**SRM INSTITUTE OF SCIENCE AND TECHNOLOGY**

**Kattankulathur, Chengalpattu District - 603203**



**18CSC304J/ COMPLIER DESIGN**

**MINI PROJECT REPORT**

**SYNTAX CHECKER**

***Guided by:***

**Submitted By:**

Kunal Kankariya (RA2011003010323)

**TABLE OF CONTENTS**

|  |  |  |
| --- | --- | --- |
| **S.no** | **Title** | **Page no.** |
| **1** | **Abstract** | **3** |
| **2** | **Problem Statement** | **4** |
| **3** | **Aim** | **5** |
| **4** | **Requirements** | **6** |
| **5** | **Code** | **7** |
| **6** | **Output** | **14** |
| **7** | **Result** | **16** |
| **8** | **References** | **17** |

**ABSTRACT**

The Syntax Checker is an application of the Compiler Design process that focusses on building a tool to check the syntax of user input for a programming language. It is used to identify and report any syntax errors in the code such as misspelled keywords, unexpected characters, or incorrect use of operators.

The Syntax Checker is built using Lexer and Parser techniques, which are fundamental concepts in compiler design. The Lexer breaks down the input source code into individual tokens, while the Parser analyses the tokens and checks the syntax of the code. The Syntax Checker can be used as a standalone tool or integrated into a larger compiler toolchain.

**PROBLEM STATEMENT**

One of the main challenges in programming is ensuring that the source code is free of syntax error. Syntax refers to the format or rule of the commands in any programming language. Syntax Errors can cause the code to fail to compile or run correctly, leading to wasted time and effort in debugging. Moreover, manually checking the syntax of code can be a tedious and error- prone process, especially for large codebases.

Our Syntax Checker tool should automate the process of checking the syntax on input source code. It should be able to identify and report any syntax errors in the code, allowing developers to quickly fix the issues and improve their productivity.

**AIM**

The aim of the “Syntax Checker” project is to build a simple compiler tool that checks the syntax of input source code for a programming language. The tool should be able to perform the following tasks:

* Break down the input source code into individual tokens using lexer techniques.
* Analyse the tokens and check the syntax of the code using parser techniques.
* Identify and report any syntax errors in the code, including misspelled keywords, unexpected characters, and incorrect use of operators.
* Provide helpful error messages that allow developers to quickly fix the issues and improve their productivity.
* Be easy to use and integrate into existing toolchains.

**REQUIREMENTS**

The code requires Python 3.6 or higher to run. No additional external libraries or dependencies are required. The tool should be able to handle input source code written in a programming language similar to Python, including support for integers, strings, addition, subtraction, multiplication, division, and exponentiation.

The grammar for the supported arithmetic operations is given below:

program ::= statement\_list

statement\_list ::= statement | statement NEWLINE statement\_list

statement ::= expr | assignment | if\_statement

expr ::= term ((ADD | SUBTRACT) term)\*

term ::= factor ((MULTIPLY | DIVIDE | EXPONENT) factor)\*

factor ::= INTEGER | STRING | LPAREN expr RPAREN

assignment ::= NAME ASSIGN expr

if\_statement ::= IF expr COLON statement\_list ENDIF

**CODE**

# Token class

class Token:

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

        self.type = type

        self.value = value

    def \_\_repr\_\_(self):

        return 'Token({type}, {value})'.format(

            type=self.type,

            value=repr(self.value)

        )

# Token types

INTEGER = 'INTEGER'

STRING = 'STRING'

PLUS = 'PLUS'

MINUS = 'MINUS'

MULTIPLY = 'MULTIPLY'

DIVIDE = 'DIVIDE'

POWER = 'POWER'

EOF = 'EOF'

# Lexical Analyzer

class Lexer:

    def \_\_init\_\_(self, source\_code):

        self.source\_code = source\_code

        self.pos = 0

        self.current\_char = self.source\_code[self.pos]

    def advance(self):

        self.pos += 1

        if self.pos < len(self.source\_code):

            self.current\_char = self.source\_code[self.pos]

        else:

            self.current\_char = None

    def skip\_whitespace(self):

        while self.current\_char is not None and self.current\_char.isspace():

            self.advance()

    def get\_next\_token(self):

        while self.current\_char is not None:

            if self.current\_char.isspace():

                self.skip\_whitespace()

                continue

            if self.current\_char.isdigit():

                return self.integer()

            if self.current\_char == '"':

                return self.string()

            if self.current\_char == '+':

                self.advance()

                return Token(PLUS, '+')

            if self.current\_char == '-':

                self.advance()

                return Token(MINUS, '-')

            if self.current\_char == '\*':

                self.advance()

                return Token(MULTIPLY, '\*')

            if self.current\_char == '/':

                self.advance()

                return Token(DIVIDE, '/')

            if self.current\_char == '^':

                self.advance()

                return Token(POWER, '^')

            raise SyntaxError("Invalid token")

        return Token(EOF, None)

    def integer(self):

        result = ''

        while self.current\_char is not None and self.current\_char.isdigit():

            result += self.current\_char

            self.advance()

        return Token(INTEGER, int(result))

    def string(self):

        self.advance()

        result = ''

        while self.current\_char is not None and self.current\_char != '"':

            result += self.current\_char

            self.advance()

        if self.current\_char == '"':

            self.advance()

            return Token(STRING, result)

        else:

            raise SyntaxError("Invalid string")

# Syntax Analyzer

class Parser:

    def \_\_init\_\_(self, lexer):

        self.lexer = lexer

        self.current\_token = self.lexer.get\_next\_token()

    def error(self):

        raise SyntaxError("Invalid syntax")

    def eat(self, token\_type):

        if self.current\_token.type == token\_type:

            self.current\_token = self.lexer.get\_next\_token()

        else:

            self.error()

    def factor(self):

        token = self.current\_token

        if token.type == INTEGER:

            self.eat(INTEGER)

            return token.value

        elif token.type == STRING:

            self.eat(STRING)

            return token.value

        else:

            self.error()

    def term(self):

        result = self.factor()

        while self.current\_token.type in (MULTIPLY, DIVIDE):

            token = self.current\_token

            if token.type == MULTIPLY:

                self.eat(MULTIPLY)

                result \*= self.factor()

            elif token.type == DIVIDE:

                self.eat(DIVIDE)

                result /= self.factor()

        return result

    def expr(self):

        result = self.term()

        while self.current\_token.type in (PLUS, MINUS):

            token = self.current\_token

            if token.type == PLUS:

                self.eat(PLUS)

                result += self.term()

            elif token.type == MINUS:

                self.eat(MINUS)

                result -= self.term()

        return result

class SyntaxChecker:

    def \_\_init\_\_(self, source\_code):

        self.lexer = Lexer(source\_code)

        self.parser = Parser(self.lexer)

    def check\_syntax(self):

        try:

            self.parser.expr()

            print("Syntax is correct")

        except SyntaxError as e:

            print("Syntax error: ", e)

exp= input("Enter the expression:")

SyntaxChecker(exp).check\_syntax()

**OUTPUT**

Text

Description automatically generated

Text

Description automatically generated

Text

Description automatically generatedText

Description automatically generated

**RESULT**

The Syntax Checker runs successfully for simple expressions including strings, integers and those involving simple operations such as Addition, Subtraction, Multiplication, Division and Exponentiation. It demonstrates the principles of Compiler Design, which involves breaking down source code into smaller components and analyzing their structure to ensure they conform to a well-defined set of rules.

The project also provides a foundation for building more advanced compilers and interpreters that can translate source code into machine code or execute it directly.

All the above is executed using Python.

**REFERENCES**

1. [https://](NULL)www.geeksforgeeks.org/introduction-to-syntax-analysis-in-compiler-design/
2. <https://www.geeksforgeeks.org/type-checking-in-compiler-design/>
3. <https://www.codeproject.com/articles/30353/designing-a-compiler>