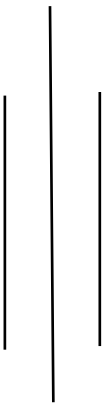
COLLEGE OF APPLIED BUSINESS AND TECHNOLOGY

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

INDEX

Experiment 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:

```
PS C:\Users\user\OneDrive - College of Applied Business\Desktop\Labs\CDC> pythc
ive - College of Applied Business\Desktop\Labs\CDC\tempCodeRunnerFile.py"
Enter the lexeme: 21
Token(NUMBER, 21, 0)
Token(EOF, , 2)
```

Experiment 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:

```
PS C:\Users\user\OneDrive - College of Applied Business\Desktop\Labs\CDC> python -u "c
ive - College of Applied Business\Desktop\Labs\CDC\2_Implement_first_of_Grammar.py"
FIRST(S) = { ε, a, b }
FIRST(A) = { ε, a }
FIRST(B) = { ε, b }
```

Experiment 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:

```
PS C:\Users\user\OneDrive - College of Applied Business\Desktop\Labs\CDC> python -u "
ive - College of Applied Business\Desktop\Labs\CDC\3_implement_follow_of_grammar.py"
Stack: [], Input: ['id', '+', 'id', '$'], Action: None
Stack: ['id'], Input: ['+', 'id', '$'], Action: Shift
Stack: ['E'], Input: ['+', 'id', '$'], Action: Reduce by E -> id
Stack: ['E', '+'], Input: ['id', '$'], Action: Shift
Stack: ['E', '+', 'id'], Input: ['$'], Action: Shift
Stack: ['E', '+', 'E'], Input: ['$'], Action: Reduce by E -> id
Stack: ['E'], Input: ['$'], Action: Accepted
```

Experiment 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:

```
PS C:\Users\user\OneDrive - College of Applied Business\Desktop\Labs\CDC> pyth
ive - College of Applied Business\Desktop\Labs\CDC\4_Shift_Reduce_parser.py"
FOLLOW(S) = { $ }
FOLLOW(A) = { b, $ }
FOLLOW(B) = { $ }
```

Experiment 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:

```
PS C:\Users\user\OneDrive - College of Applied Business\Desktop\Labs\CDC> py
ive - College of Applied Business\Desktop\Labs\CDC\5_Implement_LR_parser.py"
Error
Parsing failed.
```

Experiment 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
```

```python
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:

```
PS C:\Users\user\OneDrive - College of Applied Business\Desktop\Labs\CDC> py
ive - College of Applied Business\Desktop\Labs\CDC\6_implement_Intermediate_
t0 = 2 + 6
t1 = t0 * 3
```

Experiment 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:

```
PS C:\Users\user\OneDrive - College of Applied Business\Desktop\Labs\CDC> python -u "c:\
ive - College of Applied Business\Desktop\Labs\CDC\7_Implement_Final_Code_Generation.py"
LOADI 2, R0
LOADI 6, R3
ADD R0, R3
STORE R0, t0
LOAD t0, R1
LOADI 3, R3
MUL R1, R3
STORE R1, t1
```

Experiment 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:

```
PS C:\Users\user\OneDrive - College of Applied Business\Desktop\Labs\CDC> python -u
ive - College of Applied Business\Desktop\Labs\CDC\8_Implement_Type_Conversion.py"
t0 = (int) 6.0
t1 = 2 + t0
t2 = t1 * 3
```

Experiment 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:

```
PS C:\Users\user\OneDrive - College of Applied Business\Desktop\Labs\CDC> python -u "
ive - College of Applied Business\Desktop\Labs\CDC\9_Given_identifier_valid_orNot.py"
'while' is not a valid identifier.
'_validIdentifier1' is a valid identifier.
'1Invalid' is not a valid identifier.
'invalididentifier' is a valid identifier.
'int' is not a valid identifier.
'validIdentifier' is a valid identifier.
```

Experiment 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:

```
PS C:\Users\user\OneDrive - College of Applied Business\Desktop\Labs\CDC> python
ive - College of Applied Business\Desktop\Labs\CDC\10_String_is_within_valid_comm
True
True
False
```