National Textile University,

Faisalabad

Department of Computer Science

Group Members:	Amna Ali (22-NTU-CS-1145) Ammara Akhtar (22-NTU-CS-1144) M. Abdul Rehman Shakeel (22-NTU-CS-1183)
Project:	Mini Python Compiler with Indentation Checker
Course Code:	CSC-1074
Course Name:	Compiler Construction
Submitted To:	Dr. Hamid Ali

Table of Contents

Project Title:

Mini Python Compiler with Indentation Checker

Problem Statement:

Modern programming languages like Python depend heavily on indentation to define code blocks, making them sensitive to formatting errors. Beginner programmers often struggle with identifying and correcting such issues, which are not always obvious or clearly reported by traditional interpreters. There is a need for a simple tool that not only checks indentation and syntax but also demonstrates the key phases of compilation in an educational and visual manner.

Project Description:

This project implements a simplified Python-like compiler that guides users through the essential six phases of the compilation process: Lexical Analysis, Syntax Analysis, Semantic Analysis, Intermediate Code Generation, Code Optimization, and Code Generation.

Developed using **Python**, **PLY (Python Lex-Yacc)** for parsing, and **Tkinter** for the graphical interface, this mini compiler accepts a subset of Python syntax and compiles it into pseudo code. The tool features an indentation-aware lexer to simulate Python's block structure, highlights syntax errors, checks for semantic issues like undeclared variables, and generates intermediate code.

The main objective is to help learners understand how a compiler works internally and to provide a debugging-friendly environment that assists in correcting indentation and syntax-related mistakes.

Key Features:

- GUI-based input using Tkinter.
- Indentation analysis (using INDENT/DEDENT tokens).
- Tokenization of keywords, variables, operators, and numbers.
- Parse tree generation through PLY-based grammar rules.
- Symbol table for semantic checks.
- Intermediate three-address code generation.
- Basic constant folding optimization.
- Output of final generated pseudo code.

Functionalities:

- Accept user-written Python-like code through a GUI.

- Perform all six phases of compilation:
    1. Lexical Analysis
    2. Syntax Analysis
    3. Semantic Analysis
    4. Intermediate Code Generation
    5. Code Optimization
    6. Code Generation
- Show the generated code or errors in a pop-up dialog.

Main Objective:

To analyze and compile a subset of Python-like code, focusing on indentation, basic syntax, semantics, and converting it into intermediate code.

Six Phases of Compilation in the File:

1. Lexical Analysis

- **Objective:** Convert raw source code into tokens.
- **Algorithm Used:** *Finite State Automata (FSA)* via regex-based matching in PLY.
- Implementation:
    - Handled in t_* functions (e.g., t_NAME, t_NUMBER, t_NEWLINE, etc.).
    - Maintains an *indent stack* to simulate Python's block structure via INDENT and DEDENT tokens.

```
reserved = {
    'if': 'IF',
    'else': 'ELSE',
    'print': 'PRINT',
    'for': 'FOR',
    'in': 'IN',
    'range': 'RANGE'
}

tokens = [
    'COLON', 'NAME', 'NUMBER', 'EQUALS', 'EQEQ',
    'LPAREN', 'RPAREN', 'COMMA',
    'NEWLINE', 'INDENT', 'DEDENT'
] + list(reserved.values())

t_COLON = r':'
t_EQUALS = r'='
t_EQEQ = r'=='
t_LPAREN = r'\('
t_RPAREN = r'\)'
t_COMMA = r','

t_ignore = ' '

indent_stack = [0]
token_queue = []

def t_NUMBER(t):
    r'\d+'
    t.value = int(t.value)
    return t

def t_NAME(t):
    r'[a-zA-Z_][a-zA-Z0-9_]*'
    t.type = reserved.get(t.value, 'NAME')
    return t
```

```
def t_NEWLINE(t):
    r'\n[ \t]*'
    t.lexer.lineno += 1
    spaces = len(t.value) - 1
    last_indent = indent_stack[-1]

    new_newline = lex.LexToken()
    new_newline.type = 'NEWLINE'
    new_newline.value = '\n'
    new_newline.lineno = t.lexer.lineno
    new_newline.lexpos = t.lexpos

    if spaces > last_indent:
        indent_stack.append(spaces)
        indent = lex.LexToken()
        indent.type = 'INDENT'
        indent.value = ''
        indent.lineno = t.lexer.lineno
        indent.lexpos = t.lexpos
        token_queue.append(indent)
        return new_newline
    elif spaces < last_indent:
        while indent_stack and indent_stack[-1] > spaces:
            indent_stack.pop()
            dedent = lex.LexToken()
            dedent.type = 'DEDENT'
            dedent.value = ''
            dedent.lineno = t.lexer.lineno
            dedent.lexpos = t.lexpos
            token_queue.append(dedent)
        return new_newline
    else:
        return new_newline

def t_error(t):
    raise SyntaxError(f"Illegal character {t.value[0]!r} at line {t.lineno}")

lexer = lex.lex()
```

2. Syntax Analysis:

- **Objective:** Check grammar and build parse tree using tokens.
- **Algorithm Used:** *LALR Parser* (Look-Ahead LR) — PLY's default algorithm.
- Implementation:
  - Defined using p_* rules such as p_program, p_statement, etc.
  - Uses grammar productions like:
    - def p_assignment(p):
      'assignment : NAME EQUALS expr'

3. Semantic Analysis:

- **Objective:** Ensure logical correctness (e.g., variables used after declaration).
- **Algorithm Used:** *Symbol Table Checking*

- Implementation:
  - Maintains a symbol_table dictionary.
  - Example:
    - if p[1] not in symbol_table:
    - raise NameError(...)

4. Intermediate Code Generation:

- **Objective:** Translate parse tree into intermediate pseudo code.
- **Algorithm Used:** *Three-Address Code (TAC)* generation
- Implementation:
  - Appends expressions and statements to intermediate_code[]
  - Uses temporary variables like t0, t1, etc., via new_temp().

```
# ----------- SYNTAX, SEMANTIC, IR GENERATION -----------

start = 'program'
symbol_table = {}
intermediate_code = []
temp_count = 0

def new_temp():
    global temp_count
    temp = f"t{temp_count}"
    temp_count += 1
    return temp

def p_program(p):
    'program : statement_list'
    p[0] = p[1]

def p_statement_list(p):
    '''statement_list : statement
                      | statement_list statement'''
    pass

def p_statement(p):
    '''statement : simple_stmt NEWLINE
                 | compound_stmt
                 | NEWLINE'''
    pass

def p_simple_stmt(p):
    '''simple_stmt : assignment
                   | print_stmt'''
    pass

def p_assignment(p):
    'assignment : NAME EQUALS expr'
    symbol_table[p[1]] = 'int'
    intermediate_code.append(f"{p[1]} = {p[3]}")

def p_print_stmt(p):
    'print_stmt : PRINT LPAREN expr RPAREN'
    intermediate_code.append(f"print({p[3]})")
```

```
def p_expr_name(p):
    'expr : NAME'
    if p[1] not in symbol_table:
        raise NameError(f"Undefined variable '{p[1]}'")
    p[0] = p[1]

def p_expr_number(p):
    'expr : NUMBER'
    p[0] = str(p[1])

def p_expr_eqeq(p):
    'expr : expr EQEQ expr'
    temp = new_temp()
    intermediate_code.append(f"{temp} = {p[1]} == {p[3]}")
    p[0] = temp

def p_compound_stmt(p):
    '''compound_stmt : IF expr COLON NEWLINE INDENT statement_list DEDENT
                     | ELSE COLON NEWLINE INDENT statement_list DEDENT
                     | FOR NAME IN RANGE LPAREN NUMBER RPAREN COLON NEWLINE INDENT statement_list DEDENT'''
    if p[1] == 'if':
        intermediate_code.append(f"if {p[2]}:")
    elif p[1] == 'else':
        intermediate_code.append("else:")
    elif p[1] == 'for':
        var = p[2]
        rng = p[6]
        symbol_table[var] = 'int'
        intermediate_code.append(f"for {var} in range({rng}):")

def p_error(p):
    if p:
        raise SyntaxError(f"Syntax error at '{p.value}' line {p.lineno}")
    else:
        raise SyntaxError("Syntax error at EOF")

parser = yacc.yacc()
```

5. Code Optimization:

- **Objective:** Improve performance of intermediate code.
- **Algorithm Used:** *Constant Folding*
- Implementation:
  - Optimizes expressions like 3 == 3 into True during compilation.

```python
# ----------- OPTIMIZATION -----------

def optimize(code_list):
    optimized = []
    for line in code_list:
        if '==' in line and all(part.strip().isdigit() for part in line.split('==')):
            lhs, rhs = line.split('=')
            val = eval(rhs.strip())
            optimized.append(f"{lhs.strip()} = {val}")
        else:
            optimized.append(line)
    return optimized
```

6. Code Generation:

- **Objective:** Generate readable final code (for display, not machine code).
- **Algorithm Used:** *Linear Code Emission*
- Implementation:
  - Joins all lines of optimized intermediate code into a string with \n.

```python
#----------- Assembly code generation -------
def generate_assembly(code_lines):
    asm_code = []
    for line in code_lines:
        line = line.strip()
        # Assignment:
        if '=' in line and not line.startswith('if') and not line.startswith('for') and not line.startswith('else'):
            var, val = line.split('=', 1)
            var = var.strip()
            val = val.strip()

            asm_code.append(f"MOV {var}, {val}")
        # print statement:
        elif line.startswith('print(') and line.endswith(')'):
            inside = line[6:-1].strip()
            asm_code.append(f"PRINT {inside}")
        # if statement
        elif line.startswith('if'):
            condition = line[3:-1].strip()  # remove 'if ' and ':'
            asm_code.append(f"; IF {condition}")
        # else statement
        elif line.startswith('else'):
            asm_code.append("; ELSE")
        # for loop
        elif line.startswith('for'):
            asm_code.append(f"; {line}")
        else:
            asm_code.append(f"; {line}  ; (unknown)")
    return "\n".join(asm_code)
```

Conclusion:

The *Mini Python Compiler with Indentation Checker* project successfully demonstrates the core concepts and phases of compilation in a simplified and educational way. By simulating Python-like syntax and enforcing indentation-based block structure, the project provides an intuitive environment for understanding compiler design.

Through the use of **PLY** for lexical and syntax analysis and **Tkinter** for a user-friendly GUI, the compiler not only detects syntactic and semantic errors but also generates intermediate and optimized pseudo code. This makes it an effective learning tool for students and developers who wish to understand how a compiler processes and transforms high-level code into lower-level representations.

The project fulfills its goal of bridging the gap between abstract compiler theory and practical implementation, offering insights into:

- Tokenization and grammar parsing
- Symbol table usage
- Indentation-based block handling (unique to Python)
- Error reporting and optimization techniques

In conclusion, this mini compiler provides a strong foundation for further exploration into more advanced compiler concepts such as code optimization, backend code generation, and support for broader language features.