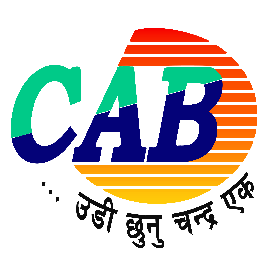
**COLLEGE OF APPLIED BUSINESS AND TECHNOLOGY**

**Gangahity, Chabahil Kathmandu**



**Compiler Design and Construction**

**PRACTICAL FILE-2081**

**Submitted by:**

Az Kafle (106)

College of Applied Business and Technology

B.Sc.CSIT 6th Semester

**Submitted to:**

Mr. Ashok Pandey

# Task 1

## Write a program to implement Lexical Analyzer to identify token.

**Source Code:**

import re

# Define token patterns

token\_specification = [

    ('NUMBER',    r'\d+(\.\d\*)?'),  # Integer or decimal number

    ('ASSIGN',    r'='),            # Assignment operator

    ('END',       r';'),            # Statement terminator

    ('ID',        r'[A-Za-z]+'),    # Identifiers

    ('OP',        r'[+\-\*/]'),      # Arithmetic operators

    ('SKIP',      r'[ \t]'),        # Skip over spaces and tabs

    ('MISMATCH',  r'.'),            # Any other character

]

# Compile the regex for performance

token\_re = '|'.join(f'(?P<{pair[0]}>{pair[1]})' for pair in token\_specification)

get\_token = re.compile(token\_re).match

# Token class to store token information

class Token:

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

        self.type = type

        self.value = value

        self.position = position

    def \_\_repr\_\_(self):

        return f'Token({self.type}, {self.value}, {self.position})'

def lex(text):

    line\_no = 1

    line\_start = 0

    pos = 0

    tokens = []

    match = get\_token(text)

    while match is not None:

        type = match.lastgroup

        value = match.group(type)

        if type == 'NUMBER':

            value = float(value) if '.' in value else int(value)

        elif type == 'ID' and value in {'if', 'else', 'for', 'while'}:

            type = value.upper()

        elif type == 'SKIP':

            pos = match.end()

            match = get\_token(text, pos)

            continue

        elif type == 'MISMATCH':

            raise RuntimeError(f'{value!r} unexpected on line {line\_no}')

        tokens.append(Token(type, value, match.start()))

        pos = match.end()

        match = get\_token(text, pos)

    tokens.append(Token('EOF', '', pos))

    return tokens

# Example usage

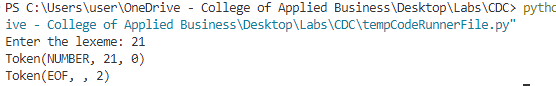
text = input('Enter the lexeme: ')

tokens = lex(text)

for token in tokens:

    print(token)

**Output:**



# Task 2

## Write a program to implement First of grammar.

**Source Code:**

# Define the grammar rules

# Example grammar:

# S -> AB

# A -> aA | ε

# B -> bB | ε

grammar = {

    'S': [['A', 'B']],

    'A': [['a', 'A'], ['ε']],

    'B': [['b', 'B'], ['ε']]

}

# Initialize the FIRST set

first = {non\_terminal: set() for non\_terminal in grammar}

# Function to compute the FIRST set for a given non-terminal

def compute\_first(non\_terminal):

    # If FIRST is already computed, return it

    if first[non\_terminal]:

        return first[non\_terminal]

    # Process each production rule

    for production in grammar[non\_terminal]:

        for symbol in production:

            if symbol.islower():  # Terminal symbol

                first[non\_terminal].add(symbol)

                break

            elif symbol == 'ε':  # Epsilon

                first[non\_terminal].add('ε')

                break

            else:  # Non-terminal symbol

                first\_set = compute\_first(symbol)

                if 'ε' in first\_set:

                    first[non\_terminal].update(first\_set - {'ε'})

                else:

                    first[non\_terminal].update(first\_set)

                    break

        else:

            first[non\_terminal].add('ε')

    return first[non\_terminal]

# Compute the FIRST set for all non-terminals

for non\_terminal in grammar:

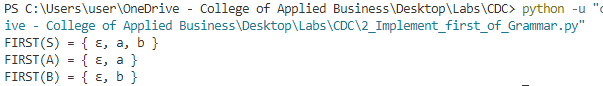
    compute\_first(non\_terminal)

# Print the FIRST sets

for non\_terminal, first\_set in first.items():

    print(f'FIRST({non\_terminal}) = {{ {", ".join(first\_set)} }}')

**Output:**



# Task 3

## Write a program to implement Follow of grammar.

**Source Code:**

class ShiftReduceParser:

    def \_\_init\_\_(self, grammar):

        self.grammar = grammar

        self.stack = []

        self.input = []

        self.action = None

    def parse(self, tokens):

        self.stack = []

        self.input = tokens + ["$"]  # End of input marker

        self.action = None

        while self.input:

            print(f"Stack: {self.stack}, Input: {self.input}, Action: {self.action}")

            if self.reduce():

                if self.stack == ["E"] and self.input == ["$"]:

                    print(f"Stack: {self.stack}, Input: {self.input}, Action: Accepted")

                    return True

                continue

            if self.shift():

                continue

            print("Error: No valid actions available")

            return False

        print(f"Stack: {self.stack}, Input: {self.input}, Action: Rejected")

        return False

    def shift(self):

        if self.input and self.input[0] != "$":

            self.action = "Shift"

            self.stack.append(self.input.pop(0))

            return True

        return False

    def reduce(self):

        for lhs, rhs\_list in self.grammar.items():

            for rhs in rhs\_list:

                if self.stack[-len(rhs):] == rhs:

                    self.action = f"Reduce by {lhs} -> {' '.join(rhs)}"

                    self.stack = self.stack[:-len(rhs)]

                    self.stack.append(lhs)

                    return True

        return False

if \_\_name\_\_ == "\_\_main\_\_":

    grammar = {

        "E": [

            ["E", "+", "E"],

            ["E", "\*", "E"],

            ["(", "E", ")"],

            ["id"]

        ]

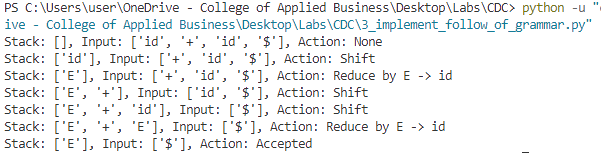
    }

    parser = ShiftReduceParser(grammar)

    tokens = ["id", "+", "id"]

    parser.parse(tokens)

**Output:**



# Task 4

## Write a program to implement Shift Reduce Parser.

**Source Code:**

# Define the grammar rules

# Example grammar:

# S -> AB

# A -> aA | ε

# B -> bB | ε

grammar = {

    'S': [['A', 'B']],

    'A': [['a', 'A'], ['ε']],

    'B': [['b', 'B'], ['ε']]

}

# Initialize the FIRST set

first = {non\_terminal: set() for non\_terminal in grammar}

# Function to compute the FIRST set for a given non-terminal

def compute\_first(non\_terminal):

    # If FIRST is already computed, return it

    if first[non\_terminal]:

        return first[non\_terminal]

    for production in grammar[non\_terminal]:

        for symbol in production:

            if symbol.islower():  # Terminal symbol

                first[non\_terminal].add(symbol)

                break

            elif symbol == 'ε':  # Epsilon

                first[non\_terminal].add('ε')

                break

            else:  # Non-terminal symbol

                first\_set = compute\_first(symbol)

                if 'ε' in first\_set:

                    first[non\_terminal].update(first\_set - {'ε'})

                else:

                    first[non\_terminal].update(first\_set)

                    break

        else:

            first[non\_terminal].add('ε')

    return first[non\_terminal]

# Compute the FIRST set for all non-terminals

for non\_terminal in grammar:

    compute\_first(non\_terminal)

# Initialize the FOLLOW set

follow = {non\_terminal: set() for non\_terminal in grammar}

follow['S'].add('$')  # End of input symbol for the start symbol

# Function to compute the FOLLOW set for all non-terminals

def compute\_follow():

    while True:

        updated = False

        for non\_terminal, productions in grammar.items():

            for production in productions:

                trailer = follow[non\_terminal].copy()

                for symbol in reversed(production):

                    if symbol in grammar:  # Non-terminal

                        if follow[symbol] != follow[symbol].union(trailer):

                            follow[symbol].update(trailer)

                            updated = True

                        if 'ε' in first[symbol]:

                            trailer.update(first[symbol] - {'ε'})

                        else:

                            trailer = first[symbol].copy()

                    else:  # Terminal

                        trailer = {symbol}

        if not updated:

            break

# Compute the FOLLOW sets

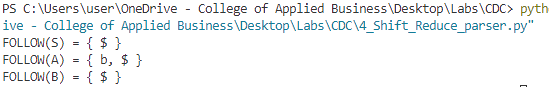
compute\_follow()

# Print the FOLLOW sets

for non\_terminal, follow\_set in follow.items():

    print(f'FOLLOW({non\_terminal}) = {{ {", ".join(follow\_set)} }}')

**Output:**



# Task 5

## Write a program to implement LR Parser.

**Source Code:**

class LRParser:

    def \_\_init\_\_(self):

        # Define the simplified grammar rules

        self.grammar = [

            ("E", ["E", "+", "E"]),  # Rule 1

            ("E", ["id"])            # Rule 2

        ]

        # Define the simplified parsing table

        self.action = {

            (0, "id"): ("S", 2),   # Shift and go to state 2

            (0, "+"): None,       # Error

            (1, "$"): ("ACC",),   # Accept

            (2, "+"): ("S", 3),   # Shift and go to state 3

            (2, "$"): ("R", 1),   # Reduce using rule 1 (E -> E + E)

            (3, "id"): ("S", 2),  # Shift and go to state 2

        }

        self.goto = {

            (0, "E"): 1,

            (3, "E"): 4

        }

        self.stack = [0]  # Initial state

    def parse(self, tokens):

        tokens.append("$")  # End of input marker

        index = 0

        while True:

            state = self.stack[-1]

            token = tokens[index]

            if (state, token) in self.action:

                action, value = self.action[(state, token)]

                if action == "S":

                    # Shift operation

                    self.stack.append(value)

                    index += 1

                elif action == "R":

                    # Reduce operation

                    rule = self.grammar[value]

                    for \_ in range(len(rule[1])):

                        self.stack.pop()

                    goto\_state = self.goto[(self.stack[-1], rule[0])]

                    self.stack.append(goto\_state)

                elif action == "ACC":

                    # Accept operation

                    print("Accepted")

                    return True

            else:

                print("Error")

                return False

# Example usage

parser = LRParser()

tokens = ["id", "+", "id"]

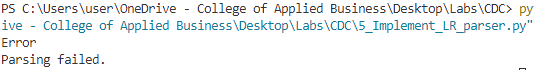
if parser.parse(tokens):

    print("Parsing successful.")

else:

    print("Parsing failed.")

**Output:**



# Task 6

## Write a program to implement Intermediate code generation.

**Source Code:**

class IntermediateCodeGenerator:

    def \_\_init\_\_(self):

        self.temp\_counter = 0

        self.instructions = []

        self.label\_counter = 0

    def generate\_temp(self):

        """Generate a new temporary variable name."""

        self.temp\_counter += 1

        return f"t{self.temp\_counter - 1}"

    def generate\_label(self):

        """Generate a new label name."""

        self.label\_counter += 1

        return f"L{self.label\_counter - 1}"

    def add\_instruction(self, instruction):

        """Add an instruction to the list of instructions."""

        self.instructions.append(instruction)

    def generate\_code(self, ast):

        """Generate intermediate code from the AST."""

        self.visit(ast)

        return self.instructions

    def visit(self, node):

        """Visit a node in the AST."""

        if isinstance(node, BinOp):

            return self.visit\_binop(node)

        elif isinstance(node, Num):

            return node.value

        elif isinstance(node, Variable):

            return node.name

    def visit\_binop(self, node):

        """Handle binary operations."""

        left = self.visit(node.left)

        right = self.visit(node.right)

        result = self.generate\_temp()

        self.add\_instruction(f"{result} = {left} {node.op} {right}")

        return result

class ASTNode:

    """Base class for all AST nodes."""

    pass

class BinOp(ASTNode):

    def \_\_init\_\_(self, left, op, right):

        self.left = left

        self.op = op

        self.right = right

class Num(ASTNode):

    def \_\_init\_\_(self, value):

        self.value = value

class Variable(ASTNode):

    def \_\_init\_\_(self, name):

        self.name = name

# Example usage

if \_\_name\_\_ == "\_\_main\_\_":

    # Constructing AST for the expression: (2 + 6) \* 3

    ast = BinOp(

        left=BinOp(left=Num(2), op='+', right=Num(6)),

        op='\*',

        right=Num(3)

    )

    # Generate intermediate code

    generator = IntermediateCodeGenerator()

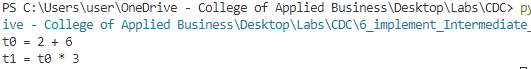
    intermediate\_code = generator.generate\_code(ast)

    # Print intermediate code

    for instruction in intermediate\_code:

        print(instruction)

**Output:**



# Task 7

## Write a program to implement Final code generation.

**Source Code:**

class FinalCodeGenerator:

    def \_\_init\_\_(self):

        self.instructions = []

        # Register mapping for temporary variables

        self.registers = {"t0": "R0", "t1": "R1", "t2": "R2", "t3": "R3"}

    def generate\_final\_code(self, intermediate\_code):

        """Generate final assembly code from intermediate code."""

        for instruction in intermediate\_code:

            self.translate\_instruction(instruction)

        return self.instructions

    def translate\_instruction(self, instruction):

        """Translate a single intermediate code instruction to assembly."""

        parts = instruction.split()

        temp\_var = parts[0]

        op1 = parts[2]

        operator = parts[3]

        op2 = parts[4]

        # Load operands into registers

        if op1.isdigit():

            self.add\_instruction(f"LOADI {op1}, {self.registers[temp\_var]}")

        else:

            self.add\_instruction(f"LOAD {op1}, {self.registers[temp\_var]}")

        if op2.isdigit():

            self.add\_instruction(f"LOADI {op2}, R3")

        else:

            self.add\_instruction(f"LOAD {op2}, R3")

        # Perform the operation

        if operator == "+":

            self.add\_instruction(f"ADD {self.registers[temp\_var]}, R3")

        elif operator == "-":

            self.add\_instruction(f"SUB {self.registers[temp\_var]}, R3")

        elif operator == "\*":

            self.add\_instruction(f"MUL {self.registers[temp\_var]}, R3")

        elif operator == "/":

            self.add\_instruction(f"DIV {self.registers[temp\_var]}, R3")

        # Store the result back to the temporary variable

        self.add\_instruction(f"STORE {self.registers[temp\_var]}, {temp\_var}")

    def add\_instruction(self, instruction):

        """Add an instruction to the list of final instructions."""

        self.instructions.append(instruction)

# Example usage

if \_\_name\_\_ == "\_\_main\_\_":

    # Example intermediate code

    intermediate\_code = [

        "t0 = 2 + 6",

        "t1 = t0 \* 3"

    ]

    # Generate final code

    final\_generator = FinalCodeGenerator()

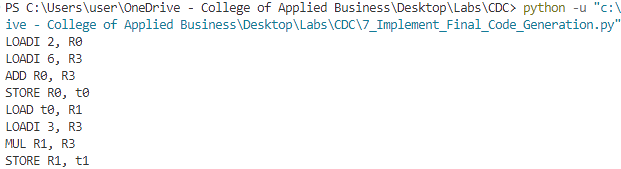
    final\_code = final\_generator.generate\_final\_code(intermediate\_code)

    # Print final code

    for instruction in final\_code:

        print(instruction)

**Output:**



# Task 8

## Write a program to implement Type Conversion.

**Source Code:**

class IntermediateCodeGenerator:

    def \_\_init\_\_(self):

        self.temp\_counter = 0

        self.instructions = []

        self.label\_counter = 0

    def generate\_temp(self):

        """Generate a new temporary variable name."""

        self.temp\_counter += 1

        return f"t{self.temp\_counter - 1}"

    def generate\_label(self):

        """Generate a new label name."""

        self.label\_counter += 1

        return f"L{self.label\_counter - 1}"

    def add\_instruction(self, instruction):

        """Add an instruction to the list of instructions."""

        self.instructions.append(instruction)

    def generate\_code(self, ast):

        """Generate intermediate code from the AST."""

        self.visit(ast)

        return self.instructions

    def visit(self, node):

        """Dispatch method for visiting AST nodes."""

        if isinstance(node, BinOp):

            return self.visit\_binop(node)

        elif isinstance(node, Num):

            return node.value

        elif isinstance(node, Variable):

            return node.name

        elif isinstance(node, TypeConversion):

            return self.visit\_type\_conversion(node)

    def visit\_binop(self, node):

        """Handle binary operations."""

        left = self.visit(node.left)

        right = self.visit(node.right)

        result = self.generate\_temp()

        self.add\_instruction(f"{result} = {left} {node.op} {right}")

        return result

    def visit\_type\_conversion(self, node):

        """Handle type conversion operations."""

        operand = self.visit(node.operand)

        result = self.generate\_temp()

        self.add\_instruction(f"{result} = ({node.target\_type}) {operand}")

        return result

class ASTNode:

    """Base class for all AST nodes."""

    pass

class BinOp(ASTNode):

    def \_\_init\_\_(self, left, op, right):

        self.left = left

        self.op = op

        self.right = right

class Num(ASTNode):

    def \_\_init\_\_(self, value):

        self.value = value

class Variable(ASTNode):

    def \_\_init\_\_(self, name):

        self.name = name

class TypeConversion(ASTNode):

    def \_\_init\_\_(self, operand, target\_type):

        self.operand = operand

        self.target\_type = target\_type

# Example usage

if \_\_name\_\_ == "\_\_main\_\_":

    # Constructing AST for the expression: (2 + 6.0) \* 3

    ast = BinOp(

        left=BinOp(left=Num(2), op='+', right=TypeConversion(Num(6.0), "int")),

        op='\*',

        right=Num(3)

    )

    # Generate intermediate code

    generator = IntermediateCodeGenerator()

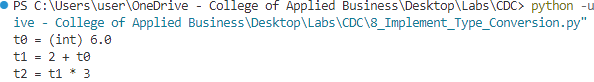
    intermediate\_code = generator.generate\_code(ast)

    # Print intermediate code

    for instruction in intermediate\_code:

        print(instruction)

**Output:**



# Task 9

## Write a program to check whether a given identifier is valid or not.

**Source Code:**

class IdentifierChecker:

    def \_\_init\_\_(self):

        # List of reserved keywords

        self.keywords = {

            "auto", "break", "case", "char", "const", "continue",

            "default", "do", "double", "else", "enum", "extern", "float",

            "for", "goto", "if", "inline", "int", "long", "register",

            "restrict", "return", "short", "signed", "sizeof", "static",

            "struct", "switch", "typedef", "union", "unsigned", "void",

            "volatile", "while", "\_Alignas", "\_Alignof", "\_Atomic", "\_Bool",

            "\_Complex", "\_Generic", "\_Imaginary", "\_Noreturn",

            "\_Static\_assert", "\_Thread\_local"

        }

    def is\_valid\_identifier(self, identifier):

        """Check if the given identifier is valid."""

        if not identifier:

            return False

        if identifier in self.keywords:

            return False

        if not (identifier[0].isalpha() or identifier[0] == '\_'):

            return False

        for char in identifier[1:]:

            if not (char.isalnum() or char == '\_'):

                return False

        return True

# Example usage

if \_\_name\_\_ == "\_\_main\_\_":

    checker = IdentifierChecker()

    identifiers = ["while", "\_validIdentifier1", "1Invalid", "invalididentifier", "int", "validIdentifier"]

    for identifier in identifiers:

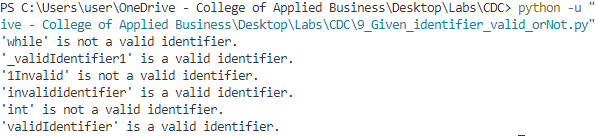
        if checker.is\_valid\_identifier(identifier):

            print(f"'{identifier}' is a valid identifier.")

        else:

            print(f"'{identifier}' is not a valid identifier.")

**Output:**



# Task 10

## Write a program to check whether a given string is within valid comment section or not.

**Source Code:**

class CommentChecker:

    def \_\_init\_\_(self):

        pass

    def is\_within\_comment(self, code, string):

        lines = code.split('\n')

        in\_multiline\_comment = False

        for line in lines:

            stripped\_line = line.strip()

            # Check for single-line comment

            if '//' in stripped\_line:

                comment\_index = stripped\_line.index('//')

                comment\_part = stripped\_line[comment\_index:]

                if string in comment\_part:

                    return True

            # Check for start of multi-line comment

            if '/\*' in stripped\_line:

                in\_multiline\_comment = True

                comment\_index = stripped\_line.index('/\*')

                comment\_part = stripped\_line[comment\_index:]

                if string in comment\_part:

                    return True

            # Check for end of multi-line comment

            if '\*/' in stripped\_line:

                if in\_multiline\_comment:

                    comment\_index = stripped\_line.index('\*/')

                    comment\_part = stripped\_line[:comment\_index + 2]

                    if string in comment\_part:

                        return True

                    in\_multiline\_comment = False

                    continue

            # Check for string within multi-line comment

            if in\_multiline\_comment:

                if string in stripped\_line:

                    return True

        return False

# Example usage

if \_\_name\_\_ == "\_\_main\_\_":

    code = '''

    // This is a single-line comment

    int main() {

        printf("Hello, world!");  // This is an inline comment

        return 0;

    }

    '''

    multi\_line\_code = '''

    /\*

    This is a multi-line comment

    spanning multiple lines

    \*/

    int main() {

        printf("Hello, world!");

        return 0;

    }

    '''

    checker = CommentChecker()

    string1 = "single-line comment"

    string2 = "multi-line comment"

    string3 = "not in comment"

    print(checker.is\_within\_comment(code, string1))   # Expected: True

    print(checker.is\_within\_comment(multi\_line\_code, string2))  # Expected: True

    print(checker.is\_within\_comment(code, string3))  # Expected: False

**Output:**

