Compiler-Pr3-99442326-99442029-99442011

July 28, 2024

1 Compiler Project

Development of a simple compiler for C++ code, executed in three distinct phases: Lexical Analysis, Top-Down Parsing, and Three-Address Code Generation.

1.0.1 Compiler Project Documentation

Introduction This project involved the development of a simple compiler for C++ code, executed in three distinct phases: Lexical Analysis, Top-Down Parsing, and Three-Address Code Generation. Each phase built upon the previous one, refining and expanding the compiler's capabilities. By the end of the project, a functional compiler capable of generating intermediate three-address code from simple C++ source code was achieved.

Phase 1: Lexical Analyzer In the first phase, a lexical analyzer was developed. The main tasks completed during this phase were:

Tokenization The lexical analyzer scanned the input C++ source code and converted it into a sequence of tokens. Tokens include keywords, identifiers and operators.

Regular Expressions Regular expressions were used to define the patterns for different tokens.

Phase 2: Top-Down Parser The second phase focused on developing a top-down parser, specifically a recursive descent parser. Key tasks accomplished included:

Grammar Definition The context-free grammar (CFG) for a subset of the C++ language was defined. This grammar was designed to be LL(1) to facilitate top-down parsing.

Parser Implementation Using the CFG, a recursive descent parser was implemented. The parser takes the token stream produced by the lexical analyzer and constructs a parse tree.

Phase 3: Three-Address Code Generator In the final phase, the compiler was extended to generate three-address code (TAC), an intermediate representation of the source code. Key developments in this phase were:

Grammar Refinement The grammar defined in the second phase was modified to better suit the needs of code generation.

Parse Tree to TAC Conversion Algorithms were developed to traverse the parse tree and generate corresponding three-address code.

Optimization Basic optimizations were implemented to produce more efficient intermediate code.

Conclusion This project culminated in a fully functional simple C++ compiler capable of performing lexical analysis, parsing, and generating three-address code. The iterative development approach ensured that each phase built upon the previous one, leading to a coherent and integrated compiler.

[1]:
```python
import re
from typing import List, Tuple
```

1.0.2 Phase 1: Lexical Analyzer

The first phase of the compiler project involves developing a lexical analyzer. The primary tasks in this phase are tokenization, defining regular expressions for various token types.

Token Types Definition The `TOKEN_SPECIFICATION` defines various token types and their corresponding regular expression patterns. Key token types include:

- **INCLUDE**: Matches preprocessor include directives (e.g., `#include <iostream>`).
- **USING**: Matches using namespace directives (e.g., `using namespace std;`).
- **STD_CIN**: Matches standard input streams (e.g., `std::cin` or `cin`).
- **STD_COUT**: Matches standard output streams (e.g., `std::cout` or `cout`).
- **TYPE**: Matches basic data types (e.g., `int`, `void`, `double`, `float`).
- **NUMBER**: Matches integer numbers.
- **WHILE**: Matches the `while` keyword.
- **IF**: Matches the `if` keyword.
- **ELSE**: Matches the `else` keyword.
- **RETURN**: Matches the `return` keyword.
- **IDENTIFIER**: Matches variable names and function names (e.g., `main`, `variableName`).
- **OP**: Matches operators (e.g., `+`, `-`, `*`, `/`, `==`, `!=`, `<=`, `>=`, `<`, `>`).
- **ASSIGN**: Matches the assignment operator (`=`).
- **SEMICOLON**: Matches the semicolon (`;`).
- **LPAREN**: Matches the left parenthesis (`(`).
- **RPAREN**: Matches the right parenthesis (`)`).
- **LBRACE**: Matches the left brace (`{`).
- **RBRACE**: Matches the right brace (`}`).
- **STRING_LITERAL**: Matches string literals (e.g., `"hello"`).
- **NEWLINE**: Matches newline characters.
- **SKIP**: Matches spaces and tabs, which are ignored.
- **MISMATCH**: Matches any other character not defined in the previous tokens, used for error handling.

Token Regex Combination The token patterns are combined into a single regular expression using named groups. This combined regex is used to match and identify tokens in the input code.

Lexical Analysis Function The `lex` function performs lexical analysis by scanning the input C++ code and producing a list of tokens. It uses regular expression matching to identify token types and their values. Special cases such as newlines and spaces are ignored, while any unexpected characters raise an error.

Conclusion This phase lays the groundwork for subsequent phases by converting the input C++ code into a sequence of tokens. These tokens represent meaningful components of the code, facilitating further parsing and code generation in later stages.

[2]:
```
# Define token types
TOKEN_SPECIFICATION = [
    ('INCLUDE',       r'#include <[a-zA-Z0-9_]+>'),
    ('USING',         r'using namespace [a-zA-Z0-9_]+;'),
    ('STD_CIN',       r'std::cin|cin'),  # Added combined token for cin
    ('STD_COUT',      r'std::cout|cout'),  # Added combined token for cout
    ('TYPE',          r'\bint\b|\bvoid\b|\bdouble\b|\bfloat\b'),
    ('NUMBER',        r'\b\d+\b'),
    ('WHILE',         r'\bwhile\b'),
    ('IF',            r'\bif\b'),
    ('ELSE',          r'\belse\b'),
    ('RETURN',        r'\breturn\b'),
    ('IDENTIFIER',    r'\b[a-zA-Z_][a-zA-Z0-9_]*\b'),
    ('OP',            r'<<|>>|\+|\-|\*|\/|==|!=|<=|>=|<|>'),
    ('ASSIGN',        r'='),
    ('SEMICOLON',     r';'),
    ('LPAREN',        r'\('),
    ('RPAREN',        r'\)'),
    ('LBRACE',        r'\{'),
    ('RBRACE',        r'\}'),
    ('STRING_LITERAL', r'"[^"]*"'),
    ('NEWLINE',       r'\n'),
    ('SKIP',          r'[ \t]+'),
    ('MISMATCH',      r'.'),  # Any other character
]

token_regex = '|'.join(f'(?P<{pair[0]}>{pair[1]})' for pair in␣
 ↪TOKEN_SPECIFICATION)

def lex(code: str) -> List[Tuple[str, str]]:
    tokens = []
    for mo in re.finditer(token_regex, code):
        kind = mo.lastgroup
        value = mo.group()
        if kind == 'NEWLINE':
            continue
        elif kind == 'SKIP':
            continue
        elif kind == 'MISMATCH':
            raise RuntimeError(f'{value!r} unexpected')
        else:
            tokens.append((kind, value))
            print(f'Token: {kind}, Value: {value}')  # Debug statement
    return tokens
```

1.0.3 Phase 2: Top-Down Parser

The second phase of the compiler project involves developing a top-down parser using a recursive descent approach. This parser takes the token stream produced by the lexical analyzer and constructs a parse tree representing the structure of the program.

Parser Class The `Parser` class is responsible for parsing the token stream. It maintains the current position in the token list and provides methods to match and process various language constructs.

Key Methods and Their Functions

- **Initialization (`__init__`)**: Initializes the parser with a list of tokens and sets the initial position to zero.
- **Parsing Entry Point (`parse`)**: Begins the parsing process by invoking the `program` method.
- **Token Matching (`match`)**: Ensures that the current token matches the expected type and advances the position. Raises a `SyntaxError` if the token does not match.

Parsing Program Structure

- **Program (`program`)**: Parses the entire program, including include directives, namespace declaration, and function definitions.
- **Include Directives (`include_directives`)**: Collects all `#include` directives at the beginning of the program.
- **Namespace Declaration (`namespace_declaration`)**: Parses the `using namespace` directive, if present.
- **Function Definitions (`function_definitions`)**: Parses one or more function definitions in the program.

Parsing Function and Statements

- **Function Definition (`function_definition`)**: Parses a function's return type, name, parameters, and body.
- **Parameters (`parameters`)**: Parses function parameters, handling multiple parameters separated by commas.
- **Statements (`statements`)**: Parses a block of statements within a function or control structure.

- **Individual Statement (`statement`)**: Determines the type of statement (variable declaration, assignment, control structures, etc.) and delegates parsing to the appropriate method.

Handling Specific Statements

- **Variable Declaration (`variable_declaration`)**: Parses variable declarations, including optional initialization.
- **Assignment (`assignment`)**: Parses assignment statements.
- **If Statement (`if_statement`)**: Parses `if` statements with optional `else` branches.
- **While Statement (`while_statement`)**: Parses `while` loops.
- **Return Statement (`return_statement`)**: Parses `return` statements.
- **Output Statement (`output_statement`)**: Parses `std::cout` statements.
- **Input Statement (`input_statement`)**: Parses `std::cin` statements.

Parsing Expressions

- **Expression (`expression`)**: Parses expressions, handling binary operators.
- **Term (`term`)**: Parses terms within expressions, handling multiplication and division.
- **Factor (`factor`)**: Parses factors, which can be numbers, string literals, identifiers, or parenthesized expressions. Handles function calls if an identifier is followed by parentheses.

Conclusion The parser is a crucial component of the compiler, transforming a linear sequence of tokens into a hierarchical structure that reflects the syntactic organization of the source code. This structured representation is essential for subsequent phases, such as semantic analysis and code generation.

[3]:
```
class Parser:
    def __init__(self, tokens: List[Tuple[str, str]]):
        self.tokens = tokens
        self.pos = 0

    def parse(self):
        return self.program()

    def match(self, expected_type):
        if self.pos < len(self.tokens) and self.tokens[self.pos][0] ==␣
↪expected_type:
            print(f'Matched {expected_type} at position {self.pos}, token:␣
↪{self.tokens[self.pos]}')  # Debug statement
            self.pos += 1
        else:
            raise SyntaxError(f'Expected {expected_type} but found {self.
↪tokens[self.pos][0]} at position {self.pos}')
```

```
    def program(self):
        includes = self.include_directives()
        namespace = self.namespace_declaration()
        functions = self.function_definitions()
        return {'type': 'program', 'includes': includes, 'namespace':␣
↪namespace, 'functions': functions}

    def include_directives(self):
        includes = []
        while self.pos < len(self.tokens) and self.tokens[self.pos][0] ==␣
↪'INCLUDE':
            includes.append(self.tokens[self.pos][1])
            self.pos += 1
        return includes

    def namespace_declaration(self):
        if self.pos < len(self.tokens) and self.tokens[self.pos][0] == 'USING':
            namespace = self.tokens[self.pos][1]
            self.pos += 1
            return namespace
        return None

    def function_definitions(self):
        functions = []
        while self.pos < len(self.tokens):
            functions.append(self.function_definition())
        return functions

    def function_definition(self):
        return_type = self.tokens[self.pos][1]
        self.pos += 1
        name = self.tokens[self.pos][1]
        self.pos += 1
        self.match('LPAREN')
        params = self.parameters()
        self.match('RPAREN')
        self.match('LBRACE')
        body = self.statements()
        self.match('RBRACE')
        return {'type': 'function', 'return_type': return_type, 'name': name,␣
↪'params': params, 'body': body}

    def parameters(self):
        params = []
        while self.pos < len(self.tokens) and self.tokens[self.pos][0] !=␣
↪'RPAREN':
            param_type = self.tokens[self.pos][1]
            self.pos += 1
            param_name = self.tokens[self.pos][1]
            self.pos += 1
            params.append((param_type, param_name))
            if self.pos < len(self.tokens) and self.tokens[self.pos][0] ==␣
↪'RPAREN':
                break
        return params

    def statements(self):
        statements = []
        while self.pos < len(self.tokens) and self.tokens[self.pos][0] !=␣
↪'RBRACE':
            statements.append(self.statement())
        return statements

    def statement(self):
        if self.tokens[self.pos][0] == 'TYPE':
            return self.variable_declaration()
        elif self.tokens[self.pos][0] == 'IDENTIFIER' and self.tokens[self.pos␣
↪+ 1][0] == 'ASSIGN':
            return self.assignment()
        elif self.tokens[self.pos][0] == 'IF':
            return self.if_statement()
        elif self.tokens[self.pos][0] == 'WHILE':
            print('into WHILE statement')
            return self.while_statement()
        elif self.tokens[self.pos][0] == 'RETURN':
            return self.return_statement()
        elif self.tokens[self.pos][0] == 'STD_COUT':
            return self.output_statement()
            print('into CIN statement ...')
        elif self.tokens[self.pos][0] == 'STD_CIN':
            return self.input_statement()
        else:
            expr = self.expression()
            if self.pos < len(self.tokens) and self.tokens[self.pos][0] ==␣
↪'SEMICOLON':
                self.match('SEMICOLON')
            return expr

    def variable_declaration(self):
        var_type = self.tokens[self.pos][1]
        self.pos += 1
        var_name = self.tokens[self.pos][1]
        self.pos += 1
        if self.pos < len(self.tokens) and self.tokens[self.pos][0] == 'ASSIGN':
            self.pos += 1
            value = self.expression()
            self.match('SEMICOLON')
            return {'type': 'var_declaration', 'var_type': var_type, 'var_name':
↪ var_name, 'value': value}
        self.match('SEMICOLON')
        return {'type': 'var_declaration', 'var_type': var_type, 'var_name':␣
↪var_name}

    def assignment(self):
        var_name = self.tokens[self.pos][1]
        self.pos += 1
        self.match('ASSIGN')
        value = self.expression()
        self.match('SEMICOLON')
        return {'type': 'assignment', 'var_name': var_name, 'value': value}

    def if_statement(self):
        self.match('IF')
        self.match('LPAREN')
        condition = self.expression()
        self.match('RPAREN')
        self.match('LBRACE')
        then_branch = self.statements()
        self.match('RBRACE')
        else_branch = None
        if self.pos < len(self.tokens) and self.tokens[self.pos][0] == 'ELSE':
            self.pos += 1
            self.match('LBRACE')
            else_branch = self.statements()
            self.match('RBRACE')
        return {'type': 'if', 'condition': condition, 'then': then_branch,␣
↪'else': else_branch}

    def while_statement(self):
        self.match('WHILE')
        self.match('LPAREN')
        condition = self.expression()
        self.match('RPAREN')
        self.match('LBRACE')
        body = self.statements()
        self.match('RBRACE')
        return {'type': 'while', 'condition': condition, 'body': body}

    def return_statement(self):
        self.match('RETURN')
        value = self.expression()
        self.match('SEMICOLON')
        return {'type': 'return', 'value': value}

    def output_statement(self):
        self.match('STD_COUT')
        self.match('OP')  # Match '<<'
        expressions = []
        expressions.append(self.expression())
        while self.pos < len(self.tokens) and self.tokens[self.pos][0] == 'OP'␣
↪and self.tokens[self.pos][1] == '<<':
            self.pos += 1
            expressions.append(self.expression())
        self.match('SEMICOLON')
        return {'type': 'output', 'expressions': expressions}

    def input_statement(self):
        self.match('STD_CIN')
        print('CIN Matched ...')
        self.match('OP')  # Match '>>'
        variables = []
        variables.append(self.expression())
        while self.pos < len(self.tokens) and self.tokens[self.pos][0] == 'OP'␣
↪and self.tokens[self.pos][1] == '>>':
            self.pos += 1
            variables.append(self.expression())
        self.match('SEMICOLON')
        return {'type': 'input', 'variables': variables}

    def expression(self):
        expr = self.term()
        while self.pos < len(self.tokens) and self.tokens[self.pos][0] == 'OP'␣
↪and self.tokens[self.pos][1] in ('+', '-', '>=', '<=', '==', '!=', '>', '<'):
            op = self.tokens[self.pos][1]
            self.pos += 1
            right = self.term()
            expr = {'type': 'binary_op', 'operator': op, 'left': expr, 'right':␣
↪right}
        return expr

    def term(self):
        expr = self.factor()
        while self.pos < len(self.tokens) and self.tokens[self.pos][0] == 'OP'␣
↪and self.tokens[self.pos][1] in ('*', '/'):
            op = self.tokens[self.pos][1]
            self.pos += 1
            right = self.factor()
            expr = {'type': 'binary_op', 'operator': op, 'left': expr, 'right':␣
↪right}
        return expr

    def factor(self):
        if self.tokens[self.pos][0] == 'LPAREN':
            self.match('LPAREN')
            expr = self.expression()
            self.match('RPAREN')
            return expr
        elif self.tokens[self.pos][0] == 'NUMBER':
            self.pos += 1
            return {'type': 'number', 'value': self.tokens[self.pos - 1][1]}
        elif self.tokens[self.pos][0] == 'STRING_LITERAL':
            self.pos += 1
            return {'type': 'string', 'value': self.tokens[self.pos - 1][1]}
        elif self.tokens[self.pos][0] == 'IDENTIFIER':
            identifier = self.tokens[self.pos][1]
            self.pos += 1
            if self.pos < len(self.tokens) and self.tokens[self.pos][0] ==␣
↪'LPAREN':
                self.match('LPAREN')
                args = []
                if self.tokens[self.pos][0] != 'RPAREN':
                    args.append(self.expression())
                    while self.tokens[self.pos][0] == 'COMMA':
                        self.match('COMMA')
                        args.append(self.expression())
                self.match('RPAREN')
                return {'type': 'function_call', 'name': identifier, 'args':␣
↪args}
            return {'type': 'identifier', 'value': identifier}
        else:
            raise SyntaxError(f'Unexpected token: {self.tokens[self.pos][0]} at␣
↪position {self.pos}')
```

1.0.4 Phase 3: Three-Address Code (TAC) Generator

The final phase of the compiler project involves generating Three-Address Code (TAC) from the Abstract Syntax Tree (AST) produced by the parser. This intermediate representation is crucial for optimization and translation into machine code.

TAC Generation Function The `generate_tac` function converts the AST into TAC instructions. It traverses the AST recursively, generating appropriate TAC instructions for each node type.

Key Components

- **Temporary Variables (`new_temp`)**: Generates unique temporary variable names for intermediate results.

Generating TAC for Various Nodes

- **Program (`program`)**: Processes each function in the program.
- **Function (`function`)**: Adds a label for the function name and generates TAC for each statement in the function body.
- **Variable Declaration (`var_declaration`)**: Generates TAC for variable initialization.
- **Assignment (`assignment`)**: Generates TAC for assignment statements.
- **Binary Operation (`binary_op`)**: Generates TAC for binary operations, using temporary variables for intermediate results.
- **Number (`number`)**: Returns the numeric value.
- **Identifier (`identifier`)**: Returns the variable name.
- **Function Call (`function_call`)**: Generates TAC for function calls, including arguments and return values.
- **While Loop (`while`)**: Generates TAC for while loops, including conditional and jump instructions.
- **If Statement (`if`)**: Generates TAC for if statements, including conditional and jump instructions.
- **Return Statement (`return`)**: Generates TAC for return statements.
- **Output Statement (`output`)**: Generates TAC for output statements (`std::cout`).
- **Input Statement (`input`)**: Generates TAC for input statements (`std::cin`).

Integration with Parsing The `parse_cpp_with_tac` function integrates lexical analysis, parsing, and TAC generation. It processes the input code to produce tokens, constructs the AST, and generates TAC. The resulting TAC is saved to a specified output file.

Conclusion The TAC generator transforms the high-level AST into a lower-level intermediate representation, bridging the gap between the source code and machine code. This phase is essential for enabling further optimizations and efficient code generation.

[4]:
```python
# TAC Generation
def generate_tac(ast):
    tac = []
    temp_count = 0  # Move this line outside of the generate function

    def new_temp():
        nonlocal temp_count
        temp = f't{temp_count}'
        temp_count += 1
        return temp

    def generate(node):
        nonlocal temp_count  # Add this line to ensure temp_count is accessible
        if node['type'] == 'program':
            for func in node['functions']:
                generate(func)
        elif node['type'] == 'function':
            tac.append((node['name'], ':'))
            for stmt in node['body']:
                generate(stmt)
        elif node['type'] == 'var_declaration':
            if 'value' in node:
                value = generate(node['value'])
                tac.append((node['var_name'], '=', value))
        elif node['type'] == 'assignment':
            value = generate(node['value'])
            tac.append((node['var_name'], '=', value))
        elif node['type'] == 'binary_op':
            left = generate(node['left'])
            right = generate(node['right'])
            result = new_temp()
            tac.append((result, '=', left, node['operator'], right))
            return result
        elif node['type'] == 'number':
            return node['value']
        elif node['type'] == 'identifier':
            return node['value']
        elif node['type'] == 'function_call':
            args = [generate(arg) for arg in node['args']]
            result = new_temp()
            tac.append((result, '=', node['name'], '(', ', '.join(args), ')'))
            return result
        elif node['type'] == 'while':
            start_label = f"L{temp_count}"
            temp_count += 1
            end_label = f"L{temp_count}"
            temp_count += 1
            tac.append((start_label, ':'))
            cond = generate(node['condition'])
            tac.append(('if', cond, '==', '0', 'goto', end_label))
            for stmt in node['body']:
                generate(stmt)
            tac.append(('goto', start_label))
            tac.append((end_label, ':'))
        elif node['type'] == 'if':
            else_label = f"L{temp_count}"
            temp_count += 1
            end_label = f"L{temp_count}"
            temp_count += 1
            cond = generate(node['condition'])
            tac.append(('if', cond, '==', '0', 'goto', else_label))
            for stmt in node['then']:
                generate(stmt)
            tac.append(('goto', end_label))
            tac.append((else_label, ':'))
            if 'else' in node:
                for stmt in node['else']:
                    generate(stmt)
            tac.append((end_label, ':'))
        elif node['type'] == 'return':
            value = generate(node['value'])
            tac.append(('return', value))
        elif node['type'] == 'output':
            for expr in node['expressions']:
                value = generate(expr)
                tac.append(('cout', value))
        elif node['type'] == 'input':
            for var in node['variables']:
                tac.append((var['value'], '=', 'cin'))

    generate(ast)
    return tac

# Modified parse_cpp function to include TAC generation
def parse_cpp_with_tac(code: str,output_filename: str):
    print('======Tokens====================================')
    tokens = lex(code)
    print('======Parser====================================')
    parser = Parser(tokens)
    ast = parser.parse()
    tac = generate_tac(ast)

    # Save TAC to a file
    with open(output_filename, 'w') as f:
        for line in tac:
            f.write(' '.join(map(str, line)) + '\n')

    return tac
```

1.0.5 Example codes to test

[5]:
```
# Example codes to test
input_code1 = """
#include <iostream>

using namespace std;

int main() {
    int x;
    int sum = 0;
    int t = 10;

    while (t >= 0) {
        cin >> x;
        t = t - 1;
        sum = sum + x;
    }

    cout << "Sum = " << sum;

    return 0;
}
"""

# Generate TAC for the input codes
tac1 = parse_cpp_with_tac(input_code1,'TAC1')
print('======TAC1====================================')
print(tac1)
```

```
======Tokens====================================
Token: INCLUDE, Value: #include <iostream>
Token: USING, Value: using namespace std;
Token: TYPE, Value: int
Token: IDENTIFIER, Value: main
Token: LPAREN, Value: (
Token: RPAREN, Value: )
Token: LBRACE, Value: {
Token: TYPE, Value: int
Token: IDENTIFIER, Value: x
Token: SEMICOLON, Value: ;
Token: TYPE, Value: int
Token: IDENTIFIER, Value: sum
Token: ASSIGN, Value: =
Token: NUMBER, Value: 0
Token: SEMICOLON, Value: ;
Token: TYPE, Value: int
Token: IDENTIFIER, Value: t
Token: ASSIGN, Value: =
Token: NUMBER, Value: 10
Token: SEMICOLON, Value: ;
Token: WHILE, Value: while
Token: LPAREN, Value: (
Token: IDENTIFIER, Value: t
Token: OP, Value: >=
Token: NUMBER, Value: 0
Token: RPAREN, Value: )
Token: LBRACE, Value: {
```

```
Token: STD_CIN, Value: cin
Token: OP, Value: >>
Token: IDENTIFIER, Value: x
Token: SEMICOLON, Value: ;
Token: IDENTIFIER, Value: t
Token: ASSIGN, Value: =
Token: IDENTIFIER, Value: t
Token: OP, Value: -
Token: NUMBER, Value: 1
Token: SEMICOLON, Value: ;
Token: IDENTIFIER, Value: sum
Token: ASSIGN, Value: =
Token: IDENTIFIER, Value: sum
Token: OP, Value: +
Token: IDENTIFIER, Value: x
Token: SEMICOLON, Value: ;
Token: RBRACE, Value: }
Token: STD_COUT, Value: cout
Token: OP, Value: <<
Token: STRING_LITERAL, Value: "Sum = "
Token: OP, Value: <<
Token: IDENTIFIER, Value: sum
Token: SEMICOLON, Value: ;
Token: RETURN, Value: return
Token: NUMBER, Value: 0
Token: SEMICOLON, Value: ;
Token: RBRACE, Value: }
======Parser===================================
Matched LPAREN at position 4, token: ('LPAREN', '(')
Matched RPAREN at position 5, token: ('RPAREN', ')')
Matched LBRACE at position 6, token: ('LBRACE', '{')
Matched SEMICOLON at position 9, token: ('SEMICOLON', ';')
Matched SEMICOLON at position 14, token: ('SEMICOLON', ';')
Matched SEMICOLON at position 19, token: ('SEMICOLON', ';')
into WHILE statement
Matched WHILE at position 20, token: ('WHILE', 'while')
Matched LPAREN at position 21, token: ('LPAREN', '(')
Matched RPAREN at position 25, token: ('RPAREN', ')')
Matched LBRACE at position 26, token: ('LBRACE', '{')
Matched STD_CIN at position 27, token: ('STD_CIN', 'cin')
CIN Matched …
Matched OP at position 28, token: ('OP', '>>')
Matched SEMICOLON at position 30, token: ('SEMICOLON', ';')
Matched ASSIGN at position 32, token: ('ASSIGN', '=')
Matched SEMICOLON at position 36, token: ('SEMICOLON', ';')
Matched ASSIGN at position 38, token: ('ASSIGN', '=')
Matched SEMICOLON at position 42, token: ('SEMICOLON', ';')
Matched RBRACE at position 43, token: ('RBRACE', '}')
Matched STD_COUT at position 44, token: ('STD_COUT', 'cout')
Matched OP at position 45, token: ('OP', '<<')
Matched SEMICOLON at position 49, token: ('SEMICOLON', ';')
Matched RETURN at position 50, token: ('RETURN', 'return')
Matched SEMICOLON at position 52, token: ('SEMICOLON', ';')
Matched RBRACE at position 53, token: ('RBRACE', '}')
======TAC1====================================
[('main', ':'), ('sum', '=', '0'), ('t', '=', '10'), ('L0', ':'), ('t2', '=',
't', '>=', '0'), ('if', 't2', '==', '0', 'goto', 'L1'), ('x', '=', 'cin'),
('t3', '=', 't', '-', '1'), ('t', '=', 't3'), ('t4', '=', 'sum', '+', 'x'),
('sum', '=', 't4'), ('goto', 'L0'), ('L1', ':'), ('cout', None), ('cout',
'sum'), ('return', '0')]
```

[6]:
```
input_code2 = """
#include <iostream>
using namespace std;

int factorial(int n) {
    if (n == 0){
        return 1;
    } else {
        return n * factorial(n - 1);
    }
}

int main() {
    int n;
    cout << "Enter a number: ";
    cin >> n;
    cout << "Factorial of " << n << " is " << factorial(n);
    return 0;
}
"""

# Generate TAC for the input codes
tac2 = parse_cpp_with_tac(input_code2,'TAC2')
print('======TAC2====================================')
print(tac2)
```

```
======Tokens====================================
Token: INCLUDE, Value: #include <iostream>
Token: USING, Value: using namespace std;
Token: TYPE, Value: int
Token: IDENTIFIER, Value: factorial
Token: LPAREN, Value: (
Token: TYPE, Value: int
Token: IDENTIFIER, Value: n
Token: RPAREN, Value: )
```

```
Token: LBRACE, Value: {
Token: IF, Value: if
Token: LPAREN, Value: (
Token: IDENTIFIER, Value: n
Token: OP, Value: ==
Token: NUMBER, Value: 0
Token: RPAREN, Value: )
Token: LBRACE, Value: {
Token: RETURN, Value: return
Token: NUMBER, Value: 1
Token: SEMICOLON, Value: ;
Token: RBRACE, Value: }
Token: ELSE, Value: else
Token: LBRACE, Value: {
Token: RETURN, Value: return
Token: IDENTIFIER, Value: n
Token: OP, Value: *
Token: IDENTIFIER, Value: factorial
Token: LPAREN, Value: (
Token: IDENTIFIER, Value: n
Token: OP, Value: -
Token: NUMBER, Value: 1
Token: RPAREN, Value: )
Token: SEMICOLON, Value: ;
Token: RBRACE, Value: }
Token: RBRACE, Value: }
Token: TYPE, Value: int
Token: IDENTIFIER, Value: main
Token: LPAREN, Value: (
Token: RPAREN, Value: )
Token: LBRACE, Value: {
Token: TYPE, Value: int
Token: IDENTIFIER, Value: n
Token: SEMICOLON, Value: ;
Token: STD_COUT, Value: cout
Token: OP, Value: <<
Token: STRING_LITERAL, Value: "Enter a number: "
Token: SEMICOLON, Value: ;
Token: STD_CIN, Value: cin
Token: OP, Value: >>
Token: IDENTIFIER, Value: n
Token: SEMICOLON, Value: ;
Token: STD_COUT, Value: cout
Token: OP, Value: <<
Token: STRING_LITERAL, Value: "Factorial of "
Token: OP, Value: <<
Token: IDENTIFIER, Value: n
Token: OP, Value: <<
Token: STRING_LITERAL, Value: " is "
Token: OP, Value: <<
Token: IDENTIFIER, Value: factorial
Token: LPAREN, Value: (
Token: IDENTIFIER, Value: n
Token: RPAREN, Value: )
Token: SEMICOLON, Value: ;
Token: RETURN, Value: return
Token: NUMBER, Value: 0
Token: SEMICOLON, Value: ;
Token: RBRACE, Value: }
======Parser=====================================
Matched LPAREN at position 4, token: ('LPAREN', '(')
Matched RPAREN at position 7, token: ('RPAREN', ')')
Matched LBRACE at position 8, token: ('LBRACE', '{')
Matched IF at position 9, token: ('IF', 'if')
Matched LPAREN at position 10, token: ('LPAREN', '(')
Matched RPAREN at position 14, token: ('RPAREN', ')')
Matched LBRACE at position 15, token: ('LBRACE', '{')
Matched RETURN at position 16, token: ('RETURN', 'return')
Matched SEMICOLON at position 18, token: ('SEMICOLON', ';')
Matched RBRACE at position 19, token: ('RBRACE', '}')
Matched LBRACE at position 21, token: ('LBRACE', '{')
Matched RETURN at position 22, token: ('RETURN', 'return')
Matched LPAREN at position 26, token: ('LPAREN', '(')
Matched RPAREN at position 30, token: ('RPAREN', ')')
Matched SEMICOLON at position 31, token: ('SEMICOLON', ';')
Matched RBRACE at position 32, token: ('RBRACE', '}')
Matched RBRACE at position 33, token: ('RBRACE', '}')
Matched LPAREN at position 36, token: ('LPAREN', '(')
Matched RPAREN at position 37, token: ('RPAREN', ')')
Matched LBRACE at position 38, token: ('LBRACE', '{')
Matched SEMICOLON at position 41, token: ('SEMICOLON', ';')
Matched STD_COUT at position 42, token: ('STD_COUT', 'cout')
Matched OP at position 43, token: ('OP', '<<')
Matched SEMICOLON at position 45, token: ('SEMICOLON', ';')
Matched STD_CIN at position 46, token: ('STD_CIN', 'cin')
CIN Matched …
Matched OP at position 47, token: ('OP', '>>')
Matched SEMICOLON at position 49, token: ('SEMICOLON', ';')
Matched STD_COUT at position 50, token: ('STD_COUT', 'cout')
Matched OP at position 51, token: ('OP', '<<')
Matched LPAREN at position 59, token: ('LPAREN', '(')
Matched RPAREN at position 61, token: ('RPAREN', ')')
Matched SEMICOLON at position 62, token: ('SEMICOLON', ';')
Matched RETURN at position 63, token: ('RETURN', 'return')
Matched SEMICOLON at position 65, token: ('SEMICOLON', ';')
Matched RBRACE at position 66, token: ('RBRACE', '}')
```

```
======TAC2====================================
[('factorial', ':'), ('t2', '=', 'n', '==', '0'), ('if', 't2', '==', '0',
'goto', 'L0'), ('return', '1'), ('goto', 'L1'), ('L0', ':'), ('t3', '=', 'n',
'-', '1'), ('t4', '=', 'factorial', '(', 't3', ')'), ('t5', '=', 'n', '*',
't4'), ('return', 't5'), ('L1', ':'), ('main', ':'), ('cout', None), ('n', '=',
'cin'), ('cout', None), ('cout', 'n'), ('cout', None), ('t6', '=', 'factorial',
'(', 'n', ')'), ('cout', 't6'), ('return', '0')]
```