Project 3: Course Project Progress Report

Tianxiang Jin

Tjin@my.harrisburgu.edu

Harrisburg University of Science and Technology

November 22, 2020

CISC 603-51-A-2020/Fall – Theory of Computation

Heading

Abstract

Compiler is a program that can translate one language to another (typically translate higher level language to lower level language). Understanding of how compiler works is important to understanding high performance computing. A programmer who really wants her code performs well on computer should understand thoroughly how compiler interpret the code, how it parses the code, builds the syntax tree, and finally evaluates expressions. In this article, we will demonstrate and explore a simple compiler that can calculate mathematical expressions to summarize the process of building and using a compiler, which could help reader get an idea of context-free-grammars, lexical analysis, parsing, building syntax tree, and compiling.

Introduction

In this article, we will build a compiler that could evaluate a set of mathematical expressions efficiently. This could include basic math operations like addition, subtraction, multiplication and division. We will also try to point to some further development which readers could explore.

For the first step, we will build a simple version with the typical compiler components:

- **Tokens** – the fundamental element of every compiler
- **Context-free grammars** – or BNF form of grammar
- **Lexical analysis** – break the input text string into tokens that parser can recognize
- **Parsing** – we will construct an AST (abstract syntax tree) to represent the structure
- **Interpreter** – will calculate expression based on AST output from parser

Afterwards, we will use python as our language to build this simple compiler. We will build syntax first and implement in python

Next step of this project would be testing the compiler, check errors and improving efficiency. This step would improve our code reliability and semantic validity. Errors and flaws like duplicate declaration, zero division or invalid input would be identified and validated before parsing to the lower level language.

Finally, some possible further work can be done is provided and briefly discussed. Things like how to implement variable declaration, control and flow statement implementation. This part is not implemented in our python code due to project timeline limit. Once additional time and resources available, reader should explore it.

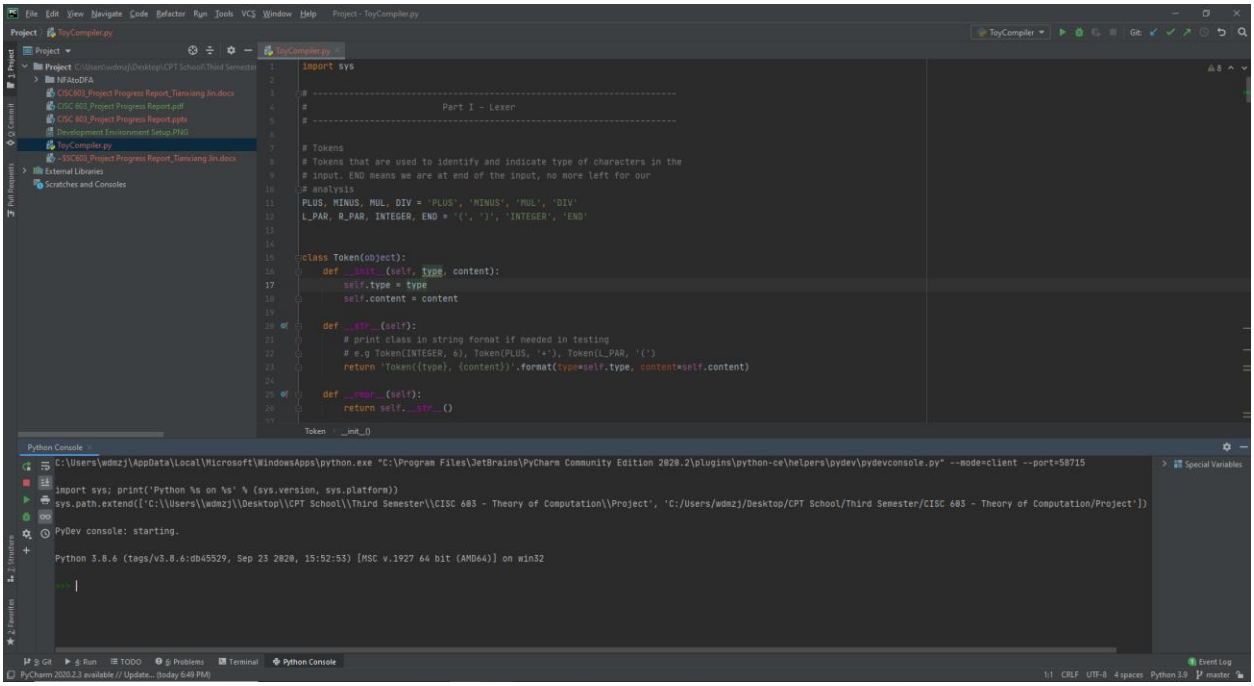
Compilation Environment Setup Instruction

Because the idea of this paper is to show how to use grammar and syntax analysis to build parser and interpreter of a compiler, all variables, functions and classes are included in one file for convenient purpose. The only file that is needed is – ToyCompiler.py

The only thing that are needed to execute this program is python 3 (version 3.5 or later). Since there are many python compiler and IDEs available on internet, we won't restrict to specific one. Even no IDEs available, as long as there is python compiler installed, you can still run the following command to bring up our calculator and play with it:

```
$python --version
$ToyCompiler.py
```

Here we use PyCharm IDE with python 3.8.6 for compiling and testing. A screenshot of the development environment setup is included below



Grammar and Design

Grammar

Firstly, in this part we will give a detailed illustration of the grammar we built for lexer analysis. We borrowed the idea of context-free-grammar, and summarize the rules into the following three part:

- **Expression** – Term ((PLUS|MINUS) Term)*
- **Term** – Factor ((MUL|DIV) Factor)*
- **Factor** – (PLUS|MINUS) Factor | Integer | LPARAN Expression RPARAN

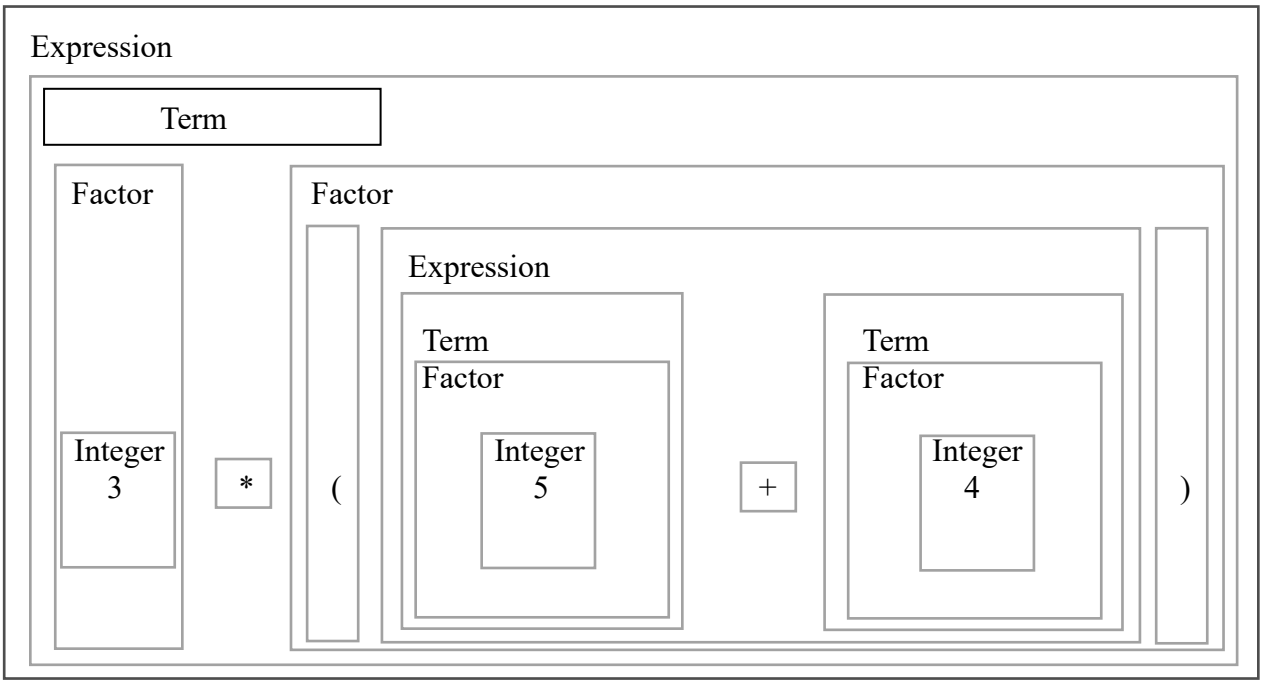
Factor is the most fundamental part in this grammar, which could be three type – either a unary operator "+" or "-" followed with an factor, or an integer, or an expression closed within pair of parenthesis.

After we form the basic structure of our language, we should deal with binary operators "+-*/", and this comes the second level structure – Term. Term is a series of factor followed by any number of "*/" together with another factor (second part could be none). We deal with "*/" first because it's higher order compared with "+-".

Third level (and the most outside level) is the expression. An expression is a series of term followed by any number of "+-" together with another term (the second part could be none).

Later we will see, with this syntax and grammar designed, our calculator compiler will be able to calculate any mathematical expression that is a valid combination of any integers (positive and negative), "+-*/" and parenthesis.

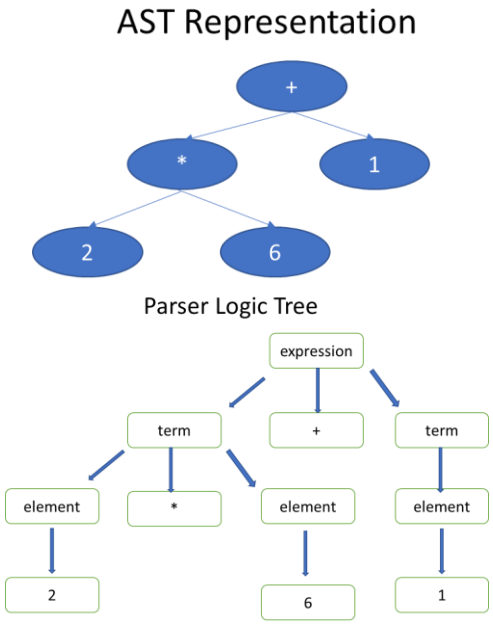
One example to illustrate above relationship is the grammar representation of 3*(5+4) as below, one can get a sense of the regular expression kind-of grammar we designed above.



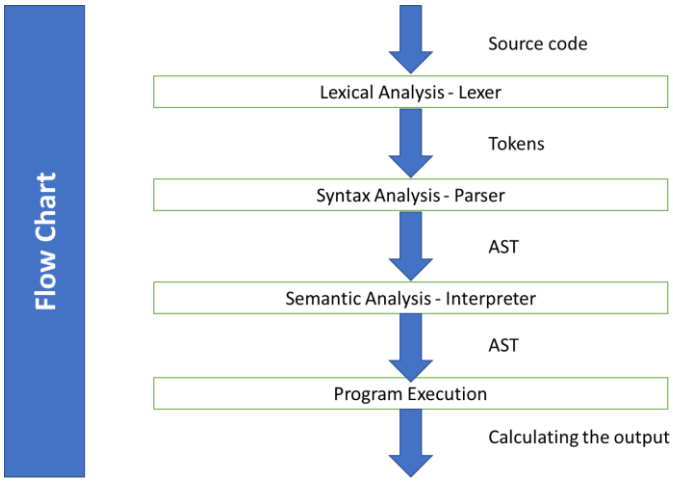
Abstract Syntax Tree and Parse Syntax Tree

In order to connect the work of parsing and interpreting, we need a more advance intermediate representation. In this project, we will use abstract-syntax-tree to represent the flow structure. Following are a brief example of AST versus Parse Tree using expression 2*6+1.

With chart below, we could see that AST is a more concise tree structure that represents the abstract structure of the mathematics expression. Potentially, one could also use AST to represent sophisticated language statements such as control statement and condition statement. Just one extra note, we choose AST here instead of Parse Logic Tree because they could both represent the same thing, but AST is a more concise and easy way to implement with Linked-list data structure.



Before we jump to next part, we should present clear here how the overall flow structure of our calculator, for better understanding what role parser and interpreter plays in the flow.



Algorithm Design Structure

There would be several parts below constructed with classes, to make our calculator work:

- Class Token
- Class Lexer
- Parent Class AST – Child Class (BinaryOperator, Numbers, UnaryOperator)
- Class Parser
- Class Interpreter
- Function Main

Here we walk through these parts one by one.

Token class is for storing token information, token will be the basic element for lexical analysis. For example, the tokens we will use include INTEGER, PLUS, MINUS, MUL, DIV, LPAR, RPAR and END, representing "+-*/" and "()" respectively.

Lexer class is for breaking input strings into tokens one by one. For example, when we input following into terminal: 3 + 22 * (-2 / (3 + 7) + - 184). The input would be breaking into Token(Integer, 3), Token(PLUS, "+"), Token(Integer, 22), Token(Mul, "*"), Token(LPAR, "(") … one by one, and the tokens will be feed to next part of the flow structure.

AST is a parent class inheritance from the most basic system object – object. AST will derived a few child classes for various implementations, like Class BinaryOperator(AST), Numbers(AST) or UnaryOperator(AST). AST (Abstract Syntax Tree) is the structure built with Tokens that Lexer gives us, and AST is the tree structure for later interpreter processing. Due to the nature of AST, interpreter could perform calculation in the correct order with pre-order traverse AST.

Parser Class is for constructing the AST, it calls Lexer class and gets Tokens one by one, and construct AST recursively.

Interpreter class iteratively "visit" nodes in AST and perform calculation based on the order of traversal. Final main class used for taking inputs from command line that user input and construct instances of above classes to do their work, and output the result back to command line.

More Details about Implementation

Here we will provide a few more details about implementation.

First one is about Lexer class. The purpose of the Lexer class is to break input string from user's command line into tokens. The method call lexer.get_next_token() is the method to implement this. Call on this method will start a while loop that takes input character one by one from input string, check which type of character it is, skip whitespace, build token accordingly, and return the tokens. Specifically, when deal with integers, the program use get_integer() method to catch integer with multiple digits. Lastly, it will perform lexical check to see if input is valid, if not, it will raise an error and terminate program.

For AST parent and child classes, we use the inheritance structure. Parent class AST didn't specify the implementation and type, while child classes UnaryOperator, BinaryOperator and Numbers, construct separately depends on token type. AST will be constructed recursively by parser class and this serve as the structure for processing calculation.

Parser class takes tokens from Lexer class, call its Parser.parse() function will build AST using the context-free-grammar. The function Parser.factor(), Parser.term() and Parser.expression() are respective implementation of expression, term and factor. Each function will build AST recursively. The Parser class will return a final node (root of the AST) to the next part.

Interpreter is the final component in our calculator compiler. The purpose of it is to perform calculation using AST, checking valid of the syntax, and return result. The way it doing its job is by calling visit() method to iteratively traversal the AST, calling specifically visit method (visit_BinaryOperator, visit_Numbers and visit_UnaryOperator). Each visit_xxx method has its own way of subtree traversal method just inline with how we construct AST.

Finally, we got the result and give it to main function, print it back to command line, and waiting for next instruction input from user.

Potential Further Development

Due to the time and resource constrict, we will leave some interesting parts for potential further development for reader.

Firstly, we could potentially add a part which could declare variables. Each language has to accept user variable declaration, this should not be difficult for our compiler as well, and potential implementation could use the Token idea we use above, together with a Symbol Table. Idea of tokens will be broaden to Keywords, and we could include things like Token("START"), Token("END"), Token("VAR")… these keywords will serve the need to identify code blocks by Parser Class. And Symbol Table will perform as the storage data structure for our variables.

keywords

Reserved Keywords including the following:

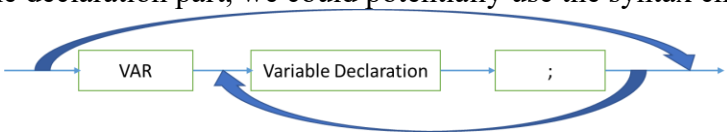
- START : Token('START')
- END.: Token('END.')
- VAR: Token('VAR')
- INT: Token('INT')
- FLOAT: Token('FLOAT')
- CALCULATOR: Token('CALCULATOR')
- ":=", ";" .....

Symbol Table

After a:= 3, b := 4

.....	.....
a	3
b	4

Secondly, we could broaden our grammar and syntax diagram. This would help use add various code blocks, flow statements to our compiler. For example, if we want a grammar representation for the declaration part, we could potentially use the syntax chart below.



Lastly, for reference purpose, we list a few more syntax diagram and grammar rules. And to implement these ideas, readers will still have to build AST. AST will feed to interpreter class and preorder traverse the AST will execute the statement one by one, and perform more complex tasks defined by the input.

Syntax Diagram	Grammar Rule
=>START=>compound statement=>END.	**Program:** START compound statement END.
=>START=>statement list=>END;	**Compound statement:** START statement list END;
=>statement=>(if ';' back to statement)=>	**Statement List:** statement | statement; statement list
	Statement: empty | assignment statement | control statement | compound statement
=>variable=> := =>expression=>	**Assignment statement:** variable := expression
	Variable: Token

Conclusion

In this article, we construct the grammar and syntax for our sample compiler, build structure for flow and construct classes to implement in python. We walk through the whole life cycle of building a compiler, including context-free grammar, regular expression, lexical analysis, syntax analysis, semantic analysis, parsing and interpreting. Although this is a relatively simple example, reader can explore further development based on the work here to add features, such as declaring variables, evaluate control statement/conditional statement/assignment statement. With all features added, reader will be able to build a working compiler himself.

Reference

[1] How to write a very basic compiler, https://softwareengineering.stackexchange.com/questions/165543/how-to-write-a-very-basic-compiler, Taken on 20200911
[2] Writing a C Compiler, https://norasandler.com/2017/11/29/Write-a-Compiler.html, Taken on 20200911
[3] Let's Build A Simple Interpreter, https://ruslanspivak.com/lsbasi-part7/

Appendix A – Related External Links

- Video Explanation of this project – https://youtu.be/vQOxHV0gI8w
- Source Code in Github – https://github.com/wdmzjjtx66/CISC603-51A.git
- Environment Setup and Compilation Instruction – in Github as well

Appendix B – Python Source Code

```
import sys

# ------------------------------------------------------------------------
#                           Part I - Lexer
# ------------------------------------------------------------------------

# Tokens
# Tokens that are used to identify and indicate type of characters in the
# input. END means we are at end of the input, no more left for our
# analysis
PLUS, MINUS, MUL, DIV = 'PLUS', 'MINUS', 'MUL', 'DIV'
L_PAR, R_PAR, INTEGER, END = '(', ')', 'INTEGER', 'END'


class Token(object):
    def __init__(self, type, content):
        self.type = type
        self.content = content

    def __str__(self):
        # print class in string format if needed in testing
        # e.g Token(INTEGER, 6), Token(PLUS, '+'), Token(L_PAR, '(')
        return 'Token({type}, {content})'.format(type=self.type,
content=self.content)

    def __repr__(self):
        return self.__str__()


class Lexer(object):
    def __init__(self, text):
        # here text means input strings e.g. "3 - (5 * (4 // 2 + 100) + -
16)"
        self.text = text
        # position is a marker on the index of the input text we are
currently processing
        self.position = 0
        # curr_char is the character that position point to in the text
        self.curr_char = self.text[self.position]

    def forward(self):
        # each time we call this method, we advance the index and character
pointed to
        self.position += 1
        if self.position > len(self.text) - 1:
            # we have already reach the end of the file
            self.curr_char = None
        else:
            # update the current character
            self.curr_char = self.text[self.position]

    def get_integer(self):
        # get an integer from input text, could be multi-digits
        temp = 0
```

```
        while self.curr_char and self.curr_char.isdigit():
            temp = temp * 10 + int(self.curr_char)
            self.forward()
        return temp

    def skip_whitespace(self):
        # we need to skip whitespace in the input string
        while self.curr_char and self.curr_char.isspace():
            self.forward()

    def raise_error(self):
        raise Exception('Input character is invalid')

    def get_next_token(self):
        # this part is the implementation of tokenizer or lexer analysis
        # we just break down input text into tokens one by one
        while self.curr_char:

            if self.curr_char.isspace():
                self.skip_whitespace()
                continue

            if self.curr_char == '+':
                self.forward()
                return Token(PLUS, '+')

            if self.curr_char == '-':
                self.forward()
                return Token(MINUS, '-')

            if self.curr_char == '*':
                self.forward()
                return Token(MUL, '*')

            if self.curr_char == '/':
                self.forward()
                return Token(DIV, '/')

            if self.curr_char.isdigit():
                return Token(INTEGER, self.get_integer())

            if self.curr_char == '(':
                self.forward()
                return Token(L_PAR, '(')

            if self.curr_char == ')':
                self.forward()
                return Token(R_PAR, ')')

            # if the input is not one of the token we recognize, raise an
error
            self.raise_error()

        # done processing the input, point at end of the text
        return Token(END, None)
```

```
# ------------------------------------------------------------------------
#                           Part II - Parser
# ------------------------------------------------------------------------


# following part we implement abstract syntax tree, each node in the tree is
# either an operator or an integer, we will remove parenthesis by add the
# content within the parenthesis in their order of they should be processed.

class AST(object):
    # this is the class derived from the most basic system object - object
    # this class is the parent class for the following detailed
implementation
    pass


class UnaryOperator(AST):
    # first inheritance of base class AST - for unary operators "+-"
    def __init__(self, operator, expression):
        # token should represent one of the unary operators e.g. Token(PLUS,
'+')
        self.token = self.operator = operator
        # according to our syntax, an expression follows an unary operator
        self.expression = expression


class BinaryOperator(AST):
    # second derived child class of AST - for binary operator "+-*/"
    def __init__(self, left, operator, right):
        # there should be 2 child nodes for a binary operator - left and
right
        self.left = left
        self.token = self.operator = operator
        self.right = right


class Numbers(AST):
    # second derived child class, members are integer tokens like
Token(INTEGER, 2)
    def __init__(self, token):
        self.token = token
        self.content = token.content


# this is the implementation of second part of compiler - parse the tokens
from
# lexer and put them into AST which could be processed by the tree traversed
order
class Parser(object):
    def __init__(self, lexer):
        # use the lexer help break the input string into token, here we
process
        # one token at a time
        self.lexer = lexer
        self.curr_token = self.lexer.get_next_token()

    def ignore(self, token_type):
```

```
        # when we are processing a token, we should jump to the
        # next one in order to get next part to process. And when
        # ignore we should check if we are ignoring the correct
        # token, otherwise raise an error
        if self.curr_token.type == token_type:
            self.curr_token = self.lexer.get_next_token()
        else:
            self.raise_error()

    def factor(self):
        # this method is to put together tokens into 'factor' in
        # our syntax analysis
        token = self.curr_token
        if token.type == PLUS:
            self.ignore(PLUS)
            node = UnaryOperator(token, self.factor())
            return node
        elif token.type == MINUS:
            self.ignore(MINUS)
            node = UnaryOperator(token, self.factor())
            return node
        elif token.type == INTEGER:
            # we should add integer tokens directly
            self.ignore(INTEGER)
            return Numbers(token)
        elif token.type == L_PAR:
            # all tokens between parenthesis should be treated as
            # expression, we shall remove parenthesis also because
            # we won't need them in AST
            self.ignore(L_PAR)
            node = self.expression()
            self.ignore(R_PAR)
            return node

    def term(self):
        # same as factor function above, term in our syntax is just
        # two parts on the side of a "*/" operator
        node = self.factor()

        while self.curr_token.type == MUL or self.curr_token.type == DIV:
            token = self.curr_token
            if token.type == MUL:
                self.ignore(MUL)
            elif token.type == DIV:
                self.ignore(DIV)

            # construct a BinaryOperator, and we process the factor
            # on the right side further using factor() function
            node = BinaryOperator(left=node, operator=token,
right=self.factor())

        return node

    def expression(self):
        # in our syntax, expression is the most outside pact, so
        # we should start from here. expression can be separated
        # by "+-" into two terms.
```

```
        node = self.term()

        while self.curr_token.type == PLUS or self.curr_token.type == MINUS:
            token = self.curr_token
            if token.type == PLUS:
                self.ignore(PLUS)
            elif token.type == MINUS:
                self.ignore(MINUS)

            # same as term, left child node is current node, then
            # recursively build right node of expression
            node = BinaryOperator(left=node, operator=token,
right=self.term())

        return node

    def raise_error(self):
        raise Exception('Found a syntax error of input text')

    def parse(self):
        # build AST recursively, and return the root node of it
        # for interpreter to work on
        root = self.expression()
        # when we reach the end of the input test but
        # not given us END token, there must be an error
        if self.curr_token.type != END:
            self.raise_error()
        return root


# ---------------------------------------------------------------------------
#                          Part III - INTERPRETER
# ---------------------------------------------------------------------------



class NodeVisitor(object):
    # class we use to iterate AST we built from parser

    def visit(self, node):
        # specify the method name we should use depends on
        # the node type, kind of inheritance idea here
        method_name = 'visit_' + type(node).__name__
        visitor = getattr(self, method_name, self.visit_error)
        return visitor(node)

    def visit_error(self, node):
        # if there is no such method, we should raise an error
        raise Exception('No such a method called
visit_{}'.format(type(node).__name__))


# the implementation of our interpreter class, which takes
# AST built in parser and evaluate the input based on syntax
class Interpreter(NodeVisitor):
    def __init__(self, parser):
        self.parser = parser
```

```
    def visit_BinaryOperator(self, node):
        # we should output result based on the binary operator
        # type
        if node.operator.type == PLUS:
            return self.visit(node.left) + self.visit(node.right)
        elif node.operator.type == MINUS:
            return self.visit(node.left) - self.visit(node.right)
        elif node.operator.type == MUL:
            return self.visit(node.left) * self.visit(node.right)
        elif node.operator.type == DIV:
            try:
                return self.visit(node.left) // self.visit(node.right)
            except ZeroDivisionError:
                print("The denominator should not be zero")
                sys.exit(1)

    def visit_Numbers(self, node):
        # if the node is an integer, just output the integer
        return node.content

    def visit_UnaryOperator(self, node):
        # if it is a unary operator, just output the expression
        # with the prefix unary operator
        op = node.operator.type
        if op == PLUS:
            return +self.visit(node.expression)
        elif op == MINUS:
            return -self.visit(node.expression)

    def interpret(self):
        # recursive evaluate the input by traverse the AST tree
        # with various visit method inherited from the base class
        tree = self.parser.parse()
        if not tree:
            return ''
        return self.visit(tree)


# ---------------------------------------------------------------------------
#                          Part IV - Main Function
# ---------------------------------------------------------------------------


def main():
    while True:
        try:
            text = input('calculator> ')
        except EOFError:
            break
        if not text:
            continue

        # calling lexer to break input into tokens
        lexer = Lexer(text)

        # calling parser to construct AST
```

```
        parser = Parser(lexer)

        # calling interpreter to do calculation using
        # AST
        interpreter = Interpreter(parser)
        re = interpreter.interpret()
        print ('result is: {re}'.format(re=re))


if __name__ == '__main__':
    main()
```