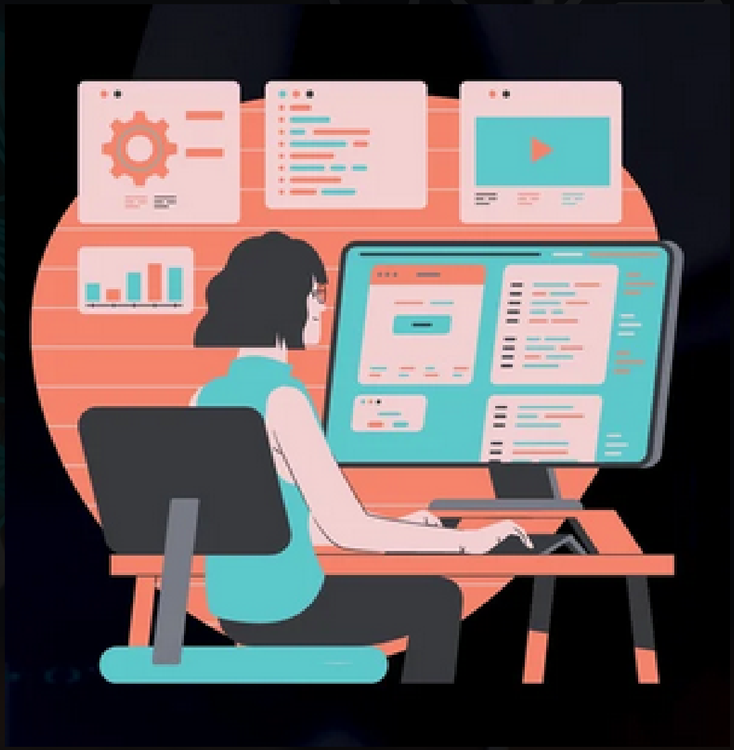
COMPILEME

Lexical and Syntax analyzer (python)

Presented By: Neveen Nabil Shukri



WELCOME TO MY COMPILER

I've always been curious about understanding how computers interpret and execute instructions From the hardware level, where electrical signals are processed by logic gates and circuits, to the software level, where high-level programming languages are translated into machine code

Compilers play a crucial role in this process by translating human-readable code into machine code that the computer can understand and execute. They analyze the syntax and semantics of the source code

TECHNOLOGY AND ARCHITECTURE:

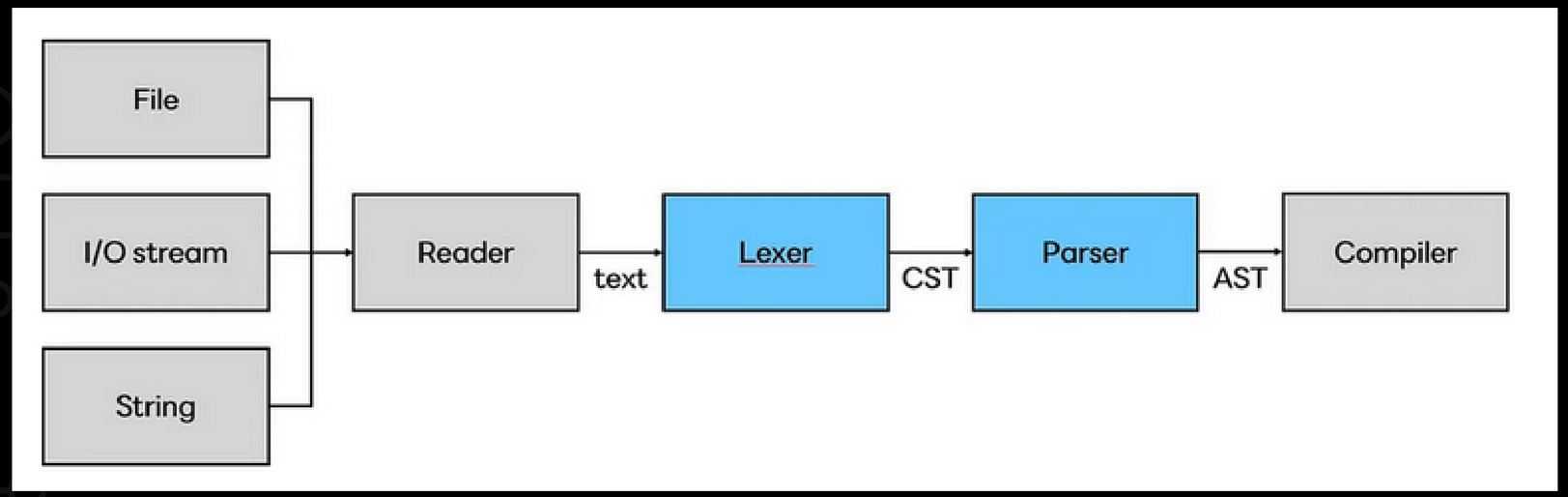
Implementation: Python
Libraries and Packages:

- Tkinter for implementing GUI
- PLY for lexical analysis and parsing in Python.
- webbrowser for visualizing the parse tree given the syntax analysis on www.webgraphviz.com

INTRODUCTION

The input Python code that can come from different sources like a file, I/O stream, or string format is first read by the reader, then fed into the lexer and parser, the lexer constructs the Concrete Syntax Tree (CST) from the input Python code, and the parser constructs the AST from the CST. The AST is fed into the compiler.

This procedure is shown in the figures below. Our focus of this article is the lexer and parser, which are colored in blue.



1. LEXICAL ANALYZER: (lexer or scanner)

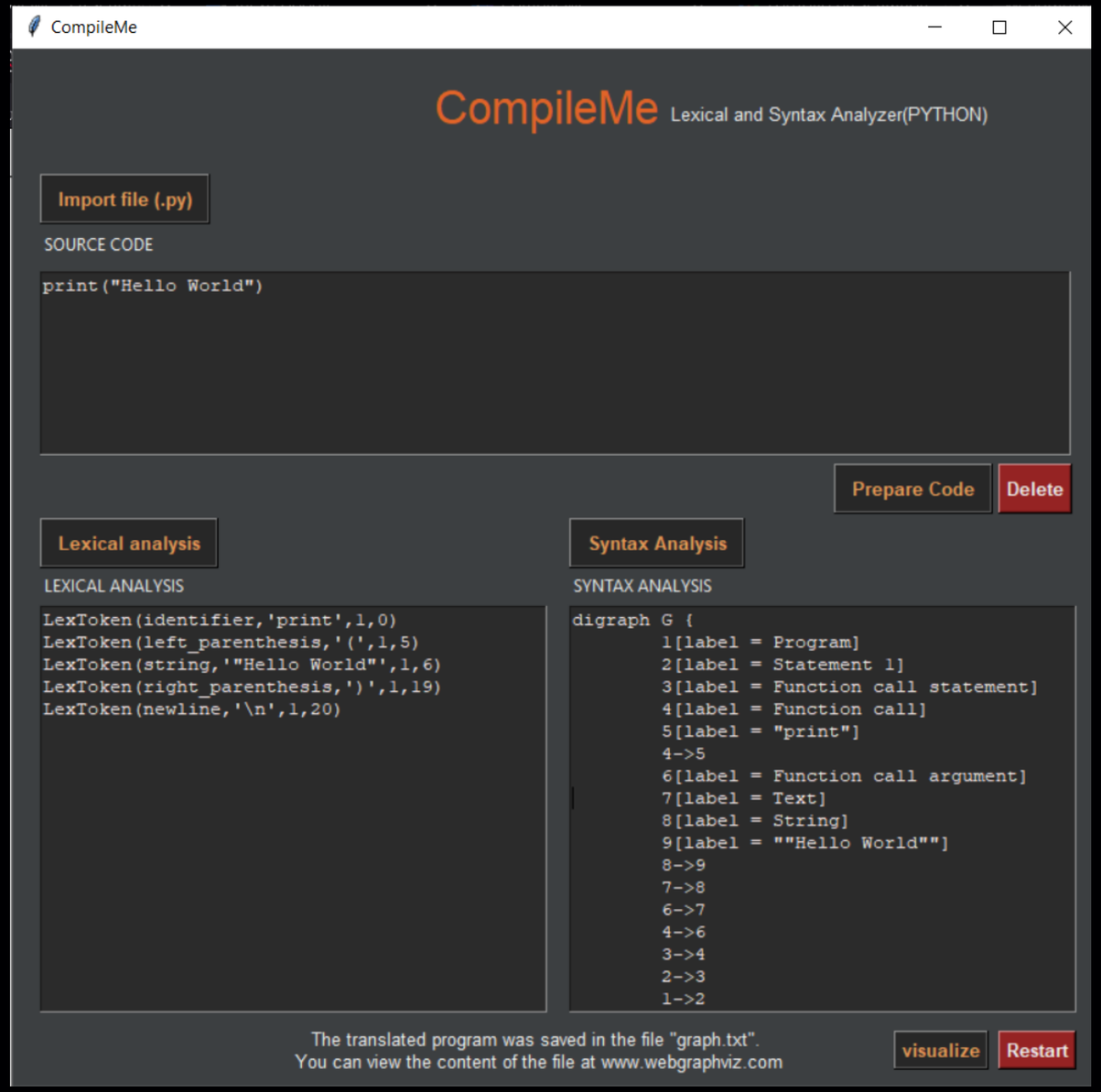
read the source code and break it down into individual tokens. **Tokens** are the smallest units of meaning in a programming language, such as keywords, identifiers, operators, and literals.

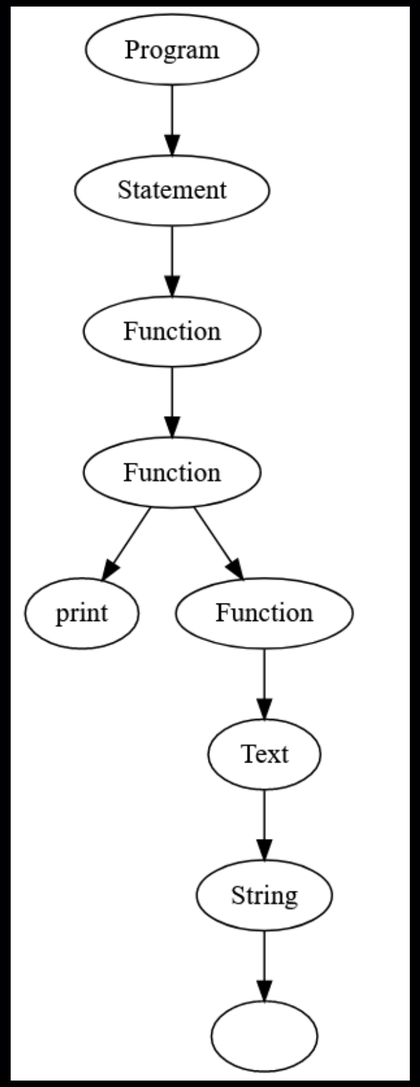
Tokens generated by the lexer serve as input for the the syntax analyzer.

2. SYNTAX ANALYZER: (parser)

examine the sequence of tokens generated by the lexer and determine whether they form a valid program according to the language's grammar rules. The syntax analyzer constructs a hierarchical structure called a syntax tree or parse tree, representing the grammatical structure of the program.

CORE ALGORITHMS AND CODE SNIPPET





GUI CODE SNIPPET

```python
from tkinter import *
import sys
import os
import webbrowser
import tkinter as tk
# Import specific functions from the tkinter library for file dialog and message box.
from tkinter.filedialog import askopenfilename
from tkinter.messagebox import showinfo
# Import the tkinter module with an alias 'tk' for convenience.

# Import custom modules syntax.py and lexical.py for additional functionality.
import syntax as sy
import lexical as lx

# Define global variables to be used across functions or parts of the program.
global input_text1  # Placeholder for a Tkinter text widget or related input field.
global input_text2  # Placeholder for another Tkinter text widget or related input field.
global input_text3  # Placeholder for yet another Tkinter text widget or related input field.
global filename  # Placeholder for storing the name of the current file.
global root
```

```python
# These global variables may be utilized to store and access inf
def main():
    global input_text1
    global input_text2
    global input_text3

    # Parent window declaration and configuration
    w = 780
    h = 745
    global root
    root = Tk()
    ws = root.winfo_screenwidth()
    hs = root.winfo_screenheight()
    x = (ws / 2) - (w / 2)
    y = (hs / 2) - (h / 2) - 35
    root.geometry('%dx%d+%d+%d' % (w, h, x, y))

    root.title("CompileMe")

    mainLabelColor = "#E16428"
    backgroundColor = "#3C3F41"
    textBackgroundColor = "#2B2B2B"
    textColor = "#F6E9E9"
    buttonBackgroundColor = "#272727"
    buttonColor = "#E79A58"
    labelColor = "#EEEEEE"

    root.configure(background=backgroundColor)
    root.resizable(0, 0)
```

```python
# Labels
Label(root, text="CompileMe", font=("Arial", 25), bg=backgroundColor, fg=mainLabelColor).place(x=300, y=20)
Label(root, text="Lexical and Syntax Analyzer(PYTHON)", font=("Arial", 10), bg=backgroundColor,
      fg=labelColor).place(x=470, y=36)
Label(root, text="The translated program was saved in the file \"graph.txt\". \nYou can view the content of the "
                 "file at www.webgraphviz.com", font=("Arial", 10), bg=backgroundColor, fg=labelColor).place(x=200,
                                                                                                             y=700)
Label(root, text="SOURCE CODE", bg=backgroundColor, fg=labelColor).place(x=20, y=130)
Label(root, text="LEXICAL ANALYSIS", bg=backgroundColor, fg=labelColor).place(x=20, y=375)
Label(root, text="SYNTAX ANALYSIS", bg=backgroundColor, fg=labelColor).place(x=400, y=375)
# Buttons
Button(root, text="Import file (.py)", font=("Arial", 10, "bold"), bg=buttonBackgroundColor,
       cursor="hand2", fg=buttonColor, padx=8, pady=5, command=lookForFile).place(x=20, y=90)
Button(root, text="Delete", pady=5, font=("Arial", 10, "bold"), cursor="hand2",
       bg="#952323", fg="#EEEEEE", command=eraseCode).place(x=708, y=298)
Button(root, text="Restart", font=("Arial", 10, "bold"), cursor="hand2",
       bg="#952323", fg="#EEEEEE", command=restart_application).place(x=708, y=705)
Button(root, text="visualize", font=("Arial", 10, "bold"), cursor="hand2",
       bg=buttonBackgroundColor, fg=buttonColor, command=visual).place(x=633, y=705)
Button(root, text="Lexical analysis", font=("Arial", 10, "bold"), bg=buttonBackgroundColor,
       cursor="hand2", padx=8, pady=5, fg=buttonColor, command=executeLexical).place(x=20, y=337)

Button(root, text="Syntax Analysis", font=("Arial", 10, "bold"), bg=buttonBackgroundColor,
       cursor="hand2", padx=8, pady=5, fg=buttonColor, command=executeSyntax).place(x=400, y=337)
Button(root, text="Prepare Code", font=("Arial", 10, "bold"), bg=buttonBackgroundColor,
       cursor="hand2", fg=buttonColor, padx=8, pady=5, command=prepareFile).place(x=590, y=298)
# Inputs
input_text1 = Text(root, width=92, height=8, bg=textBackgroundColor, fg=textColor)
input_text1.place(x=20, y=160)

input_text2 = Text(root, width=45, height=18, bg=textBackgroundColor, fg=textColor)
input_text2.place(x=20, y=400)

input_text3 = Text(root, width=45, height=18, bg=textBackgroundColor, fg=textColor)
input_text3.place(x=400, y=400)

# Start the main loop
root.mainloop()
sys.exit(0)
```

GUI CODE SNIPPET

```python
def lookForFile():
    """
    Opens a file dialog to select a Python (.py) file, reads its content,
    and displays it in the source code input area.
    """
    global filename
    global input_text1
    filename = tk.filedialog.askopenfilename()
    try:
        if filename.endswith(".py"):
            delete_empty_lines_at_end()
            file = open(filename, 'r')
            code = file.read()
            file.close()
            input_text1.delete('1.0', END)
            input_text1.insert(END, code)
        elif filename:
            tk.messagebox.showwarning("Error", "A .py file must be chosen")
            lookForFile()
    except Exception as e:
        pass
```

```python
def executeLexical():
    """
    Executes the lexical analysis using the main function from the 'lexical' module
    and displays the result in the lexical analysis output area.
    """
    global input_text2
    text = lx.main()
    input_text2.delete('1.0', END)
    input_text2.insert(END, text)


def executeSyntax():
    """
    Executes the syntax analysis using the main function from the 'syntax' module
    and displays the result in the syntax analysis output area.
    """
    global input_text3
    text = sy.main()
    input_text3.delete('1.0', END)
    input_text3.insert(END, text)
```

```python
def prepareFile():
    global filename
    global input_text1
    filename = "./input.py"
    # Writes the content of the source code input area to a file named 'input.py'.
    input_text2.delete('1.0', END)
    input_text3.delete('1.0', END)
    input_text0 = ""
    fileWrite = open('graph.txt', 'w')
    fileWrite.write(input_text0)
    fileWrite.close()
    delete_empty_lines_from_text()
    fileWrite = open('input.py', 'w')
    fileWrite.write(input_text1.get("1.0", END))
    fileWrite.close()
    return filename
```

```python
def eraseCode():
    # Clears the content of the source code input area.
    global input_text1
    input_text1.delete('1.0', END)
```

```python
def visual():
    url = "http://www.webgraphviz.com/"
    webbrowser.open(url)
```

LEXICAL MODULE

```python
# Import the 'lex' module from the 'ply' package for lexical analysis.
import ply.lex as lex
# Import the 'kwlist' from the 'keyword' module to include Python keywords in the tokens.
from keyword import kwlist
from tkinter import messagebox

# Define lists for different categories of tokens.
operators = ["plus", "minus", "product", "division", "integer_division", "module", "power", "equals", "non_equal",
             "less", "greater",
             "less_equal", "greater_equal"]
delimiters = ["left_parenthesis", "right_parenthesis", "left_bracket", "right_bracket", "left_brace", "right_brace",
              "period", "comma", "colon",
              "semicolon", "at", "assign", "increment", "decrement", "self_product", "self_division",
              "self_integer_division", "self_module", "self_power"]
others = ["identifier", "identation", "backspace", "float", "integer", "string", "newline"]
# Combine all categories to create the list of all token types.
tokens = kwlist + operators + delimiters + others

# Define a regular expression pattern to ignore spaces.
t_ignore = ' '

# Define regular expressions for each operator token.
t_plus = r'\+'
t_minus = r'-'
t_product = r'\*'
t_division = r'/'
t_integer_division = r'//'
t_module = r'%'
t_power = r'\*\*'
t_equals = r'=='
t_non_equal = r'!='
t_less = r'<'
t_greater = r'>'
t_less_equal = r'<='
t_greater_equal = r'>='
# Define regular expressions for each delimiter token.
t_left_parenthesis = r'\('
t_right_parenthesis = r'\)'
t_left_bracket = r'\['
t_right_bracket = r']'
t_left_brace = r'{'
```

```python
# Define a function to handle the integer.
def t_integer(t):
    r'-?\d+'
    t.value = int(t.value)
    return t


# Define a function to handle the string.
def t_string(t):
    r'(\" | \').*(\" | \')'
    return t


# Define a function to handle lexer errors.
def t_error(t):
    messagebox.showerror('Python Error', 'Error:' + "Invalid token %s at line %d, position %d"
                         % (t.value, t.lineno, t.lexpos))
    t.lexer.skip(1)


# Define the main function for the lexer.
def main():
    text = ""
    # Open the input file for reading.
    file = open("input.py", "r")
    # Read the content of the file.
    content = file.read()
    file.close()

    input_text0 = ""
    fileWrite = open('graph.txt', 'w')
    fileWrite.write(input_text0)
    fileWrite.close()

    # Create a lexer instance.
    lexer = lex.lex()
    # Set the lexer input to the content of the file.
    lexer.input(content)
    # Iterate over the tokens produced by the lexer.
    while True:
        token = lexer.token()
        if not token:
            break
        # Print the token and add its string representation to the text.
        text += token.__str__() + "\n"
    return text
```

SYNTAX MODULE

```python
import ply.yacc as yacc
import semantic
from semantic import *
from tkinter import messagebox

# Define the precedence of operators in the grammar
precedence = (
    ("left", "or"),
    ("left", "and"),
    ("right", "not"),
    ("right", "in"),
    ("right", "is"),
    ("right", "assign"),
    ("right", "increment", "decrement", "self_product", "self_division",
     "self_integer_division", "self_module", "self_power"),
    ("left", "equals", "non_equal"),
    ("left", "less", "greater", "less_equal", "greater_equal"),
    ("left", "plus", "minus"),
    ("left", "product", "division", "integer_division", "module"),
    ("left", "power"),
    ("left", "left_parenthesis", "right_parenthesis"),
)


def p_program(p):
    """program : statement_list"""
    # Parse a program consisting of a statement list
    p[0] = Program("Program", p[1])


def p_statement1(p):
    """statement : block_statement"""
    # Parse a statement as a block statement
    p[0] = NonTerminal("Block statement", p[1])


def p_block_statement1(p):
    """block_statement : header newline inner_statement_list backspace"""
    # Parse a block statement with a header, newline, inner statement list, and backspace
    p[0] = NonTerminal("Block statement", p[1], p[3])


def p_block_statement2(p):
    """block_statement : header newline inner_statement_list"""
    # Parse a block statement with a header, newline, and inner statement list
    p[0] = NonTerminal("Block statement", p[1], p[3])
```

```python
def p_empty(p):
    """empty :"""
    # Parse an empty production
    pass


def p_error(p):
    # Handle syntax errors during parsing
    messagebox.showerror('Compiler Error', 'Error:' + "Invalid syntax %s" % p)


global text
parser = yacc.yacc()


def restart_table():
    parser.LRTable = None


def main():
    global text

    input_text0 = ""
    fileWrite = open('graph.txt', 'w')
    fileWrite.write(input_text0)
    fileWrite.close()

    # Read content from the "input.py" file
    text = ""
    file = open("input.py", "r")
    content = file.read()
    file.close()

    # Create a parser and parse the content
    result = parser.parse(content, debug=0, tracking=True)
    text = (result.translate())

    # Save the translated program in the "graph.txt" file
    graphFile = open('graph.txt', 'w')
    graphFile.write(text)
    graphFile.close()

    del parser.statestack[:]
    del parser.symstack[:]
    semantic.count = 0
    semantic.txt = ""
    return text
```

SEMANTIC MODULE

```python
txt = ""  # Global variable for storing DOT language representation
count = 0  # Global variable for counting nodes in the graph


def add_count():
    # Function to increment the global count variable and return its value
    global count
    count += 1
    return "%s" % count


class Node:
    def __init__(self, name):
        # Constructor for the Node class, initializes a node with a given name
        self.name = name


class Program(Node):
    def __init__(self, name, *args):
        # Constructor for the Program class, a specialized Node representing a program
        # Takes a name and a variable number of child nodes (sons)
        Node.__init__(self, name)
        self.sons = []
        for x in args:
            self.sons.append(x)

    def print(self, indentation):
        # Method to print the tree structure with proper indentation
        for x in self.sons:
            x.print("\t" + indentation)
        print(indentation + "Node: " + self.name)

    def translate(self):
        # Method to translate the graph structure to DOT language
        global txt
        number = add_count()
        txt += number + "[label = " + self.name + "]\n\t"
        for x in self.sons:
            son = x.translate()
            txt += number + "->" + son + "\n\t"
        return "digraph G {\n\t" + txt + "}"
```

```python
class Terminal(Node):
    def __init__(self, name):
        Node.__init__(self, name)

    def print(self, identation):
        # Print information about the terminal node.
        print(identation + "Node: " + self.name)

    def translate(self):
        global txt
        number = add_count()
        # Conditionally set the label for specific terminal nodes.
        if self.name == "Squals" or self.name == "Non_equal" or ...:
            txt += number + "[label = \"" + str(self.name) + "\"]\n\t"
        else:
            txt += number + "[label = " + str(self.name) + "]\n\t"
        return number


class NonTerminal(Node):
    def __init__(self, name, *args):
        Node.__init__(self, name)
        self.sons = []
        for x in args:
            self.sons.append(x)

    def print(self, identation):
        # Print information about the non-terminal node and its sons.
        for x in self.sons:
            if type(x) == type(tuple()):
                x[0].print("\t" + identation)
            else:
                x.print("\t" + identation)
        print(identation + "Node: " + self.name)

    def translate(self):
        global txt
        number = add_count()
        txt += number + "[label = " + self.name + "]\n\t"
        # Translate each son and connect them to the current non-terminal node.
        for x in self.sons:
            if type(x) == type(tuple()):
                son = x[0].translate()
            else:
                son = x.translate()
            txt += number + "->" + son + "\n\t"
        return number
```

PROCESS AND TIMELINE

- Project Initiation and Planning:

Identified the need for a compiler project and defined project objectives, scope, and requirements.
Conducted research on compiler construction principles, tools, and technologies.
Developed a project plan outlining key milestones, deliverables, and a timeline for completion.

- Requirements Analysis:

Analyzed the requirements for the compiler project, including supported language features, input/output formats, and performance criteria.
Defined the grammar rules, token definitions, and syntax specifications for the target language.

PROCESS AND TIMELINE

- Implementation Phase:

Started implementation with the development of the lexical analysis module, including defining tokens, regular expressions, and lexer functions.
Proceeded with the parsing module, designing the grammar rules, constructing the parse table, and implementing parser functions.
Iteratively refined the implementation based on testing, feedback, and debugging, addressing issues and optimizing performance as needed.

- Semantic Analysis and Code Generation:

Implemented the semantic analysis module to perform semantic checks, type inference, and symbol resolution.
Developed the code generation module to translate the validated AST into target code, applying optimizations and generating executable output.

PROCESS AND TIMELINE

- Project Completion and Evaluation:

Completed the compiler project according to the defined timeline and objectives.

Conducted a thorough evaluation of the project outcomes, assessing the achievement of goals, adherence to requirements, and overall quality of the deliverables.

CHALLENGE(S) OVERCOME

- User Interface Design:

Designing an intuitive user interface for interacting with the compiler and viewing compilation results.

Conducted user testing and feedback sessions to iterate on the UI design and improve usability and user experience.

CHALLENGE(S) OVERCOME

- Error Handling:

Handling syntax errors and providing meaningful error messages to aid debugging.

Implemented error recovery mechanisms in the parser to gracefully handle syntax errors and resume parsing whenever possible.

CHALLENGE(S) OVERCOME

- Debugging and Troubleshooting:

Handling debugging and troubleshooting tasks independently, including identifying and fixing errors, optimizing performance, and ensuring correctness.

Leveraging debugging tools, logging mechanisms, and systematic problem-solving approaches to address technical challenges effectively.

CHALLENGE(S) OVERCOME

- Isolation and Collaboration:

Facing the challenge of working in isolation without the support and collaboration of team members.

Overcoming obstacles through self-motivation, seeking feedback from mentors or peers, and actively participating in online forums or communities for assistance and collaboration opportunities.

CHALLENGE(S) OVERCOME

- Time Management:

Managing time effectively to meet project deadlines and milestones while balancing other commitments and responsibilities.

Establishing a disciplined work routine, setting achievable goals, and prioritizing tasks based on their importance and urgency.

LIVE DEMO

LEARNINGS ABOUT TECHNICAL INTERESTS

- **Compiler Construction:** Developed a keen interest in compiler construction, including lexical analysis, parsing algorithms, semantic analysis, and code generation techniques.

- **Tool Development:** Acquired skills in developing tools and utilities to support compiler development, such as lexer and parser generators, syntax highlighters, and code editors.

- **Language Design:** Explored the principles of language design and implementation, including syntax design, semantics, type systems, and language features.

- **Problem-Solving Skills:** Enhanced problem-solving skills through tackling complex challenges in lexical analysis, parsing, error handling, and optimization.

- **Continuous Learning:** Embraced a culture of continuous learning and experimentation, recognizing the ever-evolving nature of technology and the importance of staying updated with the latest developments in compiler technology and programming languages.

LINKS

landing page:
https://nevennabil.github.io/CompileMe-Landing-Page/

Github Repo:
https://github.com/NevenNabil/CompileMe

Q & A

THANK YOU