SVKM's NMIMS

Mukesh Patel School of Technology Management & Engineering (Mumbai Campus)

Computer Engineering Department (B.Tech CSBS Sem V)

Compiler Design

PART A

(Part A: TO BE REFFERED BY STUDENTS)

A.1 AIM:

Development of Mini Project

A.2 Pre requisite:

Working of various phases of compiler

A.3 Outcome:

After successful completion of this experiment, students will be able to:

- Understand the internal working of various phases of a compiler
- Compare designs of few production compilers

A.4 Theory:

A.5 Procedure/Task:

- This is a group activity
- Each group will consist of 3 members
- Each group has to integrate at least 3 modules (phases of compiler) into one single program
    - lexical analysis,
    - syntax analysis,
    - intermediate code generation
    - code optimization
- The program can be a simple menu-driven program
- Since these modules are developed as part of the Lab experiments, Students have to enhance the lab experiments by either increasing the scope of the experiment or modify the program
- The group should carry out research in order to know what new changes can be implemented

SVKM's NMIMS

Mukesh Patel School of Technology Management & Engineering (Mumbai Campus)

Computer Engineering Department (B.Tech CSBS Sem V)

Compiler Design

PART B

(PART B: TO BE COMPLETED BY STUDENTS)

(Students must submit the soft copy as per following segments within two hours of the practical. The soft copy must be uploaded on the Blackboard or emailed to the concerned lab in charge faculties at the end of the practical in case the there is no Black board access available)

Roll No. : E005, E015, E034	Name: Mahvish Reyaz, Heer Dhandhukia, Kashmaya Khandelwal
Class : BTech CSBS	Batch : 1
Date of Experiment : 03-09-23	Date/Time of Submission : 25-10-23
Grade :	

B.1 Project Details

(Detailed description of project. It should include answers to following questions)

- **Group Details with task distribution**

Mahvish Reyaz (E005): Lexical Analyzer

Heer Dhandhukia (E015): Syntax Analyzer

Kashmaya Khandelwal (E034): Intermediate Code Generator

- **Explain the purpose of the phases of compiler (implemented phases)**

The phases of a compiler serve specific purposes in the process of translating source code into machine code or an equivalent form. Here's an explanation of the purposes of the phases we have implemented:

1. Lexical Analyzer:

- Purpose: The lexical analysis phase, often referred to as the lexer or scanner, serves to break down the source code into tokens or lexemes.

- Function: It scans the source code character by character, removing whitespace and comments, and identifies keywords, identifiers, operators, and literals. The purpose is to create a stream of tokens that the subsequent phases can work with efficiently.
- Example: In the C++ source code "int x = 10;", the lexical analyzer would identify tokens like "int," "x," "=", "10," and ";".

2. Syntax Analyzer:

- Purpose: The syntax analysis phase, also known as the parser, ensures that the sequence of tokens produced by the lexical analyzer adheres to the language's grammar rules.
- Function: It constructs a parse tree or abstract syntax tree, which represents the hierarchical structure of the source code. The purpose is to check for syntax errors and enforce the correct order and structure of code elements.
- Example: In a programming language, the syntax analyzer verifies that statements like "if (condition) { code }" are structured correctly.

3. Intermediate Code Generator:

- Purpose: The intermediate code generation phase plays a crucial role in the compilation process by creating an intermediate representation of the source code.
- Function: It generates code that is closer to the machine code but abstract enough to facilitate optimization. This intermediate code makes it easier to perform high-level optimizations and target multiple hardware architectures.
- Example: Instead of directly generating machine code, the intermediate code might represent operations like addition, subtraction, and control flow statements in a more platform-independent manner.

These phases work together to transform high-level source code into an executable format while ensuring correctness and enabling optimizations. The lexical analyzer simplifies code into manageable tokens, the syntax analyzer enforces language rules, and the intermediate code generator prepares a bridge between high-level code and machine code.

- **List the enhancements done in the program for each phase**

SVKM's NMIMS
Mukesh Patel School of Technology Management & Engineering (Mumbai Campus)
Computer Engineering Department (B.Tech CSBS Sem V)
Compiler Design

1. **Lexical Analyzer:** Symbol Table, Position of Tokens, Frequency of Tokens

• In this phase, we implemented a symbol table to store the lexemes, their corresponding tokens, and their positions in the input expression.

• We also calculated the frequency of each type of token, providing valuable insights into the composition of the input expression.

• This phase effectively identified and categorized tokens into operators, integers, symbols, keywords, identifiers, and constants

2.Syntax Analyzer: Syntax Tree and Operator Precedence

• The syntax analyzer phase involved constructing a syntax tree from the input expression. The tree structure represented the hierarchical relationships between the operators and operands.

• We determined the precedence of operators, which is crucial in evaluating expressions correctly.

• The operator precedence table served as a reference for the relative priorities of different operators.

3.Intermediate Code Generator: Quadruples and Triples

• This phase involved generating intermediate code for arithmetic expressions.

• We implemented the construction of Triples and Quadruples Tables.

• We implemented error handling mechanisms to deal with potential issues, such as division by zero, ensuring the robustness of the code generator

- **Paste the source code of the program**

```
#include <iostream>

#include <cstring>

#include <cstdlib>

#include <string>

#include <map>

#include <vector>

#include <bits/stdc++.h>
```

```
#include <algorithm>

using namespace std;

#define MAX_TOKEN_LENGTH 100


struct symbol_table {

   char *lexeme;

   char *token;

   int position;

   symbol_table *next;

};


bool oper(const char *str) {

   if (!strcmp(str, "+") || !strcmp(str, "-") || !strcmp(str, "*") || !strcmp(str, ">") || !strcmp(str, "<") ||
!strcmp(str, "/") || !strcmp(str, "="))

      return true;

   return false;

}


bool iden(const char *str) {

   if (str[0] >= '0' && str[0] <= '9')

      return false;

   return true;

}


bool keyw(const char *str) {
```

```
    const char *keywords[] = {"if", "else", "while", "do", "break", "continue", "int", "double", "float", "return", "char", "case", "sizeof", "long", "short", "typedef", "switch", "unsigned", "void", "static", "struct", "goto"};

    int numKeywords = sizeof(keywords) / sizeof(keywords[0]);

    for (int i = 0; i < numKeywords; i++) {

        if (!strcmp(str, keywords[i]))

            return true;

    }

    return false;

}


bool symbol(const char *str) {

    if (!strcmp(str, "(") || !strcmp(str, ")"))

        return true;

    return false;

}


bool no(const char *str) {

    int i, len = strlen(str);

    bool hasDecimal = false;

    if (len == 0)

        return false;

    for (i = 0; i < len; i++) {

        if ((str[i] < '0' || str[i] > '9') && (str[i] != '.' || (str[i] == '-' && i > 0)))

            return false;

        if (str[i] == '.')
```

```
            hasDecimal = true;
        }
        return hasDecimal;
    }

    void insert_into_symbol_table(symbol_table *table, const char *lexeme, const char *token, int position) {
        symbol_table *new_entry = (symbol_table *)malloc(sizeof(symbol_table));
        new_entry->lexeme = strdup(lexeme);
        new_entry->token = strdup(token);
        new_entry->position = position;
        new_entry->next = table->next;
        table->next = new_entry;
    }

    const char *search_symbol_table(symbol_table *table, const char *lexeme) {
        symbol_table *current_entry = table->next;
        while (current_entry != NULL) {
            if (strcmp(current_entry->lexeme, lexeme) == 0) {
                return current_entry->token;
            }
            current_entry = current_entry->next;
        }
        return nullptr;
    }
```

```
struct TreeNode {
    std::string value;
    int precedence; // Operator precedence level
    TreeNode *left;
    TreeNode *right;
};

bool isOperator(char c) {
    return (c == '+' || c == '-' || c == '*' || c == '/');
}

int getOperatorPrecedence(char op) {
    if (op == '$') return 6; // Lowest precedence
    if (op == '+') return 4;
    if (op == '-') return 5;
    if (op == '*') return 3;
    if (op == '/') return 2;
    return 1; // Variables have the highest precedence
}

TreeNode *createNode(std::string value, int precedence) {
    TreeNode *node = new TreeNode;
    node->value = value;
    node->precedence = precedence;
```

```
    node->left = nullptr;
    node->right = nullptr;
    return node;
}


TreeNode *buildSyntaxTree(std::string expression) {
    TreeNode *root = nullptr;
    TreeNode *current = nullptr;
    for (int i = 0; i < expression.length(); i++) {
        std::string currentValue(1, expression[i]);
        if (isOperator(expression[i])) {
            current = createNode(currentValue, getOperatorPrecedence(expression[i]));
            current->right = createNode(std::string(1, expression[i + 1]), getOperatorPrecedence(expression[i
+ 1])); // Create the right node
            current->left = root;
            root = current;
            i++; // Skip the next character
        } else {
            if (isalnum(expression[i])) {
                if (current == nullptr) {
                    root = createNode(currentValue, 1); // Variables have the highest precedence
                } else {
                    if (current->right == nullptr) {
                        current->right = createNode(currentValue, 1);
                    } else {
```

```
                TreeNode *temp = createNode(currentValue, 1);
                temp->left = current->right;
                current->right = temp;
            }
        }
    }
}
return root;
}

void printSyntaxTree(TreeNode *node, std::string indent = "", bool isRight = false) {
    if (node != nullptr) {
        std::cout << indent;
        if (isRight) {
            std::cout << "R---- ";
            indent += "     ";
        } else {
            std::cout << "L---- ";
            indent += "|    ";
        }

        std::cout << node->value << " (Precedence: " << node->precedence << ")" << std::endl;

        printSyntaxTree(node->left, indent, false);
```

```
        printSyntaxTree(node->right, indent, true);
    }
}

void printOperatorPrecedenceTable() {
    std::cout << std::endl;
    std::cout << "Operator Precedence Table:" << std::endl;
    std::cout << std::endl;
    std::cout << "Operator  | Precedence Level" << std::endl;
    std::cout << "--------------------------" << std::endl;
    std::cout << "Variables | 1 (Highest)" << std::endl;
    std::cout << "/         | 2" << std::endl;
    std::cout << "*         | 3" << std::endl;
    std::cout << "+         | 4" << std::endl;
    std::cout << "-         | 5" << std::endl;
    std::cout << "$         | 6 (Lowest)" << std::endl;
}

vector<pair<int, int>> updatevec(string s) {
    vector<pair<int, int>> ans;
    for (int i = 0; i < s.size(); i++) {
        if (s[i] == '%') {
            ans.push_back({1, i});
        } else if (s[i] == '/') {
            ans.push_back({2, i});
```

```
        } else if (s[i] == '*') {
            ans.push_back({3, i});
        } else if (s[i] == '+') {
            ans.push_back({4, i});
        } else if (s[i] == '-') {
            ans.push_back({5, i});
        }
    }
    sort(ans.begin(), ans.end());
    return ans;
}

int main() {
    int choice;
    do {
        std::cout<<"\n\n\n";
        std::cout<<"----------------------------------------\n";
        std::cout << "                MENU              \n";
        std::cout<<"----------------------------------------\n";
        std::cout << "1. Lexical Analyzer\n";
        std::cout << "2. Syntax Analyzer\n";
        std::cout << "3. Intermediate Code Generator\n";
        std::cout << "4. Exit\n";
        std::cout<<"----------------------------------------\n";
        std::cout << "Enter your choice: ";
```

```
std::cin >> choice;
std::cout<<"\n";
switch (choice) {
    case 1: {
        char str[MAX_TOKEN_LENGTH];
        std::cout << "Enter an expression: ";
        std::cin.ignore();
        std::cin.getline(str, sizeof(str));
        symbol_table table;
        table.next = nullptr;
        char delims[] = " ";
        char *token = strtok(str, delims);

        std::map<std::string, int> tokenFrequency;
        int position = 1; // Initialize the position

        while (token != nullptr) {
            const char *token_type = nullptr;
            if (oper(token)) {
                token_type = "Operator";
            } else if (no(token)) {
                token_type = "Integer";
            } else if (symbol(token)) {
                token_type = "Symbol";
            } else if (keyw(token)) {
```

```
            token_type = "Keyword";
        } else if (iden(token)) {
            token_type = "Identifier";
        } else {
            token_type = "Constant";
        }
        insert_into_symbol_table(&table, token, token_type, position);
        tokenFrequency[token_type]++;
        token = strtok(nullptr, delims);
        position++; // Increment the position
    }
    std::cout << "\n\nSymbol Table:\n";
    std::cout << "-----------------------------------------";
    std::cout << "\nPosition\tLexeme\t\tToken\n";
    std::cout << "-----------------------------------------\n";

    symbol_table *current_entry = table.next;
    while (current_entry != nullptr) {
        std::cout << current_entry->position << "\t\t" << current_entry->lexeme << "\t\t" <<
current_entry->token << std::endl;
        current_entry = current_entry->next;
    }

    std::cout << "\n\nToken Frequencies:\n\n";
    for (const auto &pair : tokenFrequency)
```

```
            {
            std::cout << pair.first << ": " << pair.second << std::endl;
            }

            break;
        }

        case 2: {
            std::string expression;
            std::cout << "Enter an expression: ";
            std::cin.ignore();
            std::getline(std::cin, expression);
            TreeNode *syntaxTree = buildSyntaxTree(expression);
            std::cout << "\nSyntax Tree:\n";
            printSyntaxTree(syntaxTree);
            printOperatorPrecedenceTable();
            break;
        }

        case 3: {
            string s;
cout << "Enter the Expression: ";
cin >> s;
int check = 0;
vector<pair<int, int>> v;
```

```
v = updatevec(s);

char temptoadd = 'A';


// Vector to store quadruples

vector<tuple<string, string, string, string>> quadruples;

vector<tuple<string, string, string>> triples;


try {

    while (v.size() != 0 && s.size() > 3) {

        string temp = "";

        temp.push_back(s[v[0].second - 1]);

        if (v[0].first == 1) {

            temp.push_back('%');

            if (s[v[0].second + 1] == '0') {

                throw runtime_error("Division by zero");

            }

        } else if (v[0].first == 2) {

            temp.push_back('/');

            if (s[v[0].second + 1] == '0') {

                throw runtime_error("Division by zero");

            }

        } else if (v[0].first == 3) {

            temp.push_back('*');

        } else if (v[0].first == 4) {

            temp.push_back('+');
```

```
        } else {
            temp.push_back('-');
        }
        temp.push_back(s[v[0].second + 1]);

        // Generate quadruple
        quadruples.push_back(make_tuple(string(1, temp[0]), string(1, temp[2]), string(1, temp[1]),
string(1, temptoadd)));

         // Generate triple
         string operatorSymbol = string(1, temp[1]); // Get the operator symbol
         triples.push_back(make_tuple(string(1, temp[0]), string(1, temp[2]), operatorSymbol));
        string temporary = "";
        temporary.push_back(temptoadd);
        temporary = temporary + " = " + temp;
        cout << temporary << "\t\t";
        string str = "";
        for (int i = 0; i < v[0].second - 1; i++) {
            str.push_back(s[i]);
        }
        str.push_back(temptoadd);
        for (int i = v[0].second + 2; i < s.size(); i++) {
            str.push_back(s[i]);
        }
        s = str;
```

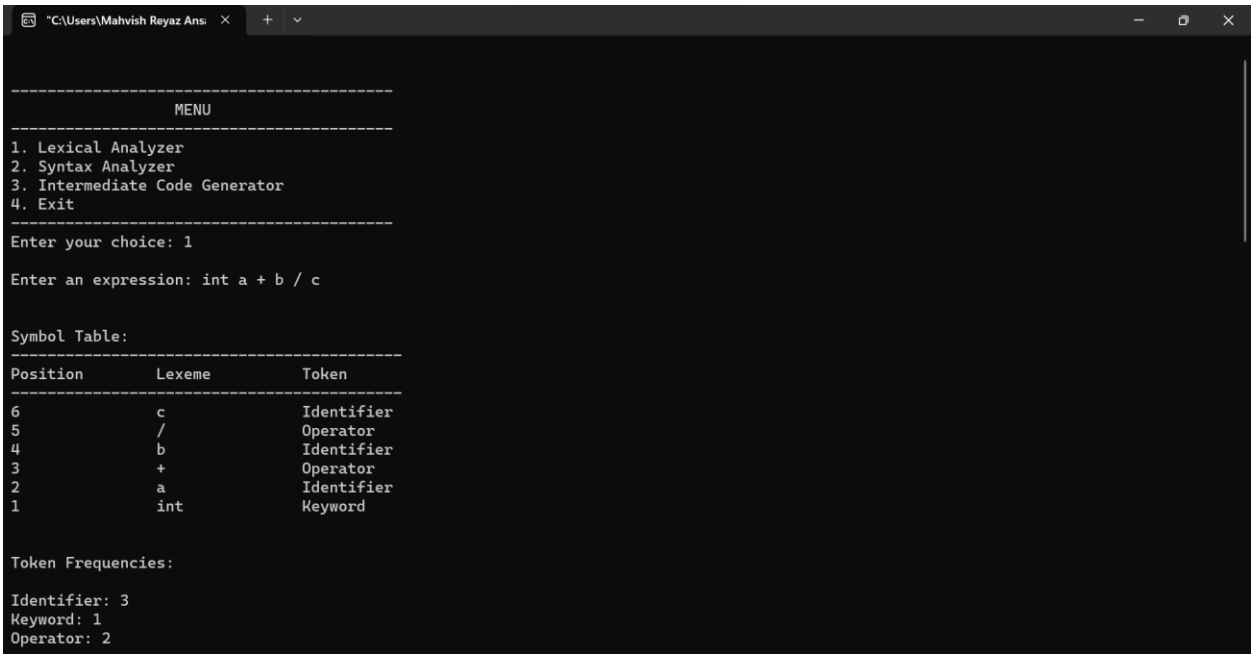
```
            cout << s << endl;
            v = updatevec(s);
            temptoadd++;
        }

        // Create a new temporary variable to store the final result
        char finalTemp = temptoadd;
        string finalExpression = s;
        cout << finalTemp << " = " << finalExpression << endl;
    } catch (const exception& e) {
        cerr << "Error: " << e.what() << endl;
    }

    // Print the heading for quadruples
    cout << "\nQuadruples: " << endl;
    cout << "----------------------------------------" << endl;
    cout << "Arg1\tArg2\tOperator\tResult" << endl;
    cout << "----------------------------------------" << endl;

    // Print the generated quadruples
    for (const auto& quadruple : quadruples) {
        cout << get<0>(quadruple) << "\t" << get<1>(quadruple) << "\t" << get<2>(quadruple) << "\t\t" <<
get<3>(quadruple) << endl;
    }
```

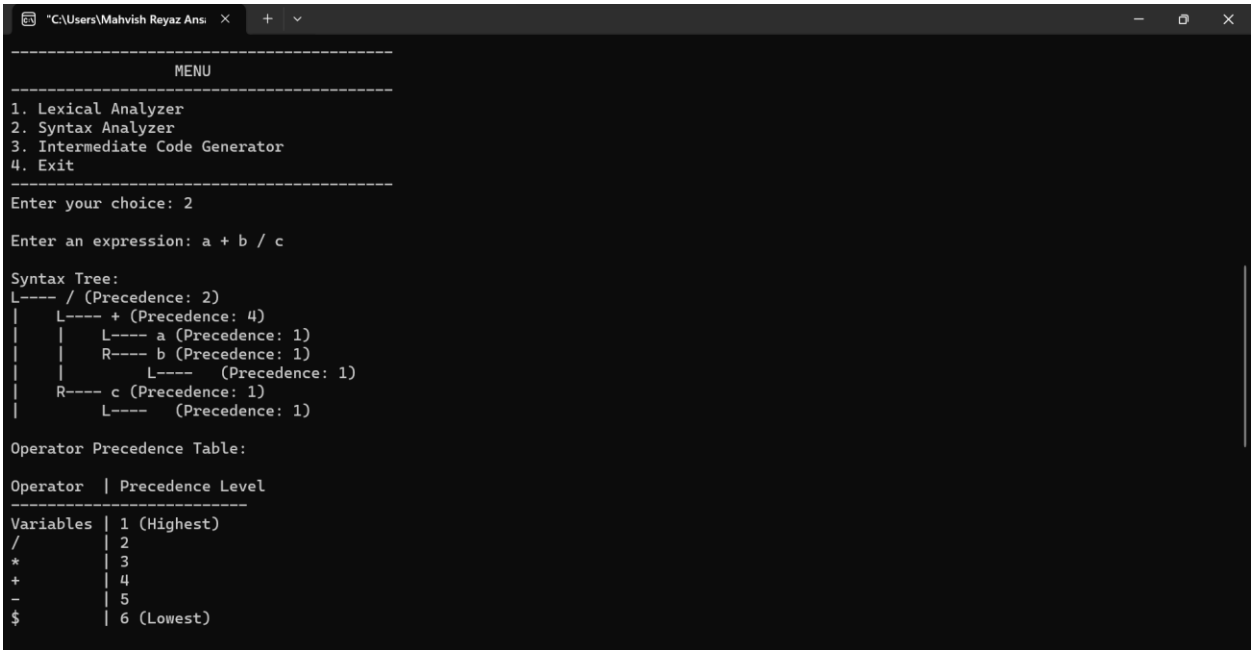
```
// Print the heading for triples
    cout << "\nTriples: " << endl;
    cout << "-----------------------------------------" << endl;
    cout << "Arg1\t\tArg2\t\tOperator" << endl;
    cout << "-----------------------------------------" << endl;
    // Print the generated triples
    for (const auto& triple : triples) {
        cout << get<0>(triple) << "\t\t" << get<1>(triple) << "\t\t" << get<2>(triple) << endl;
    }


            break;
        }


        case 4: {
            std::cout << "Exiting the program.\n";
            break;
        }


        default: {
            std::cout << "Invalid choice. Please select a valid option.\n";
        }
    }
} while (choice != 4);


return 0;
```

}

B.2 Output

(Take screen shots of the output at run time and paste it here)





```
-----------------------------------------
                  MENU
-----------------------------------------
1. Lexical Analyzer
2. Syntax Analyzer
3. Intermediate Code Generator
4. Exit
-----------------------------------------
Enter your choice: 3

Enter the Expression: a+b/c*g
A = b/c         a+A*g
B = A*g         a+B
C = a+B

Quadruples:
---------------------------------------
Arg1    Arg2    Operator        Result
---------------------------------------
b       c       /               A
A       g       *               B

Triples:
---------------------------------------
Arg1            Arg2            Operator
---------------------------------------
b               c               /
A               g               *
```

```
-----------------------------------------
                  MENU
-----------------------------------------
1. Lexical Analyzer
2. Syntax Analyzer
3. Intermediate Code Generator
4. Exit
-----------------------------------------
Enter your choice: 4

Exiting the program.

Process returned 0 (0x0)   execution time : 62.835 s
Press any key to continue.
```

B.3 Conclusion:

(Students must write the conclusion as per the attainment of individual outcome listed above)

Overall, this project provided us with a comprehensive understanding of the different phases of compiler design and emphasizes the importance of careful planning, modular development, and thorough testing. It helped us understand the practical application of theoretical concepts in building a functional compiler.

B.4 Observations and Learning:

(Students must write their observations and learnings as per the attainment of individual outcome listed above)

• **Symbol Table**: The symbol table is a fundamental data structure in lexical analysis, facilitating efficient lookup and retrieval of token information.

• **Syntax Tree:** The syntax tree is a powerful tool for representing the hierarchical structure of expressions, making it easier to evaluate complex mathematical operations.

• **Operator Precedence**: Understanding operator precedence is crucial for correctly parsing and evaluating expressions. It ensures that operators are applied in the correct order.

• **Error Handling**: Implementing error handling mechanisms is essential for creating a robust compiler. It helps identify and address potential issues that may arise during code generation.

• **Modular Design**: The project demonstrates the importance of breaking down the compiler into distinct phases (lexical analysis, syntax analysis, and code generation). This modular design allows for focused development and easier debugging.

• **Input Validation:** The project highlights the significance of validating input expressions to ensure they conform to the language's syntax and semantics. This helps in preventing runtime errors.

• **Frequency Analysis:** Analyzing the frequency of different types of tokens provides valuable insights into the composition of the input expression. This information can be used for optimization and further analysis.

• **Documentation and Comments**: The code is well-documented and contains comments that enhance readability and maintainability. Clear and concise documentation is crucial for understanding and extending the compiler