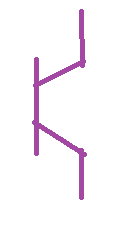
**Name of the programming lang : Journal**

**LOGIC GATES:**

**Computers are made of transistors , and transistors form the logic gates.**

****

**NOT , AND , OR , XOR , XNOR**

**BASIC ARITHMETIC:**

**Basic addition , subtraction , multiplication and division**

**MEMORY:**

**SET RESET Latch**

**Ink Drawings
Ink Drawings
Ink Drawings
Ink Drawings
Ink Drawings
Ink Drawings
Ink Drawings
Ink Drawings
Ink Drawings
Ink Drawings
Ink Drawings
Ink Drawings
Ink Drawings
Ink Drawings
Ink Drawings
Ink Drawings
Ink Drawings
Ink Drawings
Ink Drawings
**

**Note : Shit on my drawings**

**Computers remembers the state they were in and that is called saving.**

**Need for programming:**

**Inside computer we have a huge set of logic gates hooked up to understand instructions.**

**Ink Drawings
1000
4 Bit Instruction

Ink Drawings
Control Unit
Ink Drawings
Ink Drawings
Ink Drawings
Ink Drawings
Ink Drawings
Saving to Memory after
Execution
**

**Source Code Compiler (Lexing , Parsing , etc)**

**Machine Code**

**Compilers and Interpreters convert the source code into machine code but our interpreter(the one we would create) will not do that.**

**Our code would first be converted into Python code and then would run.**

**IMPLEMENTATION :**

**In our interpretation there are three steps involved :**

**Lexical Parsing Interpretation**

**Analysis**

**JOURNAL Interpreter Executable Code**

**The features which our language would incorporate are :**

* **Variables**
* **Arithmetic Operations**
* **Conditonal Operations**
* **Logical Operations**
* **Loops**

**LEXICAL ANALYSIS / TOKENIZATION :**

**It involves the broking down of input code into specific tokens.**

**Tokens can have a value or a type.**

**Output of this process would be feeded to the parser.**

**Example : 1 + 3 = [Token1(1,INT),Token2(+,Operator),Token3(3,INT)]**

**PARSING :**

**Recognition of patterns in an input is called parsing.**

**Grammer Rules:**

**BNF :- <expr> := <term> + <expr>**

**| <term> - <expr>**

**| <term>**

**Expression can be a single term or a term added or subtracted with another term or expression.**

**<term> := <factor> \* <factor>/<term>…..n times in a recursive manner.**

**<factor> := <integer> or something**

**<terminal> := basic tokens (operators or digits)**

**1 + 5 \* 2**

**The goal of the parsing process is to generate a binary tree like above.**

**OBJECT ORIENTED PROGRAMMING:**

def *fuelCalculation*(*initial* , *kilometers* , *ratio* ):

*return* initial - kilometers \* ratio

**In functional programming paradigm ,we separate the data form the actual functionality.**

**OOPs divides programs into objects.**

**DATA STRUCTURES :**

**DS we are going to use are Lists , Tuples , Dicts , Stacks**

class Stack:

    # LIFO

    def \_\_init\_\_(*self*):

*self*.stack = []

    def *push*(*self* , *item*):

*self*.stack.append(item)

    def *pop*(*self*):

*return* *self*.stack.pop() #This is not a recursive function , this pop() method is a predefined list() method

    def *peak*(*self*):

        lastElement = *self*.stack[-1:] # self.stack(len(self.stack) - 1)

*return* lastElement

stack1 = Stack()

stack1.push(3)

stack1.push(4)

stack1.push(5)

print(stack1.pop())

print(stack1.stack)

print(stack1.peak())

stack1.push(5)

print(stack1.peak())

**Binary Tree**

**Post Order Traversal will be used in our interpreter.**

**How PostOrder Traversal computes an expression ?**

# A binary tree is an hierarchical Data Structure

# Each node can have a maximum of two child nodes

# Traversal means visiting each and every node oon a tree

# PreOrder Traversal : RootNOde -> Left SubTree -> Right subTree : 1 + 2 \* 3 = 9 [R,1,2,3,\*]

# PostOrder Traversal : Left Subtree -> Right SubTree -> Root Node :=> 1 + 2 \* 3 = 7  [1,2,3,\*,R]

# CONSTRUCTION

class Node:

    def \_\_init\_\_(self , value):

        self.value = value

        self.leftChild = None

        self.rightChild = None

class Tree:

    def \_\_init\_\_(self , root):

        self.root = Node(root)

    def preOrdertraversal(self, start , records ) :

        if start is not None:    # if the node exists

            records.append(start.value)  # root NOde

            records = self.preOrdertraversal(start.leftChild , records ) # Left Subtree

            records = self.preOrdertraversal(start.rightChild , records) # Right Subtree

        return records

    def postOrdertraversal(self , start , records):

        if start is not None:

            records = self.postOrdertraversal(start.leftChild , records) #left Subtree

            records = self.postOrdertraversal(start.rightChild , records) # right subtree

            records.append(start.value)

        return records

tree = Tree(5)

tree.root.leftChild = Node('+')

tree.root.rightChild = Node(4)

tree.root.leftChild.leftChild = Node(3)

tree.root.rightChild.leftChild = Node('\*')

tree.root.rightChild.rightChild = Node(7)

#      5

#    /  \

#   +    4

#  /    / \

# 3    \*   7

print(tree.preOrdertraversal(tree.root,[]))

print(tree.postOrdertraversal(tree.root , []))

**Recursion :**

**A function calling itself.**

**LEXER:**

# Lexer will do lexical analysis , it will break our expression into tokens

class Lexer:

    digits = "0123456789" # number can be a single digit or a string of digits so we have to specify here

    operations = "+-/\*%"

    stopwords = [" "] #chunks of code which is to be ignored , currently it is blank spaces only

    def \_\_init\_\_(self,inputText):

        self.inputText = inputText

        self.idx = 0 # index

        self.tokens = list()

        self.char = self.inputText[self.idx]

        self.Token = None

    def tokenize(self):

        while self.idx < len(self.inputText):

            if self.char in Lexer.digits:

                self.Token = self.extractNumber()

            elif self.char in Lexer.operations:

                self.Token = Operation(self.char)

                self.move()

            elif self.char in Lexer.stopwords:

                self.move()

                continue   #coz we dont want to append anything to the tokens list , and we have to start a new iteration

            self.tokens.append(self.Token)

        return self.tokens

    def extractNumber(self):

        number = ""  # storing the number as a string

        isFloat = False #to check whether the number is a float or not

        while (self.char in Lexer.digits or self.char == ".") and (self.idx < len(self.inputText)):

            if self.char == ".":

                isFloat = True

            number += self.char

            self.move()

        return Integer(number) if not isFloat else Float(number)

    def move(self):  # for moving the character

        self.idx += 1

        if self.idx < len(self.inputText):

            self.char = self.inputText[self.idx]

class Token:

    def \_\_init\_\_(self , type , value):

        self.type = type

        self.value = value

    def \_\_repr\_\_(self):

        return self.value

class Integer(Token):

    def \_\_init\_\_(self,value):

        super().\_\_init\_\_("INT",value)

class Float(Token):

    def \_\_init\_\_(self,value):

        super().\_\_init\_\_("FLT",value)

class String(Token):

    pass

class Bool(Token):

    pass

class Operation(Token):

    def \_\_init\_\_(self,value):

        super().\_\_init\_\_("OP",value)

**Simple Expression Parser:**

*# 1 + 2 \* 3*

*# [1 , + , [2 ,\*,3]]*

class Parser:

    def \_\_init\_\_(self,tokens):

        self.tokens = tokens

        self.idx = 0

        self.token = self.tokens[self.idx]

    def factor(self):

        if self.token.Type == "INT" or self.token.Type == "FLT":

            return self.token

    def term(self):

*# 1 \* 2 : 1 -> LeftNode , \* -> RootNode , 2 -> RightNode*

        left\_node = self.factor()

        self.move()

        output = left\_node

        if self.token.value == "\*" or self.token.value == "/":

            operation = self.token

            self.move()

            right\_node = self.factor()

            self.move()

            output = [left\_node , operation , right\_node]

        return output

    def expression(self):

        left\_node = self.term()

        output = left\_node

        if self.token.value == "+" or self.token.value == "-":

            operation = self.token

            self.move()

            right\_node = self.term()

            output = [left\_node , operation , right\_node]

        return output

    def parse(self):

        return self.expression()

    def move(self):

        self.idx += 1

        if self.idx < len(self.tokens) :

            self.token = self.tokens[self.idx]

**Now , the parser fixed for handling multiple operations in a single expression:**

*# 1 + 2 \* 3*

*# [1 , + , [2 ,\*,3]]*

class Parser:

    def \_\_init\_\_(self,tokens):

        self.tokens = tokens

        self.idx = 0

        self.token = self.tokens[self.idx]

    def factor(self):

        if self.token.Type == "INT" or self.token.Type == "FLT":

            return self.token

    def term(self):

*# 1 \* 2 : 1 -> LeftNode , \* -> RootNode , 2 -> RightNode*

        left\_node = self.factor()

        self.move()

        output = left\_node

        while self.token.value == "\*" or self.token.value == "/":

            operation = self.token

            self.move()

            right\_node = self.factor()

            self.move()

            left\_node = [left\_node , operation , right\_node]

        return left\_node

    def expression(self):

        left\_node = self.term()

        output = left\_node

        while self.token.value == "+" or self.token.value == "-":

            operation = self.token

            self.move()

            right\_node = self.term()

            left\_node = [left\_node , operation , right\_node]

        return left\_node

    def parse(self):

        return self.expression()

    def move(self):

        self.idx += 1

        if self.idx < len(self.tokens) :

            self.token = self.tokens[self.idx]

**This below posted code is to extract the data type of the data.**

    if lNode.Type == 'INT':

            lNode = int(lNode.value)

        elif lNode.Type == 'FLT':

            lNode = float(lNode.value)

        if rNode.Type == 'INT':

            rNode = int(rNode.value)

        elif rNode.Type == 'FLT':

            rNode = float(rNode.value)

**This code can be rewritten by used get attribute method of Python.**

    def read\_INT(self,value):

        return int(value)

    def read\_FLT(self,value):

        return float(value)

    def compute\_bin(self,lNode , operator ,rNode):

        lNodetype = lNode.Type

        rNodetype = rNode.Type

        lNode = getattr(self , f"read\_{lNodetype}")(lNode.value) *#read\_INT(value)*

        rNode = getattr(self, f"read\_{rNodetype}")(rNode.value) *#read\_FLT(value)*

**Grammer Rules of Journal:**

1. **Variables assigning , saving & retrieving(done).**
2. **Unary Operations (Done)**
3. **Boolean and Comparison operators**
4. **If Elseif else statements**
5. **While Loop**