\_assign\_root(self, node: MathNode) -> 'ParseTree':  self.\_root = node  self.currentPointer = node  return self  # Assigns child and parent nodes  def \_\_assign\_child(self, parent: MathNode, child: MathNode, pos: Literal['left', 'right']) -> 'MathNode':  child.\_parent = parent  if pos == 'left':  parent.\_left = child  return parent  elif pos == 'right':  if parent.\_right is not None:  child.\_left = parent.\_right  parent.\_right = child  return parent if child.\_left is None else child  # Floats a node to its appropriate position  # Time complexity: O(h), h=tree height  def \_\_float\_child(self, currentPointer: MathNode, node: MathNode) -> 'MathNode':  if currentPointer is None:  self.\_root = self.\_\_assign\_child(parent=node, child=self.\_root, pos='left')  return self.\_root  if node > currentPointer:  return self.\_\_assign\_child(parent=currentPointer, child=node, pos='right')  return self.\_\_float\_child(currentPointer=currentPointer.\_parent, node=node)  # Reads a given expression  def read(self, expression: str):  self.\_\_expression = re.sub('\\s', ' ', expression)  # Parses a list of token objects and constructs the parse tree  def parse(self, token\_objs: List[MathNode], i: int = 0):  while i < len(token\_objs):  n = token\_objs[i]  if n == '(':  i += 1  sub\_tree, i = ParseTree().parse(token\_objs=token\_objs, i=i)  if sub\_tree.\_root is None:  raise InvalidExpressionError('Empty parentheses encountered')  sub\_tree.\_root.augment\_priority()  self.insert(sub\_tree.\_root)  elif n == ')':  return self, i  else:  self.insert(n)  i += 1  return self, i  # Combines:  # - Resetting  # - Tokenization  # - Lexical Analysis  # - Parsing/Construction  # - Validation  def build(self):  self.reset()  tokens = self.\_\_tokenizer.tokenize(self.\_\_expression)  token\_objs = self.\_\_lexer.lex(tokens)  self.parse(token\_objs)  if not self.\_\_validate\_parse\_tree():  raise InvalidExpressionError('Invalid Expression')  self.\_\_prev\_build = self.\_\_expression  # Builds the parse tree if it is not already built  def \_\_prepare(self) -> None:  if self.\_\_prev\_build != self.\_\_expression:  self.build()  # Validates the structure of the parse tree  # - Operands have no child nodes  # - Operators have exactly 2 child nodes  def \_\_validate\_parse\_tree(self) -> bool:  def \_\_internal\_recursive(node: MathNode) -> bool:  if isinstance(node, Operator):  if node.\_left is None or node.\_right is None:  return False  return \_\_internal\_recursive(node.\_left) and \_\_internal\_recursive(node.\_right)  elif isinstance(node, Operand):  if node.\_left is None and node.\_right is None:  return True  return False  if self.\_root is None:  return False  return \_\_internal\_recursive(node=self.\_root)  # Option 1 - Assert given expression is fully parenthesised  def validate\_fully\_parenthesised(self):  return self.\_\_expression.replace(' ', '') == self.reconstruct\_expression().replace(' ', '')  # Options 1, 2, 3 - Evaluate/Display the parse tree for given expression(s)  # Prints the tree (uses superclass Tree to do so)  def print\_tree(self):  self.\_\_prepare()  super().print\_tree()  # Evaluates the expression  def evaluate(self):  self.\_\_prepare()  self.\_root()  result = self.\_root.\_value  return result if type(result) is int else round(result, self.\_\_precision)  # Option 3 - Evaluate and Sort a list of expressions from a specified text file  def evaluate\_and\_sort(self):  try:  inputfile = input("Please enter input file:\t".expandtabs(4))  outputfile = input("Please enter output file:\t".expandtabs(4))  filename = open(inputfile, 'r').read().splitlines()  lst = []  for i in filename:  self.read(i)  exp = Expression(i, self.evaluate())  lst.append(exp)  mergeSort(lst)  print("\n>>> Evaluation and sorting started:", end='')  header = '\n\n\*\*\* Expressions with value ==> '  exp = []  prev\_value = None  for i in lst:  current\_value = i.get\_result()  if current\_value != prev\_value:  var = header + str(current\_value)  exp.append(var)  prev\_value = current\_value  exp.append("\n" + str(i))  processed\_expressions = ''.join(exp)  print(processed\_expressions)  print("\n>>> Evaulation and sorting completed!")  if outputfile != '':  with open(outputfile, "w") as add\_to\_output\_file:  add\_to\_output\_file.write(processed\_expressions[2:])  except (ValueError, InvalidExpressionError):  print('Input file contains one or more invalid expressions. Aborting...')  # Option 4 - Fully parenthesise a non-fully parenthesised expression  def reconstruct\_expression(self) -> str:  self.\_\_prepare()  def \_\_reconstruct\_internal(node):  if node.\_left is not None and node.\_right is not None:  return '(' + \_\_reconstruct\_internal(node.\_left) + ' ' + str(node) + ' ' + \_\_reconstruct\_internal(node.\_right) + ')'  return str(node)  if self.\_root.\_left is None and self.\_root.\_right is None:  return '(' + str(self.\_root) + ')'  return \_\_reconstruct\_internal(node=self.\_root)  # Option 6 - Register a New Operator  # Allows the user to select the symbol for their custom operator  def \_\_get\_symbol(self):  symbol = input('Select a symbol besides [{}, ., (, )] with max length of 3: '.format(', '.join(self.\_\_token\_lookup.keys()))).strip()  if symbol in '.()' or symbol in self.\_\_token\_lookup.keys():  raise InvalidOptionError(f'Illegal symbol encountered: {symbol}')  if len(symbol) > 3:  raise InvalidOptionError(f'Symbol {symbol} is too long (max 3)')  return symbol  # Allows the user to select what their custom operator does  # - They have access to Python's standard math library's functions  def \_\_get\_func(self):  return eval('lambda a, b: ' + input('Enter function (in Python format) (parameters are a and b): '))  # Allows the user to select the priority of their custom operator  def \_\_get\_priority(self):  self.\_\_print\_priority\_menu()  priority\_str = input('Operator Priority: ').strip()  priority = int(priority\_str) if priority\_str != '' else 1  if priority not in {1, 2, 3}:  raise SyntaxError('Invalid priority (expected 1, 2 or 3)')  return priority  # Prints all available priority options  @staticmethod  def \_\_print\_priority\_menu():  print('Enter the priority of your operator\n'  '\t1: {+, -} [Default]\n'  '\t2: {\*, /}\n'  '\t3: {\*\*}')  # Interface for complete registration of a new operator  def register\_new\_operator(self):  print('Here is a simple wizard that will guide you through registering a custom operator...')  symbol = self.\_\_get\_symbol()  func = self.\_\_get\_func()  priority = self.\_\_get\_priority()  new\_operator = Operator(symbol=symbol, func=func, priority=priority)  self.\_\_token\_lookup[symbol] = new\_operator  print(f'Successfully registered new operator {symbol}') |

#### src/print\_orientation.py

|  |
| --- |
| '''  Class : DAAA/FT/2B03  Member 1: Ethan Tan (P2012085)  Member 2: Reshma (P2011972)  '''  from enum import Enum  # Enum of all valid print orientations  class PrintOrientation(Enum):  HORIZONTAL = 'Horizontal'  VERTICAL = 'Vertical' |

#### src/temp\_node.py

|  |
| --- |
| '''  Class : DAAA/FT/2B03  Member 1: Ethan Tan (P2012085)  Member 2: Reshma (P2011972)  '''  from .node import Node  # Class for temporary nodes for vertical printing  class TempNode(Node):  def \_\_init\_\_(self, width) -> None:  super().\_\_init\_\_(value=' ' \* width, width=width)  # Pad with spaces  def display(self):  return ' ' \* self.\_width |

#### src/tokenizer.py

|  |
| --- |
| '''  Class : DAAA/FT/2B03  Member 1: Ethan Tan (P2012085)  Member 2: Reshma (P2011972)  '''  from typing import List, Set  import re  # Extracts tokens from a string expression  class Tokenizer:  def \_\_init\_\_(self,  registered\_tokens: Set[str]) -> None:  self.\_\_registered\_tokens = registered\_tokens  # Identifies negative numbers, and  # groups the prefix - with the corresponding operand  def \_\_combine(self, tokens: List[str]) -> List[str]:  combined = []  prev\_token = ''  for i, token in enumerate(tokens):  if token == '-' and (i == 0 or not prev\_token[-1] in '0123456789.)'):  tokens[i + 1] = '-' + tokens[i + 1]  else:  combined.append(token)  prev\_token = token  return combined  # Extracts plain tokens using regex  def tokenize(self, expression: str):  expression\_simplified = re.sub('\\s', '', expression)  known\_symbols = set(''.join(self.\_\_registered\_tokens)).difference('-')  matcher = r'([-]|[\d.]+|[{}]+|[()])'.format(''.join(known\_symbols))  plain\_tokens = re.findall(matcher, expression\_simplified)  combined\_tokens = self.\_\_combine(tokens=plain\_tokens)  return combined\_tokens |

#### src/tree\_traversal\_order.py

|  |
| --- |
| '''  Class : DAAA/FT/2B03  Member 1: Ethan Tan (P2012085)  Member 2: Reshma (P2011972)  '''  from enum import Enum  # Enum of all valid tree traversal orders (dfs)  class TreeTraversalOrder(Enum):  IN\_ORDER = 'In-Order'  PRE\_ORDER = 'Pre-Order'  POST\_ORDER = 'Post-Order' |

#### src/tree.py

|  |
| --- |
| # type: ignore  '''  Class   : DAAA/FT/2B03  Member 1: Ethan Tan (P2012085)  Member 2: Reshma    (P2011972)  '''  from .temp\_node import TempNode  from .exceptions import InvalidOptionError  from .print\_orientation import PrintOrientation  from .tree\_traversal\_order import TreeTraversalOrder  # Class for generic trees  class Tree:      def \_\_init\_\_(self,                   depth\_symbol: str = '.') -> None:          # Protected properties          self.\_root = None          self.\_currentPointer = None          # Private properties          self.\_\_depth\_symbol = depth\_symbol          self.\_\_print\_orientation: PrintOrientation = PrintOrientation.HORIZONTAL          self.\_\_print\_traversal\_order: TreeTraversalOrder = TreeTraversalOrder.IN\_ORDER      # Getter for print orientation      @property      def print\_orientation(self):          return self.\_\_print\_orientation      # Setter for print orientation      @print\_orientation.setter      def print\_orientation(self, new\_print\_orientation: str):          if new\_print\_orientation == '':              return          if new\_print\_orientation not in 'hv':              raise InvalidOptionError(f'Unknown option \'{new\_print\_orientation}\' encountered for print\_mode (expected \'h\' or \'v\')')          available\_print\_modes = {              'h': PrintOrientation.HORIZONTAL,              'v': PrintOrientation.VERTICAL          }          self.\_\_print\_orientation = available\_print\_modes[new\_print\_orientation]      # Getter for print traversal order      @property      def print\_traversal\_order(self):          return self.\_\_print\_traversal\_order      # Setter for print traversal order      @print\_traversal\_order.setter      def print\_traversal\_order(self, new\_traversal\_order: str):          if new\_traversal\_order == '':              return          if new\_traversal\_order not in 'abc':              raise InvalidOptionError(f'Unknown option \'{new\_traversal\_order}\' encountered for traversal\_order (expected a, b or c)')          possible\_traversal\_orders = {              'a': TreeTraversalOrder.IN\_ORDER,              'b': TreeTraversalOrder.PRE\_ORDER,              'c': TreeTraversalOrder.POST\_ORDER          }          self.\_\_print\_traversal\_order = possible\_traversal\_orders[new\_traversal\_order]      # Prints the available options for tree traversal orders      @staticmethod      def \_\_print\_traversal\_menu():          print()          print("Please select how you want to traverse the tree [a/b/c]:\n"                "\ta. Inorder (R, N, L)\n"                "\tb. Preorder (N, R, L)\n"                "\tc. Postorder (R, L, N)\n")      # Print methods      # DFS      def \_\_print\_inorder(self):          def \_\_internal\_recursive(node, depth: int = 0):              if node is not None:                  \_\_internal\_recursive(node=node.\_right, depth=depth + 1)                  print(self.\_\_depth\_symbol \* depth + str(node))                  \_\_internal\_recursive(node=node.\_left, depth=depth + 1)          \_\_internal\_recursive(node=self.\_root, depth=0)      def \_\_print\_preorder(self):          def \_\_internal\_recursive(node, depth: int = 0):              if node is not None:                  print(self.\_\_depth\_symbol \* depth + str(node))                  \_\_internal\_recursive(node=node.\_right, depth=depth + 1)                  \_\_internal\_recursive(node=node.\_left, depth=depth + 1)          \_\_internal\_recursive(node=self.\_root, depth=0)      def \_\_print\_postorder(self):          def \_\_internal\_recursive(node, depth: int = 0):              if node is not None:                  \_\_internal\_recursive(node=node.\_right, depth=depth + 1)                  \_\_internal\_recursive(node=node.\_left, depth=depth + 1)                  print(self.\_\_depth\_symbol \* depth + str(node))          \_\_internal\_recursive(node=self.\_root, depth=0)      # BFS      def \_\_print\_vertical(self):          self.\_root.update\_widths()          current\_nodes = [self.\_root]          next\_nodes = []          # Continue while there is at least one non-temporary node left          while any(map(lambda n: not isinstance(n, TempNode), current\_nodes)):              for n in current\_nodes:                  print(n.display(), end=' ')                  if n.\_left is None and n.\_right is None:                      next\_nodes.append(TempNode(width=n.\_width))                  else:                      if n.\_left is not None:                          next\_nodes.append(n.\_left)                      if n.\_right is not None:                          next\_nodes.append(n.\_right)              current\_nodes = next\_nodes              next\_nodes = []              print()      # Options 1 & 2 - Public method for printing the tree      #   - Uses \_\_print\_orientation and \_\_print\_traversal\_order to determine how to print the tree      def print\_tree(self):          if self.\_\_print\_orientation is PrintOrientation.VERTICAL:              self.\_\_print\_vertical()          else:              if self.\_\_print\_traversal\_order is TreeTraversalOrder.PRE\_ORDER:                  self.\_\_print\_preorder()              elif self.\_\_print\_traversal\_order is TreeTraversalOrder.IN\_ORDER:                  self.\_\_print\_inorder()              else:                  self.\_\_print\_postorder()      # Resets the pointers      def reset(self):          self.\_root = None          self.\_currentPointer = None      # Oprion 5 - Interface for users to change the print orientation and/or traversal order      def change\_print\_mode(self):          new\_print\_orientation = input('Enter new print mode (h/v): ').strip().lower()          self.print\_orientation = new\_print\_orientation          if new\_print\_orientation == 'h':              self.\_\_print\_traversal\_menu()              new\_traversal\_order = input('Enter new print mode (a/b/c): ').strip().lower()              self.print\_traversal\_order = new\_traversal\_order          print()          print("Printing Mode Updated")          print("Orientation:\t{}".format(self.print\_orientation.value).expandtabs(6))          print("Traversal Order:\t{}".format(self.print\_traversal\_order.value).expandtabs(6)) |

#### main.py

|  |
| --- |
| '''  Class   : DAAA/FT/2B03  Member 1: Ethan Tan (P2012085)  Member 2: Reshma    (P2011972)  '''  from sys import argv  from src import ParseTree, clear\_console, load\_config  from src.exceptions import InvalidExpressionError  # Global constants  DEFAULT\_CONFIG\_FILE = 'config.txt'  CONFIG\_FILE = argv[1] if len(argv) > 1 else DEFAULT\_CONFIG\_FILE  # Prints the details of the authors of this program  def print\_author\_details():      print("\*"\*48)      print("\* ST1507 DSAA: Expression Evaluator and Sorter \*")      print("\*----------------------------------------------\*")      print("\* Done By: Ethan (2012085) & Reshma (2011972)  \*")      print("\* Class  : DAAA/2B/03                          \*")      print("\*"\*48)      print()  # Prints the available options  def menu():      print("Please select your choice [1 - 7]:\n"            "1. Evaluate Fully Parenthesised Expression\n"            "2. Evaluate Any Valid Expression\n"            "3. Sort Expressions\n"            "4. Fully Parenthesise an Expression\n"            "5. Select Printing Mode\n"            "6. Register a New Operator\n"            "7. Exit")  # Entry point into the program  def main():      # Loads configuration,      #   otherwise falls back to default options      try:          depth\_symbol, operator\_mode = load\_config(config\_file=CONFIG\_FILE, default\_config=['.', 1])          t = ParseTree(depth\_symbol=depth\_symbol[0], mode=int(operator\_mode))      except:          t = ParseTree()      # Infinite loop,      #   until user exits the program      user\_choice = ''      while user\_choice != '7':          try:              # Refresh the display              clear\_console()              print\_author\_details()              menu()              # Allow user to select option              user\_choice = input("Enter choice: ").strip()              # Evaluate an expression (1 - fully parenthesised only / 2 - any valid expression)              #   - Parse tree will be displayed              #   - Followed by the result of the evaluation              if user\_choice in {'1', '2'}:                  expression = input('Please enter the expression you want to evaluate:\n')                  t.read(expression)                  t.build()                  if user\_choice == '1' and not t.validate\_fully\_parenthesised():                      raise InvalidExpressionError(f'Expression is not fully parenthesised. Did you mean \'{t.reconstruct\_expression()}\'?')                  print('\nParse Tree')                  print('----------')                  t.print\_tree()                  print(f'\nExpression evaluates to:')                  print(t.evaluate())              # Evaluate and sort expressions from a specified text file              #   - The user can choose to save the results              elif user\_choice == '3':                  t.evaluate\_and\_sort()              # Fully parenthesise a non-fully parenthesised valid expression              elif user\_choice == '4':                  expression = input('Enter an expression you want to fully parenthesise: \n')                  t.read(expression)                  print('Expression fully parenthesised:', t.reconstruct\_expression())              # Allows the user to change the mode of printing              #   - h. horizontal (employs dfs)              #       - a. in-order              #       - b. pre-order              #       - c. post-order              #   - v. vertical (employs bfs)              elif user\_choice == '5':                  t.change\_print\_mode()              # Allows the user to register a custom operator              elif user\_choice == '6':                  t.register\_new\_operator()              # Exits/Quits and thus terminates the program              elif user\_choice == '7':                  print("Bye, thanks for using ST1507 DSAA: Expression Evaluator and Sorter :D")          except FileNotFoundError as e:              # Truncates the prefix '[Errno 2]'              print(str(e)[10:])          except ValueError:              print('Invalid Expression')          except Exception as e:              print(str(e))          finally:              # Pauses program before display is refreshed              input('\nPress Enter to continue...')  main() |

### References

* Python animated\_parse\_tree package (<https://github.com/ethanolx/Animated-Parse-Tree-py>)