# CSN-352

# COMPILER DESIGN CODING PROJECT



| | |
|---|---|
| **Group Id** | **10** |
| **Project Number** | **39** |
| **Sumit Bera** | **20114023** |
| **Chinayush Waman Wasnik** | **20114027** |
| **Dhruv Shandilya** | **20114034** |

# Problem Statement

Building an Interpreter in Python language that does divide. Work with tokens, lexical analyser, and expressions that divide integers from integers.

# Working of Code

## Input

We accept input from the user in the form of input.txt. We then parse through this input using open() function in python and store all the lines in a python list(array).

.1) Code to accept .txt file as input

here 'lines' is a list containing all the lines

```
with open('input.txt') as f:
    lines = f.readlines()

print(lines)
```

Python

```
['d = 1/1.22/1.2\n', 'a = 1/2/(3/4)/2/(2/(2/9))\n', 'e = ((\n', 'f = 0 0\n', 'g = a/(0/1))\n', 'h = 2/\n', 'a /= 1\n', '
```

In the above image we can see that lines is a python list holding all the input lines for our code.

## Cleaning up input

Then we clean up each line in input by removing '\n' from end of line if it is present.

```
# first we remove '\n' from end of each line if it is present

for i in range(len(lines)):
    if lines[i][-1] == '\n':
        lines[i] = lines[i][:-1]
```

In the above code we check each line of our input and if a '\n' character is present at the end of a line we remove it.

## DFAs

We define DFA functions that check whether a lexeme is

i) An Identifier

```
def DFA_identifier(s):
    """
    DFA to check if a string is an identifier or not
    input : (string) s
    output : (bool) returns true if s is an identifier.
    """
    if isLetter(s[0]):
        for i in range(1,len(s)):
            if isLetter(s[i]) or isDigit(s[i]):
                pass
            else:
                return False
        return True

    else:
        return False
```

In the above code we have implemented a DFA that evaluates whether the input lexeme is an identifier or not. To be an identifier the first character always must be alphabetic [a-z], and the following characters may take their value from [0-9_] (zero to nine including underscore).

ii) A Number (float or int)

```
def DFA_num(s):
    """
    DFA to check if a string is an number or not
    input : (string) s
    output : (bool) returns true if s is an number.
    """
    dec=False
    if isDigit(s[0]):
        for i in range(1,len(s)):
            if isDigit(s[i]) :
                pass
            elif s[i]=='.' and dec==False:
                dec=True
            else:
                return False
        return True
    else:
        return False
```

In the above code we have defined a DFA to check if an entered lexeme is a number or not. For a lexeme to be a number the first character must be a digit [0-9], and the following characters can be any number of numbers [0-9], one decimal point [.], followed by any number of numbers. Our code does not accept lexemes like 1.2.3 as they have multiple decimal points.

iii) '=' sign

```
def DFA_equal(s):
    """
    DFA to check if a string is '='
    input : (string) s
    output : returns if string is equal to '='
    """
    return s=='='
```

Above is the straightforward DFA to check if a lexeme matches with the '=' symbol.

iv) '/' sign

```
def DFA_divide(s):
    """
    DFA to check if a string is '/'
    input : (string) s
    output : returns if string is equal to '/'
    """
    return s=='/'
```

Above is the straightforward DFA to check if a lexeme matches with the '/' symbol.

v) '(' sign

```
def DFA_lparen(s):
    """
    DFA to check if a string is '('
    input : (string) s
    output : returns if string is equal to '('
    """
    return s=='('
```

Above is the straightforward DFA to check if a lexeme matches with the '(' symbol.

vi) ')' sign

```
def DFA_rparen(s):
    """
    DFA to check if a string is ')'
    input : (string) s
    output : returns if string is equal to ')'
    """
    return s==')'
```

Above is the straightforward DFA to check if a lexeme matches with the ')' symbol.

## Lexical analysis

We then define two functions as follow:

### token()

This function accepts a lexeme and evaluates the DFA it matches to give its corresponding token. If no token is matched it returns 'none' as the token.

```
def token(s):
    """
    returns the token type of the string.
    input : (string) accepts a string and returns token type.
    output : (string) returns the token type.
    """

    if DFA_identifier(s):
        return 'IDENT'

    elif DFA_num(s):
        return 'NUM'

    elif DFA_equal(s):
        return 'EQ'

    elif DFA_divide(s):
        return 'DIV'

    elif DFA_lparen(s):
        return 'LPAREN'

    elif DFA_rparen(s):
        return 'RPAREN'

    return 'none'
```

In the above code we will be passing a lexeme and returning the associated token type by passing it through all the DFAs that we created above. If our token does not match with any of the DFAs then we return ‘none’ as the token type.

### Lexer()

This function first splits the string into substrings using delimiters as '/’, '=' , '(' and ')' and each substring we get we pass to the token() function to get the tokens of our lexemes.

```
def Lexer(s):
    """
    converts a line of code into stream of tokens.
    input : (string) accepts a line as string.
    output : (list of string) returns the stream of tokens.
    """
    lexeme = []
    tokens = []

    ptr = 0

    for i in range(len(s)):
        if search(s[i]):
            if s[ptr:i] != '':
                lexeme.append(s[ptr:i])
            if s[i]!=' ':
                lexeme.append(s[i])
            ptr = i+1

    if s[ptr:] != '':
        lexeme.append(s[ptr:])

    for i in lexeme:
        tokens.append(token(i))

    return (tokens,lexeme)
```

In the above function we parse through a line to be executed in our code. If it matches the character ‘ ‘ at any point we simply ignore it and if it matches any of the other delimiters like ‘(‘ , ‘)’ , ‘/’ and ‘=’ then we splice the part ahead of our character pointer i which has not been tokenised and we see what token it is provided ptr!=i . Also we append the token of the delimiters too to our token stream.

## Evaluation

In this we define comp_paren() method and pass our token and lexeme streams and return the evaluated value. We achieve this using the stack method used in infix expression evaluation.

```python
def comp_paren(lexeme, tokens, sym_tab):
    """
    computes expressions such as 1/((2/2)/3)
    Here, we will use the method we use to evaluate infix expressions.
    input :
        (list of string) lexeme is the list of lexemes.
        (list of string) tokens is the list of tokens.
        (dict) sym_tab is the dictionary having the symbol table of our past variables.
    output :
        (float) f is the float value we return after entire computation.
    """
    operand = []
    operator = []

    for i in range(len(tokens)):
        if tokens[i]=='NUM':
            operand.append(float(lexeme[i]))
        elif tokens[i]=='IDENT':
            operand.append(sym_tab[lexeme[i]])
        elif tokens[i]=='LPAREN':
            operator.append('LPAREN')
        elif tokens[i]=='DIV':
            if len(operator)==0:
                operator.append('DIV')
            else:
                try:
                    oprtr_top = operator.pop()
                    if oprtr_top=='DIV':
                        oprnd_top_1 = operand.pop()
                        oprnd_top_2 = operand.pop()
                        if oprnd_top_1==0:
                            return 'Error : Cannot divide by zero.'
```

(variable) oprtr_top: Any

```
        elif tokens[i]=='DIV':
            if len(operator)==0:
                operator.append('DIV')
            else:
                try:
                    oprtr_top = operator.pop()
                    if oprtr_top=='DIV':
                        oprnd_top_1 = operand.pop()
                        oprnd_top_2 = operand.pop()
                        if oprnd_top_1==0:
                            return 'Error : Cannot divide by zero.'
                        operand.append(oprnd_top_2/oprnd_top_1)
                        operator.append(tokens[i])
                    elif oprtr_top=='LPAREN':
                        operator.append(oprtr_top)
                        operator.append(tokens[i])
                    else:
                        return 'Error : Invalid expression.'
                except:
                    return 'Error : Invalid expression.'
        elif tokens[i]=='RPAREN':
            try:
                oprtr_top = operator.pop()
                while oprtr_top!='LPAREN':
                    oprnd_top_1 = operand.pop()
                    oprnd_top_2 = operand.pop()
                    if oprnd_top_1==0:
                        return 'Error : Cannot divide by zero.'
                    if oprtr_top=='DIV':
                        operand.append(oprnd_top_2/oprnd_top_1)
                    else:
                        return 'Error : Invalid operation.'
```

```
        elif tokens[i]=='RPAREN':
            try:
                oprtr_top = operator.pop()
                while oprtr_top!='LPAREN':
                    oprnd_top_1 = operand.pop()
                    oprnd_top_2 = operand.pop()
                    if oprnd_top_1==0:
                        return 'Error : Cannot divide by zero.'
                    if oprtr_top=='DIV':
                        operand.append(oprnd_top_2/oprnd_top_1)
                    else:
                        return 'Error : Invalid operation.'
                    oprtr_top = operator.pop()
            except:
                return 'Error : Invalid expression.'
        else:
            return 'Error :',tokens[i],' not a valid token.'

    while len(operator)!=0:
        try :
            oprtr_top = operator.pop()
            oprnd_top_1 = operand.pop()
            oprnd_top_2 = operand.pop()
            if oprnd_top_1==0:
                return 'Error : Cannot divide by zero.'
            operand.append(oprnd_top_2/oprnd_top_1)
        except:
            return 'Error : Invalid expression.'

    if len(operand)==1:
        return operand.pop()
    return 'Error : Invalid expression.'
```

## Final Step

Here we check our statement type. Whether it is an assignment statement or a normal expression. Then we pass the token and lexeme stream we had obtained directly to comp_paren() if it is computation like : 1/(2/2/3)

Otherwise if it is an expression like a = 1/(2/2/3) we only pass the token and lexeme stream associated with 1/(2/2/3) to our comp_paren() function and we store the returned float answer in our symbol table (sym_tab) under the variable name 'a'.

```
sym_tab = {}

f = open("output.txt", "w")

for i in range(len(list_tokens)):
    try:
        if list_tokens[i][0][1] == 'EQ':
            var = comp_paren(list_tokens[i][1][2:],list_tokens[i][0][2:],sym_tab)
            if type(var) == str:
                print('Error in line ',i+1,': ',var)
                f.write('Error in line '+str(i+1)+': '+str(var)+'\n')
            else:
                sym_tab[list_tokens[i][1][0]] = var
                print(var)
                f.write(str(var)+'\n')
        else:
            var = comp_paren(list_tokens[i][1],list_tokens[i][0],sym_tab)
            if type(var) == str:
                print('Error in line ',i+1,' : ',var)
                f.write('Error in line '+str(i+1)+': '+str(var)+'\n')
            else:
                print(var)
                f.write(str(var)+'\n')
    except:
        print('Error : Invalid code.')
        f.write('Error : Invalid code.\n')

print(sym_tab)
f.close()
```

## Test Cases:

**NOTE** : While copy pasting input from here sometimes extra random characters may come in our code editor which can be fixed by simply re-typing our input.

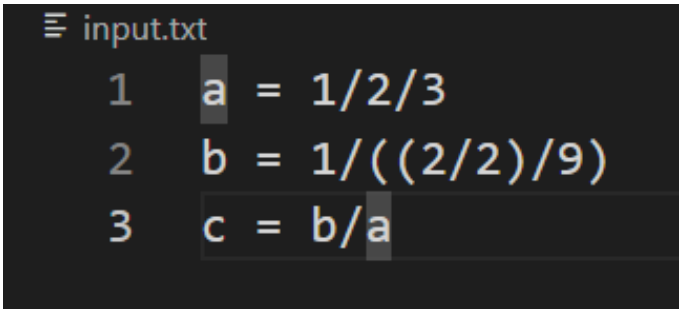
Simple Test case showing parentheses support, multiple divide in an expression and symbol table storage.

Input:

a = 1/2/3

b = 1/((2/2)/9)

c = b/a

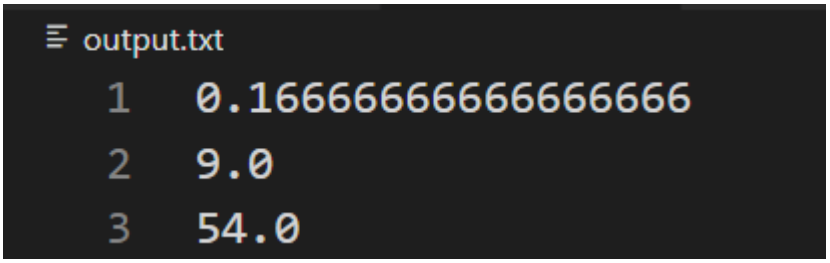



Output:

0.16666666666666666

9.0

54.0




A more complex testcase demonstrating the capabilities of our interpreter:

We have entered direct expressions too to be evaluated without assignment and it simply returns the value of the expression evaluated like a calculator.

Here, we demonstrate the capabilities of our interpreter to handle large decimal values and to handle multiple parenthesis.

Also, our identifiers can be multiple characters long.

Input:

abc = 12.228/(22/3)/2/1

3/2/(2/1)/((33/1)/22.8)

c = 9.234

def = 902.3/(100/2/(c/abc))/22.3

input.txt

```
1  abc = 12.228/(22/3)/2/1
2  3/2/(2/1)/((33/1)/22.8)
3  c = 9.234
4  def = 902.3/(100/2/(c/abc))/22.3
```

Output:

0.8337272727272728

0.5181818181818182

9.234

8.962763810471005

output.txt

```
1  0.8337272727272728
2  0.5181818181818182
3  9.234
4  8.962763810471005
```

Here, we will be demonstrating error handling capabilities of our code.

The errors will be in terms of lexemes not being any token type and invalid numbers and so on.

We will also see symbol table errors here.

Input:

dfs = 12.3.3.4

abc = 123/dfs

cds = .222/22

dd = ./9

12dfw = 23.09/21/92

input.txt

```
1   dfs = 12.3.3.4
2   abc = 123/dfs
3   cds = .222/22
4   dd = ./9
5   12dfw = 23.09/21/92
```

Output:

Error in line 1: Error : 12.3.3.4 not a valid token.

Error in line 2: Error : Identifier dfs not declared.

Error in line 3: Error : .222 not a valid token.

Error in line 4: Error : . not a valid token.

Error in line 5: Error :12dfw not a valid identifier.

output.txt

```
1  Error in line 1: Error : 12.3.3.4 not a valid token.
2  Error in line 2: Error : Identifier dfs not declared.
3  Error in line 3: Error : .222 not a valid token.
4  Error in line 4: Error : . not a valid token.
5  Error in line 5: Error :12dfw not a valid identifier.
```

Now we will be seeing some errors related to expressions in our code.

Such as, parenthesis not closed properly.

Invalid expressions not having proper operations between operands and so on.

Here, we will also see the cannot divide by zero error.

Input:

abc = 12.3/(9/(8/2)

def = 12/(7/0)

abs = ( (

cdwq = 9 9 8

sw = 21/()

input.txt

```
1   abc = 12.3/(9/(8/2)
2   def = 12/(7/0)
3   abs = ( (
4   cdwq = 9 9 8
5   sw = 21/()
```

Output:

Error in line 1: Error : Invalid expression.

Error in line 2: Error : Cannot divide by zero.

Error in line 3: Error : Invalid expression.

Error in line 4: Error : Invalid expression.

Error in line 5: Error : Invalid expression.

output.txt

```
1  Error in line 1: Error : Invalid expression.
2  Error in line 2: Error : Cannot divide by zero.
3  Error in line 3: Error : Invalid expression.
4  Error in line 4: Error : Invalid expression.
5  Error in line 5: Error : Invalid expression.
```