```
#!/user/bin/env python
"""
This module is responsible for the lexing of input into tokens defined by the
BNF
"""

# for tests
import random
import string
from utilities import *

from abc import abstractmethod

# prevents import * from other files importing anything unnecessary
__all__ = ["Token", "tokenize", "tokenize", "ScanError"]


"""
language BNF

Scanner/Lexer 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]+)*
<Number>               ::= <Digit> | <Digit> '.' <Digit>
regex: [0-9]+(\\.[0-9])?
<Digit>                ::= ('0' | '1' | '2' | '3' | '4' | '5' | '6'| '7' | '8
' | '9')+
"""
# scanner / lexer


class Token(JSONable):
    """
    interface for all tokens generated by Lexer/ Scanner
    """
    @classmethod
    @abstractmethod
```

```
54      def consume(cls, inputFeed: str) -> tuple['Token', str]:
55          """
56          tries to see if the beginning of the input feed fits the Token Type
57          :raises NotCompatibleException: when the feed is not compatible
58          :param inputFeed: string input
59          :return: an instance of the Token, and the remainder of the input feed
60          """
61          raise NotImplementedError("Take not implemented by "+cls.__name__)
62
63      @abstractmethod
64      def get_json(self) -> dict:
65          """
66          gives the JSON representation of the Token
67          :return:
68          """
69          raise NotImplementedError("JSON method not implemented by "+type(self
    ).__name__)
70
71      @abstractmethod
72      def __eq__(self, other):
73          """
74          checks if the two tokens are equal
75          :param other:
76          :return:
77          """
78          raise NotImplementedError("== not implemented by "+type(self).__name__
    )
79
80
81  class EOL(Token):
82      """
83      End of Line ";", "\n"
84      """
85      @classmethod
86      def consume(cls, inputFeed: str) -> tuple['EOL', str]:
87          if len(inputFeed) > 0 and inputFeed[0] in "\n;":
88              return EOL(), inputFeed[1:]
89          raise NotCompatibleException
90
91      def get_json(self) -> dict:
92          return {
93              "Type": "EOL"
94          }
95
96      def __eq__(self, other):
97          if not isinstance(other, EOL):
98              return False
99          return True
100
101
102 class OpenBracket(Token):
103     """
104     Open bracket "("
105     """
106     @classmethod
107     def consume(cls, inputFeed: str) -> tuple['OpenBracket', str]:
108         if len(inputFeed) > 0 and inputFeed[0] == "(":
109             return OpenBracket(), inputFeed[1:]
110         raise NotCompatibleException
```

```
111
112     def get_json(self) -> dict:
113         return {
114             "Type": "OpenBracket"
115         }
116
117     def __eq__(self, other):
118         if not isinstance(other, OpenBracket):
119             return False
120         return True
121
122
123 class ClosedBracket(Token):
124     """
125     Closed bracket ")"
126     """
127     @classmethod
128     def consume(cls, inputFeed: str) -> tuple['ClosedBracket', str]:
129         if len(inputFeed) > 0 and inputFeed[0] == ")":
130             return ClosedBracket(), inputFeed[1:]
131         raise NotCompatibleException
132
133     def get_json(self) -> dict:
134         return {
135             "Type": "ClosedBracket"
136         }
137
138     def __eq__(self, other):
139         if not isinstance(other, ClosedBracket):
140             return False
141         return True
142
143
144 class Comma(Token):
145     """
146     Comma ","
147     """
148     @classmethod
149     def consume(cls, inputFeed: str) -> tuple['Comma', str]:
150         if len(inputFeed) > 0 and inputFeed[0] == ",":
151             return Comma(), inputFeed[1:]
152         raise NotCompatibleException
153
154     def get_json(self) -> dict:
155         return {
156             "Type": "Comma"
157         }
158
159     def __eq__(self, other):
160         return isinstance(other, Comma)
161
162
163 class ComparisonOperator(Token):
164     """
165     Comparison "<",">","==","<=",">="
166     """
167     def __init__(self, typeOfComparison: str):
168         self._operator_type = typeOfComparison
169
```

```
170     @classmethod
171     def consume(cls, inputFeed: str) -> tuple['ComparisonOperator', str]:
172         if len(inputFeed) > 1 and inputFeed[:2] in ('<=', '>=', '=='):
173             return ComparisonOperator(inputFeed[:2]), inputFeed[2:]
174         elif len(inputFeed) > 0 and inputFeed[0] in "<>":
175             return ComparisonOperator(inputFeed[0]), inputFeed[1:]
176         raise NotCompatibleException
177 
178     def get_json(self) -> dict:
179         return {
180             "Type": "ComparisonOperator",
181             "ComparisonType": self._operator_type
182         }
183 
184     # noinspection PyProtectedMember
185     def __eq__(self, other):
186         if not isinstance(other, ComparisonOperator):
187             return False
188         return self._operator_type == other._operator_type
189 
190 
191 class Operator(Token):
192     """
193     Operator "+","-","*","/","^"
194     """
195     def __init__(self, typeOfOperator: str):
196         self._operator_type = typeOfOperator
197 
198     @classmethod
199     def consume(cls, inputFeed: str) -> tuple['Operator', str]:
200         if len(inputFeed) > 0 and inputFeed[0] in "+-*/^":
201             return Operator(inputFeed[0]), inputFeed[1:]
202         raise NotCompatibleException
203 
204     def get_json(self) -> dict:
205         return {
206             "Type": "Operator",
207             "OperatorType": self._operator_type
208         }
209 
210     # noinspection PyProtectedMember
211     def __eq__(self, other):
212         if not isinstance(other, Operator):
213             return False
214         return self._operator_type == other._operator_type
215 
216 
217 class ConditionalOperator(Token):
218     """
219     Conditional "|"
220     """
221     @classmethod
222     def consume(cls, inputFeed: str) -> tuple['ConditionalOperator', str]:
223         if len(inputFeed) > 0 and inputFeed[0] == "|":
224             return ConditionalOperator(), inputFeed[1:]
225         raise NotCompatibleException
226 
227     def get_json(self) -> dict:
228         return {
```

```
229             "Type": "ConditionalOperator"
230         }
231 
232     def __eq__(self, other):
233         return isinstance(other, ConditionalOperator)
234 
235 
236 class Equality(Token):
237     """
238     Equality "=","~"
239     """
240     def __init__(self, typeOfAssignment: str):
241         self._assignment_type = typeOfAssignment
242 
243     @classmethod
244     def consume(cls, inputFeed: str) -> tuple['Equality', str]:
245         if len(inputFeed) > 0 and inputFeed[0] in "=~":
246             return Equality(inputFeed[0]), inputFeed[1:]
247         raise NotCompatibleException
248 
249     def get_json(self) -> dict:
250         return {
251             "Type": "Equality",
252             "EqualityType": self._assignment_type
253         }
254 
255     # noinspection PyProtectedMember
256     def __eq__(self, other):
257         if not isinstance(other, Equality):
258             return False
259         return self._assignment_type == other._assignment_type
260 
261 
262 class NameSpace(Token):
263     """
264     Name "..."
265     """
266     def __init__(self, name: str):
267         self._string = name
268 
269     # noinspection PyPep8Naming
270     @classmethod
271     def consume(cls, inputFeed: str) -> tuple['NameSpace', str]:
272         i = 0
273         lenOfFeed = len(inputFeed)
274         # finds the index of first non-alpha-numeric character
275         while i < lenOfFeed and inputFeed[0].isalpha():
276             if not inputFeed[i].isalnum():
277                 break
278             i += 1
279         # if there is no alphabet at the start, the feed is not compatible
280         if i == 0:
281             raise NotCompatibleException
282         return NameSpace(inputFeed[:i]), inputFeed[i:]
283 
284     def get_json(self) -> dict:
285         return {
286             "Type": "NameSpace",
287             "Name": self._string
```

```
288         }
289 
290     # noinspection PyProtectedMember
291     def __eq__(self, other):
292         if not isinstance(other, NameSpace):
293             return False
294         return self._string == other._string
295 
296 
297 class Number(Token):
298     """
299     Number 131/12.313
300     """
301     def __init__(self, number: str):
302         self._num = number
303 
304     # noinspection PyPep8Naming
305     @classmethod
306     def consume(cls, inputFeed: str) -> tuple['Number', str]:
307         index = 0
308         lenOfFeed = len(inputFeed)
309         while index < lenOfFeed and inputFeed[index].isdigit():
310             index += 1
311         # if there are no numbers in the beginning of the feed
312         if index == 0:
313             raise NotCompatibleException
314         # break off if the next character is not a decimal point
315         if index == lenOfFeed or inputFeed[index] != ".":
316             return Number(inputFeed[:index]), inputFeed[index:]
317         t = index + 1
318         while t < lenOfFeed and inputFeed[t].isdigit():
319             t += 1
320         # if there are no numbers after the decimal point
321         if t == index + 1:
322             raise NotCompatibleException
323         return Number(inputFeed[:t]), inputFeed[t:]
324 
325     def get_json(self) -> dict:
326         return {
327             "Type": "Number",
328             "Value": self._num
329         }
330 
331     # noinspection PyProtectedMember
332     def __eq__(self, other):
333         if not isinstance(other, Number):
334             return False
335         return self._num == other._num
336 
337 
338 class ScanError(Exception):
339     """
340     error raised uniquely by tokenizer when a character sequence cannot be
    tokenized
341     """
342     def __eq__(self, other):
343         if not isinstance(other, ScanError):
344             return False
345         return str(self) == str(other)
```

```
346
347
348 # noinspection PyPep8Naming
349 def tokenize(inputFeed: str) -> list[Token]:
350     """
351     scans and tokenizes into list of tokens
352     :param inputFeed: string input
353     :return: list of tokens
354     :raises ScanError: when token stream does not cannot be tokenized
355     """
356     # sanitizes input
357     inputFeed = inputFeed.replace(" ", "")
358     # filter out unexpected characters
359     # unorthodox implementation
360     allowed_symbols = set("
    QWERTYUIOPASDFGHJKLZXCVBNMqwertyuiopasdfghjklzxcvbnm~1234567890^*()-+=|;<>,./\
    n")
361     for unexpected_symbol in set(inputFeed).difference(allowed_symbols):
362         inputFeed = inputFeed.replace(unexpected_symbol, "")
363
364     tokenList = []
365     while inputFeed:
366         for tokenpossibility in Token.__subclasses__():
367             try:
368                 token, inputFeed = tokenpossibility.consume(inputFeed)
369                 tokenList.append(token)
370                 # if a token has been matched with the beginning of the feed,
    skip trying the rest of the tokens
371                 break
372             except NotCompatibleException:
373                 # if not compatible, try the next token type
374                 pass
375         # if all tokens have been tried, raise ScanError
376         else:
377             raise ScanError("the given input feed \""+inputFeed+"\" cannot be
    matched to any defined token")
378     return tokenList
379
380
381 # noinspection PyPep8Naming
382 @test("Lexer")
383 def LexerTest() -> None:
384     """
385     tests capability of lexer, aka Scanner method
386     :return: None
387     """
388     logFile = "../log.txt"
389     # noinspection PyShadowingNames
390     with open(logFile, "w") as e:
391         e.write("Lexer Test Results:\n")
392         e.write("""
393  Test Result |   Input   |        Expected       |         Actual        |
394  (Pass/fail) |           |        Outcome        |         Outcome       |
395 -------------+-----------+-----------------------+-----------------------+
396 """[1:])
397
398     # noinspection PyPep8Naming
399     def tokenizerTest(toBeScanned: str, expectedResult: list[Token],
    expectedError: Exception = None) -> None:
```

```
400         """
401         Tests if Take method of Node obj will produce the expected result and
    logs result into the log.txt text file
402         :param toBeScanned: typical input feed
403         :param expectedResult: the node to be expected
404         :param expectedError: error expected(if any)
405         :return: None
406         """
407         # noinspection PyShadowingNames
408         expected_outcome = str([*map(lambda a:type(a).__name__, expectedResult
    )]) if expectedError is None else (type(expectedError).__name__)
409         result_text = toBeScanned.replace("\n", "\\n")
410         if len(result_text) > 9:
411             result_text = result_text[:6] + "..."
412         result_text = f"\"{result_text}\""
413 
414         try:
415             result = (outcome := tokenize(toBeScanned)) == expectedResult and
    expectedError is None
416             if expectedError is not None:
417                 outcome = "Error expected: " + type(expectedError).__name__ +
    str(expectedError)
418             outcome = str([*map(lambda a:type(a).__name__, outcome)])
419         except Exception as e:
420             result = e == expectedError
421             if not result:
422                 outcome = type(e).__name__ + " " + str(e)
423             else:
424                 outcome = type(e).__name__ + " " + str(e)
425             if expectedError is None:
426                 result = False
427         print("{:<13}|{:<11}|{:<23}|{:<23}|".format("Pass" if result else "
    Fail", result_text, expected_outcome, outcome), file=open(logFile, "a"))
428 
429     # testing nothing
430     tokenizerTest("", [])
431     # testing EOL
432     tokenizerTest("\n", [EOL()])
433     tokenizerTest(";", [EOL()])
434     # testing OpenBracket
435     tokenizerTest("(", [OpenBracket()])
436     # testing ClosingBracket
437     tokenizerTest(")", [ClosedBracket()])
438     # testing Comma
439     tokenizerTest(",", [Comma()])
440     # testing ComparisonOperator
441     tokenizerTest("<", [ComparisonOperator("<")])
442     tokenizerTest(">", [ComparisonOperator(">")])
443     tokenizerTest("<=", [ComparisonOperator("<=")])
444     tokenizerTest(">=", [ComparisonOperator(">=")])
445     tokenizerTest("==", [ComparisonOperator("==")])
446     # testing Operator
447     tokenizerTest("+", [Operator("+")])
448     tokenizerTest("-", [Operator("-")])
449     tokenizerTest("*", [Operator("*")])
450     tokenizerTest("/", [Operator("/")])
451     tokenizerTest("^", [Operator("^")])
452     # testing ConditionalOperator
453     tokenizerTest("|", [ConditionalOperator()])
```

```
454     # testing Equality
455     tokenizerTest("=", [Equality("=")])
456     tokenizerTest("~", [Equality("~")])
457     # testing NameSpace
458     # testing first line of BNF
459     tokenizerTest("a", [NameSpace("a")])
460     tokenizerTest("letter", [NameSpace("letter")])
461     # testing character number rule
462     tokenizerTest("a1", [NameSpace("a1")])
463     tokenizerTest("ae3", [NameSpace("ae3")])
464     tokenizerTest("ae31", [NameSpace("ae31")])
465     # testing interspersing numbers in letters rule
466     possibleLetters = [*string.ascii_letters]
467     possibleDigits = [*string.digits]
468     for _ in range(1):
469         sample = [random.choice(possibleLetters)]+[random.choice(
    possibleDigits)]+[random.choice(possibleLetters)]
470         test = "".join(sample)
471         tokenizerTest(test, [NameSpace(test)])
472     # testing for ending with digit
473     for _ in range(1):
474         sample = [random.choice(possibleLetters)]+[random.choice(
    possibleDigits)]+[random.choice(possibleLetters)]+[random.choice(
    possibleDigits)]
475         test = "".join(sample)
476         tokenizerTest(test, [NameSpace(test)])
477
478     # testing Number
479     tokenizerTest("0", [Number("0")])
480     tokenizerTest("123", [Number("123")])
481     tokenizerTest("2.4", [Number("2.4")])
482     tokenizerTest("2.44", [Number("2.44")])
483     tokenizerTest("22.4", [Number("22.4")])
484     tokenizerTest("62.42", [Number("62.42")])
485     # testing fail/ invalid inputs
486     # invalid number
487     tokenizerTest(".429r4", [], ScanError("the given input feed \".429r4\"
    cannot be matched to any defined token"))
488     tokenizerTest("84889.393.24", [], ScanError("the given input feed \".24\"
    cannot be matched to any defined token"))
489     # letter number mash
490     tokenizerTest("2944r834,e.32r.", [],
491                   ScanError("the given input feed \".32r.\" cannot be matched
    to any defined token"))
492     # testing multiple tokens in one line
493     tokenizerTest("|)(7647*,<=/319.42uein~-\n2+iko2f3;ry13",
494                   [ConditionalOperator(), ClosedBracket(), OpenBracket(),
    Number("7647"), Operator("*"), Comma(),
495                    ComparisonOperator("<="), Operator("/"), Number("319.42"),
    NameSpace("uein"), Equality("~"),
496                    Operator("-"), EOL(), Number("2"), Operator("+"), NameSpace
    ("iko2f3"), EOL(), NameSpace("ry13")])
497
498
499 if __name__ == "__main__":
500     LexerTest()
501
```

File - C:\Users\Andy\PycharmProjects\NEAs\code\monad.py

```python
from typing import Any

testing = __name__ == "__main__"


class Monad:
    def __init__(self, state):
        self._state = state

    @classmethod
    def just(cls, target) -> 'Monad':
        """
        wraps target
        :param target: value to be wraped
        :return:
        """
        raise NotImplementedError(f"not implemented by {cls.__name__}")

    def map(self, func) -> 'Monad':
        """
        applies func to wrapped value
        :param func:
        :return:
        """
        raise NotImplementedError(f"not implemented by {type(self).__name__}")

    def bind(self, func) -> 'Monad':
        """
        applies func and unwraps value
        :param func: function to be bound
        :return: new wrapped value
        """
        raise NotImplementedError(f"not implemented by {type(self).__name__}")

    def get_internal_implementation(self):
        """
        retrieves wrapped value
        :return: internal value
        """
        return self._state

    def __rshift__(self, function) -> 'Monad':
        """
        syntax sugar to mimic haskell bind syntax
        :param function: function to be bound
        :return: new wrapped value
        """
        return self.bind(function)


class Maybe(Monad):
    def __init__(self, target):
        super(Maybe, self).__init__(target)
        self.__target = target  # duplicate of _state of Monad

    @classmethod
    def just(cls, target):
        return Maybe(target)

```

```
60      def map(self, func):
61          return (Maybe(func(self.__target)) if not isinstance(self.__target,
    Monad) else Maybe(self.__target.map(func))) if self.__target is not None else
    self
62
63      def bind(self, func):
64          return Maybe(func(self.__target).__target if self.__target is not None
     else None)
65
66      def __eq__(self, other):
67          if not isinstance(other, Maybe):
68              return False
69          return self.__target == other.__target
70
71      def __str__(self):
72          return f"Maybe({self.__target})"
73
74      def __repr__(self):
75          return repr(self.__target)
76
77
78  def MaybeTest():
79      print("\ntesting maybeMonad")
80      maybe_divide = lambda num2: lambda num1: Maybe(None) if ((num2 == 0) or (
    num1 % num2 != 0)) else Maybe(num1 // num2)
81      print(
82          Maybe(20)
83          .bind(maybe_divide(5))
84          .bind(maybe_divide(6))
85          .bind(maybe_divide(2))
86      )
87      print(
88          Maybe(210)
89          >> maybe_divide(5)
90          >> maybe_divide(2)
91          >> maybe_divide(7)
92      )
93      r = Maybe(2)
94      r >>= maybe_divide(2)
95      print(r)
96      print("finished testing maybeMonad\n")
97
98
99  if testing:
100     MaybeTest()
101
102
103 class ListMonad(Monad):
104     def __init__(self, *items):
105         # as None is used as the end identifier
106         # it is critical to remove every instance of None when instantiating
    ListMonad
107         # to prevent a percieved false end
108         if None in items:
109             items = list(items)
110             items.remove(None)
111         # to set _state in Monad for get_internal_implementation
112         # for consistency
113         super(ListMonad, self).__init__(list(items))
```

```
114         if len(items) == 0:
115             self._head = Maybe(None)
116             self._tail = Maybe(None)
117         else:
118             self._head = Maybe(items[0])
119             self._tail = Maybe(ListMonad(*items[1:]))
120 
121     @classmethod
122     def just(cls, *target):
123         return ListMonad(*target)
124 
125     def head(self) -> Any | None:
126         """
127         getter for the first element of the list
128         :return: first element of the list or None if list is empty
129         """
130         match self._head:
131             case Maybe(_state=None):
132                 return None
133             case Maybe(_state=state):
134                 return state
135 
136     def tail(self) -> 'ListMonad | None':
137         """
138         getter for tail of the list, None if end of list
139         [].tail() -> None
140         :return: tail of the list
141         """
142         match self._tail:
143             # case when there is no head or there is no more tail
144             case Maybe(_state=None) | Maybe(_state=ListMonad(_head=None)):
145                 return None
146             case Maybe(_state=ListMonad() as t):
147                 return t
148 
149     def contents(self) -> tuple[None, None] | tuple[Any, 'ListMonad']:
150         """
151         mimics x:xs matching in haskell
152         :return: a tuple containing the head and tail of the listMonad
153         """
154         return self.head(), self.tail()
155 
156     def map(self, func) -> 'ListMonad':
157         match len(self):
158             # match empty array case
159             case 0:
160                 return self
161             # match if only one element
162             case 1:
163                 return ListMonad(head.map(func) if isinstance((head := self.
    head()), Monad) else func(head))
164             # if there is more than one element in the list
165             case int(x) if x > 1:
166                 temp = ListMonad(head.map(func) if isinstance((head := self.
    head()), Monad) else func(head))
167                 if temp.head() is None:
168                     return self.tail().map(func)
169                 temp **= self.tail().map(func)
170                 return temp
```

```
171         raise KeyError("unknown case encountered")
172
173     def bind(self, func) -> 'ListMonad':
174         return self.flatmap(func)
175
176     def flatmap(self, func) -> 'ListMonad':
177         """
178         takes in a subroutine that transforms elements in the list into
    ListMonads and then concatenates all the ListMonads
179         :param func: function that transforms elements into ListMonads
180         :return: ListMonad with func applied to elements of the list and
    flattens into list
181         """
182         match len(self):
183             # match empty array case
184             case 0:
185                 return self
186             # match if only one element
187             case 1:
188                 return func(head) if not isinstance((head := self.head()),
    Monad) else head.map(func)
189             # when there is more than one element
190             case _:
191                 return (func(head) if not isinstance((head := self.head()),
    Monad) else head.map(func)) ** self.tail().flatmap(func)
192
193     def __pow__(self, other, modulo=None) -> 'ListMonad':
194         """
195         List concatenation to override python ** operator mimicing Haskell
    syntax
196         :param other: the other ListMonad to concatenate with
197         :param modulo: not supported
198         :return: concatenated ListMonad
199         """
200         if modulo is not None:
201             raise NotImplementedError("modulo not implemented by ListMonad")
202         match len(self):
203             # match empty array case
204             case 0:
205                 temp = ListMonad()
206                 temp._head = Maybe(other.head())
207                 temp._tail = Maybe(other.tail())
208                 return temp
209             # match if only one element
210             case 1:
211                 temp = ListMonad()
212                 temp._head = Maybe(self.head())
213                 temp._tail = Maybe(other)
214                 return temp
215             case _:
216                 temp = ListMonad()
217                 temp._head = Maybe(self.head())
218                 temp._tail = Maybe(self.tail() ** other)
219                 return temp
220
221     def __len__(self) -> int:
222         """
223         calculates length of the ListMonad
224         :return: length of list
```

```
225         """
226         match (self._head, self._tail):
227             # match empty array case
228             case (Maybe(_state=None), Maybe(_state=None)):
229                 # base case
230                 return 0
231             # match if only one element
232             case (Maybe(_state=head), Maybe(_state=ListMonad(_head=Maybe(
    _state=None)))):
233                 # base case
234                 return 1
235             case (Maybe(_state=head), Maybe(_state=ListMonad() as tail)):
236                 # recursive case
237                 return 1 + len(tail)
238 
239     def to_list(self) -> list:
240         """
241         converts back into python list
242         :return: list
243         """
244         match len(self):
245             # match empty array case
246             case 0:
247                 return []
248             # match if only one element
249             case 1:
250                 return [self.head()]
251             case _:
252                 return [self.head()]+self.tail().to_list()
253 
254     def get_internal_implementation(self) -> list:
255         """
256         gets the ListMonad into list
257         :return:
258         """
259         return self.to_list()
260 
261     def __str__(self):
262         """
263         get the string form of the ListMonad
264         :return:
265         """
266         match (self._head, self._tail):
267             # match empty array case
268             case (Maybe(_state=None), Maybe(_state=None)):
269                 return "[]"
270             # match if only one element
271             case (Maybe(_state=head), Maybe(_state=ListMonad(_head=Maybe(
    _state=None)))):
272                 return f"[{head}]"
273             case (Maybe(_state=head), Maybe(_state=ListMonad() as tail)):
274                 return f"[{head},{str(tail)[1:-1]}]"
275             case _:
276                 raise KeyError("unknown case encountered")
277 
278     def __repr__(self):
279         # for debugging
280         return str([self._head, repr(self._tail)])
281 
```

```
282
283 def ListMonad_test():
284     """testing listMonad"""
285     print("\ntesting listMonad")
286     l = ListMonad(3, 5, 2, 4, 5, 2)
287     # print(repr(l))
288     print("6 [3,5,2,4,5,2]: expected")
289     print(len(l), l)
290     l = l.map(lambda a: a+3)
291     print("6 [6,8,5,7,8,5]: expected")
292     print(len(l), l)
293     l = l.map(lambda a: a//2 if a % 2 == 0 else None)
294     print("3 [3,4,4]: expected")
295     print(len(l), l)
296     l1 = ListMonad("T", "r", "s", "t")
297     print("4 [T,r,s,t]: expected")
298     print(len(l1), l1)
299     l2 = l ** l1
300     print("7 [3,4,4,T,r,s,t]: expected")
301     print(len(l2), l2)
302     l1 = l1.map(lambda a: a if a != "r" else "e")
303     print("4 [T,e,s,t]: expected")
304     print(len(l1), l1)
305     print("7 [3,4,4,T,r,s,t]: expected")
306     print(len(l2), l2)
307     print("[T,e,s,t,3,4,4,T,r,s,t]: expected")
308     print(l1**l2)
309     print("['T', 'e', 's', 't', 3, 4, 4, 'T', 'r', 's', 't']: expected")
310     print((l1 ** l2).to_list())
311     l = ListMonad(24)
312     split = lambda num: (lambda a: ListMonad(num, a//num) if a % num == 0 and
    a != num else ListMonad(a))
313     split2 = split(2)
314     l = l.flatmap(split2)
315     print(l)
316     l = l.flatmap(split(3))
317     print(l)
318     l = l.flatmap(split2)
319     print(l)
320     l = l.flatmap(split2)
321     print(l)
322
323     print("done testing listMonad\n")
324
325
326 if testing:
327     ListMonad_test()
328
```

```
"""
Module responsible for parsing the tokens generated by the Lexer/Scanner into an AST
"""

from abc import abstractmethod
from typing import Type

# hackish method
from lexer import tokenize, Token, EOL, OpenBracket, ClosedBracket, Comma, ComparisonOperator, Operator, ConditionalOperator, Equality, NameSpace, Number
from utilities import *

__all__ = ["parse", "ParseError"]
# prevent language BNF from becoming docstring


"""
Language BNF

Parser Rules
<Lines>             ::= <Line>
                      | <Line> <Lines>

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

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

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

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

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

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

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

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

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

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

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

```

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

Note to future coders looking into this code
the reason the token lists can be consecutively passed and items retrieved from the list without effecting
the original list when a NotCompatibleException is raised
is because "removal" is getting/indexing the value and reassigning the token variable to the new list slice without the retrieved values,
this slice is a new list reference with the same values from the old list and hence does not affect the original list
"""


class Node(JSONable):
    """
    abstract bass class for all nodes/rule in the expression tree
    """
    @classmethod
    @abstractmethod
    def consume(cls, tokens: list[Token], statements=()) -> tuple['Node', list[Token]]:
        """
        matches the token feed into the rules of the Node
        chain of responsibility pattern
        object factory
        :param statements: statements to be identified
        :raises NotCompatibleException: raised when the token feed does not match the rules of the node
        :param tokens: token Input
        :return: the node and the remaining tokens
        """
        raise NotImplementedError("Take method is not implemented by "+cls.__name__)

    @abstractmethod
    def get_json(self) -> dict:
        """
        serializes object
        returns the node in a JSON format
        :return: "JSON" object
        """
        raise NotImplementedError("JSON method is not implemented by "+type(self).__name__)

    @abstractmethod
    def __eq__(self, other) -> bool:
        """
        determines if 2 Nodes are the same
        :param other: the other node
        :return: if the two nodes have the same properties
        """
        raise NotImplementedError("eq is not implemented by "+type(self).__name__)


class Inequality(Node):
    # Inequality>        ::= <Term> <ComparisonOperator> <Term> | <Term> <ComparisonOperator> <Term> <ComparisonOperator> <Term>
```

```
108     def __init__(self, term1: 'Term', operator1: ComparisonOperator, term2: '
    Term', operator2: ComparisonOperator = None, term3: 'Term' = None):
109         self._is_sandwiched = operator2 is not None and term3 is not None
110         # if it is not valid/ complete sandwich
111         if not self._is_sandwiched and (operator2 is not None or term3 is not
    None):
112             raise ValueError("there must be 3 terms and 2 operators or 2 terms
     and 1 operator, there must be no in-between")
113         self._terms = [term1, term2]
114         self._operators = [operator1]
115         if self._is_sandwiched:
116             self._terms.append(term3)
117             self._operators.append(operator2)
118         if any(map(lambda a: not isinstance(a, Term), self._terms)) or any(map
    (lambda a: not isinstance(a, ComparisonOperator), self._operators)):
119             raise TypeError("terms and operators must be Term Node and
    ComparisonOperator Token respectively")
120 
121     # noinspection PyTypeChecker
122     @classmethod
123     def consume(cls, tokens: list[Token], statements=()) -> tuple['Inequality'
    , list[Token]]:
124         # heuristic
125         if len(tokens) == 0:
126             raise NotCompatibleException
127         term1, tokens = Term.consume(tokens, statements)
128         # an inequality must have an operator
129         if len(tokens) == 0 or not isinstance(tokens[0], ComparisonOperator):
130             raise NotCompatibleException
131 
132         operator1, tokens = tokens[0], tokens[1:]
133         term2, tokens = Term.consume(tokens, statements)
134 
135         # if there is no second operator, that's it
136         if len(tokens) == 0 or not isinstance(tokens[0], ComparisonOperator):
137             return Inequality(term1, operator1, term2), tokens
138 
139         operator2, tokens = tokens[0], tokens[1:]
140         term3, tokens = Term.consume(tokens, statements)
141         return Inequality(term1, operator1, term2, operator2, term3), tokens
142 
143     def get_json(self) -> dict:
144         return {
145             "Type": "Inequality",
146             "Terms": list(map(lambda a: a.get_json(), self._terms)),
147             "Comparisons": list(map(lambda a: a.get_json(), self._operators))
148         }
149 
150     # noinspection PyProtectedMember
151     def __eq__(self, other):
152         if not isinstance(other, Inequality):
153             return False
154         return self._terms == other._terms and self._operators == other.
    _operators
155 
156 
157 class Conditional(Node):
158     # <Conditional>       ::= <Inequality> <ConditionalOperator> <Inequality>
159     def __init__(self, conditions: list[Inequality]):
```

```
160         if len(conditions) != 2 or any(map(lambda a: not isinstance(a,
    Inequality), conditions)):
161             raise ValueError("a Conditonal must take 2 conditions/
    Inequalities")
162         self._conditions = conditions
163 
164     @classmethod
165     def consume(cls, tokens: list[Token], statements=()) -> tuple['Conditional
    ', list[Token]]:
166         # heuristic
167         if len(tokens) == 0:
168             raise NotCompatibleException
169         inequality1, tokens = Inequality.consume(tokens, statements)
170         # a conditional must have a conditional operator
171         if len(tokens) == 0 or not isinstance(tokens[0], ConditionalOperator):
172             raise NotCompatibleException
173         inequality2, tokens = Inequality.consume(tokens[1:], statements)
174         return Conditional([inequality1, inequality2]), tokens
175 
176     def get_json(self) -> dict:
177         return {
178             "Type": "Conditional",
179             "Conditions": list(map(lambda a: a.get_json(), self._conditions))
180         }
181 
182     # noinspection PyProtectedMember
183     def __eq__(self, other):
184         if not isinstance(other, Conditional):
185             return False
186         return self._conditions == other._conditions
187 
188 
189 class Operand(Node):
190     # <Operand>           ::= <Conditional> | <Inequality> | <Expression>
191     def __init__(self, operand: 'Expression | Inequality | Conditional'):
192         if not isinstance(operand, (Expression, Inequality, Conditional)):
193             raise ValueError("operand must be either an Expression, an
    Inequality, ot a Conditional, not a "+type(operand).__name__)
194         self._operand = operand
195 
196     @classmethod
197     def consume(cls, tokens: list[Token], statements=()) -> tuple['Operand',
    list[Token]]:
198         # heuristic
199         if len(tokens) == 0:
200             raise NotCompatibleException
201         # operand is either a conditional
202         try:
203             conditional, tokens = Conditional.consume(tokens, statements)
204             return Operand(conditional), tokens
205         except NotCompatibleException:
206             pass
207         # or an inequality
208         try:
209             inequality, tokens = Inequality.consume(tokens, statements)
210             return Operand(inequality), tokens
211         except NotCompatibleException:
212             pass
213         # or an expression
```

```
214         try:
215             expression, tokens = Expression.consume(tokens, statements)
216             return Operand(expression), tokens
217         except NotCompatibleException:
218             pass
219         # otherwise, it's not an operand
220         raise NotCompatibleException
221 
222     def get_json(self) -> dict:
223         return {
224             "Type": "Operand",
225             "Operand": self._operand.get_json()
226         }
227 
228     # noinspection PyProtectedMember
229     def __eq__(self, other):
230         if not isinstance(other, Operand):
231             return False
232         return self._operand == other._operand
233 
234 
235 class Operands(Node):
236     # <Operands>          ::= ε | <Operand> | <Operand> <Comma> <Operands>
237     # following are unexpected but permitted behaviours/syntax as defined by
    the BNF
238     # <Operand> <Comma>
239     # <Operand> <Comma> <Operand> <Comma>
240     # ...
241     def __init__(self, *operands: Operand):
242         if any(map(lambda a: not isinstance(a, Operand), operands)):
243             raise TypeError("operands are supposed to be type Operand")
244         self._operands = list(operands)
245 
246     @classmethod
247     def consume(cls, tokens: list[Token], statements=()) -> tuple['Operands',
    list[Token]]:
248         # if there is no operand to be matched
249         try:
250             operand, tokens = Operand.consume(tokens, statements)
251         except NotCompatibleException:
252             return Operands(), tokens
253         # if there is nothing to follow up or a comma does not follow, it's
    done
254         if len(tokens) == 0 or not isinstance(tokens[0], Comma):
255             return Operands(operand), tokens
256         # remove comma token
257         tokens = tokens[1:]
258         # recursive definition
259         operands, tokens = Operands.consume(tokens, statements)
260         return Operands(operand, *operands._operands), tokens
261 
262     # noinspection PyProtectedMember
263     def __eq__(self, other):
264         if not isinstance(other, Operands):
265             return False
266         return self._operands == other._operands
267 
268     def get_json(self) -> dict:
269         return {
```

```
270             "Type": "Operands",
271             "Operands": list(map(lambda a: a.get_json(), self._operands))
272         }
273 
274 
275 class Function(Node):
276     # <Function>          ::= <NameSpace> <OpenBracket> <Operands> <
    ClosedBracket>
277     def __init__(self, name: NameSpace, operands: Operands):
278         if not isinstance(name, NameSpace) or not isinstance(operands,
    Operands):
279             raise TypeError("name and operands should be NameSpace and
    Operands respectively")
280         self._name = name
281         self._operands = operands
282 
283     # noinspection PyTypeChecker
284     @classmethod
285     def consume(cls, tokens: list[Token], statements=()) -> tuple['Function',
    list[Token]]:
286         # if there is no nameSpace in the beginning, there's no chance it's a
    function
287         if len(tokens) == 0 or not isinstance(tokens[0], NameSpace):
288             raise NotCompatibleException
289         # assign the nameSpace to variable name
290         name, tokens = tokens[0], tokens[1:]
291         if name.get_json()["Name"] in statements:
292             raise NotCompatibleException("function expected, not statement")
293         # if there is no openBracket following the name, it's not a function
294         if len(tokens) == 0 or not isinstance(tokens[0], OpenBracket):
295             raise NotCompatibleException
296         operands, tokens = Operands.consume(tokens[1:], statements)
297         if len(tokens) == 0 or not isinstance(tokens[0], ClosedBracket):
298             raise NotCompatibleException
299         return Function(name, operands), tokens[1:]
300 
301     def get_json(self) -> dict:
302         return {
303             "Type": "Function",
304             "Name": self._name.get_json(),
305             "Operands": self._operands.get_json()
306         }
307 
308     # noinspection PyProtectedMember
309     def __eq__(self, other):
310         if not isinstance(other, Function):
311             return False
312         return self._name == other._name and self._operands == other._operands
313 
314 
315 class Assignable(Node):
316     # <Assignable>        ::= <Function> | <NameSpace>
317     def __init__(self, variable: Function | NameSpace):
318         self._variable = variable
319 
320     # noinspection PyTypeChecker
321     @classmethod
322     def consume(cls, tokens: list[Token], statements=()) -> tuple['Assignable'
    , list[Token]]:
```

```
323         # heuristic
324         # if the first token is not a NameSpace, it is not an assignable
325         # this is because an assignable is either a function or a namespace,
    both of which start with a namespace token
326         if len(tokens) == 0 or not isinstance(tokens[0], NameSpace):
327             raise NotCompatibleException
328         try:
329             func, tokens = Function.consume(tokens, statements)
330             return Assignable(func), tokens
331         except NotCompatibleException:
332             pass
333         return Assignable(tokens[0]), tokens[1:]
334 
335     def get_json(self) -> dict:
336         return {
337             "Type": "Assignable",
338             "Assignee": self._variable.get_json()
339         }
340 
341     # noinspection PyProtectedMember
342     def __eq__(self, other):
343         if not isinstance(other, Assignable):
344             return False
345         return self._variable == other._variable
346 
347 
348 class BracketedTerm(Node):
349     # <BracketedTerm>     ::= <OpenBracket> <Expression> <ClosedBracket>
350     def __init__(self, expression: 'Expression'):
351         if not isinstance(expression, Expression):
352             raise TypeError("expression should be an Expression, not "+type(
    expression).__name__)
353         self._expression = expression
354 
355     # noinspection PyTypeChecker
356     @classmethod
357     def consume(cls, tokens: list[Token], statements=()) -> tuple['
    BracketedTerm', list[Token]]:
358         # bracketed term starts with an openbracket
359         if len(tokens) == 0 or not isinstance(tokens[0], OpenBracket):
360             raise NotCompatibleException
361         expression, tokens = Expression.consume(tokens[1:], statements)
362         # and ends with a closed bracket
363         if len(tokens) == 0 or not isinstance(tokens[0], ClosedBracket):
364             raise NotCompatibleException
365         return BracketedTerm(expression), tokens[1:]
366 
367     def get_json(self) -> dict:
368         return {
369             "Type": "BracketedTerm",
370             "expression": self._expression.get_json()
371         }
372 
373     # noinspection PyProtectedMember
374     def __eq__(self, other):
375         if not isinstance(other, BracketedTerm):
376             return False
377         return self._expression == other._expression
378 
```

```
379
380 class Term(Node):
381     # <Term>              ::= <Number> | <Function> | <NameSpace> | <Operator
    > <Term> | <BracketedTerm>
382     def __init__(self, item: Number | Function | NameSpace | BracketedTerm,
    negated: bool = False):
383         if not isinstance(item, (Number, Function, NameSpace, BracketedTerm)):
384             raise TypeError("item should not be "+type(item).__name__)
385         self._item = item
386         self._isNegated = negated
387
388     # noinspection PyTypeChecker,PyPep8Naming
389     @classmethod
390     def consume(cls, tokens: list[Token], statements=()) -> tuple['Term', list
    [Token]]:
391         # heuristic
392         if len(tokens) == 0:
393             raise NotCompatibleException
394         tokenHeadType = type(tokens[0])
395
396         if tokenHeadType == Operator:
397             if tokens[0].get_json()["OperatorType"] != "-":
398                 raise NotCompatibleException("only negating is allowed")
399             term, tokens = Term.consume(tokens[1:], statements)
400             return Term(term._item, not term._isNegated), tokens
401
402         elif tokenHeadType == Number:
403             return Term(tokens[0]), tokens[1:]
404
405         elif tokenHeadType == OpenBracket:
406             # try bracketed term, it can't be anything else
407             bracketed, tokens = BracketedTerm.consume(tokens, statements)
408             return Term(bracketed), tokens
409
410         elif tokenHeadType == NameSpace:
411             try:
412                 func, tokens = Function.consume(tokens, statements)
413                 return Term(func), tokens
414             except NotCompatibleException:
415                 # if it's not a function, it's a variable
416                 return Term(tokens[0]), tokens[1:]
417         else:
418             raise NotCompatibleException
419
420     def get_json(self) -> dict:
421         return {
422             "Type": ("Negated" if self._isNegated else "")+"Term",
423             "Content": self._item.get_json()
424         }
425
426     # noinspection PyProtectedMember
427     def __eq__(self, other):
428         if not isinstance(other, Term):
429             return False
430         return self._item == other._item and self._isNegated == other.
    _isNegated
431
432
433 class Expression(Node):
```

```
434     # <Expression>        ::= <Term> | <Term> <Operator> <Expression>
435     def __init__(self, *terms: Term | Operator):
436         if any(map(lambda a: not isinstance(a, (Term, Operator)), terms)):
437             raise TypeError("terms must be of the type Term node or Operator
    token")
438         self._terms = list(terms)
439 
440     # noinspection PyTypeChecker
441     @classmethod
442     def consume(cls, tokens: list[Token], statements=()) -> tuple['Expression'
    , list[Token]]:
443         # heuristic
444         if len(tokens) == 0:
445             raise NotCompatibleException
446         term, tokens = Term.consume(tokens, statements)
447         # if an operator does not follow, the expression is over/ does not fit
     the other definition
448         if len(tokens) == 0 or not isinstance(tokens[0], Operator):
449             return Expression(term), tokens
450         operator, tokens = tokens[0], tokens[1:]
451         # recursive rule
452         expression, tokens = Expression.consume(tokens, statements)
453         return Expression(term, operator, *expression._terms), tokens
454 
455     def get_json(self) -> dict:
456         return {
457             "Type": "Expression",
458             "TermsAndOperators": list(map(lambda a: a.get_json(), self._terms
    ))
459         }
460 
461     # noinspection PyProtectedMember
462     def __eq__(self, other):
463         if not isinstance(other, Expression):
464             return False
465         return self._terms == other._terms
466 
467 
468 class Assignment(Node):
469     # <Assignment>        ::= <Assignable> <Equality> <Expression>
470     def __init__(self, assignee: Assignable, equality: Equality, expression:
    Expression):
471         if (not isinstance(assignee, Assignable) or
472             not isinstance(equality, Equality) or
473                 not isinstance(expression, Expression)):
474             raise TypeError("assignee, equality, and expression must be of
    type Assignable node, Equality token, and Expression node")
475         self._assigned = assignee
476         self._equals = equality
477         self._expression = expression
478 
479     # noinspection PyTypeChecker
480     @classmethod
481     def consume(cls, tokens: list[Token], statements=()) -> tuple['Assignment'
    , list[Token]]:
482         # heuristic
483         if len(tokens) == 0:
484             raise NotCompatibleException
485         assignable, tokens = Assignable.consume(tokens, statements)
```

```
486         if len(tokens) == 0 or not isinstance(tokens[0], Equality):
487             raise NotCompatibleException
488         equals, tokens = tokens[0], tokens[1:]
489         expression, tokens = Expression.consume(tokens, statements)
490         return Assignment(assignable, equals, expression), tokens
491 
492     def get_json(self) -> dict:
493         return {
494             "Type": "Assignment",
495             "Assigned": self._assigned.get_json(),
496             "Equality": self._equals.get_json(),
497             "Expression": self._expression.get_json()
498         }
499 
500     # noinspection PyProtectedMember
501     def __eq__(self, other):
502         if not isinstance(other, Assignment):
503             return False
504         return (self._assigned == other._assigned and
505                 self._equals == other._equals and
506                 self._expression == other._expression)
507 
508 
509 class Statement(Node):
510     # <Statement>: := <NameSpace> <OpenBracket> <Operands> <ClosedBracket>
511     def __init__(self, name: NameSpace, operands: Operands):
512         if not isinstance(name, NameSpace) or not isinstance(operands,
    Operands):
513             raise TypeError("name and operands must be of type NameSpace and
    Operands respectively, not "+type(name).__name__+", "+type(operands).__name__)
514         # mutable default values causes pointer issues(ie referencing mess),
    easy bug origin
515         self._name = name
516         self._operands = operands
517 
518     # noinspection PyTypeChecker
519     @classmethod
520     def consume(cls, tokens: list[Token], statements=None) -> tuple['Statement
    ', list[Token]]:
521         # heuristic
522         if len(tokens) < 3 or not isinstance(tokens[0], NameSpace):
523             raise NotCompatibleException
524         name, tokens = tokens[0], tokens[1:]
525         if name.get_json()["Name"] not in statements:
526             raise NotCompatibleException
527         if not isinstance(tokens[0], OpenBracket):
528             raise NotCompatibleException
529         tokens = tokens[1:]
530         operands, tokens = Operands.consume(tokens, statements)
531         if not isinstance(tokens[0], ClosedBracket):
532             raise NotCompatibleException
533         tokens = tokens[1:]
534         return Statement(name, operands), tokens
535 
536     def get_json(self) -> dict:
537         return {
538             "Type": "Statement",
539             "Name": self._name.get_json(),
540             "Operands": self._operands.get_json()
```

```
541         }
542 
543     # noinspection PyProtectedMember
544     def __eq__(self, other) -> bool:
545         if not isinstance(other, Statement):
546             return False
547         return self._name == other._name and self._operands == other._operands
548 
549 
550 class Line(Node):
551     # <Line>              ::= <Assignment> <EOL> | <Expression> <EOL> | <
    Statement> <EOL> | <EOL>
552     def __init__(self, action: Assignment | Expression | Statement | None):
553         if not isinstance(action, (Assignment, Expression, Statement, type(
    None))):
554             raise TypeError("action must be an Assignment, Expression, or None
    ")
555         self._action = action
556 
557     @classmethod
558     def consume(cls, tokens: list[Token], statements=None) -> tuple['Line',
    list[Token]]:
559         try:
560             action, tokens = Statement.consume(tokens, statements)
561         except NotCompatibleException:
562             try:
563                 action, tokens = Assignment.consume(tokens, statements)
564             except NotCompatibleException:
565                 try:
566                     action, tokens = Expression.consume(tokens, statements)
567                 except NotCompatibleException:
568                     # placeholder for Nothing (last bit of BNF)
569                     action = None
570         if len(tokens) == 0 or not isinstance(tokens[0], EOL):
571             raise NotCompatibleException
572         return Line(action), tokens[1:]
573 
574     def get_json(self) -> dict:
575         return {
576             # recreates last rule
577             "Type": ("Empty" if self._action is None else "")+"Line",
578             "Action": self._action.get_json() if self._action is not None else
     "None"
579         }
580 
581     # noinspection PyProtectedMember
582     def __eq__(self, other):
583         if not isinstance(other, Line):
584             return False
585         return self._action == other._action
586 
587 
588 class Lines(Node):
589     # <Lines>             ::= <Line> | <Line> <Lines>
590     def __init__(self, *line: Line):
591         if any(map(lambda a: not isinstance(a, Line), line)):
592             raise NotCompatibleException
593         self._lines = list(line)
594 
```

```
595     @classmethod
596     def consume(cls, tokens: list[Token], statements=None) -> tuple['Lines',
    list[Token]]:
597         line, tokens = Line.consume(tokens, statements)
598         if len(tokens) == 0:
599             return Lines(line), tokens
600         try:
601             # recursive definition
602             lines, tokens = Lines.consume(tokens, statements)
603             return Lines(line, *lines._lines), tokens
604         except NotCompatibleException:
605             return Lines(line), tokens
606 
607     def get_json(self) -> dict:
608         return {
609             "Type": "Lines",
610             "Lines": list(map(lambda a: a.get_json(), self._lines))
611         }
612 
613     # noinspection PyProtectedMember
614     def __eq__(self, other):
615         if not isinstance(other, Lines):
616             return False
617         return self._lines == other._lines
618 
619 
620 @test("Parser")
621 def parseTest():
622     """
623     test for parser
624     :return: None
625     """
626     # noinspection PyPep8Naming
627     logFile = "../log.txt"
628     with open(logFile, "w") as e:
629         e.write("Parser Test Results:\n")
630         e.write("""
631  Test Result |  Type Tested  |       Input       |                    Expected
                    |                      Actual                    |
632  (Pass/fail) |               |                   |                    Outcome
                     |                      Outcome                     |
633 -------------+---------------+-------------------+----------------------------
    ------------------+-----------------------------------------------+
634 """[1:])
635 
636     def tester(inp: str, obj: Type[Node], expectedOutput: Node,
    expectedRemainder: list[Token], expectedError: Exception = None) -> None:
637         """
638         Tests if Take method of Node obj will produce the expected result and
    logs result into the log.txt text file
639         :param inp: typical input feed
640         :param obj: the node to be taking the tokens
641         :param expectedOutput: the node to be expected
642         :param expectedRemainder: the trailing tokens after the
643         :param expectedError: error expected(if any)
644         :return: None
645         """
646         # noinspection PyShadowingNames
647         # ("Settings",) as list of statements for testing purposes
```

```
648
649         try:
650             result = (outcome := obj.consume(tokenize(inp), ("Settings"
    ,))) == (expectedOutput, expectedRemainder) and expectedError is None
651             outcome = outcome[0]
652             if expectedError is not None:
653                 outcome = type(expectedError).__name__
654         except Exception as e:
655             result = e == expectedError
656             if not result:
657                 outcome = type(e).__name__
658             else:
659                 outcome = type(e).__name__
660             if expectedError is None:
661                 result = False
662         inp = inp.replace("\n", "\\n")
663         inp = f"\"{inp}\""
664         expected_output = expectedOutput if expectedError is None else type(
    expectedError).__name__
665         if expectedError is None:
666             outcome, *other_lines = dict_beautify(outcome.get_json()).split("\
    n")
667             expected_output, *expected_lines = dict_beautify(expected_output.
    get_json()).split("\n")
668         else:
669             other_lines = [" "]
670             expected_lines = [" "]
671         print(inp, expected_output, outcome)
672         print("{:^13}| {:<14}| {:<18}| {:<46}| {:<46}|".format(("Pass" if
    result else "Fail"), obj.__name__, inp, expected_output, str(outcome)), file=
    open(logFile, "a"))
673         for index in range(max(len(other_lines), len(expected_lines))):
674             expected = expected_lines[index] if index < len(expected_lines)
    else ""
675             actual = other_lines[index] if index < len(other_lines) else ""
676             print(" "*13+"|"+" "*15+"|"+" "*19+"| {:<46}| {:<46}|".format(
    expected, actual), file=open(logFile, "a"))
677         print(
    "-------------+---------------+------------------+-------------------------
    ---------------------+-----------------------------------------------+", file=
    open(logFile, "a"))
678
679         # if expectedError is None:
680         #     print(dict_beautify(expectedOutput.get_json()), file=open(
    logFile, "a"))
681     # testing basic terms
682     tester("3", Term, Term(Number("3")), [])
683     tester("a", Term, Term(NameSpace("a")), [])
684     tester("-3", Term, Term(Number("3"), negated=True), [])
685     tester("--3", Term, Term(Number("3")), [])
686     tester("+3", Term, Node(), [], expectedError=NotCompatibleException("only
    negating is allowed"))
687     # testing Inequality
688     tester("4<2", Inequality, Inequality(Term(Number("4")), ComparisonOperator
    ("<"), Term(Number("2"))), [])
689     tester("a>=3.4", Inequality, Inequality(Term(NameSpace("a")),
    ComparisonOperator(">="), Term(Number("3.4"))), [])
690     tester("4==2>o", Inequality,
691            Inequality(Term(Number("4")), ComparisonOperator("=="), Term(Number
```

```
691 ("2")), ComparisonOperator(">"),
692                         Term(NameSpace("o"))), [])
693     # testing Conditional
694     tester("X>3|Y<=2", Conditional, Conditional(
695         [Inequality(Term(NameSpace("X")), ComparisonOperator(">"), Term(Number
    ("3"))),
696          Inequality(Term(NameSpace("Y")), ComparisonOperator("<="), Term(
    Number("2"))))]), [])
697     # testing basic Operand(no expressions)
698     tester("a>=3.4", Operand, Operand(Inequality(Term(NameSpace("a")),
    ComparisonOperator(">="), Term(Number("3.4")))),
699            [])
700     tester("X>3|Y<=2", Operand, Operand(Conditional(
701         [Inequality(Term(NameSpace("X")), ComparisonOperator(">"), Term(Number
    ("3"))),
702          Inequality(Term(NameSpace("Y")), ComparisonOperator("<="), Term(
    Number("2"))))])), [])
703     # testing Operands
704     tester("", Operands, Operands(), [])
705     tester("a>=3.4", Operands,
706            Operands(Operand(Inequality(Term(NameSpace("a")),
    ComparisonOperator(">="), Term(Number("3.4"))))), [])
707     tester("a>=3.4,", Operands,
708            Operands(Operand(Inequality(Term(NameSpace("a")),
    ComparisonOperator(">="), Term(Number("3.4"))))), [])
709     tester("X>3|Y<=2,a>=3.4", Operands, Operands(Operand(Conditional(
710         [Inequality(Term(NameSpace("X")), ComparisonOperator(">"), Term(Number
    ("3"))),
711          Inequality(Term(NameSpace("Y")), ComparisonOperator("<="), Term(
    Number("2"))))]), Operand(
712         Inequality(Term(NameSpace("a")), ComparisonOperator(">="), Term(Number
    ("3.4"))))), [])
713     # tester("X>3|Y<=2,a>=3.4,", Operands, Operands(Operand(Conditional(
714     #     [Inequality(Term(NameSpace("X")), ComparisonOperator(">"), Term(
    Number("3"))),
715     #      Inequality(Term(NameSpace("Y")), ComparisonOperator("<="), Term(
    Number("2"))))]), Operand(
716     #     Inequality(Term(NameSpace("a")), ComparisonOperator(">="), Term(
    Number("3.4"))))), [])
717     # testing Function
718     tester("func()", Function, Function(NameSpace("func"), Operands()), [])
719     tester("func)", Function, Node(), [], expectedError=NotCompatibleException
    ())
720     tester("P(a>=3.4)", Function, Function(NameSpace("P"), Operands(
721         Operand(Inequality(Term(NameSpace("a")), ComparisonOperator(">="),
    Term(Number("3.4")))))), [])
722     tester("Settings()", Function, Node(), [], expectedError=
    NotCompatibleException("function expected, not statement"))
723     # testing Expression
724     tester("1+3", Expression, Expression(Term(Number("1")), Operator("+"),
    Term(Number("3"))), [])
725     tester("1+", Expression, Node(), [], expectedError=NotCompatibleException
    ())
726     # tester("func(abba1==-3.4)+3", Expression, Expression(Term(Function(
    NameSpace("func"), Operands(
727     #     Operand(Inequality(Term(NameSpace("abba1")), ComparisonOperator
    ("=="), Term(Number("3.4"), negated=True))))),
728     #                                                 Operator("+"), Term
    (Number("3"))), [])
```

```
729     # testing BracketedTerm
730     tester("(1+3)", BracketedTerm, BracketedTerm(Expression(Term(Number("1"
    )), Operator("+"), Term(Number("3")))), [])
731     # tester("(func(abba1==3.4)+3)", BracketedTerm, BracketedTerm(Expression(
    Term(Function(NameSpace("func"), Operands(
732     #     Operand(Inequality(Term(NameSpace("abba1")), ComparisonOperator
    ("=="), Term(Number("3.4")))))), Operator("+"),
733     #
    Term(Number("3")))), [])
734     # testing all Terms
735     tester("(1+3)", Term, Term(BracketedTerm(Expression(Term(Number("1")),
    Operator("+"), Term(Number("3"))))), [])
736     # tester("(func(abba1==3.4)+3)", Term, Term(BracketedTerm(Expression(Term(
    Function(NameSpace("func"), Operands(
737     #     Operand(Inequality(Term(NameSpace("abba1")), ComparisonOperator
    ("=="), Term(Number("3.4")))))), Operator("+"),
738     #                                                                 Term(
    Number("3"))))), [])
739     # testing Assignable
740     tester("var1", Assignable, Assignable(NameSpace("var1")), [])
741     tester("Func1()", Assignable, Assignable(Function(NameSpace("Func1"),
    Operands())), [])
742     tester("Func1(a+2,d)", Assignable, Assignable(Function(NameSpace("Func1"
    ), Operands(
743         Operand(Expression(Term(NameSpace("a")), Operator("+"), Term(Number("2
    "))))),
744         Operand(Expression(Term(NameSpace("d")))))), []) 
745     # testing Assignment
746     tester("a=4", Assignment, Assignment(Assignable(NameSpace("a")), Equality(
    "="), Expression(Term(Number("4")))), [])
747     tester("f(x)~x+1", Assignment, Assignment(Assignable(Function(NameSpace("f
    "), Operands(Operand(Expression(Term(NameSpace("x")))))),
748                                               Equality("~"), Expression(Term(
    NameSpace("x")), Operator("+"), Term(Number("1")))), [])
749     # tester("gen(x)~B(x,4/e)", Assignment,
750     #        Assignment(Assignable(Function(NameSpace("gen"), Operands(Operand
    (Expression(Term(NameSpace("x")))))),
751     #                   Equality("~"), Expression(Term(Function(NameSpace("B
    "),
752     #                                                   Operands(
    Operand(Expression(Term(NameSpace("x")))),
753     #
    Operand(
754     #
    Expression(Term(Number("4")), Operator("/"),
755 
    #
          Term(NameSpace("e")))))))))), [])
756     # testing Statements
757     tester("Settings()", Statement, Statement(NameSpace("Settings"), Operands
    ()), [])
758     # Tester("randomFunc()", Statement, Node(), [], expectedError=
    NotCompatibleException())
759     # testing Line
760     tester(";", Line, Line(None), [])
761     tester("1+3\n", Line, Line(Expression(Term(Number("1")), Operator("+"),
    Term(Number("3")))), [])
762     tester("Settings();", Line, Line(Statement(NameSpace("Settings"), Operands
    ())), [])
```

```
763     tester("func()\n", Line, Line(Expression(Term(Function(NameSpace("func"),
    Operands())))), [])
764     tester("a=4\n", Line, Line(Assignment(Assignable(NameSpace("a")), Equality
    ("="), Expression(Term(Number("4"))))),
765             [])
766     # testing Lines
767     tester(";", Lines, Lines(Line(None)), [])
768     tester(";;", Lines, Lines(Line(None), Line(None)), [])
769     tester(";1+3\n;", Lines,
770             Lines(Line(None), Line(Expression(Term(Number("1")), Operator("+"
    ), Term(Number("3")))), Line(None)), [])
771     tester("", Lines, Node(), [], expectedError=NotCompatibleException())
772     tester("l", Lines, Node(), [], expectedError=NotCompatibleException())
773     tester("whatis(a,b) = a+b;4/3+", Lines, Lines(Line(Assignment(Assignable(
    Function(NameSpace("whatis"), Operands(
774         Operand(Expression(Term(NameSpace("a")))), Operand(Expression(Term(
    NameSpace("b")))))))), Equality("="),
775                                                                 Expression(
    Term(NameSpace("a")), Operator("+"),
776
    Term(NameSpace("b"))))))),
777             [Number("4"), Operator("/"), Number("3"), Operator("+")])
778     return
779     # with open("StatsTest.txt", "r") as f:
780     #     for line in f:
781     #         print(dictBeautify(parse(scanner(line+";"))[0].JSON()))
782
783
784 class ParseError(Exception):
785     pass
786
787
788 def parse(inp: list[Token], statements=()) -> tuple[Node, list[Token]]:
789     """
790     parses input, if inp does not match language grammar, (None, inp) is
    returned
791     :param inp: the input to be parsed
792     :param statements: statements to be identified
793     :return: (SyntaxNode, remaining string)
794     :raises ParseError: raised when the token sequence is not parsable
795     """
796     try:
797         return Lines.consume(inp, statements)
798     except NotCompatibleException:
799         raise ParseError("The token sequence cannot be parsed into lines")
800
801
802 if __name__ == "__main__":
803     parseTest()
804
805     def tests():
806         """
807         subroutine that used to visually test the functionality of the parser
808         :return:
809         """
810         # stop it from being run
811         quit()
812
813         # noinspection PyPep8Naming
```

```
814         def testMatching(cls: Type[Node], line: list[Token]):
815             """
816             gets the first match from a rule in a node class
817             (only to be used within file)
818             :param cls: the node class to be matched
819             :param line: the feed
820             :return:
821             """
822             if __name__ != "__main__":
823                 return None
824             try:
825                 rootNode = cls.consume(line)
826                 return rootNode
827             except NotCompatibleException:
828                 return [None]
829         # visual tests
830         print("matched", *testMatching(Operands, tokenize("3,x,2.4")))
831         print("matched", *testMatching(Operands, tokenize("3),<=,2.4")))
832         print("matched", *testMatching(Operands, tokenize("3,x),==")))
833         print("matched", *testMatching(Inequality, tokenize("pi<x<2.4566666<0"
    )))
834         print("matched", *testMatching(Expression, tokenize("4*2+f(5<3)/f(3,3,
    x)")))
835         print("matched", *testMatching(Assignment, tokenize("var=4*2+func(5<3
    )/f(3,3.1415,x)")))
836         print("matched", *testMatching(Expression, tokenize("a*X^2+b*x+c-sqrt(
    4)+(y-3.2)^2-2\n")))
837         print("matched", *testMatching(Line, tokenize("Var=a*X^2+b*x+c-sqrt(4
    )+(y-3.2)^2-2\n")))
838         print("matched", *testMatching(Expression, tokenize("4-(vat(2>4.2<2))^
    hello(water, 4,6)")))
839         print("matched", *testMatching(Line, tokenize("v=4-(vat(2>4.2<2))^
    hello(water, 4,6)\n")))
840 
841         print(dict_beautify(NameSpace.consume("fghgfvh")[0].get_json()))
842         print(mystery := tokenize("Var=a*X^2+b*x+c-sqrt(4)+(y-3.2)^2-2\n"))
843         # noinspection PyProtectedMember
844         print("thikng", Lines.consume(mystery)[0]._lines[0]._action.
    _expression)
845         print(dict_beautify(Lines.consume(tokenize("Var=a*X^2+b*x+c-sqrt(4)+(y
    -3.2)^2-2\n"))[0].get_json()))
846         print(dict_beautify(Lines.consume(tokenize("Func();"))[0].get_json()))
847         quit()
848 
```

File - C:\Users\Andy\PycharmProjects\NEAs\code\mathobj.py

```
import math
import numpy as np
from abc import abstractmethod
from enum import Enum
from utilities import *


class TrigMode(Enum):
    """
    defines possible trig modes
    """
    DEGREES = True
    RADIANS = False


class Value:
    """
    bass class for all values that the calculator will interact with
    """
    def __new__(cls, *args, **kwargs):
        """
        intelligent object creation using the Value Constructor
        eg Value(np.nan) -> Undefined()
        Value(1) -> Number(1)
        Value([]) -> NPArray([])
        :param args: the value to be turned into values
        :param kwargs: key values
        """
        if len(args) != 1:
            raise TypeError("Incorrect number of arguments")
        if isinstance(args[0], np.ndarray):
            return object.__new__(NPArray)
        elif args[0] == np.nan:
            return object.__new__(Undefined)
        elif isinstance(args[0], (float, int)):
            return object.__new__(Number)
        else:
            return NotImplemented

    @abstractmethod
    def __call__(self, *args, range: int | np.ndarray = 100, **kwargs) -> '
Value':
        raise NotImplementedError


class NPArray(Value, Wrapper):
    """
    the value wrapper for numpy arrays
    """
    def __new__(cls, *args, **kwargs):
        return object.__new__(cls)

    def __init__(self, contents: np.ndarray):
        if not isinstance(contents, np.ndarray):
            raise TypeError("contents must be of type np.ndarray, not"+type(
contents).__name__)
        super(NPArray, self).__init__(contents)

    def __call__(self, *args, **kwargs):
```

```
58          return self
59
60      # trig function support
61      def sin(self, mode=TrigMode.DEGREES):
62          assert isinstance(mode, TrigMode)
63          multiplier = 1 if mode == TrigMode.RADIANS else math.pi / 180
64          return NPArray(np.sin(self.get_wrapped() * multiplier))
65
66      def cos(self, mode=TrigMode.DEGREES):
67          assert isinstance(mode, TrigMode)
68          multiplier = 1 if mode == TrigMode.RADIANS else math.pi / 180
69          return NPArray(np.cos(self.get_wrapped() * multiplier))
70
71      def tan(self, mode=TrigMode.DEGREES):
72          assert isinstance(mode, TrigMode)
73          multiplier = 1 if mode == TrigMode.RADIANS else math.pi / 180
74          return NPArray(np.tan(self.get_wrapped() * multiplier))
75
76      def arcsin(self, mode=TrigMode.DEGREES):
77          assert isinstance(mode, TrigMode)
78          multiplier = 1 if mode == TrigMode.RADIANS else 180 / math.pi
79          return NPArray(np.arcsin(self.get_wrapped()) * multiplier)
80
81      def arcos(self, mode=TrigMode.DEGREES):
82          assert isinstance(mode, TrigMode)
83          multiplier = 1 if mode == TrigMode.RADIANS else 180 / math.pi
84          return NPArray(np.arccos(self.get_wrapped()) * multiplier)
85
86      def arctan(self, mode=TrigMode.DEGREES):
87          assert isinstance(mode, TrigMode)
88          multiplier = 1 if mode == TrigMode.RADIANS else 180 / math.pi
89          return NPArray(np.arctan(self.get_wrapped()) * multiplier)
90
91      # miscellaneous other functions
92      def log(self, *args):
93          assert len(args) <= 1
94          return NPArray(np.log(self.get_wrapped())/np.log(np.e if len(args) ==
    0 else args[0]))
95
96      def sqrt(self):
97          return NPArray(np.sqrt(self.get_wrapped()))
98
99      # operator overloading to support +, -, *, /, **(aka ^)
100     def __add__(self, other):
101         if isinstance(other, Wrapper):
102             other = other.get_wrapped()
103             return NPArray(self._val+other)
104         return NotImplemented
105
106     def __radd__(self, other):
107         if isinstance(other, Wrapper):
108             other = other.get_wrapped()
109             return NPArray(other+self._val)
110         return NotImplemented
111
112     def __sub__(self, other):
113         if isinstance(other, Wrapper):
114             other = other.get_wrapped()
115             return NPArray(self._val-other)
```

```
116         return NotImplemented
117
118     def __rsub__(self, other):
119         if isinstance(other, Wrapper):
120             other = other.get_wrapped()
121             return NPArray(other-self._val)
122         return NotImplemented
123
124     def __mul__(self, other):
125         if isinstance(other, Wrapper):
126             other = other.get_wrapped()
127             return NPArray(self._val*other)
128         return NotImplemented
129
130     def __rmul__(self, other):
131         if isinstance(other, Wrapper):
132             other = other.get_wrapped()
133             return NPArray(other*self._val)
134         return NotImplemented
135
136     def __truediv__(self, other):
137         if isinstance(other, Wrapper):
138             other = other.get_wrapped()
139             return NPArray(self._val/other)
140         return NotImplemented
141
142     def __rtruediv__(self, other):
143         if isinstance(other, Wrapper):
144             other = other.get_wrapped()
145             return NPArray(other/self._val)
146         return NotImplemented
147
148     def __pow__(self, power, modulo=None):
149         if isinstance(power, Wrapper):
150             power = power.get_wrapped()
151             return NPArray(self._val**power)
152         return NotImplemented
153
154     def __rpow__(self, power, modulo=None):
155         if isinstance(power, Wrapper):
156             power = power.get_wrapped()
157             return NPArray(power**self._val)
158         return NotImplemented
159
160     def __neg__(self):
161         return NPArray(-self._val)
162
163
164 class Number(Value, Wrapper):
165     """
166     Value wrapper for numbers eg 0, 1, 3.2, ...
167     """
168     def __new__(cls, *args, **kwargs):
169         return object.__new__(cls)
170
171     def __init__(self, val: float):
172         if not isinstance(val, (int, float)):
173             try:
174                 super(Number, self).__init__(float(val))
```

```
175                 return
176             except TypeError:
177                 raise TypeError("val must be of type int or float, not "+type(
    val).__name__)
178         super(Number, self).__init__(float(val))
179 
180     def __float__(self):
181         return self._val
182 
183     def __call__(self, *args, **kwargs):
184         if isinstance(args[0], NPArray):
185             return NPArray(np.linspace(self._val, self._val, args[0].
    get_wrapped().size))
186         elif isinstance(args[0], Value):
187             return self
188         else:
189             return NotImplemented
190 
191     # trig function support
192     def sin(self, mode=TrigMode.DEGREES):
193         assert isinstance(mode, TrigMode)
194         multiplier = 1 if mode == TrigMode.RADIANS else math.pi/180
195         return Number(math.sin(self.get_wrapped() * multiplier))
196 
197     def cos(self, mode=TrigMode.DEGREES):
198         assert isinstance(mode, TrigMode)
199         multiplier = 1 if mode == TrigMode.RADIANS else math.pi/180
200         return Number(math.cos(self.get_wrapped() * multiplier))
201 
202     def tan(self, mode=TrigMode.DEGREES):
203         assert isinstance(mode, TrigMode)
204         multiplier = 1 if mode == TrigMode.RADIANS else math.pi/180
205         return Number(math.tan(self.get_wrapped() * multiplier))
206 
207     def arcsin(self, mode=TrigMode.DEGREES):
208         assert isinstance(mode, TrigMode)
209         multiplier = 1 if mode == TrigMode.RADIANS else 180/math.pi
210         if not (-1 <= self.get_wrapped() <= 1):
211             return Undefined()
212         return Number(math.asin(self.get_wrapped()) * multiplier)
213 
214     def arcos(self, mode=TrigMode.DEGREES):
215         assert isinstance(mode, TrigMode)
216         multiplier = 1 if mode == TrigMode.RADIANS else 180/math.pi
217         if not (-1 <= self.get_wrapped() <= 1):
218             return Undefined()
219         return Number(math.asin(self.get_wrapped()) * multiplier)
220 
221     def arctan(self, mode=TrigMode.DEGREES):
222         assert isinstance(mode, TrigMode)
223         multiplier = 1 if mode == TrigMode.RADIANS else 180/math.pi
224         if not (-1 <= self.get_wrapped() <= 1):
225             return Undefined()
226         return Number(math.atan(self.get_wrapped()) * multiplier)
227 
228     # miscellaneous function support
229     def log(self, *args):
230         assert len(args) <= 1
231         if self.get_wrapped() <= 0:
```

```
232             return Undefined()
233         return Number(math.log(self.get_wrapped(), *map(float, args)))
234 
235     def sqrt(self):
236         if self.get_wrapped() < 0:
237             return Undefined()
238         return Number(math.sqrt(self.get_wrapped()))
239 
240     # operator overloading to support +, -, *, /, **(aka ^)
241     def __add__(self, other):
242         if isinstance(other, Number):
243             return Number(self._val+other._val)
244         return NotImplemented
245 
246     def __sub__(self, other):
247         if isinstance(other, Number):
248             return Number(self._val-other._val)
249         return NotImplemented
250 
251     def __mul__(self, other):
252         if isinstance(other, Number):
253             return Number(self._val*other._val)
254         return NotImplemented
255 
256     def __truediv__(self, other):
257         try:
258             if isinstance(other, Number):
259                 return Number(self._val/other._val)
260             return NotImplemented
261         except ZeroDivisionError:
262             return Undefined()
263 
264     def __pow__(self, power, modulo=None):
265         if isinstance(power, Number):
266             return Number(self._val**power._val)
267         return NotImplemented
268 
269     def __neg__(self):
270         return Number(-self._val)
271 
272     # for debugging & testing
273     def __repr__(self):
274         return "Number: " + str(self._val)
275 
276 
277 class Undefined(Value):
278     """
279     used to represent math errors
280     """
281     def __new__(cls, *args, **kwargs):
282         return object.__new__(cls)
283 
284     def __call__(self, *args, **kwargs):
285         return self
286 
287     # trig function compatibility
288     def sin(self, mode=TrigMode.DEGREES):
289         return self
290 
```

```
291     def cos(self, mode=TrigMode.DEGREES):
292         return self
293 
294     def tan(self, mode=TrigMode.DEGREES):
295         return self
296 
297     def arcsin(self, mode=TrigMode.DEGREES):
298         return self
299 
300     def arcos(self, mode=TrigMode.DEGREES):
301         return self
302 
303     def arctan(self, mode=TrigMode.DEGREES):
304         return self
305 
306     # other function compatibility
307     def log(self, *args):
308         assert len(args) <= 1
309         return self
310 
311     def sqrt(self):
312         return self
313 
314     # operator overloading to support +, -, *, /, **(aka ^)
315     def __add__(self, other):
316         return Undefined()
317 
318     def __radd__(self, other):
319         return Undefined()
320 
321     def __sub__(self, other):
322         return Undefined()
323 
324     def __rsub__(self, other):
325         return Undefined()
326 
327     def __mul__(self, other):
328         return Undefined()
329 
330     def __rmul__(self, other):
331         return Undefined()
332 
333     def __truediv__(self, other):
334         return Undefined()
335 
336     def __rtruediv__(self, other):
337         return Undefined()
338 
339     def __pow__(self, power, modulo=None):
340         return Undefined()
341 
342     def __rpow__(self, other):
343         return Undefined()
344 
345     def __neg__(self):
346         return Undefined()
347 
348     def __repr__(self):
349         return "Undefined"
```

```
350
351
352 @test("Value")
353 def value_test():
354     o = Number(5.2)
355     a = Number(3.2)
356     one = Number(1)
357
358     print(o)
359     print(o/one)
360     print(float(o))
361     print(o+a)
362     print((o+Number(2)))
363
364     print("weird stuff")
365     print(o+Undefined())
366     print(Undefined()+o)
367     print("done adding undefined")
368     print(o/Number(0))
369     print(o/Number(0)+Undefined())
370
371     print("printing undefined", Undefined())
372     print("printing 6.2", Number(6.2))
373     print("2/0", Number(2)/Number(0))
374     try:
375         print(Number(4)+Number(3))
376         print("True")
377     except TypeError:
378         print("False")
379     print("4+0.4+undef", Number(4)+Number(0.4)+Undefined())
380     print("4^4", Number(4)**Number(4))
381     print("4^Undefined", Number(4)**Undefined())
382     print("Undefined^4", Undefined()**Number(4))
383     pass
384     print((NPArray(np.linspace(-10, 10, 100))+Undefined()))
385     print(Number(4)/NPArray(np.linspace(-10, 10, 100)))
386
387
388 if __name__ == "__main__":
389     value_test()
390     quit()
391
```

```python
from typing import Type, Callable, TypeVar
from abc import abstractmethod

from monad import ListMonad

__all__ = ["dict_beautify", "JSONable", "NotCompatibleException", "Wrapper", "Stack", "test", "unwrap_dict"]


class Wrapper:
    """
    unified interface for all classes that wrap a value
    specifies class has get_wrapped method that returns the wrapped value
    """
    def __init__(self, val):
        """
        initializes the _val attribute for the instance
        :param val: value being wrapped
        """
        self._val = val

    def get_wrapped(self):
        """
        returns the wrapped value
        :return: wrapped value of the instance
        """
        return self._val


class JSONable:
    """
    abstract interface that specifies all instances of subclasses will implement get_json method
    that returns a dictionary representative of itself
    """
    @abstractmethod
    def get_json(self) -> dict:
        """
        gets the instance in the form of a dictionary
        :return:
        """
        raise NotImplementedError("Not Implemented by "+type(self).__name__)


class NotCompatibleException(Exception):
    """
    raised when a match attempt has failed
    used during tokenizing and parsing
    """
    def __eq__(self, other):
        """
        used to determine if an exception is equivalent to another exception
        :param other:
        :return:
        """
        if not isinstance(other, NotCompatibleException):
            return False
        return str(self) == str(other)

```

```
 58
 59 # noinspection PyShadowingNames
 60 # noinspection PyPep8Naming
 61 # noinspection PyPep8:E741
 62 def dict_beautify(inp: list | tuple | set | dict) -> str:
 63     """
 64     prints dictionaries in readable manner
 65     :param inp:
 66     :return: inp "JSON" in proper JSON Format
 67     """
 68     # reverses the monadic list recursively
 69     reverse = (lambda inp:
 70                ListMonad()
 71                if len(inp) == 0 else
 72                reverse(inp.tail()) ** ListMonad(inp.head())
 73                )
 74
 75     # method that adds an indent to the front of a string
 76     add_indent = lambda a: "  " + a
 77
 78     def deal_with_commas(commaed_lines):
 79         """
 80         removes the trailing comma in the last line in lines
 81         :param commaed_lines: lines with terminal comma
 82         :return: lines without terminal comma
 83         """
 84         # if there are no items in list
 85         # code smell
 86         if len(commaed_lines) == 0:
 87             return ListMonad("")
 88         commaed_lines = reverse(commaed_lines)
 89         commaed_lines = reverse(
 90             ListMonad(commaed_lines.head()[:-1]) ** commaed_lines.tail()
 91         )
 92         return commaed_lines
 93
 94     lines = ListMonad()
 95     if not isinstance(inp, (dict, list, tuple, set)):
 96         raise TypeError("Input for dictBeautify should be a dictionary, list,
    tuple, or set, not "+type(inp).__name__)
 97     if isinstance(inp, (list, tuple, set)):
 98         for value in inp:
 99             # obtains beautified JSON format from dicts, list, sets, and
    tuples
100             if type(value) in (dict, list, tuple, set):
101                 lines **= ListMonad(*(dict_beautify(value) + ",").split("\n"))
102             else:
103                 lines **= ListMonad("\"" + str(value) + "\",")
104         lines = deal_with_commas(lines)
105         brackets = {list: "[]", tuple: "()", set: "{}"}
106         lines = (
107                 # adds opening bracket
108                 ListMonad(brackets[type(inp)][0]) **
109                 # indents contents
110                 lines.map(add_indent) **
111                 # adds closing bracket
112                 ListMonad(brackets[type(inp)][1])
113         )
114         return "\n".join(lines.to_list())
```

```
115
116     for key, value in inp.items():
117         key = f"\"{key}\""
118         # obtain beautified JSON if value is a dictionary, list, set, or tuple
119         if type(value) in (dict, list, tuple, set):
120             # define a function to add key and colon to the first line
121             addToStart = (lambda item, sequence:
122                           ListMonad(item + sequence.head()) ** sequence.tail()
123                           )
124             # obtains beautified JSON format from dicts, list, sets, and
    tuples and transforms into ListMonad
125             lines **= addToStart(key + ": ", ListMonad(*(dict_beautify(value
    ) + ",").split("\n")))
126         else:
127             lines **= ListMonad(key + ": \"" + str(value) + "\",")
128     # removes trailing comma
129     lines: ListMonad = deal_with_commas(lines)
130     # adds {} brackets to front and end
131     lines = (
132             ListMonad("{") **
133             # and indents all contents
134             lines.map(add_indent) **
135             ListMonad("}")
136     )
137     # transforms ListMonad into list, then concatenates each item with a
    newline
138     return "\n".join(lines.to_list())
139
140
141 class Stack:
142     """
143     a stack, FILO data type
144     """
145     def __init__(self):
146         self._stack = []
147
148     def is_empty(self) -> bool:
149         """
150         checks if stack is empty
151         :return:
152         """
153         return len(self._stack) == 0
154
155     def push(self, item):
156         """
157         pushes item onto stack
158         :param item:
159         :return:
160         """
161         self._stack.append(item)
162
163     def pop(self):
164         """
165         pops and return top item from stack
166         :return: top of stack
167         """
168         if self.is_empty():
169             raise IndexError("cannot pop from empty Stack")
170         return self._stack.pop()
```

```
171
172     def peek(self):
173         """
174         returns top of stack
175         :return: top of stack
176         """
177         if self.is_empty():
178             raise IndexError("cannot peek from empty stack")
179         return self._stack[-1]
180
181     def __len__(self) -> int:
182         """
183         returns number of items in stack
184         :return: number of items in the stack
185         """
186         return len(self._stack)
187
188
189 def test(name):
190     """
191     test wrapper
192     declares tests py printing name of test (if given) at start and end of
    test
193     :param name: name of the test
194     :return:
195     """
196     def actual_wrapper(func):
197         """
198         function that returns a function that calls the test function passed
    in and prints declaration before and after the test is called
199         :param func: test subroutine
200         :return: function that prints a declaration before and after calling
    the test function
201         """
202         def _(*args, **kwargs):
203             """
204             function that prints the declaration before and after the test
    function is called
205             :param args:
206             :param kwargs:
207             :return: what the test function returns
208             """
209             t = name  # stops syntax error?
210             if not isinstance(t, Callable):
211                 t += " "
212             else:
213                 t = ""
214             print(f"\nstart of {t}tests")
215             try:
216                 res = func(*args, **kwargs)
217             except Exception as e:
218                 print(f"{t}test failed, with "+type(e).__name__+" "+str(e))
219                 res = None
220             print(f"end of {t}tests\n")
221             return res
222         return _
223
224     if not isinstance(name, Callable):
225         return actual_wrapper
```

```
226     return actual_wrapper(name)
227 
228 
229 # declaring generics
230 K = TypeVar("K")
231 G = TypeVar("G")
232 
233 
234 def unwrap_dict(dict_obj: dict[K, G], target_key: K):
235     """
236     manipulates a dictionary object in reference
237     unwraps dictionary into target key's dictionary
238     :param dict_obj:
239     :param target_key:
240     :return:
241     """
242     if type(dict_obj) != dict:
243         raise TypeError("dict_obj must be a dictionary object, not "+type(
    dict_obj).__name__)
244     if target_key not in dict_obj:
245         raise KeyError(target_key + " not found in " + str(dict_obj))
246     # save the target dictionary reference/pointer into a temporary variable
247     saved_reference = dict_obj[target_key]
248     if not isinstance(saved_reference, dict):
249         raise TypeError("contents in key value must also by a dictionary
    object, not "+type(saved_reference).__name__)
250     dict_obj.clear()  # .clear() also clears up *some* memory
251     # redefine saved dictionary in the old dictionary reference
252     # copy over key value pairs in the saved dictionary into the cleared
    dictionary
253     for key in saved_reference:
254         dict_obj[key] = saved_reference[key]
255 
256 
257 @test("unwrap_dict")
258 def unwrap_dict_tests():
259     test1 = {"Hey": "what", "inside": {"Hey": "what"}}
260     print("before:", test1, sep="\n")
261     unwrap_dict(test1, "inside")
262     print("after:", test1, test1 == {"Hey": "what"})
263 
264 
265 if __name__ == "__main__":
266     unwrap_dict_tests()
267 
```

```python
#!/user/bin/env python

"""
Takes in the Abstract Syntax Tree as a JSON/dictionary format
and transforms into actions

semantics enforcer
ie enforces the coherence of the contents in the input(does it run)
example: "More people have been to Berlin than I have." looks correct, but isn't
example1: "factorial(e)" is not valid, although it fits the grammar
"""
import math
import warnings
import time
from abc import abstractmethod
from typing import Callable
import numpy as np


from lexer import *
from parser import *
from utilities import *
from mathobj import *

__all__ = ["UserInterfaceInterface", "CalculationUnit", "Visitor", "SubTree", "Nothing", "Assignment", "Expression"]


class Visitor:
    """
    base class for all visitors to specify all instances has visited method
    to comply with Visitor pattern
    """
    @abstractmethod
    def visited(self, visitee, *args):
        """
        method to customize response to certain objects
        :param visitee: the object visited
        :param args:
        :return:
        """
        raise NotImplementedError("visited method not yet implemented by subclass of Visitor, "+type(self).__name__)


class TwoStepVisitor(Visitor):
    """a subclass of visitor whose visit method occurs in two steps using generators"""
    @abstractmethod
    def visited(self, visitee):
        """
        :param visitee:
        args: arguments yielded later on
        :return:
        """
        super(TwoStepVisitor, self).visited(visitee)


```

```
 56 class Visitee:
 57     """
 58     an object that is visited
 59     """
 60     @abstractmethod
 61     def visit(self, visitor: Visitor):
 62         raise NotImplementedError("visit method not yet implemented by
    subclass of Visitee, "+type(self).__name__)
 63 
 64 
 65 class SubTree(Visitee):
 66     """
 67     a visitable tree that exposes its branches using the visitor pattern
 68     """
 69     def __init__(self, *branches: 'SubTree'):
 70         if any(map(lambda a: not isinstance(a, SubTree), branches)):
 71             raise TypeError("SubTree contents must be of SubTree type")
 72         self._branches = branches
 73 
 74     def visit(self, visitor: Visitor):
 75         if not isinstance(visitor, TwoStepVisitor):
 76             return visitor.visited(self, *map(lambda a: a.visit(visitor), self
    ._branches))
 77         else:
 78             # first visit current node
 79             v = visitor.visited(self)
 80             next(v)  # start generator
 81             # then visit branches
 82             return v.send(list(map(lambda a: a.visit(visitor), self._branches
    )))
 83 
 84     def __repr__(self):
 85         return type(self).__name__+"\n\t"+"\n\t".join([*sum(map(lambda a: repr
    (a).split("\n"), self._branches), [])])
 86 
 87 
 88 class Action(SubTree):
 89     """
 90     the "root" of the tree object
 91     """
 92     def __new__(cls, json: dict):
 93         if json["Type"] == "EmptyLine":
 94             return object.__new__(Nothing)
 95 
 96         unwrap_dict(json, "Action")
 97         typ = json["Type"]
 98         if typ == "Statement":
 99             return object.__new__(Statement)
100         elif typ == "Assignment":
101             return object.__new__(Assignment)
102         elif typ == "Expression":
103             return object.__new__(Expression)
104         else:
105             raise TypeError("json must be type Line, not "+typ)
106 
107 
108 class Nothing(Action):
109     def __new__(cls, json: dict):
110         return super(Action, cls).__new__(cls)
```

```
111
112     # not really needed
113     def __init__(self, json: dict):
114         super(Nothing, self).__init__()
115
116
117 # TODO: remove test
118 @test("Nothing")
119 def Nothing_Action_test():
120     print(o := Action({"Type": "EmptyLine", "Action": "None"}))
121     print(o)
122     print(type(o))
123
124
125 if __name__ == "__main__":
126     Nothing_Action_test()
127
128
129 class Statement(Action):
130     def __new__(cls, json: dict):
131         return super(Action, cls).__new__(cls)
132
133     def __init__(self, json: dict):
134         # set the name of the statement
135         self._name = json["Name"]["Name"]
136         operands: list[SubTree] = []
137         for operand in json["Operands"]["Operands"]:
138             operands.append(Operand(operand))
139         super(Statement, self).__init__(*operands)
140
141     def get_name(self):
142         return self._name
143
144     def __repr__(self):
145         head, tail = super(Statement, self).__repr__().split("\n", 1)
146         head += ": "+self.get_name()
147         return "\n".join([head, tail])
148
149
150 # TODO: remove tests
151 @test("Statement")
152 def Statement_tests():
153     print(Action({'Type': 'Line', 'Action': {'Type': 'Statement', 'Name': {'
    Type': 'NameSpace', 'Name': 'Settings'}, 'Operands': {'Type': 'Operands', '
    Operands': []}}}))
154     print(Action(parse(tokenize("Setting();"), ("Setting",))[0].get_json()["
    Lines"][0]))
155
156     print(Action(parse(tokenize("Settings();"), ("Settings",))[0].get_json()["
    Lines"][0]))
157
158
159 if __name__ == "__main__":
160     Statement_tests()
161
162
163 class BinaryOperations(SubTree):
164     """
165     base class for all binary operations
```

```
166     """
167     def __init__(self, left: SubTree, right: SubTree):
168         if not isinstance(left, SubTree) or not isinstance(right, SubTree):
169             raise TypeError("Operands are not of type SubTree")
170         super(BinaryOperations, self).__init__(left, right)
171 
172 
173 # all binary operations subclasses inherit the same constructor
174 # binary operations are all created in Expression initiation, so no __new__
    needed
175 class Plus(BinaryOperations):
176     pass
177 
178 
179 class Minus(BinaryOperations):
180     pass
181 
182 
183 class Multiply(BinaryOperations):
184     pass
185 
186 
187 class Divide(BinaryOperations):
188     pass
189 
190 
191 class Exponentiation(BinaryOperations):
192     pass
193 
194 
195 class Comparisons(BinaryOperations):
196     """
197     Base class for all comparisons
198     """
199 
200     def __new__(cls, left: SubTree, right: SubTree, json: dict, *args, **
    kwargs):
201         try:
202             typ = json["ComparisonType"]
203         except KeyError:
204             raise TypeError("ComparisonOperator Expected, not "+json["Type"])
205         if typ == ">":
206             return object.__new__(GreaterThan)
207         elif typ == "<":
208             return object.__new__(LessThan)
209         elif typ == ">=":
210             return object.__new__(GreaterThanEquals)
211         elif typ == "<=":
212             return object.__new__(LessThanEquals)
213         elif typ == "==":
214             return object.__new__(Equals)
215         else:
216             raise TypeError("ComparisonOperator Expected, not "+json["Type"])
217 
218     def __init__(self, left: SubTree, right: SubTree, json: dict):
219         super(Comparisons, self).__init__(left, right)
220 
221 
222 # no need to redeclare new since overridden new does the same if input is
```

```
222 correct
223 # when incorrect json given to subClass of Comparisons, initialisation will
    fail, leading to Attribute Error
224 # ie GreaterThan({"ComparisonType": "<"})  fail
225 # debugger used
226 # cause of bug still not yet found/understood
227
228 class GreaterThan(Comparisons):
229     def __new__(cls, *args, **kwargs):
230         return object.__new__(GreaterThan)
231
232
233 class LessThan(Comparisons):
234     def __new__(cls, *args, **kwargs):
235         return object.__new__(LessThan)
236
237
238 class GreaterThanEquals(Comparisons):
239     def __new__(cls, *args, **kwargs):
240         return object.__new__(GreaterThanEquals)
241
242
243 class LessThanEquals(Comparisons):
244     def __new__(cls, *args, **kwargs):
245         return object.__new__(LessThanEquals)
246
247
248 class Equals(Comparisons):
249     def __new__(cls, *args, **kwargs):
250         return object.__new__(Equals)
251
252
253 class Operand(SubTree):
254     def __new__(cls, json):
255         typ = json["Operand"]["Type"]
256         unwrap_dict(json, "Operand")
257         if typ == "Expression":
258             return object.__new__(Expression)
259         elif typ == "Inequality":
260             return object.__new__(Inequality)
261         elif typ == "Conditional":
262             return object.__new__(Conditional)
263         else:
264             raise TypeError("json must be type Operand, ")
265
266
267 class Inequality(Operand):
268     def __new__(cls, json: dict):
269         return object.__new__(cls)
270
271     def __init__(self, json: dict):
272         terms = list(map(Term, json["Terms"]))
273
274         node = Comparisons(terms[0], terms[1], json["Comparisons"][0])
275         if len(json["Comparisons"]) == 2:
276             node = Comparisons(node, terms[2], json["Comparisons"][1])
277         super(Inequality, self).__init__(node)
278
279
```

```
280 class Conditional(Operand):
281     def __new__(cls, json: dict):
282         return object.__new__(Conditional)
283 
284     def __init__(self, json: dict):
285         super(Conditional, self).__init__(Inequality(json["Conditions"][0]),
    Inequality(json["Conditions"][1]))
286 
287 
288 # TODO: remove tests
289 @test("Inequality and Conditional")
290 def incon_tests():
291     json = parse(tokenize("f(6>5);"))[0].get_json()["Lines"][0]
292     # print(dictBeautify(json))
293     print(Action(json))
294     json = parse(tokenize("f(6>5|3<=2<1);"))[0].get_json()["Lines"][0]
295     # print(dictBeautify(json))
296     print(Action(json))
297 
298 
299 if __name__ == '__main__':
300     incon_tests()
301 
302 
303 class Term(SubTree):
304     def __new__(cls, json: dict, *args, **kwargs):
305         """
306         creates the correct object type when using the Term constructor
307         :param json: dict being passed in
308         :param args: other miscellaneous arguments
309         :param kwargs: other keyword arguments
310         """
311         typ = "Undefined"
312         if "Type" not in json or (typ := json["Type"]) not in ("Term", "
    NegatedTerm"):
313             raise TypeError("json must be of type term, not "+typ)
314         typ = json["Content"]["Type"] if json["Type"] not in ("NegatedTerm",)
    else json["Type"]
315 
316         # unite different implementations of Terms and BracketedTerm in
    parsing
317         if "Content" in json:
318             unwrap_dict(json, "Content")
319 
320         if typ == "Number":
321             return object.__new__(Literal)
322         elif typ == "NameSpace":
323             return object.__new__(Variable)
324         elif typ == "Function":
325             return object.__new__(Function)
326         elif typ == "NegatedTerm":
327             return object.__new__(NegatedTerm)
328         elif typ == "BracketedTerm":
329             return object.__new__(BracketedTerm)
330         else:
331             raise TypeError("Unspecified Term Type")
332 
333 
334 class Literal(Term):
```

```
335     def __new__(cls, json: dict, *args, **kwargs):
336         # disallow Literal({"Function":...})
337         # basically object.__new__(Literal)
338         return super(Term, cls).__new__(cls)
339 
340     def __init__(self, json: dict | int | float | Value):
341         """
342         a container for values class
343         :param json:
344         """
345         if not (isinstance(json, dict) and "Value" in json or isinstance(json
    , (int, float, Value))):
346             raise TypeError("json must be of type dict with a \"Value key\",
    or an int or float or Value")
347         if isinstance(json, dict):
348             self._value = Number(json["Value"])
349         elif isinstance(json, (int, float)):
350             self._value = Number(json)
351         else:
352             self._value = json
353         super(Literal, self).__init__()
354 
355     def __call__(self, *args, **kwargs):
356         return self._value(*args)
357 
358     def get_value(self):
359         return self._value
360 
361     def __repr__(self):
362         return super(Literal, self).__repr__()[:-2]+": "+str(self._value)
363 
364 
365 class Variable(Term):
366     def __new__(cls, json: dict, *args, **kwargs):
367         return super(Term, cls).__new__(cls)
368 
369     def __init__(self, json: dict):
370         self._name = json["Name"]
371         super(Variable, self).__init__()
372 
373     def get_name(self) -> str:
374         return self._name
375 
376     def __repr__(self):
377         return super(Variable, self).__repr__()[:-2]+": "+self.get_name()
378 
379 
380 class Function(Term):
381     def __new__(cls, json: dict, *args, **kwargs):
382         return super(Term, cls).__new__(cls)
383 
384     def __init__(self, json: dict):
385         self._name = json["Name"]["Name"]
386         super(Function, self).__init__(*map(Operand, json["Operands"]["
    Operands"]))
387 
388     def get_name(self):
389         return self._name
390 
```

```
391     def __repr__(self):
392         head, tail = super(Function, self).__repr__().split("\n", 1)
393         head += ": "+self.get_name()
394         return "\n".join([head, tail])
395 
396 
397 class Expression(Action, Operand, Term):
398     # delegation pattern
399     def __new__(cls, json: dict):
400         return object.__new__(cls)
401 
402     def __init__(self, json: dict):
403         # dijkstra's shunting yard algorithm
404         stack = Stack()
405         operator_stack = Stack()
406         operator_precedence_key = {
407             "+": 0, "-": 0,
408             "*": 1, "/": 1,
409             "^": 2
410         }
411         for index, item in enumerate(json["TermsAndOperators"]):
412             # every other item in list is a Term
413             if index % 2 == 0:
414                 term = Term(item)
415                 stack.push(term)
416                 continue
417             current_operator = item["OperatorType"]
418             # empty operator stack
419             # if precedence of the top of the operator stack >= precedence of
    current operator
420             while not operator_stack.is_empty() and operator_precedence_key[
    operator_stack.peek()] >= operator_precedence_key[current_operator]:
421                 top = operator_stack.pop()
422                 right = stack.pop()
423                 left = stack.pop()
424                 stack.push({"+": Plus, "-": Minus, "*": Multiply, "/": Divide
    , "^": Exponentiation}[top](left, right))
425             # push current operator onto operator_stack
426             operator_stack.push(current_operator)
427             continue  # redundant
428 
429         # empty operator_stack again
430         while not operator_stack.is_empty():
431             top = operator_stack.pop()
432             right = stack.pop()
433             left = stack.pop()
434             stack.push({"+": Plus, "-": Minus, "*": Multiply, "/": Divide, "^"
    : Exponentiation}[top](left, right))
435 
436         # assumed all expressions are valid,
437         # so there should be one item on the stack remaining
438         super(Expression, self).__init__(stack.pop())
439 
440 
441 class NegatedTerm(Term):
442     def __new__(cls, json: dict, *args, **kwargs):
443         return object.__new__(NegatedTerm)
444 
445     def __init__(self, json: dict):
```

```
446         super(NegatedTerm, self).__init__(Term({"Type": "Term", "Content":
    json}))
447 
448 
449 class BracketedTerm(Term):
450     def __new__(cls, json: dict, *args, **kwargs):
451         return object.__new__(BracketedTerm)
452 
453     def __init__(self, json: dict):
454         super(BracketedTerm, self).__init__(*Expression(json["expression"]).
    _branches)
455 
456 
457 @test("Expression")
458 def Expression_tests():
459     json = parse(tokenize("e^-(2+r)+x*3-func(31-2,2);"))[0].get_json()["Lines"
    ][0]
460     temp = Action(json)
461     print(temp)
462     json = parse(tokenize("m*(x+b)+c;"))[0].get_json()["Lines"][0]
463     # print(dictBeautify(json))
464     temp = Action(json)
465     print(temp)
466 
467 
468 if __name__ == "__main__":
469     Expression_tests()
470 
471 
472 class Assignment(Action):
473     class AssignmentError(Exception):
474         pass
475 
476     def __new__(cls, json: dict):
477         return super(Action, cls).__new__(cls)
478 
479     def __init__(self, json: dict):
480         assigned_type = json["Assigned"]["Assignee"]["Type"]
481         self._equals_symbol = json["Equality"]["EqualityType"]
482 
483         if assigned_type == "NameSpace":
484             pass
485         elif assigned_type == "Function":
486             assignee = json["Assigned"]["Assignee"]
487 
488             # if the number of parameters of user defined function is greater
    than 1
489             # artificial limit number of parameters in user defined functions
490             if len(operands := assignee["Operands"]["Operands"]) > 1:
491                 raise self.AssignmentError("defined functions must only have
    one or less parameters")
492 
493             # if there is one operand and that operand is not just a variable
494             if len(operands) == 1 and (len(operands[0]["Operand"]["
    TermsAndOperators"]) != 1 or operands[0]["Operand"]["TermsAndOperators"][0]["
    Content"]["Type"] != "NameSpace"):
495                 raise self.AssignmentError("parameter(s) must be a variable")
496 
497             # collapse/compactify json
```

```
498             assignee["Operands"] = assignee["Operands"]["Operands"]
499             for index in range(len(assignee["Operands"])):
500                 assignee["Operands"][index] = assignee["Operands"][index]["
    Operand"]["TermsAndOperators"][0]["Content"]
501 
502         self._assignee = json["Assigned"]["Assignee"]
503         expression = Expression(json["Expression"])
504         super(Assignment, self).__init__(expression)
505 
506     def get_assignee(self):
507         return self._assignee  # leaky reference
508 
509     def __repr__(self):
510         head, tail = super(Assignment, self).__repr__().split("\n", 1)
511         head += "\nAssignee:" + dict_beautify(self.get_assignee())
512         return "\n".join([head, tail])
513 
514 
515 @test("Assignment")
516 def assignment_tests():
517     print(Action(parse(tokenize("x = 2;"))[0].get_json()["Lines"][0]))
518     print(Action(parse(tokenize("f() = x-4^r(6,9.8,-5);"))[0].get_json()["
    Lines"][0]))
519 
520 
521 if __name__ == "__main__":
522     assignment_tests()
523 
524 
525 class CalculationUnit(Visitor):
526     class CalculationUnitExceptions(Exception):
527         """
528         abstract base class for all exceptions raised from/by the Calculation
    Unit
529         """
530         pass
531 
532     class VariableNameNotFoundException(CalculationUnitExceptions, TypeError):
533         def __init__(self, prompt: str):
534             """
535             custom Exception raised by the Calculation unit when a variable is
     called but not declared
536             :param prompt: the name of the variable not found
537             """
538             super(TypeError, self).__init__(f"\"{prompt}\" has not been
    declared and therefore has no value and cannot be used")
539 
540     class ImmutableVariablesException(CalculationUnitExceptions):
541         def __init__(self, var_name: str):
542             """
543             exception raised when an attempt was mode to change immutable
    variable/functions through instructions
544             :param var_name: name of the immutable variable/function
545             """
546             super(CalculationUnit.CalculationUnitExceptions, self).__init__(f"
    \"{var_name}\" cannot be reassigned another value")
547 
548     class ParameterMisMatchException(CalculationUnitExceptions):
549         def __init__(self, expected: int | str, given: int | str):
```

```
550             """
551             exception raised when calling a function, there is a parameter
    mismatch(wrong number of parameters, type of parameters)
552             :param expected: what was expected
553             :param given: what it was in reality
554             """
555             super(CalculationUnit.CalculationUnitExceptions, self).__init__(f"
    function takes {expected} arguments, {given} given")
556 
557     def __init__(self):
558 
559         # noinspection PyMethodParameters
560         class Callable(SubTree, Wrapper):
561             def __init__(self, functor):
562                 Wrapper.__init__(self, functor)
563                 SubTree.__init__(self)
564 
565         self._functor_wrapper = Callable
566 
567         # all Functors (and its subclasses) should have a corresponding
    CalculationUnit,
568         # ie, the existence of a Functor object is dependent on the existence
    of a CalculationUnit
569         # so hence why it is declared in the constructor of calculation unit
570         # so when the reference to the associated calculation unit is required
571         # the typical self is renamed to self1 to avoid collision and
572         # access self from outside the scope of the method and into the scope
    of the calculation unit constructor
573         # noinspection PyMethodParameters
574         class Functor(Value):
575             """
576             abstract base class for all functors(functions)
577             """
578 
579             def __new__(cls, *args, **kwargs):
580                 # redefined/overridden to avoid inheriting implementation from
     Value
581                 return object.__new__(cls)
582 
583             def __init__(self1):
584                 """
585                 set the variables associated with calling the function f(a,b
    ) -> ["a", "b"]
586                 default: single variable "x"
587                 """
588                 self1._vars = self._params if self._params is not None else ["
    x"]
589 
590             def get_vars(self):
591                 """
592                 gets the variables required to call it
593                 :return:
594                 """
595                 return [var for var in self._vars]  # avoid escaping reference
596 
597             def node_version(self):
598                 """
599                 get the functor expression tree as the SubTree
600                 :return:
```

```
601                 """
602                 raise NotImplementedError
603 
604             def __call__(self, *args, **kwargs):
605                 raise NotImplementedError
606 
607             # operator overloading
608             def __add__(self, other):
609                 first_root = self.node_version()
610                 if isinstance(other, Number):
611                     second_root = Literal(other)
612                 elif isinstance(other, Functor):
613                     second_root = other.node_version()
614                 else:
615                     return
616                 return UserDefinedFunctor(Plus(first_root, second_root))
617 
618             def __sub__(self, other):
619                 first_root = self.node_version()
620                 if isinstance(other, Number):
621                     second_root = Literal(other)
622                 elif isinstance(other, Functor):
623                     second_root = other.node_version()
624                 else:
625                     return
626                 return UserDefinedFunctor(Minus(first_root, second_root))
627 
628             def __mul__(self, other):
629                 first_root = self.node_version()
630                 if isinstance(other, Number):
631                     second_root = Literal(other)
632                 elif isinstance(other, Functor):
633                     second_root = other.node_version()
634                 else:
635                     return
636                 return UserDefinedFunctor(Multiply(first_root, second_root))
637 
638             def __truediv__(self, other):
639                 first_root = self.node_version()
640                 if isinstance(other, Number):
641                     second_root = Literal(other)
642                 elif isinstance(other, Functor):
643                     second_root = other.node_version()
644                 else:
645                     return
646                 return UserDefinedFunctor(Divide(first_root, second_root))
647 
648             def __pow__(self, other, modulo=None):
649                 first_root = self.node_version()
650                 if isinstance(other, Number):
651                     second_root = Literal(other)
652                 elif isinstance(other, Functor):
653                     second_root = other.node_version()
654                 else:
655                     return
656                 return UserDefinedFunctor(Exponentiation(first_root,
    second_root))
657 
658             def __radd__(self, other):
```

```
659                 first_root = self.node_version()
660                 if isinstance(other, Number):
661                     second_root = Literal(other)
662                 elif isinstance(other, Functor):
663                     second_root = other.node_version()
664                 else:
665                     return
666                 return UserDefinedFunctor(Plus(second_root, first_root))
667 
668             def __rsub__(self, other):
669                 first_root = self.node_version()
670                 if isinstance(other, Number):
671                     second_root = Literal(other)
672                 elif isinstance(other, Functor):
673                     second_root = other.node_version()
674                 else:
675                     return
676                 return UserDefinedFunctor(Minus(second_root, first_root))
677 
678             def __rmul__(self, other):
679                 first_root = self.node_version()
680                 if isinstance(other, Number):
681                     second_root = Literal(other)
682                 elif isinstance(other, Functor):
683                     second_root = other.node_version()
684                 else:
685                     return
686                 return UserDefinedFunctor(Multiply(second_root, first_root))
687 
688             def __rtruediv__(self, other):
689                 first_root = self.node_version()
690                 if isinstance(other, Number):
691                     second_root = Literal(other)
692                 elif isinstance(other, Functor):
693                     second_root = other.node_version()
694                 else:
695                     return
696                 return UserDefinedFunctor(Divide(second_root, first_root))
697 
698             def __rpow__(self, other, modulo=None):
699                 first_root = self.node_version()
700                 if isinstance(other, Number):
701                     second_root = Literal(other)
702                 elif isinstance(other, Functor):
703                     second_root = other.node_version()
704                 else:
705                     return
706                 return UserDefinedFunctor(Exponentiation(second_root,
    first_root))
707 
708             def __neg__(self):
709                 first_root = self.node_version()
710                 SubTree.__init__(root := object.__new__(NegatedTerm),
    first_root)
711                 return UserDefinedFunctor(root)
712 
713             def __lt__(self, other):
714                 first_root = self.node_version()
715                 if isinstance(other, Number):
```

```
716                     second_root = Literal(other)
717                 elif isinstance(other, Functor):
718                     second_root = other.node_version()
719                 else:
720                     return
721                 return UserDefinedFunctor(LessThan(first_root, second_root))
722 
723             def __gt__(self, other):
724                 first_root = self.node_version()
725                 if isinstance(other, Number):
726                     second_root = Literal(other)
727                 elif isinstance(other, Functor):
728                     second_root = other.node_version()
729                 else:
730                     return
731                 return UserDefinedFunctor(Divide(first_root, second_root))
732 
733             def __repr__(self):
734                 return repr(self.node_version())
735 
736         self.functor = Functor
737 
738         # noinspection PyMethodParameters
739         class ValueDefinedFunctor(Functor):
740             def __init__(self, name: str, *operands):
741                 assert len(operands) > 0
742                 super(ValueDefinedFunctor, self).__init__()
743                 self._name = name
744                 self._operands = operands
745 
746             def get_name(self):
747                 return self._name
748 
749             def node_version(self):
750                 return Callable(self)
751 
752             def __call__(self1, *args, **kwargs):
753                 result = []
754                 for operand in self1._operands:
755                     if isinstance(operand, Functor):
756                         result.append(self.call_func(operand, *args))
757                     else:
758                         result.append(operand)
759                 # if it is function composition
760                 if any(map(lambda a: isinstance(a, Functor), result)):
761                     return self.value_defined_Functor(self1._name, *result)
762                 # check if name is a method of the value object
763                 obj = object()
764                 func = getattr(result[0], self1._name, obj)
765                 if func is obj:
766                     raise CalculationUnit.VariableNameNotFoundException(self1.
    _name)
767                 return self.call_func(func, *result[1:], name=self1._name)
768         self.value_defined_Functor = ValueDefinedFunctor
769 
770         # noinspection PyMethodParameters
771         class UserDefinedFunctor(Functor):
772             def __init__(self, tree: SubTree):
773                 super(UserDefinedFunctor, self).__init__()
```

```
774                 if not isinstance(tree, SubTree):
775                     raise TypeError("type SubTree expected, not "+type(tree).
    __name__)
776                 self._root = tree
777 
778             def node_version(self):
779                 return self._root
780 
781             def __call__(self1, *args, **kwargs):
782                 assert len(args) == len(self1.get_vars())
783                 self.add_function_frame(**{key: value for key, value in zip(
    self1.get_vars(), args)})
784                 # evaluate functor using calculation unit
785                 result = self1._root.visit(self)
786                 self.remove_function_frame()
787                 return result
788         self.user_defined_functor = UserDefinedFunctor
789 
790         self._control_states = {
791             TrigMode: TrigMode.RADIANS
792         }
793 
794         self._params = None  # variable names to look out for when creating a
    function
795         self._function_frame = Stack()  # call stack for identifying variables
796 
797         self._variables = {
798             "Ans": Number(0),
799             "x": UserDefinedFunctor(Variable({"Type": "Variable", "Name": "x"
    })),
800             "e": Number(np.e),
801             "pi": Number(np.pi)
802         }
803 
804         self._IMMUTABLE_VARIABLES = {"x", "e", "pi"}  # specifies what
    variables cannot be changed
805 
806         self._functions = {}
807         self._IMMUTABLE_FUNCTIONS = {"sqrt", "ln", "sin", "cos", "arcsin", "
    arcos"}  # specifies what functions cannot be user defined
808 
809     def add_function_frame(self, **kwargs):
810         """
811         the calculation unit is also used when resolving SubTrees in functors,
812         to prevent mix-up of variables, a call stack is created to ensure
    variables defined outside the functor is not used when resolving the functor
813         :param kwargs:
814         :return:
815         """
816         self._function_frame.push(kwargs)
817 
818     def remove_function_frame(self):
819         """
820         pops function frame stack
821         :return:
822         """
823         self._function_frame.pop()
824 
825     def begin_function_creation(self, var: list[str]):
```

```
826         """
827         sets flag variables to set calculation unit into function creation
    mode
828         :param var:
829         :return:
830         """
831         if self._params is not None:
832             warnings.warn("Calculation Unit is in the process of creating a
    function")
833             time.sleep(0)
834         if not all(map(lambda a: isinstance(a, str), var)):
835             raise TypeError("all variables to be passed in to var is to be
    type str")
836         self._params = var
837         self.add_function_frame(**{name: self.user_defined_functor(Variable({"
    Name": name})) for name in self._params})
838 
839     def end_function_creation(self):
840         """
841         sets flag variables to set calculation unit out of function creation
    mode
842         :return:
843         """
844         if self._params is None:
845             warnings.warn("Calculation Unit was not creating a function")
846             time.sleep(0)
847         self._params = None
848         self.remove_function_frame()
849 
850     def reset_visited_trackers(self):
851         """
852         resets calculation unit flag variables
853         :return:
854         """
855         if __name__ == "__main__" and (not self._function_frame.is_empty() or
    self._params is not None):
856             print("anomaly detected", self._params, self._function_frame)
857         self._params = None
858         self._function_frame = Stack()
859 
860     def set_control_states(self, new_states: dict):
861         """
862         sets the control states of the calculation unit
863         :param new_states:
864         :return:
865         """
866         if not isinstance(new_states, dict):
867             raise TypeError(f"dict type expected, got {type(new_states).
    __name__} instead")
868         for key, value in new_states.items():
869             if key not in self._control_states:
870                 continue
871             if value not in key:
872                 continue
873             self._control_states[key] = value
874 
875     def get_control_states(self) -> dict[Enum: Enum]:
876         """
877         getter for control states
```

```
878         :return:
879         """
880         return {key: value for key, value in self._control_states.items()}  # disallow escaping reference
881 
882     def call_func(self, func, *params, name=None):
883         # gives a function the proper contexts for mainly trigonometric functions
884         if isinstance(func, self.value_defined_Functor):
885             name = func.get_name()
886         elif name is None and not isinstance(func, self.user_defined_functor):
887             raise TypeError("name expected for custom functors/functions")
888 
889         trig_related_functions = ("sin", "cos", "tan", "arcsin", "arcos", "arctan")
890         if name in trig_related_functions:
891             return func(*params, mode=self._control_states[TrigMode])
892         else:
893             return func(*params)
894 
895     def visited(self, visitee, *args):
896         """
897         rules for each Node variant/implementation
898         :param visitee: node
899         :param args: results from branches using the same rules
900         :return: Value
901         """
902         # if visiting a literal
903         if isinstance(visitee, Literal):
904             return visitee.get_value()
905         elif isinstance(visitee, self._functor_wrapper):
906             if isinstance(visitee.get_wrapped(), self.value_defined_Functor):
907                 return self.call_func(visitee.get_wrapped(), *self._function_frame.peek().values())
908             # return visitee(*args)
909         # if visiting a variable
910         elif isinstance(visitee, Variable):
911             if not self._function_frame.is_empty() and visitee.get_name() in self._function_frame.peek():
912                 return self._function_frame.peek()[visitee.get_name()]
913             name = visitee.get_name()
914             if name not in self._variables:
915                 raise self.VariableNameNotFoundException(name)
916             return self._variables[name]
917         # if visiting an expression or bracketed Term
918         elif isinstance(visitee, (Expression, BracketedTerm, Inequality)):
919             return args[0]
920         elif isinstance(visitee, Plus):
921             return args[0] + args[1]
922         elif isinstance(visitee, Minus):
923             return args[0] - args[1]
924         elif isinstance(visitee, Multiply):
925             return args[0] * args[1]
926         elif isinstance(visitee, Divide):
927             return args[0] / args[1]
928         elif isinstance(visitee, Exponentiation):
929             return args[0] ** args[1]
930         elif isinstance(visitee, NegatedTerm):
931             return -args[0]
```

```
932         elif isinstance(visitee, Function):
933             name = visitee.get_name()
934 
935             composition = any(map(lambda a: isinstance(a, self.functor), args
    ))
936             if composition:
937                 if name not in self._functions:
938                     return self.value_defined_Functor(visitee.get_name(), *
    args)
939                 else:
940                     return self.call_func(self._functions[name], *args)
941             else:
942                 if len(args) == 0:
943                     raise self.VariableNameNotFoundException(visitee.get_name
    ())
944                 obj = object()  # used to identify if a method is present in
    the first argument
945                 func = getattr(args[0], name, obj)
946                 if func is not obj:  # if there is a function defined by the
    argument ie sin
947                     return self.call_func(func, *args[1:], name=name)
948                 del obj
949                 if name not in self._functions:
950                     raise self.VariableNameNotFoundException(visitee.get_name
    ())
951                 else:
952                     func = self._functions[name]
953                     try:
954                         return self.call_func(func, *args)
955                     except TypeError as e:
956                         if "positional argument" in str(e):  # if the
    incorrect number of arguments is given
957                             raise CalculationUnit.ParameterMisMatchException(f
    "not {len(args)}", len(args))
958                         else:
959                             raise e
960         elif isinstance(visitee, Conditional):
961             return args[0] | args[1]
962         elif isinstance(visitee, GreaterThan):
963             return args[0] > args[1]
964         elif isinstance(visitee, GreaterThanEquals):
965             return args[0] >= args[1]
966         elif isinstance(visitee, LessThan):
967             return args[0] < args[1]
968         elif isinstance(visitee, LessThanEquals):
969             return args[0] <= args[1]
970         elif isinstance(visitee, Equals):
971             return args[0] == args[1]
972         else:
973             raise NotImplementedError("Calculations for " + type(visitee).
    __name__ + " not yet implemented")
974 
975     def set_variable(self, name: str, value):
976         """
977         an exposed interface for Control unit to change values of variables
    internally
978         :param name: name of variable
979         :param value: the value to set to
980         :return:
```

```
 981         """
 982         if not isinstance(name, str):
 983             raise TypeError("str expected, not "+type(name).__name__)
 984         if name in self._IMMUTABLE_VARIABLES:
 985             raise self.ImmutableVariablesException(name)
 986         self._variables[name] = value
 987 
 988     def get_variables(self) -> dict[str: Value]:
 989         """
 990         an exposed interface for control unit to access variables declared
 991         :return:
 992         """
 993         return {key: value for key, value in self._variables.items()}
 994 
 995     def set_function(self, name: str, function):
 996         """
 997         an exposed interface for Control unit to change values of functions
     internally
 998         :param name: name of variable
 999         :param function: the function to set to
1000         :return:
1001         """
1002         if name in self._IMMUTABLE_FUNCTIONS:
1003             raise self.ImmutableVariablesException(name)
1004         self._functions[name] = function
1005         pass
1006 
1007     def get_functions(self) -> dict:
1008         """
1009         an exposed interface for Control unit to access functions declared
1010         :return:
1011         """
1012         return {key: value for key, value in self._functions.items()}
1013 
1014 
1015 class ControlUnit:
1016     """
1017     an object that is associated via composition to an interface
1018     processes input from the interface
1019     proxy for
1020     """
1021     def __init__(self, interface: 'UserInterfaceInterface'):
1022         if not isinstance(interface, UserInterfaceInterface):
1023             raise TypeError("UserInterfaceInterface expected, not "+type(
     interface).__name__)
1024         # Observer pattern/ aggregation
1025         self._interface: UserInterfaceInterface = interface
1026 
1027         # the statements/commands the interface supports
1028         self._statements: dict[str, Callable] = self._interface.
     get_custom_statements()
1029 
1030         # the working current working lines
1031         self._lines: list[Action] = []
1032 
1033         # delegation pattern/ composition
1034         self._VM = CalculationUnit()
1035 
1036     def _interpret(self, inp: str) -> None | dict:
```

```
1037         """
1038         protected method
1039         tokenizes and parses input into action
1040         :param inp: user input in the form of a string
1041         :return:
1042         """
1043         try:
1044             tokens: list[Token] = tokenize(inp)
1045 
1046             parse_result = parse(tokens, self._statements)
1047 
1048             return parse_result[0].get_json()  # discard remainder of tokens
1049 
1050         except ScanError:
1051             self._interface.alert_improper_input("Unable to tokenize")
1052             return
1053         except ParseError:
1054             self._interface.alert_improper_input("Unable to parse")
1055             return
1056 
1057     def set_lines(self, inp: str) -> None:
1058         """
1059         method that the interface calls to give the control unit user input
1060         :param inp: user input in the form of a string
1061         :return:
1062         """
1063         # if an alert is raised
1064         # result: tokenized and parsed input
1065         if (result := self._interpret(inp)) is None:
1066             self._lines = []
1067             return
1068         self._lines: list[Action] = []
1069 
1070         # could have added more defensive programming
1071         # just assuming format is correct
1072         for line in result["Lines"]:
1073             try:
1074                 self._lines.append(Action(line))
1075             except Assignment.AssignmentError:
1076                 self._interface.alert_improper_input("assignment failed")
1077 
1078 
1079     def get_lines_visited(self, visual_generator: Visitor) -> list:
1080         """
1081         returns the visualization of the current working lines dictated by
     the visitor
1082         :param visual_generator: visitor containing rules that dictate how
     the Actions are to be visually represented
1083         :return: visual representations in a list
1084         """
1085         if not isinstance(visual_generator, Visitor):
1086             raise TypeError("Visitor expected, not "+type(visual_generator).
     __name__)
1087 
1088         return [line.visit(visual_generator) for line in self._lines]
1089 
1090     class ActionVisitor(TwoStepVisitor):
1091         """
1092         visitor to generate action for get_Lines_Action method
```

```
1093        responsible for assignment, statement, and Nothing handling
1094        """
1095        def __init__(self, vm: CalculationUnit, statements: dict[str,
    Callable]):
1096            if not isinstance(vm, CalculationUnit):
1097                raise TypeError("CalculationUnit expected, not "+type(vm).
    __name__)
1098            self._calc = vm
1099            self._statements = statements
1100            self._entered = False
1101
1102        def visited(self, visitee):
1103            is_first_entry = False
1104            if not self._entered:
1105                is_first_entry = True
1106                self._entered = True
1107
1108            if not isinstance(visitee, (Nothing, Statement, Assignment)):
1109                # delegation pattern
1110                args = yield
1111                val = self._calc.visited(visitee, *args)
1112                if is_first_entry:
1113                    self._calc.set_variable("Ans", val)
1114                yield val
1115
1116            if isinstance(visitee, Nothing):
1117                yield  # dummy yield
1118                yield visitee
1119            elif isinstance(visitee, Statement):
1120                args = yield
1121                name = visitee.get_name()
1122                # not going to happen
1123                if name not in self._statements:
1124                    raise CalculationUnit.VariableNameNotFoundException(name)
1125                try:
1126                    statement_result = self._statements[name](*args)
1127                except TypeError as e:
1128                    if "positional argument" in str(e):
1129                        raise CalculationUnit.ParameterMisMatchException(f"
    not {len(args)}", len(args))
1130                    raise e
1131                if isinstance(statement_result, Value):
1132                    self._calc.set_variable("Ans", statement_result)
1133                yield statement_result
1134
1135            elif isinstance(visitee, Assignment):
1136                typ = visitee.get_assignee()
1137                if typ["Type"] == "NameSpace":
1138                    val = yield
1139                    self._calc.set_variable(typ["Name"], val[0])
1140                elif typ["Type"] == "Function":
1141                    names = [variable["Name"] for variable in typ["Operands"
    ]]
1142                    self._calc.begin_function_creation(names)
1143                    val = yield
1144                    self._calc.end_function_creation()
1145                    # if there is no variable in the expression defining the
    function
1146                    # make it a Functor
```

```
1147                     if not isinstance(val[0], self._calc.functor):
1148                         val[0] = self._calc.user_defined_functor(Literal(val[
     0]))
1149                     self._calc.set_function(typ["Name"]["Name"], val[0])
1150                 else:
1151                     # TODO: investigate
1152                     input("wierid spot")
1153                     yield
1154                 yield None
1155             else:
1156                 raise TypeError("logic for " + type(visitee).__name__ + " not
      yet implemented")
1157 
1158     def get_lines_action(self) -> list:
1159         """
1160         gets and returns the current working lines Actions
1161         :return:
1162         """
1163         result_list = []
1164         for lines in self._lines:
1165             try:
1166                 result = lines.visit(self.ActionVisitor(self._VM, self.
     _statements))
1167                 result_list.append(result)
1168             # relay all custom errors into alerts
1169             except CalculationUnit.CalculationUnitExceptions as e:
1170                 self._interface.alert_improper_input(str(e))
1171                 result_list.append(None)
1172             except NotImplementedError as e:
1173                 self._interface.alert_improper_input(str(e)+"unsupported
     operation occurred while processing")
1174                 result_list.append(None)
1175             finally:
1176                 self._VM.reset_visited_trackers()
1177         return result_list
1178 
1179     def set_settings(self, parameters):
1180         """
1181         an interface for a UserInterfaceInterface to change the internal VM's
      setting/state
1182         :param parameters:
1183         :return:
1184         """
1185         self._VM.set_control_states(parameters)
1186 
1187     def get_settings(self) -> dict[Enum: Enum]:
1188         """
1189         glass door for interface to access VM control states
1190         :return:
1191         """
1192         return self._VM.get_control_states()
1193 
1194     def get_variables(self) -> dict[str, Value]:
1195         """
1196         allows interface access to variables set
1197         :return: dictionary of variables
1198         """
1199         return self._VM.get_variables()
1200 
```

```
1201     def get_functions(self) -> dict:
1202         """
1203         allows interface access to function set
1204         :return: dictionary of functions
1205         """
1206         return self._VM.get_functions()
1207 
1208 
1209 class UserInterfaceInterface:
1210     """
1211     proxy for control unit
1212     """
1213 
1214     def __init__(self):
1215         """
1216         an interface that dictate what an interface must implement
1217         & separate user interface responsibility from controller
1218         """
1219         self.__controller = ControlUnit(self)
1220 
1221     def set_instruction(self, instruction: str):
1222         """
1223         sets instructions into the control unit
1224         :param instruction:
1225         :return:
1226         """
1227         if not isinstance(instruction, str):
1228             raise TypeError("str expected for instruction, not "+type(
     instruction).__name__)
1229         self.__controller.set_lines(instruction)
1230 
1231     def get_calculated_results(self) -> list:
1232         """
1233         returns results of instructions set within the control unit
1234         :return:
1235         """
1236         return self.__controller.get_lines_action()
1237 
1238     def send_visitor(self, visitor: Visitor):
1239         """
1240         allows interface to pass in a visitor to traverse the syntax tree
1241         :param visitor:
1242         :return:
1243         """
1244         if not isinstance(visitor, Visitor):
1245             raise TypeError("type Visitor expected, not "+type(visitor).
     __name__)
1246         return self.__controller.get_lines_visited(visitor)
1247 
1248     def set_settings(self, args: dict):
1249         """
1250         called to change/set settings of the virtual machine/calculation unit
1251         :return:
1252         """
1253         if not isinstance(args, dict):
1254             raise NotImplementedError("Settings not implemented for interface
     ")
1255         # do user input
1256         self.__controller.set_settings(args)
```

```
1257
1258    def get_settings(self) -> dict:
1259        """
1260        allows interface to grab/access current settings
1261        :return:
1262        """
1263        return self.__controller.get_settings()
1264
1265    def get_variables(self) -> dict:
1266        """
1267        gets all defined variables
1268        :return:
1269        """
1270        return self.__controller.get_variables()
1271
1272    def get_functions(self) -> dict:
1273        """
1274        gets all defined functions
1275        :return:
1276        """
1277        return self.__controller.get_functions()
1278
1279    def get_custom_statements(self) -> dict[str, Callable]:
1280        """
1281        gets callable statements from the interface(itself)
1282        :return: a dictionary of the string names of the methods and the
     methods themselves
1283        """
1284        dict_of_methods = {}
1285        for attr in dir(self):
1286            thing = self.__getattribute__(attr)
1287            if type(thing).__name__ == "method" and "_" not in attr:
1288                dict_of_methods[attr] = thing
1289        return dict_of_methods
1290
1291    @abstractmethod
1292    def alert_improper_input(self, improper_ness: str) -> None:
1293        """
1294        an alert to the user/interface why an unexpected action has occurred
1295        :param improper_ness: the reason why something unexpected has
     happened
1296        :return:
1297        """
1298        raise NotImplementedError
1299
1300
1301 # noinspection PyTypeChecker
1302 @test("interface")
1303 def final_test():
1304
1305    class DummyInterface(UserInterfaceInterface):
1306        def get_custom_statements(self) -> dict[str, Callable]:
1307            return {
1308                "settings": self.Settings,
1309                "setIntoDeg": self.setIntoDeg
1310            }
1311
1312        def test(self, test_id: str, inp: str, expected_output: list,
     new_values: None | dict[str, Value] = None, new_functions: None | dict[str,
```

```
Value] = None, new_states: None | dict=None):
            """
            tests inputs against expected outputs and variable sets
            :param test_id: the requirement being tested
            :param inp: tested input
            :param expected_output: the expected output
            :param new_values: the "new" values to be checked
            :return: None
            """
            assert isinstance(inp, str)
            assert isinstance(expected_output, list)
            assert new_values is None or isinstance(new_values, dict)
            assert new_functions is None or isinstance(new_functions, dict)
            assert new_states is None or isinstance(new_states, dict)
            passed = True
            self.set_instruction(inp)
            actual_results = self.get_calculated_results()
            if new_values is not None:
                actual_values = self.get_variables()
                for name in new_values:
                    passed &= repr(actual_values[name]) == repr(new_values[
    name])
            if new_functions is not None:
                actual_functions = self.get_functions()
                for name in new_functions:
                    passed &= repr(actual_functions[name]) == repr(
    new_functions[name])
                    if not passed:
                        print(repr(actual_functions[name]))
            if new_states is not None:
                actual_states = self.get_settings()
                for state in new_states:
                    passed &= actual_states[state] == new_states[state]

            passed &= len(actual_results) == len(expected_output)
            for actual, expected in zip(actual_results, expected_output):
                passed &= repr(actual) == repr(expected)
            print("{:<9}| {:<5}| {:<14}| {:<15}| {:<15}".format(test_id, str(
    passed), inp, str(expected_output), str(actual_results)))

        def Settings(self):
            print("you are in settings")
            pass

        def graph(self, value):
            print("graphing")
            print(value)
            return value

        def setIntoRad(self):
            self.set_settings({TrigMode: TrigMode.RADIANS})

        def setIntoDeg(self):
            self.set_settings({TrigMode: TrigMode.DEGREES})

        def showMode(self):
            print(self.get_settings())

        def alert_improper_input(self, improper_ness: str) -> None:
```

```
1368            print(improper_ness)
1369
1370
1371    control = DummyInterface()
1372    control.test("FR 2.1", "2+4;", [Number("6")])
1373    control.test("FR 2.2", "2-4;", [Number("-2")])
1374    control.test("FR 2.3", "2*3;", [Number("6")])
1375    control.test("FR 2.4", "6/2;", [Number("3")])
1376    control.test("FR 2.4.1", "1/0;", [Undefined()])
1377    control.test("FR 2.5", "2^3;", [Number("8")])
1378    control.test("FR 2.6", "-3;", [Number("-3")])
1379    control.test("FR 2.7", "(2+3)*4;", [Number("20")])
1380    control.test("FR 2.8.1", "a=3;", [None], new_values={"a": Number("3")})
1381    def dummy_init(self, value):
1382        self.value = value
1383    def dummy_repr(self):
1384        return self.value
1385    dummy = type("dummy", (object,), {"__init__": dummy_init, "__repr__":
    dummy_repr})
1386    control.test("FR 2.8.2", "f(x)=x+1;", [None], new_functions={"f": dummy("
    Plus\n  Variable: x\n   Literal: Number: 1.0")})
1387    control.set_instruction("g(x)=2*x;")
1388    control.get_calculated_results()
1389    control.test("FR 2.9", "f(g(4));", [Number("9")])
1390    control.test("FR 2.10", "4*3^2+5;", [Number("41")])
1391    control.test("FR 2.11", "sin(pi);", [Number("0")])
1392    control.test("FR 2.11", "tan(pi);", [Number("0")])
1393    control.test("FR 3.1", "setIntoDeg();", [None], new_states={TrigMode:
    TrigMode.DEGREES})
1394    control.test("FR 3.2", "sin(pi);", [Number(str(math.sin(math.pi**2/180
    ))[:5])])
1395    control.test("FR 5", "2^(3+1);2^3+1;", [])
1396    def temp_visited(self, visitee, *args):
1397        if isinstance(visitee, Literal):
1398            return str(visitee.get_value().get_wrapped())
1399        elif isinstance(visitee, Plus):
1400            return args[0] + "+" + args[1]
1401        elif isinstance(visitee, Exponentiation):
1402            return " "*len(args[0]) + args[1] + "\n" + args[0] + " "*len(args
    [1])
1403        else:
1404            return args[0]
1405    TempVisitor = type("TempVisitor", (Visitor,), {"visited": temp_visited})
1406    print(*control.send_visitor(TempVisitor()), sep="\n\n")
1407
1408
1409 if __name__ == '__main__':
1410    final_test()
1411
```

File - C:\Users\Andy\PycharmProjects\NEAs\code\user_interface.py

```
from mathobj import *
from controlunit import *
import numpy as np
from matplotlib import pyplot as plt
plt.rcParams["figure.figsize"] = [7.50, 3.50]
plt.rcParams["figure.autolayout"] = True
import tkinter


class Matrix(Value):
    def __new__(cls, *args, **kwargs):
        return object.__new__(Matrix)

    def __init__(self, h: int, l: int, contents, default=0):
        if not isinstance(h, int) or not isinstance(l, int):
            raise TypeError("type int expected for h and l")
        if h < 1 or l < 1:
            raise ValueError("dimensions of the matrix should be greater than 1x1")
        if len(contents) > h*l:
            raise ValueError("contents should be less than the size of the matrix")
        for _ in range(h*l-len(contents)):
            contents.append(default)
        self._dimension = (h, l)
        self._grid = [[contents[i*l+j] for j in range(l)] for i in range(h)]

    def get_dimension(self) -> tuple[int, int]:
        return self._dimension

    def __call__(self, *args, **kwargs):
        return self

    def get_matrix_of_minor(self, intersect: tuple[int, int]):
        if self._dimension[0] != self._dimension[1]:
            raise ValueError("cannot get matrix of minor for non-square matrices")
        n = self._dimension[0]
        array = []
        for i in range(n):
            for j in range(n):
                if i == intersect[0] or j == intersect[1]:
                    continue
                array.append(self._grid[i][j])
        return Matrix(n-1, n-1, array)

    def get_sub_matrix(self, top_left: tuple[int, int], bottom_right: tuple[int, int]):
        h = bottom_right[0] - top_left[0]+1
        l = bottom_right[1] - top_left[0]+1
        array = []
        for i in range(top_left[0], bottom_right[0]+1):
            for j in range(top_left[1], bottom_right[1]+1):
                array.append(self._grid[i][j])
        return Matrix(h, l, array)

    def transpose(self):
        return Matrix(*self._dimension[::-1], contents=[self._grid[j][i] for i in range(self._dimension[0]) for j in range(self._dimension[1])])
```

```
 55
 56     def Det(self):
 57         if self._dimension[0] != self._dimension[1]:
 58             raise CalculationUnit.CalculationUnitExceptions("cannot get
    determinant of non-square matrix")
 59         if self._dimension[0] == 2:
 60             return self._grid[0][0]*self._grid[1][1]-self._grid[0][1]*self.
    _grid[1][0]
 61         total = 0
 62         for i in range(self._dimension[0]):
 63             total += (1 - 2 * (i % 2)) * self._grid[0][i] * self.
    get_matrix_of_minor((0, i)).Det()
 64         return Number(total)
 65 
 66     def Inv(self):
 67         if self._dimension[0] != self._dimension[1]:
 68             raise CalculationUnit.CalculationUnitExceptions("cannot get
    determinant of non-square matrix")
 69         if self._dimension[0] == 2:
 70             return Matrix(2, 2, [self._grid[1][1], -self._grid[0][1], -self.
    _grid[1][0], self._grid[0][0]])/self.Det()
 71         array = []
 72         for i in range(self._dimension[0]):
 73             for j in range(self._dimension[1]):
 74                 array.append(
 75                     # swapped i and j to achieve transposition
 76                     (1-2*((i+j) % 2)) * self.get_matrix_of_minor((j, i)).Det()
 77                 )
 78         return Matrix(*self._dimension, contents=array)/Number(self.Det())
 79 
 80     def __add__(self, other):
 81         if isinstance(other, Matrix):
 82             if self._dimension != other._dimension:
 83                 raise CalculationUnit.CalculationUnitExceptions("cannot add
    matrices of different sizes, {}X{} and {}X{}".format(*self._dimension, *other.
    _dimension))
 84             return Matrix(*self._dimension, contents=[self._grid[i][j]+other.
    _grid[i][j] for i in range(self._dimension[0]) for j in range(self._dimension[
    1])])
 85         elif isinstance(other, Number):
 86             return Undefined()
 87         return NotImplemented
 88 
 89     def __radd__(self, other):
 90         if isinstance(other, Number):
 91             return Undefined()
 92         return NotImplemented
 93 
 94     def __sub__(self, other):
 95         if isinstance(other, Matrix):
 96             if self._dimension != other._dimension:
 97                 raise CalculationUnit.CalculationUnitExceptions("cannot add
    matrices of different sizes, {}X{} and {}X{}".format(*self._dimension, *other.
    _dimension))
 98             return Matrix(*self._dimension, contents=[self._grid[i][j] - other
    ._grid[i][j] for i in range(self._dimension[0]) for j in range(self._dimension
    [1])])
 99         elif isinstance(other, Number):
100             return Undefined()
```

```
101         return NotImplemented
102 
103     def __rsub__(self, other):
104         if isinstance(other, Number):
105             return Undefined()
106         return NotImplemented
107 
108     def __mul__(self, other):
109         if isinstance(other, Number):
110             other = other.get_wrapped()
111             return Matrix(*self._dimension, contents=[j*other for i in self.
    _grid for j in i])
112         elif isinstance(other, Matrix):
113             if self._dimension[1] != other._dimension[0]:
114                 raise CalculationUnit.CalculationUnitExceptions("cannot
    multiply {}X{} matrices with {}X{} matrices".format(*self._dimension, *other.
    _dimension))
115             array = []
116             for new_i in range(self._dimension[0]):
117                 new_row = []
118                 for new_j in range(other._dimension[1]):
119                     total = 0
120                     for t in range(self._dimension[1]):
121                         total += self._grid[new_i][t]*other._grid[t][new_j]
122                     new_row.append(total)
123                 array += new_row
124             return Matrix(self._dimension[0], other._dimension[1], array)
125         return NotImplemented
126 
127     def __rmul__(self, other):
128         if isinstance(other, Number):
129             return self.__mul__(other)
130         return NotImplemented
131 
132     def __truediv__(self, other):
133         if isinstance(other, Matrix):
134             raise CalculationUnit.CalculationUnitExceptions("matrix division
    not supported, use the Inv function instead")
135         if isinstance(other, Number):
136             return self.__mul__(Number(1)/other)
137         return NotImplemented
138 
139     def __rtruediv__(self, other):
140         if isinstance(other, Number):
141             return Undefined()
142         return NotImplemented
143 
144     def __repr__(self):
145         return "\n".join([str(line) for line in self._grid])
146 
147 
148 # testing interface
149 class CommandLineInterface(UserInterfaceInterface):
150     """
151 
152     """
153     class UserDoneEvent(Exception):
154         """
155         Custom exception raised to indicate user is done with the program
```

```
156         """
157         pass
158 
159     def get_custom_statements(self) -> dict:
160         return {
161             "quit": self.quit,
162             "exit": self.quit,
163             "makeMatrix": self.makeMatrix,
164             "plot": self.plot,
165             "settings": self.Settings,
166             "help": self.help,
167             "findQuadraticRoots": self.quadraticRoots,
168             "changeToRad": lambda: self.set_settings({TrigMode: TrigMode.
    RADIANS}),
169             "changeToDeg": lambda: self.set_settings({TrigMode: TrigMode.
    DEGREES})
170         }
171 
172     def makeDefault3X3Matrix(self):
173         return Matrix(3, 3, [1, 0, 0, 0, 1, 0, 0, 0, 1])
174 
175     def makeMatrix(self):
176         while (height := input("input the height of the matrix (0<)")).isdigit
    () and int(height) == 0:
177             pass
178         while (width := input("input the width of the matrix (0<)")).isdigit
    () and int(width) == 0:
179             pass
180         height, width = int(height), int(width)
181         default = 0
182         array = ["   " for _ in range(height*width)]
183         for index in range(height*width):
184             for row_index in range(height):
185                 print("[", ",".join(map(lambda a: str(a).center(3, " "), array
    [row_index*width: row_index*width+width])), "]")
186             while True:
187                 inp = input(">>>")
188                 if not inp:
189                     inp = default
190                     break
191                 try:
192                     inp = int(inp)
193                     break
194                 except ValueError:
195                     pass
196                 try:
197                     inp = float(inp)
198                     break
199                 except ValueError:
200                     pass
201             array[index] = inp
202 
203         print(array)
204         return Matrix(height, width, array)
205 
206     def __init__(self, visual=None, autostart=True):
207         super(CommandLineInterface, self).__init__()
208         self._prev_error = []
209         self._history = []
```

```
210         if visual is not None and not isinstance(visual, Visitor):
211             raise TypeError("visual_display must be a visitor type, not "+type
    (visual).__name__)
212         self.visual = visual
213         if autostart:
214             self.main_loop()
215 
216     def main_loop(self):
217         """
218         main program loop
219         runs until user specifies to stop
220         :return:
221         """
222         self.help()
223         # pyramid of doom
224         try:
225             try:
226                 while True:
227                     # user defined
228                     inp = self.get_input_from_user()
229                     # predefined
230                     self.set_instruction(inp)
231                     # user defined
232                     if self.visual is not None:
233                         # predefined
234                         self.send_visitor(self.visual)
235                     # user defined
236                     self.return_output(result := self.get_calculated_results
    ())
237             except KeyboardInterrupt:
238                 self.Quit()  # quit raises UserDoneEvent
239         except self.UserDoneEvent:
240             pass
241 
242     def plot(self, func):
243         separations = 10000
244         x = np.linspace(-10, 10, separations)
245         plt.plot(x, func(NPArray(x)).get_wrapped(), color="red")
246         plt.show()
247 
248     def Settings(self):
249         print("here are the settings you can change:")
250         changed = {}
251         for state, mode in self.get_settings().items():
252             decision = input(f"do you wish to change {state.__name__}, it is
    currently in {mode.name}:")
253             if not decision:
254                 continue
255             index = "-1"
256             for index, key in enumerate((possibilities := state.__members__).
    keys()):
257                 print(f"{index}) {key}")
258             while not (inp := input("which mode do you want to change to?0-"+
    str(index))).isdigit():
259                 pass
260             changed[state] = possibilities[list(possibilities.keys())[int(inp
    )]]
261         self.set_settings(changed)
262 
```

```
263     def help(self):
264         print("you can perform regular calculations such as:")
265         print("""
266 >>>3-1
267 >>>0.2*3
268 >>>-0.25/3.1
269 >>>2^5
270 >>>(2+3)*5""")
271         print("variables can be assigned using the format:")
272         print("<name>=<value>")
273         print("")
274         print("functions can be declared as:")
275         print("<name>(<parameter names>)=<expression with or without
    parameters>")
276         print("for example: f(x)=x+2")
277         print("you can call statements and functions using the following
    format <name>(<Operands>)")
278         print("eg f(2+4)")
279         print("these are all the statements you can use")
280         print(list(self.get_custom_statements().keys()), sep="\n")
281         print("enter \"help()\" to revisit this page")
282         input("press enter to continue")
283 
284     def Quit(self):
285         print("Thank you for using this calculator!")
286         raise self.UserDoneEvent()
287 
288     def quit(self):
289         self.Quit()
290 
291     def quadraticRoots(self, *coefficients: Value) -> None:
292         """
293         example statement that a user can call to calculate the roots of a
    quadratic equation
294         :param coefficients: the three possible coefficients in
295         :return:
296         """
297         if len(coefficients) != 3:
298             raise CalculationUnit.ParameterMisMatchException(3, len(
    coefficients))
299         previous_errors = [*self._prev_error]
300         self.set_instruction("a;b;c;")
301         priors = zip("abc", self.get_calculated_results())
302         a, b, c = coefficients
303         print(a, b, c)
304         self.set_instruction(f"a={float(a)};b={float(b)};c={float(c)};")
305         self.get_calculated_results()
306         self.set_instruction("(-b+sqrt(b^2-4*a*c))/(2*a);(-b-sqrt(b^2-4*a*c
    ))/(2*a);")
307         # m, d = -b/2/a, self.get_functions()["sqrt"](b**2-4*a*c)/2/a
308         self._history[-1] = (self._history[-1][0], self.get_calculated_results
    ())
309         for variable in priors:
310             if variable[1] is None:
311                 continue
312             self.set_instruction(variable[0] + "=" + str(float(variable[1
    ])) + ";")
313             self.get_calculated_results()
314         self._prev_error = previous_errors
```

```
315
316     def get_input_from_user(self) -> str:
317         # re-inject error statement
318         if len(self._history) > 0 and self._history[-1][1] == []:
319             self._history[-1] = (self._history[-1][0], self._prev_error)
320             self._prev_error = []
321         print("-"*50)
322         print(", ".join([topic.__name__ + ":" + str(state).split(".")[1] for
    topic, state in self.get_settings().items()]))
323         print("\n"*(7-2*min(len(self._history), 3)))
324         for i in range(min(len(self._history), 3)):
325             print(">>>"+self._history[i-min(len(self._history), 3)][0][:-1])
326             print(*(self._history[i-min(len(self._history), 3)][1]))
327         inp = input(">>>")+";"
328         self._history.append((inp, []))
329         return inp
330
331     def return_output(self, result: list | None):
332         """
333         returns and or displays the output to the user
334         :param result: the result of the line
335         :return: None
336         """
337         if result is None:  # if an error is thrown
338             raise NotImplementedError
339         if self._history[-1][1] not in ([], None):
340             return
341         lines = []
342         for line in result:
343             if line is None:
344                 continue
345             if isinstance(line, Assignment):
346                 continue
347             lines.append(line)
348         self._history[-1] = (self._history[-1][0], lines)
349
350     def alert_improper_input(self, improper_ness: str) -> None:
351         self._prev_error.append(improper_ness)
352
353
354 if __name__ == "__main__":
355
356     interface = CommandLineInterface(autostart=True)
357
358     # print(list(map(lambda a: UserInterfaceInterface.__getattribute__(a), )))
359
```