Andy CHUNG

Abstract

A ground up reconstruction of a calculator backend to provide a customizable interface API

Customizable Calculator Interface

Table of Contents

[2 Introduction 2](#_Toc161649604)

[3 Analysis 8](#_Toc161649605)

[3.1 Problem statement 8](#_Toc161649606)

[3.2 Identifying users 8](#_Toc161649607)

[3.3 Existing solutions 8](#_Toc161649608)

[3.4 Dialogue with client 8](#_Toc161649609)

[3.5 Acceptable limits 12](#_Toc161649610)

[3.6 Project requirements 12](#_Toc161649611)

[3.6.1 Functional Requirements 12](#_Toc161649612)

[3.6.2 Non-Functional Requirements 13](#_Toc161649613)

[3.6.3 Technical Requirements 13](#_Toc161649614)

[3.6.4 Non-Client Specified Technical Requirements 14](#_Toc161649615)

[3.7 Extension objectives 18](#_Toc161649616)

[4 Documented Design 19](#_Toc161649617)

[4.1 System Design 19](#_Toc161649618)

[4.1.1 Model 19](#_Toc161649619)

[4.1.2 Controller 19](#_Toc161649620)

[4.1.3 View 19](#_Toc161649621)

[4.2 Control flows 20](#_Toc161649622)

[4.2.1 Set\_instruction 20](#_Toc161649623)

[4.2.2 send\_visitor 21](#_Toc161649624)

[4.2.3 get\_calculated\_results 22](#_Toc161649625)

[4.2.4 get\_(setting/variables/functions)/set\_settings 23](#_Toc161649626)

[4.3 File layout and contents 24](#_Toc161649627)

[4.3.1 monad.py 25](#_Toc161649628)

[4.3.2 utlities.py 28](#_Toc161649629)

[4.3.3 mathobj.py 33](#_Toc161649630)

[4.3.4 lexer.py 34](#_Toc161649631)

[4.3.5 parser.py 35](#_Toc161649632)

[4.3.6 controlunit.py 37](#_Toc161649633)

[4.4 Data Structures 40](#_Toc161649634)

[4.4.1 Stack 40](#_Toc161649635)

[4.4.2 Rooted Tree (SubTree) 40](#_Toc161649636)

[4.5 Algorithms 42](#_Toc161649637)

[4.5.1 Tokenizer 42](#_Toc161649638)

[4.5.2 Parser 43](#_Toc161649639)

[4.5.3 Dijkstra’s shunting yard algorithm 45](#_Toc161649640)

[4.5.4 Tree traversal algorithm (Visitor pattern) 49](#_Toc161649641)

[5 Testing (automated regression testing) 50](#_Toc161649642)

[5.1 Non-client required testing 50](#_Toc161649643)

[5.1.1 Lexer tests 50](#_Toc161649644)

[5.1.2 Parser tests 51](#_Toc161649645)

[5.2 Project requirement tests 59](#_Toc161649646)

[5.2.1 Functional requirements 59](#_Toc161649647)

[5.2.2 Non-Functional Requirements 60](#_Toc161649648)

[5.2.3 Technical requirements 60](#_Toc161649649)

[5.3 Additional testing 61](#_Toc161649650)

[6 Evaluation 69](#_Toc161649651)

[6.1 Requirements 69](#_Toc161649652)

[6.2 Improvements / Potential extensions 69](#_Toc161649653)

[6.3 Feedback from user 70](#_Toc161649654)

[6.4 What I learned & how I would use it to recreate the same program. 70](#_Toc161649655)

[7 Citations 70](#_Toc161649656)

[8 Code listing 71](#_Toc161649657)

# Introduction

At first the aim of this project was to create a calculator that can perform simple arithmetic, functional, and graphical operations. As the project progressed, this project evolved to be just a backend for any calculator interface that provides an interface for user interfaces to be implemented by other programmers.

When you interact with a physical calculator, you first enter a push a sequence of buttons that represent want you want the calculator to calculate. As you enter the request, the calculator will display what you have input in their own format. Then when you have finished entering what you want, the calculator then displays what you have inputted as well as the calculated result.

Simplified into a list of procedures, it would look like:

1. User enters input, while calculator displays what user is inputtingA calculator on a table

   Description automatically generated
2. User confirms input as complete

A person pressing a calculator

Description automatically generated

1. Calculator does the calculations users specifies
2. Calculator displays user’s original input + calculated result

A calculator on a table

Description automatically generated

What my program does is mainly step 3 of the list above, performing non-user related(backend) tasks, in addition to providing customizability during operations of steps 1 and 4.

With most calculator inputs and or any language that is used to run instructions for a machine, there are three main steps from receiving the instructions in the form of characters into the desired results of the instructions encoded in the input.

1. The recognition of familiar segments within the input (aka **tokenizing**: labeling segments of text with certain meanings(**tokens**))
2. Stringing the identified segments into the intended sequential instructions or into a structure that defines the order of execution. (**parsing**: to consider the context of the token in relation other tokens and then rearrange into a more structured form)
3. Apply the subroutines/operations as required by the structure made in the step prior.

Before we can dive into an example, it is first important to understand that for an input to be parsed, there must be a standard or a set of language rules it coheres to. These rules can often be expressed in a form called **Backus Naur Form** (**BNF**) as well as **Syntax diagrams**.

Both forms express a language in terms of rules, where each rule could either be independent, self-dependent, or interdependent on other rules.

For instance, the rule for a whole positive number would look like this in Backus Naur Form:

<number> ::= <number><digit>|<digit>

Where “::=” indicates the creation of the rule for “number”, and on the right, it shows that a number could be a number with a “digit” at the end of a number or (“|”) it could just be a “digit” where “digit” indicates for another rule for digits as it is in angle brackets <>.

Back to an example to illustrate the processing operations, “2+3”:

First, “2+3” is first isolated into the segments, “2”, “+”, and “3”; which will be considered as tokens.

Then with the second step, the tokens will then be parsed into a format that the calculator can operate with. In the case with calculators and mathematical expressions, that format would be an **expression tree**.

An **expression tree** is a structure that represents a mathematical expression commonly with values and operations as “nodes”. Connections between the nodes indicate the application of the operation on the nodes connected below it (**branches**). When resolving an expression tree, it is convention to start from the bottom up.

To reuse the example, an expression tree of the expression 2+3 would look like:

A diagram of a connection

Description automatically generated

Back to the example of “2+3”, once step 2 is finished and the expression tree has been constructed, the final step would be to apply the operations as dictated by the expression tree constructed.

The application of the instruction would be carried out during a traversal of the expression tree. This traversal can be carried out using a **Visitor behavioral pattern**.

The Visitor pattern is a way of arranging logic and data such that they are independent of each other. This is done by having the data object (often called the **Visitee**) provide a way for the logic (**Visitor**) to operate with the data carried by the object.

In the case of evaluating an expression tree, each node would have a certain rule (operation for operations, and the number values for the numbers).

This would yield the following results if done from a left to right, bottom to top (**postfix**) order on the expression tree:

(2) -> 2

(3) -> 3

(+) -> 5

In the last result, due to the recursive nature of the expression tree, we can assume that the visitor also has access to the connected nodes below the node it is operating on; the visitor can then operate on those nodes and obtain the result, then apply the corresponding operation on the results.

For a more complex example, let’s see how “2+3\*5” is processed:

First, it is tokenized into “2”, “+”, “3”, “\*”, “5”.

Then the order of operations is often considered using **Dijkstra’s shunting yard algorithm** (for more detail, refer to the more in-depth break down in the Algorithms section p45) and parsed into an expression tree with the later operations being placed above operations that should come before it.

A diagram of a network

Description automatically generated

The above diagram represents the expression 2+3\*5 in that way because, due to the order of operations, multiplication is applied first. This is indicated by the 3 and 5 nodes being connected to the multiply operator, implying that 3 and 5 are being multiplied first, then the result of that operation has the add operation applied in conjunction with the “2” node connected to the add operator as well.

In the final step, the results from applying the operations would be:

(2) -> 2

(3) -> 3

(5) -> 5

(\*) -> 15

(+) -> 17

At the end we would have obtained the expected value from the expression “2+3\*5”, which is equals to 17.

# Analysis

## Problem statement

There are many calculators that are available on the internet, but they all have their own interfaces and preset displays with minimal customization abilities and access to their backends.

## Identifying users

The users of the program would be programmers that could either patch together a quick calculator with a simple or sophisticated user interface.

## Existing solutions

There are many options for calculators both physical (Casio fx-CG50, Ti-inspire) and digital (Desmos, Google, windows calculator, python), but none provide an entirely customizable interface for users.

## Dialogue with client

What kind of operations would you like the calculator backend to be capable of?

For starters, the calculator should be able to support basic mathematical operations, +, -, \*, /, ^. The calculator should also be able to work with all real numbers.

What should the response be when a math error such as 1/0 occurs?

I would like for a undefined value to be created.

What form of user input would be the most preferable?

I think a sequence of characters would be the most adaptable to most systems.

For the customization feature, would a limitation to only recursive structures be too strict?

No, that would actually make it much simpler, reducing the number of rules I would have to consider.

When the inputted character sequence is not valid or other various errors pop up during processing of the input, would you like for the error to be propagated back up to the interface call, or would you like to have a specified method that the program can pass the error to?

If possible, I think the latter would be more preferable as it would be not favorable to stop every possible error that can pop up.

Ok, let me get you a brief list with functionalities and you can see if that fits what you want.

|  |  |  |  |
| --- | --- | --- | --- |
| Exposed Methods/functions | Functionality | Input | Output |
| set\_instruction | Sets the internal record/ expression tree | Sequence of characters | Nothing or warning that the sequence cannot be tokenized or parsed into an expression tree |
| get\_calculated | Calculates the result of the expression tree | None | The result of arithmetic operations on the expression tree |
| send\_visitor | Pass in a collection of rules to operate on the expression tree | Rules (Visitor object) | Whatever the rules dictate |
| alert\_improper\_input | Called by program to notify interface of error during processing | Error | None |

With support for arithmetic operations such as addition, subtraction, multiplication, division, and exponentiation; returns undefined for math errors.

Here you are, I have also included a demo implementation that demonstrates what the API can be capable of.

That looks good, could you also add support for assigning values to variables, as well as the declaring of functions, and the use of them as well?

Yes, that could be done.

Could there be a method that retrieves all the variables and functions that have been declared/saved via the instructions? Oh, and while you are at it, could you have all results of instructions be saved to an “Ans” variable?

Ok.

Could the demo also have a help menu displayed on startup?

Yes.

Could you add support for trigonometric functions like sin, cos, tan, arcsin, arcos, arctan?

Ok, and with the trigonometric functions, would you like to have a distinction between radians and degrees when using those functions?

Yes, I would, and could there be a way or method to change between the two modes(degree and radians) as well as another one to get the current mode?

Certainly.

Could you add a feature where I can pass in named functions that the program can call?

Ok, but what would the functions do?

The functions could provide calculations, display a user interface, create a custom object etc.

That sounds like an interesting idea, I’ll be sure to try and implement that into the demo as well.

I think that will be it.

Ok, just to summarise, is this what you want?

|  |  |  |  |
| --- | --- | --- | --- |
| Exposed Methods/functions | Functionality | Input | Output |
| set\_instruction | Sets the internal record/ expression tree | Sequence of characters | Nothing or warning that the sequence cannot be tokenized or parsed into an expression tree |
| get\_calculated | Calculates the result of the expression tree | None | The result of arithmetic operations on the expression tree |
| send\_visitor | Pass in a collection of rules to operate on the expression tree | Rules (Visitor object) | Whatever the rules dictate |
| alert\_improper\_input | Called by program to notify interface of error during processing | Error | None |
| get\_variables | Gets the variables that have been declared/saved | None | Variable names and their values |
| get\_functions | Gets the functions that have been declared/saved | None | Function names and their corresponding functions |
| get\_settings | Gets the current mode it is in | None | Setting categories and their state |
| set\_settings | Sets the mode | Mode to set to | None |
| get\_custom\_statements | Returns list of named functions that can be called | None | Named functions |

With language support for:

* Arithmetic

<a><operation><b> -> <a operation b>

* + Addition
  + Subtraction
  + Multiplication
  + Division
  + Exponentiation
* Trigonometry
  + Sin
  + Cos
  + Tan
  + Arcsin
  + Arcos
  + Arctan
* Math errors
* Variables
  + setting

<name>=<value>

* + use
* Functions
  + setting

<name>(<parameters>)=<expression>

* + use
* Calling of custom defined functions

## Acceptable limits

The program should be able to accept user input in a mathematical expression like string, perform basic arithmetic, support trigonometric functions, support radian and degrees modes, allow for user created variables and functions. The program shall operate with real numbers and undefined values. The program should detect methods of the implemented interface and subsequently call them as part of the instruction passed in the string.

## Project requirements

### Functional Requirements

1 Take string input from interface

1.1 support/create a mathematical expression like language

2 Perform/support arithmetic

2.1 addition

<a>+<b> -> <a+b>

2.2 subtraction

<a>-<b> -> <a-b>

2.3 multiplication

<a>\*<b> -> <a\*b>

2.4 division

<a>/<b> -> <a/b>

2.4.1 division by 0

<a>/0 -> Undefined

2.5 exponentiation

<a>^<b> -> <ab>

2.6 negation

-<a> -> <-a>

2.7 brackets

(<expression>) <operator> <expression> -> <(expression) operator expression>

2.8 assignment

2.8.1 Variable assignment

<name> = <value>

2.8.2 function assignment

<name>(<parameters>) = <expression>

2.9 function composition

f(3+g(2))

2.10 order of operations is followed when no brackets are present

2.10.1 exponentiation before anything

2.10.2 then followed by multiplication or division

2.10.3 then followed by addition or subtraction

2.10.4 if two operations have the same order, the one of the left has precedence

2.11 trigonometric functions

- sin

- cos

- tan

- arcsin

- arcos

- arctan

3 has operation modes

3.1 can set operation mode

3.2 operation mode must affect trigonometric functions

4 method that returns the custom named functions called by the program

5 user/implementer must be able to customize output by passing in a Visitor

### Non-Functional Requirements

1. the interface must be simple
   1. must not expose methods not discussed
   2. must not be another way of manipulating program data other than through specified methods

### Technical Requirements

1 the abstract base interface class must have the following methods

* 1. set\_instruction

to set and create the expression tree from instruction input

* 1. send\_visitor

specifies what rules operate on the expression tree and returns result

* 1. get\_calculated

returns results of carrying out functions and or expressions

* 1. set\_settings

sets the mode of operation

* 1. get\_settings

gets the mode of operation

* 1. get\_variables

gets the declared variables

* 1. get\_functions

gets the declared functions

* 1. alert\_improper\_input

called by program to alert interface of error when processing input

mock class diagram showing client requirements

A white paper with black text

Description automatically generated

### Non-Client Specified Technical Requirements

1 Lexer/tokenizer

1.1 must only accept legal characters (QWERTYUIOPASDFGHJKLZXCVBNMqwertyuiopasdfghjklzxcvbnm~1234567890^\*()-+=|;<>,./\n)

#### Valid Language syntax(BNFs)

##### Tokens & rules

<EOL> ::= ';' | '\n'

<OpenBracket> ::= '('

<ClosedBracket> ::= ')'

<Comma> ::= ','

<ComparisonOperator> ::= '<' | '>' | '<=' | '>=' | '=='

<Operator> ::= '+' | '-' | '\*' | '/' | '^'

<ConditionalOperator> ::= '|'

<Equality> ::= '=' | '~'

<NameSpace> ::= ('A' | 'B' | 'C' | 'D' | 'E' | 'F' | 'G' | 'H' | 'I' | 'J' | 'K' | 'L' | 'M' | 'N' | 'O' | 'P' | 'Q' | 'R' | 'S' | 'T' | 'U' | 'V' | 'W' | 'X' | 'Y' | 'Z' | 'a' | 'b' | 'c' | 'd' | 'e' | 'f' | 'g' | 'h' | 'i' | 'j' | 'k' | 'l' | 'm' | 'n' | 'o' | 'p' | 'q' | 'r' | 's' | 't' | 'u' | 'v' | 'w' | 'x' | 'y' | 'z')

<NameSpace> ::= <Namespace> <Digit> <NameSpace> | <Namespace> <Digit>

regex: [A-Za-z]+([A-Za-z0-9]+)\*

A diagram of a circuit board

Description automatically generated

<Number> ::= <Digit> | <Digit> '.' <Digit>

regex: [0-9]+(\\.[0-9])?

A diagram of a diagram

Description automatically generated

<Digit> ::= ('0' | '1' | '2' | '3' | '4' | '5' | '6'| '7' | '8' | '9')+

##### Parser Rules

<Lines> ::= <Line> | <Line> <Lines>

A diagram of lines and words

Description automatically generated

<Line> ::= <Assignment> <EOL> | <Expression> <EOL> | <Statement> <EOL> | <EOL>

A diagram of a system

Description automatically generated

<Statement> ::= <NameSpace> <OpenBracket> <Operands> <ClosedBracket>

A diagram of a block diagram

Description automatically generated with medium confidence

<Assignment> ::= <Assignable> <Equality> <Expression>

A green rectangle with black text

Description automatically generated

<Expression> ::= <Term> | <Term> <Operator> <Expression>

A diagram of a function

Description automatically generated

<Term> ::= <Number> | <Function> | <NameSpace> | <Operator> <Term> | <BracketedTerm>

A diagram of a computer

Description automatically generated

<BracketedTerm> ::= <OpenBracket> <Expression> <ClosedBracket>

A green and black rectangular with black text

Description automatically generated

<Assignable> ::= <Function> | <NameSpace>

A diagram of function and name

Description automatically generated

<Function> ::= <NameSpace> <OpenBracket> <Operands> <ClosedBracket>

A diagram of a block diagram

Description automatically generated

<Operands> ::= ε | <Operand> | <Operand> <Comma> <Operands>

A diagram of a comma

Description automatically generated

<Operand> ::= <Conditional> | <Inequality> | <Expression>

A diagram of a function

Description automatically generated with medium confidence

<Conditional> ::= <Inequality> <ConditionalOperator> <Inequality>

A close-up of a logo

Description automatically generated

<Inequality> ::= <Term> <ComparisonOperator> <Term> | <Term> <ComparisonOperator> <Term> <CompOperator> <Term>

A diagram of a diagram

Description automatically generated

3. Abstract syntax tree constructed

4. Dijkstra’s shunting yard algorithm used to determine order of operations in Expressions

## Extension objectives

1. prototype interfaces
   1. command line interface
   2. graphical user interface
2. support for more/additional types of values
   1. distributions
   2. complex values

# Documented Design

## System Design

The program employes a model view controller design. This isolates the algorithms (**model**) from the decision-making (**controller**) compartments of the program and the interactions to the program (**view**). As a side effect, this modularizes the program into parts that that has sperate responsibilities.

A diagram of a software application

Description automatically generated

### Model

In the program, the lexer, parser, and the calculation unit constitute the model, and are all called/used by the controller, responsible for tokenizing, parsing, and performing operations on the expression tree respectively. (the order *may* be called at random, but in the program, it follows that lexer is called before parser and parser before calculation unit)

### Controller

The controller sits in the middle between the models and the view, where it coordinates which models should be used for each request from the view.

### View

The view acts as the interface to the “user” (true user interface) but also functions to restrict and simplify the access of the interface to the controller.

## Control flows

### Set\_instruction

The set\_instruction method provided by the interface transforms a string of characters into the traversable expression tree. It does this by first passing the characters to the control unit which then passes it into the tokenizer to convert the characters into recognizable tokens. Then once the control unit receives the tokens, it then passes the tokens into the parser where it waits for the expression tree. Once the expression tree has been received from the parser, it is then passed long to the tree constructor where the tree is made traversable before being returned to the control unit where the prior expression trees will be cleared and the new expression tree will be stored for later instructions.

A diagram of a tree structure

Description automatically generated

The diagram shows the order of calls chronologically along with what is passed to where and what is returned.

Returning to the “2+3” example:

Once set\_instruction(“2+3”) is called, then the intermediary respective method in the control unit is called with the same parameter.

The control unit then passes “2+3” to the tokenizer which returns the tokens 2, +, 3.

Then the tokens gets passed by the control unit to the parser which returns the traversable expression tree to the control unit.

Once the control unit has the tree, it will first clear out the previous expression trees and then stores the expression tree from “2+3”.

### send\_visitor

“send\_visitor” method provides the option for customization when it comes to the display of the expression entered prior in “set\_instruction”. This is achieved using the aforementioned Visitor pattern in the introduction by the customization being conveyed through a Visitor object. When send\_visitor is called with a visitor, it is received by the control unit, which then uses the visitor to visit each of the stored expression trees, which will apply the pattern. Once the results are in from each of the expression trees, the control unit will then return the result to the interface.

A diagram of a user interface

Description automatically generated

This diagram shows the control flow when “send\_visitor” is called by the Interface.

The following pseudocode conveys the same idea as the diagram.

Subroutine Interface.send\_visitor( visitor )

Call and Return Control Unit.get\_lines\_visited( visitor )

Subroutine Control Unit.get\_lines\_visited( visitor )

Call and Return Traversable Tree Object.visit( visitor )

Subroutine Traversable Tree Object.visit( visitor )

If self is leaf node Then

Call and Return visitor.visited( self )

Set results to empty list

For each node in branches Then

Call and Add node.visit( visitor )

Call and Return Visitor.visited( self, results )

To borrow from the same example, where “2+3” has been loaded into the control unit:

Before the method can be called, the Visitor and the corresponding rules must be defined first:

If is a number

Return number

Else if is +

Return left + “+” + right

This is then passed in though the visitor object into the expression tree where it is first received by the node +.

The + node then passes the same Visitor object to its branches 2 and 3.

First the 2 node passes its contents to the visitor, which returns 2, the same will occur for 3.

Then the results of the two branches (as left and right) along with + are passed to the visitor by the + node.

The visitor then consults the rules and returns “2+3” which is then passed back to the control unit, then back to the interface.

### get\_calculated\_results

The control flow for the “get\_calculated\_results” method is very similar to the “send\_visitor” method, with the only difference being the Visitor object in this instance is a Calculation Unit object, supplied by the control unit.

The calculation unit stores the necessary information, stored in the Control Unit and with rules configured to return the mathematical result from the expression tree.

A screenshot of a computer

Description automatically generated

### get\_(setting/variables/functions)/set\_settings

The methods get\_variables, get\_functions, get\_settings, and set\_settings are all methods that act as portals to access the attributes/data stored within the program that could aid the programming of the interface. All requests are passed to the control unit which then itself requests to the calculation unit for the respective information. The results from the calculation unit are then returned to the control unit and then returned to the interface. The set\_settings follow the same procedure with the exception that the request is replaced with the details of what to set the setting to and there is no return after it has reached the calculation unit.

A diagram of a function

Description automatically generated

Diagram shows control flow when calling getters/setters.

## File layout and contents

The files of the program are separated in such a way that all components that could be independent are encapsulated within their own file.

The following diagram shows the dependances of each file in the program, this closely reflects the dependencies of each of the objects/subroutines.

The following shows what important classes/subroutines are in each file in the format `File -> class`. The & is used to shows multiple important classes in one file:

Controlunit.py -> controlunit & calculationunit

parser.py -> parser

lexer.py -> tokenizer

A diagram of a company

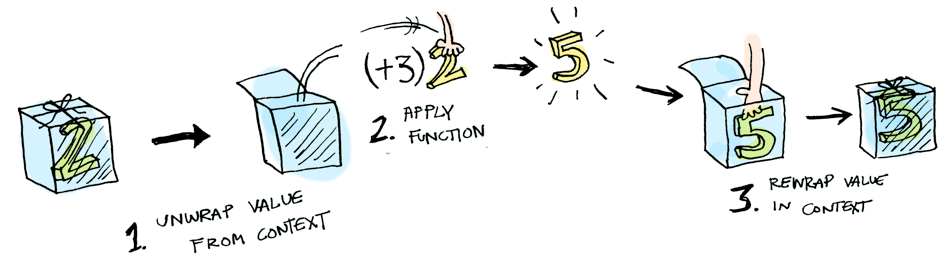
Description automatically generated

This serves to make it easier to test individual parts, as well as reducing the naming clashes that could result from multiple objects having the same token. This is achieved with the \_\_all\_\_ attribute being custom defined to hint to users what is to be imported from the file.

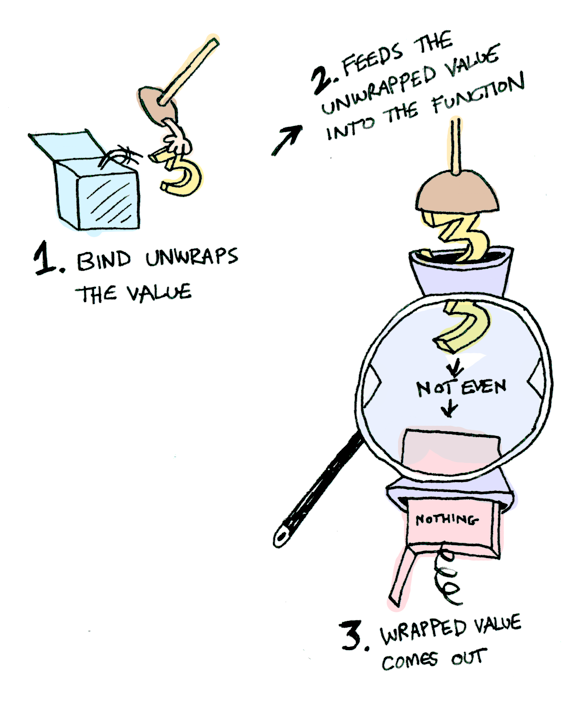
### monad.py

Monads are a type of objects that serve as wrappers for other values which also hosts methods that can apply other methods to the wrapped values. These methods are map, and bind.

The map method takes another function and returns the result of the function applied to the wrapped value wrapped again.

<https://www.adit.io/imgs/functors/fmap_just.png>

The bind method accepts a function that returns the same type of wrapper(monad) and returns the result of the value passed into the functions, or first applies the function to the value wrapped, then unwrapping the result.



<https://www.adit.io/imgs/functors/monad_just.png>

For more detailed information/breakdown, visit <https://www.adit.io/posts/2013-04-17-functors,_applicatives,_and_monads_in_pictures.html>.

The following class diagram shows what classes the file monad.py contains.

A diagram of a data flow

Description automatically generated with medium confidence

The monad.py class hosts only monads, which are types of classes which as explained above have a method, “map”, which can apply a method to its contents and returns the modified contents wrapped in the same class and another method, “bind”, which accepts a subroutine which returns a monad and passes the wrapped internal value and returns the value returned by the method. The implementation of monads also include other methods such as “Just” and “get\_internal\_implemntation” where “Just” acts like a constructor and creates the specific monad (Just(3) creates a monad wrapping the value 3) and “get\_internal\_implememtations” returns the values wrapped by the monad.

In monad.py, there are two monad classes/implememtations, Maybe and ListMonad.

#### Maybe

Maybe is a monad that has a specific implementation of bind and map when the wrapped value is none.

Maybe wraps any value or nothing. When there is no wrapped value, map and bind methods both return a Maybe object wrapping nothing. A white board with arrows and symbols

Description automatically generated

<https://www.adit.io/imgs/functors/fmap_nothing.png>

#### ListMonad

ListMonad is a monad, for lists, meaning it wraps a list of values.

A list of tasks with black text

Description automatically generated with medium confidence

ListMonad uses a singly linked list like implementation where the tail of the list is recursively stored as an attribute. The \*\* “\_\_pow\_\_” method has been overridden to emulate list concatenation in Haskell. There is also a method “flatmap”, which provides the functionality for bind, but concatenates the result from all the elements of the list into a new ListMonad.

### utlities.py

utilities.py hosts all of the miscellaneous subroutines and classes that are used across the program, these include: Wrapper class, JSONable class, NotCompatibleException Exception, Stack class, and unwrap\_dict subroutine, dict\_beautify subroutine, and test subroutine.

A screenshot of a computer

Description automatically generated

Note\* dict\_beautify and test subroutine are only used as part of testing and should not be called as part of the final program

#### Wrapper

The Wrapper class provides an implemented constructor and method to encapsulate and retrieve the encapsulated value.

#### JSONable

JSONable is an abstract base class containing a virtual abstract “get\_json” method, which when inherited by a class indicates that the inherited class has provided an implementation to the “get\_json” method that returns a dictionary object.

#### NotCompatibleException

The Exception is an error/ interrupt that is raised when a matching attempt has failed and is only raised and caught during parsing and tokenizing.

#### Stack

The Stack class represents a stack data structure. The stack is used in various points in the program with the most notable being parsing and evaluation of the expression tree. To further elaborate, a stack is used as part of the Dijkstra’s shunting yard algorithm during parsing and the call stack during evaluation by the calculation unit both of which closely model the features of the stack when storing and retrieving data.

#### unwrap\_dict()

The “unwrap\_dict” subroutine takes in a dictionary and a string value and if the value corresponding to the string key is a dictionary, replaces the passed in dictionary object with the dictionary referenced by the string.

A diagram of a flowchart

Description automatically generated

This works since in python, dictionaries are mutable and passed by reference, so any mutations to the dictionary within the subroutine will also mutate the same object from the caller.

The subroutine checks if the string taken is a key in the dictionary, if it is not, then a KeyError is raised.

Then it stores the contents of the value corresponding to the key into a temporary variable.

If the type of the value is not a dictionary, a TypeError is raised.

The original dictionary then has its contents cleared, after which the contents of the dictionary in the temporary variable are then copied over into the original dictionary.

To demonstrate it with an example original={“one”: 1, “two”: {“three”: 3}}, key=“two”:

First it checks if “two” is a key of the dictionary.

Then it assigns {“three”: 3} to a temporary variable.

Then it clears out original such that it becomes empty ({}).

Then it copies over each key value pair in the temporary variable back into the original.

Finally original will become {“three”: 3}, just like the nested dictionary as it was passed in.

#### dict\_beautify()

The “dict\_beautify” subroutine takes in a list, tuple, set, or dictionary, and returns a multiline string that is indented and represents the object passes in as well as the nested components like the JSON format. The subroutine is only called during testing and may be called by the user when they see fit.

A diagram of a flowchart

Description automatically generated

If the object passed in is not of the types dictionary, list, tuple, or set, a TypeError will be raised.

A lines ListMonad will be declared to represent the lines of the resultant string.

Then if it is a dictionary, it iterates through the dictionary and checks each value to see if dict\_beautify can be called recursively, if not, the default string value of the value object will be used, otherwise, dict\_beautify will be called, and the result split by line and added to the lines with the key string at the front.

Each line in lines then has an indent added to it, and the front and back added with a curly bracket

before transforming into a list and converted into a string joined by newlines.

To give an example, lets say that the dictionary {“3”: 5, “two”: [3,2,1]} is to be turned into a JSON like string using the dict\_beautify subroutine.

First it identifies the input as a dictionary.

Then it iterates though the key value pairs (“3”, 5), (“two”, [3,2,1]).

For each pair, it determines if the value (second object of the pair) can be called recursively. (No, Yes)

For (“3”: 5), it adds “”3”: 5,” to the lines.

For those that can be, it will be called recursively.

This returns:

[

3,

2,

1

]

for the list and the key is added to the first line, and comma to the last

“two”: [

3,

2,

1

],

And broken up and added to lines.

So lines now consists of:

“3”: 5,

“two”: [

3,

2,

1,

],

An indent is added to each line, comma on the last line removed, and { and } to the front and back of lines.

{

“3”: 5,

“two”: [

3,

2,

1,

]

}

Now lines are stitched back up with newline characters between each line into one string of characters and returned to the caller.

#### test()

“test” is a python decorator used on test subroutines in files and displays a message indicating that the following prints have been made as part of a test.

Test also accepts a string as an additional parameter to specify the test being carried out.

Note\* see example usage in unwrap\_dict\_test subroutine in utilities.py

### mathobj.py

A screenshot of a computer

Description automatically generated

The mathobj.py is where objects that are operated on are defined. These data types are Number and NPArray, which are wrapper classes that encapsulate a float and numpy.array object types respectively, and the Undefined data type. This file also includes the TrigMode enum definition which is used in the trigonometrc methods of the datatypes(sin, cos, tan, arcsin, arcos, arctan).

### lexer.py

lexer.py is the location where the subroutines and classes that support the tokenizing of sequences of characters into a sequence of tokens that more concisely convey the information encoded within the string with respect to the language defined for the parser. This is done right after the user has entered in the instruction into the UI and before it is parsed. The subroutine that controls the tokenizing is in the “tokenize” subroutine.

A screenshot of a computer

Description automatically generated

There are 4 main components in lexer.py, the abstract base class Token, all implementations of Token, ScanError Exception, and the tokenize subroutine.

\* Note: all the subclasses of Token are not shown in the class diagram as they are all very similar with only minor adjustments to reflect the token they are representing. To show all the subclasses would be a superfluous endeavor and render the diagram moot.

#### Token

The Token abstract base class specifies that all inherited subclasses have a consume class method and get\_json instance method.

##### consume()

accepts string as the argument and returns an instance of itself and the remainder of the string.

##### get\_json()

A method that returns a dictionary object representing the token instance

##### \_\_eq\_\_()

Checks for equality between methods.

#### implementations of Token

The list of classes that are subclasses of Token are: EOL, OpenBracket, ClosedBracket, Comma, ComparisonOperator, Operator, Conditional Operator, Eqiality, NameSpace, and Number.

All the classes above have a unique implementation of each of the methods in Token.

#### ScanError

ScanError is raised by the “tokenize” subroutine when the initial string sequence passed in cannot be completely turned into tokens.

#### tokenize()

The tokenize subroutine takes in a string of characters and returns those characters mapped to a list of specific tokens carrying the same information.

For details on the implementation of the tokenize subroutine, please go to the tokenizer section of Algorithms on p42.

### parser.py

The parser.py is where the parse subroutine is defined along with any classes/ object type that is only involved in creation of the parsed syntax tree. Parsing is done after the user input has been tokenized and the resultant expression tree will be made traversable before being stored and evaluated.

A screenshot of a computer

Description automatically generated

parser.py includes the Node class, all of its subclasses, ParseError, and the “parse” subroutine.

\* Note: all the subclasses of Node are not shown in the class diagram as they are all very similar with only minor adjustments to reflect the node they are representing. To show all the subclasses would be a superfluous endeavor and render the diagram moot.

#### Node

Node defines the methods all Nodes of the syntax tree should have implemented, a class method “consume” and “get\_json”.

##### consume()

accepts token sequence as argument, then matches to the implemented Node’s rules, if the implemented Node is defined in terms of other Nodes, then call the respective implementations consume method.

If all definitions have raised a NotCompatibleException, then raise NotCompatibleException.

If a definition does not raise a NotCompatibleException, initiate an instance of the Node and return the remainder of the token sequence.

To give an analogy, when describing an object, as you list off its properties, the listener will slowly form an image of what the object looks like and what it could be. In the case for “consume”, it is given an object and must check itself what the object’s properties are. It can do this itself, but it could also “ask” (call) other nodes utilizing their “consume” method to construct what the object (in this case, expression tree) looks like from a 1-dimensional view (list of tokens).

##### get\_json

returns a dictionary representation of the Node.

#### Implementations of Node

Implementations of node are as defined by the Backus Naur Form(BNF) of the language prior.

#### ParseError

An exception raised by the parse subroutine if the token sequence cannot be matched to any of the rules of “Lines” Node

#### parse()

the parse subroutine calls the Lines.consume method to see if the token sequence could be matched to any possible lines representation, and if a NotCompatibleException is raised, the exception is caught and ParseError is raised instead to distinguish the source of the incompatibility as originating from during parsing.

### controlunit.py

The controlunit.py hosts 5 components of the program, visitor abstract base class, the traversable tree (SubTree and its subclasses), the abstract base class for Interfaces (UserInterfaceInterface), the control unit, and the calculation unit.

A screenshot of a computer program

Description automatically generated

#### Visitor

Visitor is an abstract base class that dictates that all subclasses must implement the method visited for Visitee instances’ visit method. Ie that it can be used as part of the visitor behavioral pattern.

#### SubTree & its subclasses

A diagram of a diagram

Description automatically generated

The SubTree class inherits from Visitee and provides a constructor that instantiates the \_branches variable representing the branches, and a visit method that enables traversal of the data structure using visitor classes.

The following diagram shows what are the implementations of the nodes. The nodes that are not connected by other nodes (**leaf node**) are the nodes that contain the implementations and could be part of the expression tree.

A black and white image of a plane

Description automatically generated with medium confidence

To give an explanation:

All nodes are subclasses of SubTrees as all nodes can be traced upwards to the class SubTree.

Assignment is a node that could appear in an expression tree as it is a leaf node.

Expression, Inequality, and Conditional are all Operands, whereas Variable is not an Operand.

\*Simplified inheritance diagram of all SubTree classes as a full class diagram would contain too much information and become unreadable and meaningless.

#### CalculationUnit

CalculationUnit is an datatype that inherits the interface Visitor and visits SubTree objects to interpret the SubTree structure into mathematical operations in the correct sequence. The

CalculationUnit object also stores the setting, variables, and functions defined during the runtime of the program.

The calculationUnit is composed to a ControlUnit object

#### ControlUnit

The ControlUnit serves to control access and encapsulate operations on SubTree objects, from creation, to traversal, and interpretation. The ControlUnit also encapsulates the CalculationUnit and provides getters and setters to certain attributes in the CalculationUnit to the UserInterface.

#### UserInterfaceInterface

UserInterfaceInterface is a abstract base class that encapsulates the control unit as well as providing a default implementation for getting methods in subclasses for the ControlUnit. The UserInterfaceInterface also the interface that all subclasses must implement, such as the case with the “alert\_improper\_input”.

## Data Structures

### Stack

The stack is a First In Last Out(FILO) data structure that operates similarly to an array, but access to the data stored is only limited to the top of the data structure, with push pop and peek, performing adding removing and accessing on the element last added to the structure.

### Rooted Tree (SubTree)

The rooted tree is used to store nested recursive data structures. This is achieved using objects called nodes. In a node, it stores references to any number of nodes. In a rooted tree they are linked in such a way that no circular references are formed and there is one node that is not referenced by any other node, this is called the root node. This is evident in the case of a mathematical equation, where it could be composed of sub-equations and functions and operators which are themselves composed of branches.

Eg “a=f(2,1)+3”

A diagram of a diagram

Description automatically generated

## Algorithms

### Tokenizer

The tokenizer algorithm maps a string input into a sequence of tokens. This is done by iteratively checking each token’s rules\* with the start of the string.

If a token’s rule is found to fit the start of the string, the respective segment is spliced out of the string an the respective token is appended to the token sequence.

If none of the token’s rules fit the start of the string sequence, then the algorithm is done and returns the token sequence and the remainder of the inputted string.

The diagram and subroutine below demonstrate the specific instructions/steps taken by the tokenizer to lexically analyze the input into a list of tokens.

A diagram of a process

Description automatically generated

Subroutine tokenize (string)

Set string to filtered string

Set tokens to empty list

While string is not empty then

For each token type

Call token type.consume(string)

If call does not raise NotCompatableException then

Add result from call to tokens

Set string to call result

Continue while loop

Else

Continue for loop

Raise ScanError

Return tokens

To run through a working example “\2+3;]”:

First the tokenizer will filter out all unrecognized characters yielding “2+3;”.

Then it will consult each token type for a match.

Number will give an indication of a match and return Number(2) with the remainder of the input “+3;”.

This repeats for Operator, Number, and EOL Tokens, returning (Operator(+), “3;”), (Number(3), “;”), and (EOL(), “”) respectively.

Once the input has been exhausted, it will stop consulting Tokens and return the sequence of Tokens in order in a list.

Now for a faulty input “2..3”:

The input has no illegal characters so it stays the same.

Then Number is recognized returning Number(2), “..3”.

For this sequence, no token recognizes it and hence a ScanError will be raised.

Note\*: the tokens are chosen specifically such that there is only one token that can fit a sequence of characters

### Parser

The parser algorithm creates nodes from a sequence of tokens using a method similar to the decreasing bin fitting algorithm. It tries to fit the token sequence first into the most complex/specific rule for the Lines node, where each composed node’s rule then recursively checks if the token sequence also matches its most complex rule, before checking the next strictest rule.

This is done recursively for all possible rules until a rule is found or all rules are exhausted.

When a rule is found to match a node and its composed nodes, a node object will be created from the information encoded within the tokens.

Program parse into node

For node rules

if token sequence fit node rule Then

return node

Raise NotCompatibleException

To give an example, with the following tokens [Number, Operator, NameSpace, OpenBracket, Number, ClosedBracket, EOL] (which represents some form of a+b(c) expression, where b is a function)

The list of tokens will be passed into lines to determine if the sequence is valid.

During which it will be passed to line (as lines is composed of lines) to determine if the sequence is valid.

Line then passes the sequence to assignment first, as it is the most complex route, to determine if the sequence is valid.

Assignment then passes the sequence to Assignable to determine if the sequence is valid.

Assignment then determines that the sequence is not valid as the first token is not a NameSpace, afterwhich it indicates to assignment that the sequence is not valid.

Assignment then raises the same indication to line which then passes the same sequence to expression.

Expression passes the sequence to term which as the BNF defined, Number is a valid term.

Term then removes the Number Token from the sequence and creates a new term with the number it had removed and returns both the new Term and the sequence to Expression.

Sequence: [Operator, NameSpace, OpenBracket, Number, ClosedBracket, EOL]

Expression then stores the Term into a list.

Expression then notes there is an Operator in the front of the sequence and removes it into the list with the Term.

Expression then passes the sequence back to Term, which notes the front is not a Number and passes to Function.

Function then recognizes the first two tokens as NameSpace and OpenBracket, which then removes them from the sequence.

Sequence: [Number, ClosedBracket, EOL]

Function then passes the sequence to Operands, which passes to Operand.

To simplify, Conditional, and Inequality were both called, and both indicated the sequence was not valid for them.

This leaves Expression, and to skip some details as the calling of Expression is already being covered, the first term is stored, then Expression does not recognize the first Token of the sequence afterwards, so it finishes checking.

After Expression finishes validating, it creates an Expression object and returns it to Operand, which using the Expression, creates an Operand, and returns that to Operands.

Sequence: [ClosedBracket, EOL]

Operands then stores the Operand in a list and repeats the same operation with Operand, which will indicate the sequence does not start with a valid Operand.

Operands will then create a new Operands object using the stored list and return that to Function.

Function will then see that the start of the sequence is a ClosedBracket which is valid and take the Operands along with the initial NameSpace and create a Function object and pass it back to Expression.

Sequence: [EOL]

Expression will store the Function object into the list with the Number and Operation.

Expression will then note that the sequence does not start with an Operation, and create and return a new Expression Object from the stored list.

The Expression is received by Line which notes the sequence starts with EOL and confirms the sequence is valid and creates and returns a new Line object to Lines.

Sequence: []

Lines stores the Line into a list, then it “sees” that the sequence is now empty, which prompts it to return a new Lines object from the stores list.

Lines is returned to the parse subroutine and the structure of the expression is now embedded into the Lines object.

### Dijkstra’s shunting yard algorithm

This algorithm is used to correctly determine the operator precedence while translating a 1-dimensional array of terms and operators into a tree structure.

The algorithm is called/used during the reconstruction of the expression tree to make it traversable where it takes the stores list of Terms and Operations and converts them into an expression tree.

Set operator stack to empty stack

Set result stack to empty stack

For items in expression sequence

If item is a term Then

Add to result stack

Else

While item precedence is less than operator stack top precedence

Apply top two items in result stack using the stack’s top operator

Add result to top of result stack

Remove top of operator stack

While operator stack is not empty

Apply top two items in result stack using the stack’s top operator

Add result to top of result stack

Remove top of operator stack

Peek/pop and Return result from result stack

Note\*: brackets are accounted for during parsing

\*\* This implementation assumes that the items in the expression sequence are in correct order and no checking is required, therefore there should be 1 element remaining in in the result stack to peek and return

To give an example of the algorithm in action, let’s process 8+2\*3+6.

You would expect the tree to look something like:

A diagram of a tree

Description automatically generated

This expression tree takes into account the higher precedence of the multiply operation and the left to right precedence of the addition operators.

Now let’s get back to the algorithm at hand, the worded expression would first be broken up into a list of terms and operators, like: [8, +, 2, \* 3, +, 6]

Then two stacks are created, one for the “result”, and another for operators.

Result: []

Operators: []

It first stores the 8 onto the result stack.

Result: [8]

Operators: []

Then the precedence of the + is compared with the top of the operators stack, since there is nothing in the operator stack, + is pushed onto the operator stack

Result: [8]

Operators: [+]

Then the 2 is pushed onto the result stack

Result: [8, 2]

Operators: [+]

Then the \* precedence is compared with the precedence of the top of the operator stack, +.

Since \* has a higher precedence than +, \* is pushed onto the operator stack.

Result: [8, 2]

Operators: [+, \*]

Then the 3 is pushed onto the result stack

Result: [8, 2, 3]

Operators: [+, \*]

Then the precedence of + is compared with the precedence of the top of the operator stack, \*.

Since the precedence of + is lower or equal to \*, the operator stack is popped and the top two items in the result stack is attached to the \*, and the result is pushed back onto the result stack.

Result: [8, A black and white diagram

Description automatically generated]

Operators: [+]

The same is repeated for the + in the Operators stack, leaving:

Result: [A diagram of a network

Description automatically generated]

Operators: []

After the Operators stack is emptied or there is a operator with a lower precedence at the top of the operator stack, + is pushed onto the operator stack

Result: [A diagram of a network

Description automatically generated, 6]

Operators: [+]

Then the 6 is pushed onto the result stack

Once the list of terms and operators are empty, the operator stack is emptied again, resulting in:

A diagram of a tree

Description automatically generated

Which is exactly what we have come to expect.

### Tree traversal algorithm (Visitor pattern)

The method/ pattern used to traverse the syntax tree generated is the Visitor pattern. The Visitor pattern is used to traverse a tree using two kinds of classes, a “Visitor”(one that visits) and a “Visitee”(one that is visited). The Visitor class dictates the rules for what occurs when a visitee is visited in the method “visited”, where the visitee is passed in as an argument. For all nodes(Visitee) there is a method called “visit” where the visitor is passed in and its visited method is called, along with recursively calling visit on its sub-nodes with the visitor, and passing the results into the visited method as well.

Class Visitor

Method visited (Visitee, arguments, …)

Return call operation visitee

Class Node

Method visit (Visitor)

Set results to empty array

For branches of Node

Add branches.visit(Node) to results

Return call Node.visited(Node, results)

This results in a postfix traversal of the tree.

\*There is a variation(TwoStepVisitor) which enables prefix traversal of the tree through the use of generators.

# Testing (automated regression testing)

I have coded my own automated regression testing for the testing of the program requirements. The automated nature of testing enables continuous testing to not be an exhaustive process when testing if a different/ new feature has effected prior features.

## Non-client required testing

### Lexer tests

Lexer Test Results:

Test Result | Input | Expected | Actual |

(Pass/fail) | | Outcome | Outcome |

-------------+-----------+-----------------------+-----------------------+

Pass |"" |[] |[] |

Pass |"\n" |['EOL'] |['EOL'] |

Pass |";" |['EOL'] |['EOL'] |

Pass |"(" |['OpenBracket'] |['OpenBracket'] |

Pass |")" |['ClosedBracket'] |['ClosedBracket'] |

Pass |"," |['Comma'] |['Comma'] |

Pass |"<" |['ComparisonOperator'] |['ComparisonOperator'] |

Pass |">" |['ComparisonOperator'] |['ComparisonOperator'] |

Pass |"<=" |['ComparisonOperator'] |['ComparisonOperator'] |

Pass |">=" |['ComparisonOperator'] |['ComparisonOperator'] |

Pass |"==" |['ComparisonOperator'] |['ComparisonOperator'] |

Pass |"+" |['Operator'] |['Operator'] |

Pass |"-" |['Operator'] |['Operator'] |

Pass |"\*" |['Operator'] |['Operator'] |

Pass |"/" |['Operator'] |['Operator'] |

Pass |"^" |['Operator'] |['Operator'] |

Pass |"|" |['ConditionalOperator']|['ConditionalOperator']|

Pass |"=" |['Equality'] |['Equality'] |

Pass |"~" |['Equality'] |['Equality'] |

Pass |"a" |['NameSpace'] |['NameSpace'] |

Pass |"letter" |['NameSpace'] |['NameSpace'] |

Pass |"a1" |['NameSpace'] |['NameSpace'] |

Pass |"ae3" |['NameSpace'] |['NameSpace'] |

Pass |"ae31" |['NameSpace'] |['NameSpace'] |

Pass |"p3I" |['NameSpace'] |['NameSpace'] |

Pass |"e7p8" |['NameSpace'] |['NameSpace'] |

Pass |"0" |['Number'] |['Number'] |

Pass |"123" |['Number'] |['Number'] |

Pass |"2.4" |['Number'] |['Number'] |

Pass |"2.44" |['Number'] |['Number'] |

Pass |"22.4" |['Number'] |['Number'] |

Pass |"62.42" |['Number'] |['Number'] |

Pass |".429r4" |ScanError |ScanError the given input feed ".429r4" cannot be matched to any defined token|

Pass |"84889...."|ScanError |ScanError the given input feed ".24" cannot be matched to any defined token|

Pass |"2944r8..."|ScanError |ScanError the given input feed ".32r." cannot be matched to any defined token|

Pass |"|)(764..."|['ConditionalOperator', 'ClosedBracket', 'OpenBracket', 'Number', 'Operator', 'Comma', 'ComparisonOperator', 'Operator', 'Number', 'NameSpace', 'Equality', 'Operator', 'EOL', 'Number', 'Operator', 'NameSpace', 'EOL', 'NameSpace']|['ConditionalOperator', 'ClosedBracket', 'OpenBracket', 'Number', 'Operator', 'Comma', 'ComparisonOperator', 'Operator', 'Number', 'NameSpace', 'Equality', 'Operator', 'EOL', 'Number', 'Operator', 'NameSpace', 'EOL', 'NameSpace']|

\* code for tests can be found in lexer.py

### Parser tests

Parser Test Results:  
Test Result | Type Tested | Input | Expected | Actual |

(Pass/fail) | | | Outcome | Outcome |

-------------+---------------+-------------------+-----------------------------------------------+-----------------------------------------------+

Pass | Term | "3" | { | { |

| | | "Type": "Term", | "Type": "Term", |

| | | "Content": { | "Content": { |

| | | "Type": "Number", | "Type": "Number", |

| | | "Value": "3" | "Value": "3" |

| | | } | } |

| | | } | } |

-------------+---------------+-------------------+-----------------------------------------------+-----------------------------------------------+

Pass | Term | "a" | { | { |

| | | "Type": "Term", | "Type": "Term", |

| | | "Content": { | "Content": { |

| | | "Type": "NameSpace", | "Type": "NameSpace", |

| | | "Name": "a" | "Name": "a" |

| | | } | } |

| | | } | } |

-------------+---------------+-------------------+-----------------------------------------------+-----------------------------------------------+

Pass | Term | "-3" | { | { |

| | | "Type": "NegatedTerm", | "Type": "NegatedTerm", |

| | | "Content": { | "Content": { |

| | | "Type": "Number", | "Type": "Number", |

| | | "Value": "3" | "Value": "3" |

| | | } | } |

| | | } | } |

-------------+---------------+-------------------+-----------------------------------------------+-----------------------------------------------+

Pass | Term | "--3" | { | { |

| | | "Type": "Term", | "Type": "Term", |

| | | "Content": { | "Content": { |

| | | "Type": "Number", | "Type": "Number", |

| | | "Value": "3" | "Value": "3" |

| | | } | } |

| | | } | } |

-------------+---------------+-------------------+-----------------------------------------------+-----------------------------------------------+

Pass | Term | "+3" | NotCompatibleException | NotCompatibleException |

| | | | |

-------------+---------------+-------------------+-----------------------------------------------+-----------------------------------------------+

Pass | Inequality | "4<2" | { | { |

| | | "Type": "Inequality", | "Type": "Inequality", |

| | | "Terms": [ | "Terms": [ |

| | | { | { |

| | | "Type": "Term", | "Type": "Term", |

| | | "Content": { | "Content": { |

| | | "Type": "Number", | "Type": "Number", |

| | | "Value": "4" | "Value": "4" |

| | | } | } |

| | | }, | }, |

| | | { | { |

| | | "Type": "Term", | "Type": "Term", |

| | | "Content": { | "Content": { |

| | | "Type": "Number", | "Type": "Number", |

| | | "Value": "2" | "Value": "2" |

| | | } | } |

| | | } | } |

| | | ], | ], |

| | | "Comparisons": [ | "Comparisons": [ |

| | | { | { |

| | | "Type": "ComparisonOperator", | "Type": "ComparisonOperator", |

| | | "ComparisonType": "<" | "ComparisonType": "<" |

| | | } | } |

| | | ] | ] |

| | | } | } |

-------------+---------------+-------------------+-----------------------------------------------+-----------------------------------------------+

Pass | Inequality | "a>=3.4" | { | { |

| | | "Type": "Inequality", | "Type": "Inequality", |

| | | "Terms": [ | "Terms": [ |

| | | { | { |

| | | "Type": "Term", | "Type": "Term", |

| | | "Content": { | "Content": { |

| | | "Type": "NameSpace", | "Type": "NameSpace", |

| | | "Name": "a" | "Name": "a" |

| | | } | } |

| | | }, | }, |

| | | { | { |

| | | "Type": "Term", | "Type": "Term", |

| | | "Content": { | "Content": { |

| | | "Type": "Number", | "Type": "Number", |

| | | "Value": "3.4" | "Value": "3.4" |

| | | } | } |

| | | } | } |

| | | ], | ], |

| | | "Comparisons": [ | "Comparisons": [ |

| | | { | { |

| | | "Type": "ComparisonOperator", | "Type": "ComparisonOperator", |

| | | "ComparisonType": ">=" | "ComparisonType": ">=" |

| | | } | } |

| | | ] | ] |

| | | } | } |

-------------+---------------+-------------------+-----------------------------------------------+-----------------------------------------------+

Pass | Inequality | "4==2>o" | { | { |

| | | "Type": "Inequality", | "Type": "Inequality", |

| | | "Terms": [ | "Terms": [ |

| | | { | { |

| | | "Type": "Term", | "Type": "Term", |

| | | "Content": { | "Content": { |

| | | "Type": "Number", | "Type": "Number", |

| | | "Value": "4" | "Value": "4" |

| | | } | } |

| | | }, | }, |

| | | { | { |

| | | "Type": "Term", | "Type": "Term", |

| | | "Content": { | "Content": { |

| | | "Type": "Number", | "Type": "Number", |

| | | "Value": "2" | "Value": "2" |

| | | } | } |

| | | }, | }, |

| | | { | { |

| | | "Type": "Term", | "Type": "Term", |

| | | "Content": { | "Content": { |

| | | "Type": "NameSpace", | "Type": "NameSpace", |

| | | "Name": "o" | "Name": "o" |

| | | } | } |

| | | } | } |

| | | ], | ], |

| | | "Comparisons": [ | "Comparisons": [ |

| | | { | { |

| | | "Type": "ComparisonOperator", | "Type": "ComparisonOperator", |

| | | "ComparisonType": "==" | "ComparisonType": "==" |

| | | }, | }, |

| | | { | { |

| | | "Type": "ComparisonOperator", | "Type": "ComparisonOperator", |

| | | "ComparisonType": ">" | "ComparisonType": ">" |

| | | } | } |

| | | ] | ] |

| | | } | } |

-------------+---------------+-------------------+-----------------------------------------------+-----------------------------------------------+

Pass | Conditional | "X>3|Y<=2" | { | { |

| | | "Type": "Conditional", | "Type": "Conditional", |

| | | "Conditions": [ | "Conditions": [ |

| | | { | { |

| | | "Type": "Inequality", | "Type": "Inequality", |

| | | "Terms": [ | "Terms": [ |

| | | { | { |

| | | "Type": "Term", | "Type": "Term", |

| | | "Content": { | "Content": { |

| | | "Type": "NameSpace", | "Type": "NameSpace", |

| | | "Name": "X" | "Name": "X" |

| | | } | } |

| | | }, | }, |

| | | { | { |

| | | "Type": "Term", | "Type": "Term", |

| | | "Content": { | "Content": { |

| | | "Type": "Number", | "Type": "Number", |

| | | "Value": "3" | "Value": "3" |

| | | } | } |

| | | } | } |

| | | ], | ], |

| | | "Comparisons": [ | "Comparisons": [ |

| | | { | { |

| | | "Type": "ComparisonOperator", | "Type": "ComparisonOperator", |

| | | "ComparisonType": ">" | "ComparisonType": ">" |

| | | } | } |

| | | ] | ] |

| | | }, | }, |

| | | { | { |

| | | "Type": "Inequality", | "Type": "Inequality", |

| | | "Terms": [ | "Terms": [ |

| | | { | { |

| | | "Type": "Term", | "Type": "Term", |

| | | "Content": { | "Content": { |

| | | "Type": "NameSpace", | "Type": "NameSpace", |

| | | "Name": "Y" | "Name": "Y" |

| | | } | } |

| | | }, | }, |

| | | { | { |

| | | "Type": "Term", | "Type": "Term", |

| | | "Content": { | "Content": { |

| | | "Type": "Number", | "Type": "Number", |

| | | "Value": "2" | "Value": "2" |

| | | } | } |

| | | } | } |

| | | ], | ], |

| | | "Comparisons": [ | "Comparisons": [ |

| | | { | { |

| | | "Type": "ComparisonOperator", | "Type": "ComparisonOperator", |

| | | "ComparisonType": "<=" | "ComparisonType": "<=" |

| | | } | } |

| | | ] | ] |

| | | } | } |

| | | ] | ] |

| | | } | } |

-------------+---------------+-------------------+-----------------------------------------------+-----------------------------------------------+

Pass | Operand | "a>=3.4" | { | { |

| | | "Type": "Operand", | "Type": "Operand", |

| | | "Operand": { | "Operand": { |

| | | "Type": "Inequality", | "Type": "Inequality", |

| | | "Terms": [ | "Terms": [ |

| | | { | { |

| | | "Type": "Term", | "Type": "Term", |

| | | "Content": { | "Content": { |

| | | "Type": "NameSpace", | "Type": "NameSpace", |

| | | "Name": "a" | "Name": "a" |

| | | } | } |

| | | }, | }, |

| | | { | { |

| | | "Type": "Term", | "Type": "Term", |

| | | "Content": { | "Content": { |

| | | "Type": "Number", | "Type": "Number", |

| | | "Value": "3.4" | "Value": "3.4" |

| | | } | } |

| | | } | } |

| | | ], | ], |

| | | "Comparisons": [ | "Comparisons": [ |

| | | { | { |

| | | "Type": "ComparisonOperator", | "Type": "ComparisonOperator", |

| | | "ComparisonType": ">=" | "ComparisonType": ">=" |

| | | } | } |

| | | ] | ] |

| | | } | } |

| | | } | } |

-------------+---------------+-------------------+-----------------------------------------------+-----------------------------------------------+

Pass | Operand | "X>3|Y<=2" | { | { |

| | | "Type": "Operand", | "Type": "Operand", |

| | | "Operand": { | "Operand": { |

| | | "Type": "Conditional", | "Type": "Conditional", |

| | | "Conditions": [ | "Conditions": [ |

| | | { | { |

| | | "Type": "Inequality", | "Type": "Inequality", |

| | | "Terms": [ | "Terms": [ |

| | | { | { |

| | | "Type": "Term", | "Type": "Term", |

| | | "Content": { | "Content": { |

| | | "Type": "NameSpace", | "Type": "NameSpace", |

| | | "Name": "X" | "Name": "X" |

| | | } | } |

| | | }, | }, |

| | | { | { |

| | | "Type": "Term", | "Type": "Term", |

| | | "Content": { | "Content": { |

| | | "Type": "Number", | "Type": "Number", |

| | | "Value": "3" | "Value": "3" |

| | | } | } |

| | | } | } |

| | | ], | ], |

| | | "Comparisons": [ | "Comparisons": [ |

| | | { | { |

| | | "Type": "ComparisonOperator", | "Type": "ComparisonOperator", |

| | | "ComparisonType": ">" | "ComparisonType": ">" |

| | | } | } |

| | | ] | ] |

| | | }, | }, |

| | | { | { |

| | | "Type": "Inequality", | "Type": "Inequality", |

| | | "Terms": [ | "Terms": [ |

| | | { | { |

| | | "Type": "Term", | "Type": "Term", |

| | | "Content": { | "Content": { |

| | | "Type": "NameSpace", | "Type": "NameSpace", |

| | | "Name": "Y" | "Name": "Y" |

| | | } | } |

| | | }, | }, |

| | | { | { |

| | | "Type": "Term", | "Type": "Term", |

| | | "Content": { | "Content": { |

| | | "Type": "Number", | "Type": "Number", |

| | | "Value": "2" | "Value": "2" |

| | | } | } |

| | | } | } |

| | | ], | ], |

| | | "Comparisons": [ | "Comparisons": [ |

| | | { | { |

| | | "Type": "ComparisonOperator", | "Type": "ComparisonOperator", |

| | | "ComparisonType": "<=" | "ComparisonType": "<=" |

| | | } | } |

| | | ] | ] |

| | | } | } |

| | | ] | ] |

| | | } | } |

| | | } | } |

-------------+---------------+-------------------+-----------------------------------------------+-----------------------------------------------+

Pass | Operands | "" | { | { |

| | | "Type": "Operands", | "Type": "Operands", |

| | | "Operands": [ | "Operands": [ |

| | | | |

| | | ] | ] |

| | | } | } |

-------------+---------------+-------------------+-----------------------------------------------+-----------------------------------------------+

Pass | Operands | "a>=3.4" | { | { |

| | | "Type": "Operands", | "Type": "Operands", |

| | | "Operands": [ | "Operands": [ |

| | | { | { |

| | | "Type": "Operand", | "Type": "Operand", |

| | | "Operand": { | "Operand": { |

| | | "Type": "Inequality", | "Type": "Inequality", |

| | | "Terms": [ | "Terms": [ |

| | | { | { |

| | | "Type": "Term", | "Type": "Term", |

| | | "Content": { | "Content": { |

| | | "Type": "NameSpace", | "Type": "NameSpace", |

| | | "Name": "a" | "Name": "a" |

| | | } | } |

| | | }, | }, |

| | | { | { |

| | | "Type": "Term", | "Type": "Term", |

| | | "Content": { | "Content": { |

| | | "Type": "Number", | "Type": "Number", |

| | | "Value": "3.4" | "Value": "3.4" |

| | | } | } |

| | | } | } |

| | | ], | ], |

| | | "Comparisons": [ | "Comparisons": [ |

| | | { | { |

| | | "Type": "ComparisonOperator", | "Type": "ComparisonOperator", |

| | | "ComparisonType": ">=" | "ComparisonType": ">=" |

| | | } | } |

| | | ] | ] |

| | | } | } |

| | | } | } |

| | | ] | ] |

| | | } | } |

-------------+---------------+-------------------+-----------------------------------------------+-----------------------------------------------+

Pass | Operands | "a>=3.4," | { | { |

| | | "Type": "Operands", | "Type": "Operands", |

| | | "Operands": [ | "Operands": [ |

| | | { | { |

| | | "Type": "Operand", | "Type": "Operand", |

| | | "Operand": { | "Operand": { |

| | | "Type": "Inequality", | "Type": "Inequality", |

| | | "Terms": [ | "Terms": [ |

| | | { | { |

| | | "Type": "Term", | "Type": "Term", |

| | | "Content": { | "Content": { |

| | | "Type": "NameSpace", | "Type": "NameSpace", |

| | | "Name": "a" | "Name": "a" |

| | | } | } |

| | | }, | }, |

| | | { | { |

| | | "Type": "Term", | "Type": "Term", |

| | | "Content": { | "Content": { |

| | | "Type": "Number", | "Type": "Number", |

| | | "Value": "3.4" | "Value": "3.4" |

| | | } | } |

| | | } | } |

| | | ], | ], |

| | | "Comparisons": [ | "Comparisons": [ |

| | | { | { |

| | | "Type": "ComparisonOperator", | "Type": "ComparisonOperator", |

| | | "ComparisonType": ">=" | "ComparisonType": ">=" |

| | | } | } |

| | | ] | ] |

| | | } | } |

| | | } | } |

| | | ] | ] |

| | | } | } |

-------------+---------------+-------------------+-----------------------------------------------+-----------------------------------------------+

Pass | Operands | "X>3|Y<=2,a>=3.4" | { | { |

| | | "Type": "Operands", | "Type": "Operands", |

| | | "Operands": [ | "Operands": [ |

| | | { | { |

| | | "Type": "Operand", | "Type": "Operand", |

| | | "Operand": { | "Operand": { |

| | | "Type": "Conditional", | "Type": "Conditional", |

| | | "Conditions": [ | "Conditions": [ |

| | | { | { |

| | | "Type": "Inequality", | "Type": "Inequality", |

| | | "Terms": [ | "Terms": [ |

| | | { | { |

| | | "Type": "Term", | "Type": "Term", |

| | | "Content": { | "Content": { |

| | | "Type": "NameSpace", | "Type": "NameSpace", |

| | | "Name": "X" | "Name": "X" |

| | | } | } |

| | | }, | }, |

| | | { | { |

| | | "Type": "Term", | "Type": "Term", |

| | | "Content": { | "Content": { |

| | | "Type": "Number", | "Type": "Number", |

| | | "Value": "3" | "Value": "3" |

| | | } | } |

| | | } | } |

| | | ], | ], |

| | | "Comparisons": [ | "Comparisons": [ |

| | | { | { |

| | | "Type": "ComparisonOperator", | "Type": "ComparisonOperator", |

| | | "ComparisonType": ">" | "ComparisonType": ">" |

| | | } | } |

| | | ] | ] |

| | | }, | }, |

| | | { | { |

| | | "Type": "Inequality", | "Type": "Inequality", |

| | | "Terms": [ | "Terms": [ |

| | | { | { |

| | | "Type": "Term", | "Type": "Term", |

| | | "Content": { | "Content": { |

| | | "Type": "NameSpace", | "Type": "NameSpace", |

| | | "Name": "Y" | "Name": "Y" |

| | | } | } |

| | | }, | }, |

| | | { | { |

| | | "Type": "Term", | "Type": "Term", |

| | | "Content": { | "Content": { |

| | | "Type": "Number", | "Type": "Number", |

| | | "Value": "2" | "Value": "2" |

| | | } | } |

| | | } | } |

| | | ], | ], |

| | | "Comparisons": [ | "Comparisons": [ |

| | | { | { |

| | | "Type": "ComparisonOperator", | "Type": "ComparisonOperator", |

| | | "ComparisonType": "<=" | "ComparisonType": "<=" |

| | | } | } |

| | | ] | ] |

| | | } | } |

| | | ] | ] |

| | | } | } |

| | | }, | }, |

| | | { | { |

| | | "Type": "Operand", | "Type": "Operand", |

| | | "Operand": { | "Operand": { |

| | | "Type": "Inequality", | "Type": "Inequality", |

| | | "Terms": [ | "Terms": [ |

| | | { | { |

| | | "Type": "Term", | "Type": "Term", |

| | | "Content": { | "Content": { |

| | | "Type": "NameSpace", | "Type": "NameSpace", |

| | | "Name": "a" | "Name": "a" |

| | | } | } |

| | | }, | }, |

| | | { | { |

| | | "Type": "Term", | "Type": "Term", |

| | | "Content": { | "Content": { |

| | | "Type": "Number", | "Type": "Number", |

| | | "Value": "3.4" | "Value": "3.4" |

| | | } | } |

| | | } | } |

| | | ], | ], |

| | | "Comparisons": [ | "Comparisons": [ |

| | | { | { |

| | | "Type": "ComparisonOperator", | "Type": "ComparisonOperator", |

| | | "ComparisonType": ">=" | "ComparisonType": ">=" |

| | | } | } |

| | | ] | ] |

| | | } | } |

| | | } | } |

| | | ] | ] |

| | | } | } |

-------------+---------------+-------------------+-----------------------------------------------+-----------------------------------------------+

Pass | Function | "func()" | { | { |

| | | "Type": "Function", | "Type": "Function", |

| | | "Name": { | "Name": { |

| | | "Type": "NameSpace", | "Type": "NameSpace", |

| | | "Name": "func" | "Name": "func" |

| | | }, | }, |

| | | "Operands": { | "Operands": { |

| | | "Type": "Operands", | "Type": "Operands", |

| | | "Operands": [ | "Operands": [ |

| | | | |

| | | ] | ] |

| | | } | } |

| | | } | } |

-------------+---------------+-------------------+-----------------------------------------------+-----------------------------------------------+

Pass | Function | "func)" | NotCompatibleException | NotCompatibleException |

| | | | |

-------------+---------------+-------------------+-----------------------------------------------+-----------------------------------------------+

Pass | Function | "P(a>=3.4)" | { | { |

| | | "Type": "Function", | "Type": "Function", |

| | | "Name": { | "Name": { |

| | | "Type": "NameSpace", | "Type": "NameSpace", |

| | | "Name": "P" | "Name": "P" |

| | | }, | }, |

| | | "Operands": { | "Operands": { |

| | | "Type": "Operands", | "Type": "Operands", |

| | | "Operands": [ | "Operands": [ |

| | | { | { |

| | | "Type": "Operand", | "Type": "Operand", |

| | | "Operand": { | "Operand": { |

| | | "Type": "Inequality", | "Type": "Inequality", |

| | | "Terms": [ | "Terms": [ |

| | | { | { |

| | | "Type": "Term", | "Type": "Term", |

| | | "Content": { | "Content": { |

| | | "Type": "NameSpace", | "Type": "NameSpace", |

| | | "Name": "a" | "Name": "a" |

| | | } | } |

| | | }, | }, |

| | | { | { |

| | | "Type": "Term", | "Type": "Term", |

| | | "Content": { | "Content": { |

| | | "Type": "Number", | "Type": "Number", |

| | | "Value": "3.4" | "Value": "3.4" |

| | | } | } |

| | | } | } |

| | | ], | ], |

| | | "Comparisons": [ | "Comparisons": [ |

| | | { | { |

| | | "Type": "ComparisonOperator", | "Type": "ComparisonOperator", |

| | | "ComparisonType": ">=" | "ComparisonType": ">=" |

| | | } | } |

| | | ] | ] |

| | | } | } |

| | | } | } |

| | | ] | ] |

| | | } | } |

| | | } | } |

-------------+---------------+-------------------+-----------------------------------------------+-----------------------------------------------+

Pass | Function | "Settings()" | NotCompatibleException | NotCompatibleException |

| | | | |

-------------+---------------+-------------------+-----------------------------------------------+-----------------------------------------------+

Pass | Expression | "1+3" | { | { |

| | | "Type": "Expression", | "Type": "Expression", |

| | | "TermsAndOperators": [ | "TermsAndOperators": [ |

| | | { | { |

| | | "Type": "Term", | "Type": "Term", |

| | | "Content": { | "Content": { |

| | | "Type": "Number", | "Type": "Number", |

| | | "Value": "1" | "Value": "1" |

| | | } | } |

| | | }, | }, |

| | | { | { |

| | | "Type": "Operator", | "Type": "Operator", |

| | | "OperatorType": "+" | "OperatorType": "+" |

| | | }, | }, |

| | | { | { |

| | | "Type": "Term", | "Type": "Term", |

| | | "Content": { | "Content": { |

| | | "Type": "Number", | "Type": "Number", |

| | | "Value": "3" | "Value": "3" |

| | | } | } |

| | | } | } |

| | | ] | ] |

| | | } | } |

-------------+---------------+-------------------+-----------------------------------------------+-----------------------------------------------+

Pass | Expression | "1+" | NotCompatibleException | NotCompatibleException |

| | | | |

-------------+---------------+-------------------+-----------------------------------------------+-----------------------------------------------+

Pass | BracketedTerm | "(1+3)" | { | { |

| | | "Type": "BracketedTerm", | "Type": "BracketedTerm", |

| | | "expression": { | "expression": { |

| | | "Type": "Expression", | "Type": "Expression", |

| | | "TermsAndOperators": [ | "TermsAndOperators": [ |

| | | { | { |

| | | "Type": "Term", | "Type": "Term", |

| | | "Content": { | "Content": { |

| | | "Type": "Number", | "Type": "Number", |

| | | "Value": "1" | "Value": "1" |

| | | } | } |

| | | }, | }, |

| | | { | { |

| | | "Type": "Operator", | "Type": "Operator", |

| | | "OperatorType": "+" | "OperatorType": "+" |

| | | }, | }, |

| | | { | { |

| | | "Type": "Term", | "Type": "Term", |

| | | "Content": { | "Content": { |

| | | "Type": "Number", | "Type": "Number", |

| | | "Value": "3" | "Value": "3" |

| | | } | } |

| | | } | } |

| | | ] | ] |

| | | } | } |

| | | } | } |

-------------+---------------+-------------------+-----------------------------------------------+-----------------------------------------------+

Pass | Term | "(1+3)" | { | { |

| | | "Type": "Term", | "Type": "Term", |

| | | "Content": { | "Content": { |

| | | "Type": "BracketedTerm", | "Type": "BracketedTerm", |

| | | "expression": { | "expression": { |

| | | "Type": "Expression", | "Type": "Expression", |

| | | "TermsAndOperators": [ | "TermsAndOperators": [ |

| | | { | { |

| | | "Type": "Term", | "Type": "Term", |

| | | "Content": { | "Content": { |

| | | "Type": "Number", | "Type": "Number", |

| | | "Value": "1" | "Value": "1" |

| | | } | } |

| | | }, | }, |

| | | { | { |

| | | "Type": "Operator", | "Type": "Operator", |

| | | "OperatorType": "+" | "OperatorType": "+" |

| | | }, | }, |

| | | { | { |

| | | "Type": "Term", | "Type": "Term", |

| | | "Content": { | "Content": { |

| | | "Type": "Number", | "Type": "Number", |

| | | "Value": "3" | "Value": "3" |

| | | } | } |

| | | } | } |

| | | ] | ] |

| | | } | } |

| | | } | } |

| | | } | } |

-------------+---------------+-------------------+-----------------------------------------------+-----------------------------------------------+

Pass | Assignable | "var1" | { | { |

| | | "Type": "Assignable", | "Type": "Assignable", |

| | | "Assignee": { | "Assignee": { |

| | | "Type": "NameSpace", | "Type": "NameSpace", |

| | | "Name": "var1" | "Name": "var1" |

| | | } | } |

| | | } | } |

-------------+---------------+-------------------+-----------------------------------------------+-----------------------------------------------+

Pass | Assignable | "Func1()" | { | { |

| | | "Type": "Assignable", | "Type": "Assignable", |

| | | "Assignee": { | "Assignee": { |

| | | "Type": "Function", | "Type": "Function", |

| | | "Name": { | "Name": { |

| | | "Type": "NameSpace", | "Type": "NameSpace", |

| | | "Name": "Func1" | "Name": "Func1" |

| | | }, | }, |

| | | "Operands": { | "Operands": { |

| | | "Type": "Operands", | "Type": "Operands", |

| | | "Operands": [ | "Operands": [ |

| | | | |

| | | ] | ] |

| | | } | } |

| | | } | } |

| | | } | } |

-------------+---------------+-------------------+-----------------------------------------------+-----------------------------------------------+

Pass | Assignable | "Func1(a+2,d)" | { | { |

| | | "Type": "Assignable", | "Type": "Assignable", |

| | | "Assignee": { | "Assignee": { |

| | | "Type": "Function", | "Type": "Function", |

| | | "Name": { | "Name": { |

| | | "Type": "NameSpace", | "Type": "NameSpace", |

| | | "Name": "Func1" | "Name": "Func1" |

| | | }, | }, |

| | | "Operands": { | "Operands": { |

| | | "Type": "Operands", | "Type": "Operands", |

| | | "Operands": [ | "Operands": [ |

| | | { | { |

| | | "Type": "Operand", | "Type": "Operand", |

| | | "Operand": { | "Operand": { |

| | | "Type": "Expression", | "Type": "Expression", |

| | | "TermsAndOperators": [ | "TermsAndOperators": [ |

| | | { | { |

| | | "Type": "Term", | "Type": "Term", |

| | | "Content": { | "Content": { |

| | | "Type": "NameSpace", | "Type": "NameSpace", |

| | | "Name": "a" | "Name": "a" |

| | | } | } |

| | | }, | }, |

| | | { | { |

| | | "Type": "Operator", | "Type": "Operator", |

| | | "OperatorType": "+" | "OperatorType": "+" |

| | | }, | }, |

| | | { | { |

| | | "Type": "Term", | "Type": "Term", |

| | | "Content": { | "Content": { |

| | | "Type": "Number", | "Type": "Number", |

| | | "Value": "2" | "Value": "2" |

| | | } | } |

| | | } | } |

| | | ] | ] |

| | | } | } |

| | | }, | }, |

| | | { | { |

| | | "Type": "Operand", | "Type": "Operand", |

| | | "Operand": { | "Operand": { |

| | | "Type": "Expression", | "Type": "Expression", |

| | | "TermsAndOperators": [ | "TermsAndOperators": [ |

| | | { | { |

| | | "Type": "Term", | "Type": "Term", |

| | | "Content": { | "Content": { |

| | | "Type": "NameSpace", | "Type": "NameSpace", |

| | | "Name": "d" | "Name": "d" |

| | | } | } |

| | | } | } |

| | | ] | ] |

| | | } | } |

| | | } | } |

| | | ] | ] |

| | | } | } |

| | | } | } |

| | | } | } |

-------------+---------------+-------------------+-----------------------------------------------+-----------------------------------------------+

Pass | Assignment | "a=4" | { | { |

| | | "Type": "Assignment", | "Type": "Assignment", |

| | | "Assigned": { | "Assigned": { |

| | | "Type": "Assignable", | "Type": "Assignable", |

| | | "Assignee": { | "Assignee": { |

| | | "Type": "NameSpace", | "Type": "NameSpace", |

| | | "Name": "a" | "Name": "a" |

| | | } | } |

| | | }, | }, |

| | | "Equality": { | "Equality": { |

| | | "Type": "Equality", | "Type": "Equality", |

| | | "EqualityType": "=" | "EqualityType": "=" |

| | | }, | }, |

| | | "Expression": { | "Expression": { |

| | | "Type": "Expression", | "Type": "Expression", |

| | | "TermsAndOperators": [ | "TermsAndOperators": [ |

| | | { | { |

| | | "Type": "Term", | "Type": "Term", |

| | | "Content": { | "Content": { |

| | | "Type": "Number", | "Type": "Number", |

| | | "Value": "4" | "Value": "4" |

| | | } | } |

| | | } | } |

| | | ] | ] |

| | | } | } |

| | | } | } |

-------------+---------------+-------------------+-----------------------------------------------+-----------------------------------------------+

Pass | Assignment | "f(x)~x+1" | { | { |

| | | "Type": "Assignment", | "Type": "Assignment", |

| | | "Assigned": { | "Assigned": { |

| | | "Type": "Assignable", | "Type": "Assignable", |

| | | "Assignee": { | "Assignee": { |

| | | "Type": "Function", | "Type": "Function", |

| | | "Name": { | "Name": { |

| | | "Type": "NameSpace", | "Type": "NameSpace", |

| | | "Name": "f" | "Name": "f" |

| | | }, | }, |

| | | "Operands": { | "Operands": { |

| | | "Type": "Operands", | "Type": "Operands", |

| | | "Operands": [ | "Operands": [ |

| | | { | { |

| | | "Type": "Operand", | "Type": "Operand", |

| | | "Operand": { | "Operand": { |

| | | "Type": "Expression", | "Type": "Expression", |

| | | "TermsAndOperators": [ | "TermsAndOperators": [ |

| | | { | { |

| | | "Type": "Term", | "Type": "Term", |

| | | "Content": { | "Content": { |

| | | "Type": "NameSpace", | "Type": "NameSpace", |

| | | "Name": "x" | "Name": "x" |

| | | } | } |

| | | } | } |

| | | ] | ] |

| | | } | } |

| | | } | } |

| | | ] | ] |

| | | } | } |

| | | } | } |

| | | }, | }, |

| | | "Equality": { | "Equality": { |

| | | "Type": "Equality", | "Type": "Equality", |

| | | "EqualityType": "~" | "EqualityType": "~" |

| | | }, | }, |

| | | "Expression": { | "Expression": { |

| | | "Type": "Expression", | "Type": "Expression", |

| | | "TermsAndOperators": [ | "TermsAndOperators": [ |

| | | { | { |

| | | "Type": "Term", | "Type": "Term", |

| | | "Content": { | "Content": { |

| | | "Type": "NameSpace", | "Type": "NameSpace", |

| | | "Name": "x" | "Name": "x" |

| | | } | } |

| | | }, | }, |

| | | { | { |

| | | "Type": "Operator", | "Type": "Operator", |

| | | "OperatorType": "+" | "OperatorType": "+" |

| | | }, | }, |

| | | { | { |

| | | "Type": "Term", | "Type": "Term", |

| | | "Content": { | "Content": { |

| | | "Type": "Number", | "Type": "Number", |

| | | "Value": "1" | "Value": "1" |

| | | } | } |

| | | } | } |

| | | ] | ] |

| | | } | } |

| | | } | } |

-------------+---------------+-------------------+-----------------------------------------------+-----------------------------------------------+

Pass | Statement | "Settings()" | { | { |

| | | "Type": "Statement", | "Type": "Statement", |

| | | "Name": { | "Name": { |

| | | "Type": "NameSpace", | "Type": "NameSpace", |

| | | "Name": "Settings" | "Name": "Settings" |

| | | }, | }, |

| | | "Operands": { | "Operands": { |

| | | "Type": "Operands", | "Type": "Operands", |

| | | "Operands": [ | "Operands": [ |

| | | | |

| | | ] | ] |

| | | } | } |

| | | } | } |

-------------+---------------+-------------------+-----------------------------------------------+-----------------------------------------------+

Pass | Line | ";" | { | { |

| | | "Type": "EmptyLine", | "Type": "EmptyLine", |

| | | "Action": "None" | "Action": "None" |

| | | } | } |

-------------+---------------+-------------------+-----------------------------------------------+-----------------------------------------------+

Pass | Line | "1+3\n" | { | { |

| | | "Type": "Line", | "Type": "Line", |

| | | "Action": { | "Action": { |

| | | "Type": "Expression", | "Type": "Expression", |

| | | "TermsAndOperators": [ | "TermsAndOperators": [ |

| | | { | { |

| | | "Type": "Term", | "Type": "Term", |

| | | "Content": { | "Content": { |

| | | "Type": "Number", | "Type": "Number", |

| | | "Value": "1" | "Value": "1" |

| | | } | } |

| | | }, | }, |

| | | { | { |

| | | "Type": "Operator", | "Type": "Operator", |

| | | "OperatorType": "+" | "OperatorType": "+" |

| | | }, | }, |

| | | { | { |

| | | "Type": "Term", | "Type": "Term", |

| | | "Content": { | "Content": { |

| | | "Type": "Number", | "Type": "Number", |

| | | "Value": "3" | "Value": "3" |

| | | } | } |

| | | } | } |

| | | ] | ] |

| | | } | } |

| | | } | } |

-------------+---------------+-------------------+-----------------------------------------------+-----------------------------------------------+

Pass | Line | "Settings();" | { | { |

| | | "Type": "Line", | "Type": "Line", |

| | | "Action": { | "Action": { |

| | | "Type": "Statement", | "Type": "Statement", |

| | | "Name": { | "Name": { |

| | | "Type": "NameSpace", | "Type": "NameSpace", |

| | | "Name": "Settings" | "Name": "Settings" |

| | | }, | }, |

| | | "Operands": { | "Operands": { |

| | | "Type": "Operands", | "Type": "Operands", |

| | | "Operands": [ | "Operands": [ |

| | | | |

| | | ] | ] |

| | | } | } |

| | | } | } |

| | | } | } |

-------------+---------------+-------------------+-----------------------------------------------+-----------------------------------------------+

Pass | Line | "func()\n" | { | { |

| | | "Type": "Line", | "Type": "Line", |

| | | "Action": { | "Action": { |

| | | "Type": "Expression", | "Type": "Expression", |

| | | "TermsAndOperators": [ | "TermsAndOperators": [ |

| | | { | { |

| | | "Type": "Term", | "Type": "Term", |

| | | "Content": { | "Content": { |

| | | "Type": "Function", | "Type": "Function", |

| | | "Name": { | "Name": { |

| | | "Type": "NameSpace", | "Type": "NameSpace", |

| | | "Name": "func" | "Name": "func" |

| | | }, | }, |

| | | "Operands": { | "Operands": { |

| | | "Type": "Operands", | "Type": "Operands", |

| | | "Operands": [ | "Operands": [ |

| | | | |

| | | ] | ] |

| | | } | } |

| | | } | } |

| | | } | } |

| | | ] | ] |

| | | } | } |

| | | } | } |

-------------+---------------+-------------------+-----------------------------------------------+-----------------------------------------------+

Pass | Line | "a=4\n" | { | { |

| | | "Type": "Line", | "Type": "Line", |

| | | "Action": { | "Action": { |

| | | "Type": "Assignment", | "Type": "Assignment", |

| | | "Assigned": { | "Assigned": { |

| | | "Type": "Assignable", | "Type": "Assignable", |

| | | "Assignee": { | "Assignee": { |

| | | "Type": "NameSpace", | "Type": "NameSpace", |

| | | "Name": "a" | "Name": "a" |

| | | } | } |

| | | }, | }, |

| | | "Equality": { | "Equality": { |

| | | "Type": "Equality", | "Type": "Equality", |

| | | "EqualityType": "=" | "EqualityType": "=" |

| | | }, | }, |

| | | "Expression": { | "Expression": { |

| | | "Type": "Expression", | "Type": "Expression", |

| | | "TermsAndOperators": [ | "TermsAndOperators": [ |

| | | { | { |

| | | "Type": "Term", | "Type": "Term", |

| | | "Content": { | "Content": { |

| | | "Type": "Number", | "Type": "Number", |

| | | "Value": "4" | "Value": "4" |

| | | } | } |

| | | } | } |

| | | ] | ] |

| | | } | } |

| | | } | } |

| | | } | } |

-------------+---------------+-------------------+-----------------------------------------------+-----------------------------------------------+

Pass | Lines | ";" | { | { |

| | | "Type": "Lines", | "Type": "Lines", |

| | | "Lines": [ | "Lines": [ |

| | | { | { |

| | | "Type": "EmptyLine", | "Type": "EmptyLine", |

| | | "Action": "None" | "Action": "None" |

| | | } | } |

| | | ] | ] |

| | | } | } |

-------------+---------------+-------------------+-----------------------------------------------+-----------------------------------------------+

Pass | Lines | ";;" | { | { |

| | | "Type": "Lines", | "Type": "Lines", |

| | | "Lines": [ | "Lines": [ |

| | | { | { |

| | | "Type": "EmptyLine", | "Type": "EmptyLine", |

| | | "Action": "None" | "Action": "None" |

| | | }, | }, |

| | | { | { |

| | | "Type": "EmptyLine", | "Type": "EmptyLine", |

| | | "Action": "None" | "Action": "None" |

| | | } | } |

| | | ] | ] |

| | | } | } |

-------------+---------------+-------------------+-----------------------------------------------+-----------------------------------------------+

Pass | Lines | ";1+3\n;" | { | { |

| | | "Type": "Lines", | "Type": "Lines", |

| | | "Lines": [ | "Lines": [ |

| | | { | { |

| | | "Type": "EmptyLine", | "Type": "EmptyLine", |

| | | "Action": "None" | "Action": "None" |

| | | }, | }, |

| | | { | { |

| | | "Type": "Line", | "Type": "Line", |

| | | "Action": { | "Action": { |

| | | "Type": "Expression", | "Type": "Expression", |

| | | "TermsAndOperators": [ | "TermsAndOperators": [ |

| | | { | { |

| | | "Type": "Term", | "Type": "Term", |

| | | "Content": { | "Content": { |

| | | "Type": "Number", | "Type": "Number", |

| | | "Value": "1" | "Value": "1" |

| | | } | } |

| | | }, | }, |

| | | { | { |

| | | "Type": "Operator", | "Type": "Operator", |

| | | "OperatorType": "+" | "OperatorType": "+" |

| | | }, | }, |

| | | { | { |

| | | "Type": "Term", | "Type": "Term", |

| | | "Content": { | "Content": { |

| | | "Type": "Number", | "Type": "Number", |

| | | "Value": "3" | "Value": "3" |

| | | } | } |

| | | } | } |

| | | ] | ] |

| | | } | } |

| | | }, | }, |

| | | { | { |

| | | "Type": "EmptyLine", | "Type": "EmptyLine", |

| | | "Action": "None" | "Action": "None" |

| | | } | } |

| | | ] | ] |

| | | } | } |

-------------+---------------+-------------------+-----------------------------------------------+-----------------------------------------------+

Pass | Lines | "" | NotCompatibleException | NotCompatibleException |

| | | | |

-------------+---------------+-------------------+-----------------------------------------------+-----------------------------------------------+

Pass | Lines | "l" | NotCompatibleException | NotCompatibleException |

| | | | |

-------------+---------------+-------------------+-----------------------------------------------+-----------------------------------------------+

Pass | Lines | "whatis(a,b) = a+b;4/3+"| { | { |

| | | "Type": "Lines", | "Type": "Lines", |

| | | "Lines": [ | "Lines": [ |

| | | { | { |

| | | "Type": "Line", | "Type": "Line", |

| | | "Action": { | "Action": { |

| | | "Type": "Assignment", | "Type": "Assignment", |

| | | "Assigned": { | "Assigned": { |

| | | "Type": "Assignable", | "Type": "Assignable", |

| | | "Assignee": { | "Assignee": { |

| | | "Type": "Function", | "Type": "Function", |

| | | "Name": { | "Name": { |

| | | "Type": "NameSpace", | "Type": "NameSpace", |

| | | "Name": "whatis" | "Name": "whatis" |

| | | }, | }, |

| | | "Operands": { | "Operands": { |

| | | "Type": "Operands", | "Type": "Operands", |

| | | "Operands": [ | "Operands": [ |

| | | { | { |

| | | "Type": "Operand", | "Type": "Operand", |

| | | "Operand": { | "Operand": { |

| | | "Type": "Expression", | "Type": "Expression", |

| | | "TermsAndOperators": [ | "TermsAndOperators": [ |

| | | { | { |

| | | "Type": "Term", | "Type": "Term", |

| | | "Content": { | "Content": { |

| | | "Type": "NameSpace",| "Type": "NameSpace",|

| | | "Name": "a" | "Name": "a" |

| | | } | } |

| | | } | } |

| | | ] | ] |

| | | } | } |

| | | }, | }, |

| | | { | { |

| | | "Type": "Operand", | "Type": "Operand", |

| | | "Operand": { | "Operand": { |

| | | "Type": "Expression", | "Type": "Expression", |

| | | "TermsAndOperators": [ | "TermsAndOperators": [ |

| | | { | { |

| | | "Type": "Term", | "Type": "Term", |

| | | "Content": { | "Content": { |

| | | "Type": "NameSpace",| "Type": "NameSpace",|

| | | "Name": "b" | "Name": "b" |

| | | } | } |

| | | } | } |

| | | ] | ] |

| | | } | } |

| | | } | } |

| | | ] | ] |

| | | } | } |

| | | } | } |

| | | }, | }, |

| | | "Equality": { | "Equality": { |

| | | "Type": "Equality", | "Type": "Equality", |

| | | "EqualityType": "=" | "EqualityType": "=" |

| | | }, | }, |

| | | "Expression": { | "Expression": { |

| | | "Type": "Expression", | "Type": "Expression", |

| | | "TermsAndOperators": [ | "TermsAndOperators": [ |

| | | { | { |

| | | "Type": "Term", | "Type": "Term", |

| | | "Content": { | "Content": { |

| | | "Type": "NameSpace", | "Type": "NameSpace", |

| | | "Name": "a" | "Name": "a" |

| | | } | } |

| | | }, | }, |

| | | { | { |

| | | "Type": "Operator", | "Type": "Operator", |

| | | "OperatorType": "+" | "OperatorType": "+" |

| | | }, | }, |

| | | { | { |

| | | "Type": "Term", | "Type": "Term", |

| | | "Content": { | "Content": { |

| | | "Type": "NameSpace", | "Type": "NameSpace", |

| | | "Name": "b" | "Name": "b" |

| | | } | } |

| | | } | } |

| | | ] | ] |

| | | } | } |

| | | } | } |

| | | } | } |

| | | ] | ] |

| | | } | } |

-------------+---------------+-------------------+-----------------------------------------------+-----------------------------------------------+\* code for tests can be found in parser.py

## Project requirement tests

### Functional requirements

Functional

Reqs |pass? | input | expected number| actual number

FR 2.1 | True | 2+4; | [Number: 6.0] | [Number: 6.0]

FR 2.2 | True | 2-4; | [Number: -2.0] | [Number: -2.0]

FR 2.3 | True | 2\*3; | [Number: 6.0] | [Number: 6.0]

FR 2.4 | True | 6/2; | [Number: 3.0] | [Number: 3.0]

FR 2.4.1 | True | 1/0; | [Undefined] | [Undefined]

FR 2.5 | True | 2^3; | [Number: 8.0] | [Number: 8.0]

FR 2.6 | True | -3; | [Number: -3.0] | [Number: -3.0]

FR 2.7 | True | (2+3)\*4; | [Number: 20.0] | [Number: 20.0]

FR 2.8.1 | True | a=3; | [None] | [None]

FR 2.8.2 | True | f(x)=x+1; | [None] | [None]

FR 2.9 | True | f(g(4)); | [Number: 9.0] | [Number: 9.0]

FR 2.10 | True | 4\*3^2+5; | [Number: 41.0] | [Number: 41.0]

FR 2.11 | False| sin(pi); | [Number: 0.0] | [Number: 1.2246467991473532e-16]

FR 2.11 | False| tan(pi); | [Number: 0.0] | [Number: -1.2246467991473532e-16]

FR 3.1 | True | setIntoDeg(); | [None] | [None]

FR 3.2 | False| sin(pi); | [Number: 0.054]| [Number: 0.05480366514878953]

FR 4 | True | get\_custom\_statements()

FR 5 | False| 2^(3+1);2^3+1;| 3.0+1.0 | 3.0+1.0

| | | 2.0 | 2.0

| | | 3.0 | 3.0

| | | 2.0 +1.0 | 2.0 +1.0

### Non-Functional Requirements

|  |  |  |
| --- | --- | --- |
| Non-Functional Requirement code/number | Pass or fail | Explanation |
| 1.1 | Pass | refer to code or class diagram in controlunit.py section above |
| 1.2 | Pass | Is this true because in the getters, all references to the variables that store the internal values are thrown away and replaced by a new reference before being passed to the interface. And since all values cannot be changed, it is secure overall and reasonably defends against edits to the retrieved data affecting internal data. |

### Technical requirements

All technical requirements passed as evident from testing of functional requirements above.

## Additional testing

The following images were taken from the Command line interface implementation in userinterface.py.

The images demonstrates the result from testing for the same functional and non-functional requirements.

A screenshot of a computer

Description automatically generated

A screenshot of a computer

Description automatically generated

A screenshot of a computer

Description automatically generated

A screenshot of a computer program

Description automatically generated

A screenshot of a math program

Description automatically generated

A screenshot of a computer

Description automatically generated

A screenshot of a computer program

Description automatically generated

A screenshot of a computer program

Description automatically generated

A screenshot of a computer program

Description automatically generated

# Evaluation

## Requirements

The program has met most the Functional, non-functional, and technical requirements. With the only exception being FR 2.11, where there was what appears to be a rounding error. I would expect under certain circumstances where a value cannot be reasonably represented by a binary system, there would also be a rounding error.

The objectives set for the project was to be able to accept input, which can be achieved through the set\_instructions method in UserInterfaceInterfacd class; performance of basic arithmetic, done using ActionVisitor Visitor pattern tree traversal; trigonometric functions, through explicit declaration in CalculationUnit; changing and accessing internal state, though getter and setter method in UderInterfaceInterface.

As part of the extension objectives and testing, and demonstration purposes, I have created a command line implementation of the calculator. This implementation serves a proof-of-concept implementation that demonstrates that capabilities of the API program.

## Improvements / Potential extensions

Some of the ways the program could be extended would be:

* Refactor number system to use a base 10 number system behind the scenes implementation to prevent rounding errors exposed during testing.
* More intelligent whitespace remover (integrate it into the parser and BNF) to parse whitespace instead of filtering it out (as this could cause unintended behavior two tokens “e pi” would be transformed into one token “epi”).

Eg <Expression> ::= <Term> | <Term> <WhiteSpace> <Operation> <WhiteSpace> <Expression>

* More options for entering numbers/decimals, by rewriting the rule for Number to accept “.3” as 0.3 as well as “1.” as 1.
* Creation of a new extension of the program that can create the code to follow custom BNF rules that could be set by the interface designers to enable further customization down to the language the user may input.
* Other value types such as Matrixes, Probability distributions, Imaginary numbers
* More operations (eg Root finding, Differentiation, Integration)

“d(x^2)” -> “2\*x”

Or

“int(e^x)” -> “e^x”

* Native graphing support: currently there is only a NPArray Value object which could be used by users to incorporate a graph functionality using matplotlib in combination with numpy arrays. More research into graphing tools available to python could lead to a more comprehensive support integrated into the program instead of having to be built by the user.

## Feedback from user

This is working quite well, but the naming of the methods in the UserInterfaceInterface class could use some more work so that it is clear what actions the respective methods perform, other than that, I would also agree with the assessment that I would have been better if there were more object types, but seeing that it is possible to personally extend the types of Values, this issue can be overlooked. I also noted there was a bug when adding 0.1 and 0.2, I would like that to be fixed.

## What I learned & how I would use it to recreate the same program.

During this project, I developed a fascination with overcomplicated programming patterns and niche python functionalities. This combination has led to many eccentric buggy outcomes which required specific knowledge to debug, which greatly slowed development progress. If I were to start again, I would stick to more simple data structures and programming patterns unless they are strictly necessary (as with the case for Dijkstra’s shunting yard algorithm).

# Citations

Expression tree inspiration: <https://people.cs.ksu.edu/~schmidt/300s05/Lectures/GrammarNotes/bnf.html>,

[The ANTLR Mega Tutorial (tomassetti.me)](https://tomassetti.me/antlr-mega-tutorial/#chapter37)

Expression trees: <https://www.graphviz.org/>, <https://sketchviz.com/new>

Syntax diagrams: <https://tabatkins.github.io/railroad-diagrams/>

Class diagrams: [https://app.diagrams.net](https://app.diagrams.net/)

Flow diagrams/charts: [https://app.smartdraw.com](https://app.smartdraw.com/)

Dijkstra’s shunting yard algorithm: <https://en.wikipedia.org/wiki/Shunting_yard_algorithm>

Visitor pattern: <https://refactoring.guru/design-patterns/visitor>

Monad explanation pictures: <https://www.adit.io/posts/2013-04-17-functors,_applicatives,_and_monads_in_pictures.html>

# Code listing