DELHI TECHNOLOGICAL UNIVERSITY

(Formerly Delhi College of Engineering)
Shahbad Daulatpur, Bawana Road, Delhi 110042

DEPARTMENT OF COMPUTER SCIENCE AND ENGINEERING

CO302 : COMPILER DESIGN

LAB FILE

Submitted To:
Dr. Pawan Singh Mehra
Department of Computer Science
And Engineering

Submitted By:
Ashwani Kumar
B. Tech. COE IIIrd Year (SEM - VI)
2K22/CO/113

INDEX

S. No.	Title	Date	Signature
1.	Write a program to check whether an entered string is a keyword or not.	15-01-2025	
2.	Write a program to count the spaces and number of lines.	15-01-2025	
3.	Write a program in LEX to count the number of comment lines in a C program. Also, eliminate them and copy the resulting program into separate file.	22-01-2025	
4.	Construction of DFA from NFA.	29-01-2025	
5.	Implementation of SHIFT REDUCE PARSING ALGORITHM.	05-02-2025	
6.	Implementation of OPERATOR PRECEDENCE PARSER.	19-02-2025	
7.	Implementation of RECURSIVE DESCENT PARSER.	19-02-2025	
8.	Implementation of CODE OPTIMIZATION TECHNIQUES.	25-02-2025	
9.	Implementation of CODE GENERATOR.	12-03-2025	

EXPERIMENT 1

AIM

Write a program to check whether an entered string is a keyword or not.

THEORY

In Compiler Design, Keywords are reserved words that have special meanings in a programming language and cannot be used as identifiers, such as variable names or function names. The Lexical Analyzer of a compiler is responsible for identifying keywords during the tokenization phase.

Keywords are predefined and recognized by the compiler, helping in syntax and semantic analysis.

Some common keywords include:

- int
- float
- if
- else
- while
- return

To determine if a given string is a keyword:

1. Store all keywords in a pre-defined list.
2. Compare the input string against the list.
3. If a match is found, the string is a keyword; otherwise, it is not.

PROGRAM CODE

```
#include <iostream>

#include <set>

#include <string>


using namespace std;


// Function to Check if a Word is a Keyword

bool isKeyword(const string& word)
```

```
{
    // Set of Keywords
    set<string> keywords = {
        "auto", "break", "case", "char", "const", "continue", "default", "do", "double",
        "else", "enum", "extern", "float", "for", "goto", "if", "inline", "int", "long",
        "register", "return", "short", "signed", "sizeof", "static", "struct", "switch",
        "typedef", "union", "unsigned", "void", "volatile", "while", "class", "public",
        "private", "protected", "virtual", "friend", "namespace", "template", "try",
        "catch", "throw", "new", "delete", "using"
    };

    return keywords.find(word) != keywords.end();
}

int main()
{
    string word;

    cout << "Enter a Word: ";
    cin >> word;

    // Check if the Word is a Keyword
    if (isKeyword(word))
        cout << word << " is a Keyword." << endl;
    else
        cout << word << " is NOT a Keyword." << endl;

    return 0;
}
```

OUTPUT FILE

```
PS C:\Users\Acer\Desktop\Compiler Design\Experiments> cd "c:\Users\Acer\Desktop\Compiler Design\Experiments\" ; if ($?) { g++ Experiment_1.cpp -o Experiment_1 } ; if ($?) { .\Experiment_1 }
Enter a Word: new
new is a Keyword.
PS C:\Users\Acer\Desktop\Compiler Design\Experiments> cd "c:\Users\Acer\Desktop\Compiler Design\Experiments\" ; if ($?) { g++ Experiment_1.cpp -o Experiment_1 } ; if ($?) { .\Experiment_1 }
Enter a Word: for
for is a Keyword.
PS C:\Users\Acer\Desktop\Compiler Design\Experiments> cd "c:\Users\Acer\Desktop\Compiler Design\Experiments\" ; if ($?) { g++ Experiment_1.cpp -o Experiment_1 } ; if ($?) { .\Experiment_1 }
Enter a Word: Hello
Hello is NOT a Keyword.
```

FINDINGS AND LEARNINGS

1. Learned how compilers identify and differentiate keywords from identifiers.
2. Used a set for efficient keyword storage and quick lookup.
3. Practiced string comparisons, input handling, and search operations.
4. Gained insight into how reserved words impact the syntax and semantics of a language.

EXPERIMENT 2

AIM

Write a program to count the spaces and number of lines.

THEORY

Text processing is essential in various applications, such as text editors, compilers, and data analysis tools. A fundamental aspect of text processing is analyzing the structure of a document by counting spaces and lines.

- Spaces (' '): These characters separate words in a sentence and contribute to text formatting.
- Newlines ('\n'): A newline character signifies the end of a line and the beginning of a new one.

To count spaces and lines in a given text:

1. Read the input text line by line.
2. Increment the line counter each time a new line is read.
3. Iterate through each character in the line and count spaces.
4. Display the total number of spaces and lines at the end of processing.

This approach efficiently processes multi-line input using getline() for flexible handling of different text inputs.

PROGRAM CODE

```
#include <iostream>

#include <string>


using namespace std;


int main()

{

    string line;

    int spaceCount = 0, lineCount = 0;
```

```
    cout << "Enter Text (Press Ctrl+D or Ctrl+Z to Stop Input):" << endl;

    // Read input Line by Line
    while (getline(cin, line))
    {
        lineCount++;                 // Count Lines

        for (char ch : line)
        {
            if (ch == ' ')
            {
                spaceCount++;       // Count Spaces
            }
        }
    }

    cout << "\nTotal Spaces: " << spaceCount << endl;
    cout << "Total Lines: " << lineCount << endl;

    return 0;
}
```

OUTPUT FILE

```
PS C:\Users\Acer\Desktop\Compiler Design\Experiments> cd "c:\Users\Acer\Desktop\Compiler Design\Experiments\" ; if ($?) { g++ Ex
periment_2.cpp -o Experiment_2 } ; if ($?) { .\Experiment_2 }
Enter Text (Press Ctrl+D or Ctrl+Z to Stop Input):
Hello World!
This is test for CD Practical.
Count the Lines and Spaces.
^Z

Total Spaces: 10
Total Lines: 3
```

FINDINGS AND LEARNINGS

1. Learned how to read multi-line input efficiently using getline(), making it easier to process large text data.
2. Understood the role of spaces and newlines in text formatting and document structure.
3. Practiced iterating through strings and counting specific characters like spaces to analyze text content.
4. Gained insight into basic text-processing techniques, which are useful in applications such as compilers, text editors, and data analytics.
5. Developed a better understanding of character manipulation in strings to extract useful information.

EXPERIMENT 3

AIM

Write a program in LEX to count the number of comment lines in a C++ program. Also, eliminate them and copy the resulting program into separate file.

THEORY

LEX is a lexical analyzer generator used for pattern-based text processing. It reads input files and applies rules to recognize specific patterns.

In this experiment, we need to:

1. Identify and count comments in a C++ program.
   - Single-line comments: // This is a comment
   - Multi-line comments: /* This is a
     multi-line comment */
2. Remove the comments while preserving the remaining code.
3. Store the cleaned code in a separate file.

Regular expressions to be handled:

- Single-line comments: Any line starting with //.
- Multi-line comments: Any content between /* and */.

PROGRAM CODE

```
#include <iostream>
#include <fstream>
#include <string>

using namespace std;

void removeComments(const string& inputFile, const string& outputFile)
{
    ifstream inFile(inputFile);
    ofstream outFile(outputFile);
```

```
    if (!inFile || !outFile)
    {
        cerr << "Error Opening Files!" << endl;
        return;
    }

    string line;
    bool inMultiLineComment = false;
    int commentCount = 0;

    while (getline(inFile, line))
    {
        string cleanLine;
        size_t i = 0;

        while (i < line.length())
        {
            if (!inMultiLineComment && i + 1 < line.length() && line[i] == '/' && line[i + 1] ==
'/')
            {
                commentCount++;                 // Count Single-Line Comment
                break;                          // Stop at the Comment but keep Spaces before it
            }
            else if (!inMultiLineComment && i + 1 < line.length() && line[i] == '/' && line[i +
1] == '*')
            {
                inMultiLineComment = true;
                commentCount++;                 // Count Start of Multi-Line Comment
                i += 2;
            }
```

```
            else if (inMultiLineComment && i + 1 < line.length() && line[i] == '*' && line[i +
1] == '/')
            {
                inMultiLineComment = false;
                i += 2;
                continue;
            }
            else if (!inMultiLineComment)
            {
                cleanLine += line[i];             // Preserve all Spaces and Code and Add Non-
Comment Sharacters to Output
            }
            i++;
        }

        // Write the Exact Line Format with Spaces Preserved
        outFile << cleanLine << endl;
    }

    cout << "Total Comment Lines: " << commentCount << endl;

    inFile.close();
    outFile.close();
}

int main()
{
    string inputFile = "Input.cpp";
    string outputFile = "Output.cpp";

    removeComments(inputFile, outputFile);
```

```
    cout << "Comment-free Code Saved in " << outputFile << endl;

    return 0;
}
```

INPUT FILE (Input.cpp)

```
Input.cpp
Input.cpp > ...
#include <iostream>
using namespace std;

int main() {
    int a = 10;  // This is a comment
    int b = 20;  /* This is a
                    multi-line comment */
    cout << a + b << endl; // Print sum
    return 0;
}
```

OUTPUT FILE (Output.cpp)

```
Output.cpp
Output.cpp > ...
#include <iostream>
using namespace std;

int main() {
    int a = 10;
    int b = 20;

    cout << a + b << endl;
    return 0;
}

```

OUTPUT

```
PS C:\Users\Acer\Desktop\Practicals\Compiler Design\Experiments> cd "c:\Users\Acer\Desktop\Practicals\Compiler Design\Experiment
s\" ; if ($?) { g++ Experiment_3.cpp -o Experiment_3 } ; if ($?) { .\Experiment_3 }
Total Comment Lines: 3
Comment-free Code Saved in Output.cpp
```

FINDINGS AND LEARNINGS

1. Learned how to read and process a file line by line.
2. Understood the structure of single-line (//) and multi-line (/* */) comments.
3. Practiced using ifstream and ofstream for file handling.
4. Implemented logic to remove comments efficiently while preserving code structure.

EXPERIMENT 4

AIM

Construction of DFA from NFA.

THEORY

A Finite Automaton is a model of computation used in pattern recognition and lexical analysis.

There are two main types:

1. Non-Deterministic Finite Automaton (NFA):
    - Multiple transitions for the same input symbol.
    - Can have ε (epsilon) transitions.
    - Can be in multiple states at once.
2. Deterministic Finite Automaton (DFA):
    - Only one transition per input symbol from any given state.
    - No ε (epsilon) transitions.
    - Always in exactly one state at any given time.

Steps to Convert NFA to DFA:

1. Find the ε-closure of states (states reachable using only ε-transitions).
2. Construct a new DFA state for each unique set of NFA states.
3. Determine transitions based on input symbols.
4. Mark final states if any NFA final state is present in the DFA state.

PROGRAM CODE

```
#include <iostream>

#include <vector>

#include <set>

#include <map>

#include <queue>


using namespace std;


// Structure for NFA

struct NFA
```

```
{
    int states;
    map<int, map<string, set<int>>> transitions;
    set<int> finalStates;
};

// Function to Find Epsilon Closure of a State
set<int> epsilonClosure(int state, const map<int, map<string, set<int>>>& transitions)
{
    set<int> closure;
    queue<int> q;
    q.push(state);
    closure.insert(state);

    while (!q.empty()) {
        int current = q.front();
        q.pop();

        if (transitions.count(current) && transitions.at(current).count("ε"))
        {
            for (int next : transitions.at(current).at("ε"))
            {
                if (closure.find(next) == closure.end())
                {
                    closure.insert(next);
                    q.push(next);
                }
            }
        }
    }
```

```
    return closure;
}

// Function to Convert NFA to DFA
void convertNFAtoDFA(const NFA& nfa)
{
    map<set<int>, int> dfaStates;
    vector<map<string, int>> dfaTransitions;
    set<int> dfaFinalStates;
    queue<set<int>> stateQueue;
    int stateCounter = 0;

    // Initial DFA State (ε-Closure of NFA Start State)
    set<int> startState = epsilonClosure(0, nfa.transitions);
    dfaStates[startState] = stateCounter++;
    stateQueue.push(startState);
    dfaTransitions.push_back({});

    while (!stateQueue.empty())
    {
        set<int> currentState = stateQueue.front();
        stateQueue.pop();
        int dfaStateIndex = dfaStates[currentState];

        // Check if this State Contains any Final NFA State
        for (int nfaState : currentState)
        {
            if (nfa.finalStates.count(nfaState))
            {
                dfaFinalStates.insert(dfaStateIndex);
```

```
        break;
    }
}

// Process Transitions for Each Input Symbol
set<string> inputSymbols;
for (int nfaState : currentState)
{
    if (nfa.transitions.count(nfaState))
    {
        for (auto transition : nfa.transitions.at(nfaState))
        {
            if (transition.first != "ε")
            {
                // Ignore Epsilon Transitions
                inputSymbols.insert(transition.first);
            }
        }
    }
}

for (string symbol : inputSymbols)
{
    set<int> newState;
    for (int nfaState : currentState)
    {
        if (nfa.transitions.count(nfaState) && nfa.transitions.at(nfaState).count(symbol))
        {
            for (int nextState : nfa.transitions.at(nfaState).at(symbol))
            {
```

```
                set<int> closure = epsilonClosure(nextState, nfa.transitions);
                newState.insert(closure.begin(), closure.end());
            }
        }
    }

    if (!newState.empty())
    {
        if (dfaStates.find(newState) == dfaStates.end())
        {
            dfaStates[newState] = stateCounter++;
            stateQueue.push(newState);
            dfaTransitions.push_back({});
        }
        dfaTransitions[dfaStateIndex][symbol] = dfaStates[newState];
    }
  }
}

// Display DFA
cout << "DFA States:\n";
for (auto state : dfaStates)
{
  cout << "State " << state.second << ": { ";
  for (int nfaState : state.first)
  {
    cout << nfaState << " ";
  }
  cout << "}\n";
}
```

```
    cout << "\nDFA Transitions:\n";
    for (int i = 0; i < dfaTransitions.size(); i++)
    {
        for (auto transition : dfaTransitions[i])
        {
            cout << "State " << i << ": " << transition.first << " --> State " << transition.second
<< endl;
        }
    }

    cout << "\nDFA Final States:\n";
    for (int state : dfaFinalStates)
    {
        cout << "State " << state << endl;
    }
}

int main()
{
    NFA nfa;
    nfa.states = 3;

    // Define NFA Transitions Using "ε" for Epsilon
    nfa.transitions[0]["ε"] = {1};
    nfa.transitions[1]["a"] = {1, 2};
    nfa.transitions[2]["b"] = {2};

    // Define final state
    nfa.finalStates = {2};
```

```
    // Convert NFA to DFA
    convertNFAtoDFA(nfa);

    return 0;
}
```

OUTPUT FILE

```
PS C:\Users\Acer\Desktop\Practicals\Compiler Design\Experiments> cd "c:\Users\Acer\Desktop\Practicals\Compiler Design\Experiment
s\" ; if ($?) { g++ Experiment_4.cpp -o Experiment_4 } ; if ($?) { .\Experiment_4 }
DFA States:
State 0: { 0 1 }
State 1: { 1 2 }
State 2: { 2 }

DFA Transitions:
State 0: a --> State 1
State 1: a --> State 1
State 1: b --> State 2
State 2: b --> State 2

DFA Final States:
State 1
State 2
```

FINDINGS AND LEARNINGS

1. Understood the difference between NFA and DFA in automata theory.
2. Learned how to compute ε-closure and construct DFA states step by step.
3. Implemented an efficient algorithm for NFA-to-DFA conversion.
4. Practiced handling sets, maps, and queues for state transitions and processing.
5. Gained insight into state minimization and optimization in DFA construction.

EXPERIMENT 5

AIM

Implementation of SHIFT REDUCE PARSING ALGORITHM.

THEORY

Shift-Reduce Parsing is a bottom-up parsing technique used in syntax analysis. It attempts to reduce an input string to the start symbol of a grammar by performing Shifts and Reductions based on a predefined set of rules.

Operations in Shift-Reduce Parsing:

1. Shift: Move the next input symbol onto the stack.
2. Reduce: Replace a sequence of stack symbols matching a grammar rule's right-hand side (RHS) with its left-hand side (LHS).
3. Accept: If the stack contains only the start symbol and the input is fully processed, parsing is successful.
4. Error: If no valid shift or reduction is possible, the string is rejected.

Steps in Shift-Reduce Parsing:

1. Initialize an empty stack and set the input pointer to the first symbol.
2. Shift Step: Move the next input symbol to the stack and advance the input pointer.
3. Reduce Step: If the top of the stack matches the right-hand side (RHS) of a grammar rule, replace it with the corresponding left-hand side (LHS).
4. Repeat shifting and reducing until:
5. The stack contains only the start symbol, and the input is fully processed (Accepted).
6. No valid shift or reduction is possible (Error and reject the input).

This method is widely used in compilers, particularly in LR parsers, to efficiently analyze and process syntax.

Example Grammar:

$E \rightarrow E + E$

$E \rightarrow E * E$

$E \rightarrow (E)$

$E \rightarrow id$

Here, id represents an identifier (a variable or number).

PROGRAM CODE

```
#include <iostream>
#include <stack>
#include <vector>
#include <string>

using namespace std;

// Grammar Rules (Shift-Reduce)
vector<vector<string>> rules = {
    {"E", "+", "E"},
    {"E", "*", "E"},
    {"(", "E", ")"},
    {"id"}
};

// Function to Check if a Symbol is a Non-Terminal
bool isNonTerminal(const string& s)
{
    return (s == "E");
}

// Function to Check if a Sequence can be Reduced
bool isValidReduction(vector<string>& stackContent)
{
    for (const auto& rule : rules)
    {
        if (stackContent.size() >= rule.size())
        {
            vector<string> lastTokens(stackContent.end() - rule.size(), stackContent.end());
```

```
            if (lastTokens == rule)
            {
                return true;
            }
        }
    }
    return false;
}

// Perform Reduction if Possible
bool reduce(stack<string>& st)
{
    vector<string> stackContent;
    stack<string> tempStack = st;

    // Extract Current Stack Content
    while (!tempStack.empty())
    {
        stackContent.insert(stackContent.begin(), tempStack.top());
        tempStack.pop();
    }

    // Try Reducing with Grammar Rules
    for (const auto& rule : rules)
    {
        if (stackContent.size() >= rule.size())
        {
            vector<string> lastTokens(stackContent.end() - rule.size(), stackContent.end());
            if (lastTokens == rule)
            {
```

```
            // Pop Elements from Stack
            for (size_t i = 0; i < rule.size(); ++i)
            {
                st.pop();
            }
            // Push Reduced Non-Terminal
            st.push("E");
            cout << "Reduce: ";
            for (const string& token : lastTokens)
            {
                cout << token << " ";
            }
            cout << "-> E" << endl;
            return true;
        }
    }
}
return false;
}

// Shift-Reduce Parsing
void shiftReduceParsing(vector<string> input)
{
    stack<string> st;
    int i = 0;
    cout << "\nParsing Steps:\n";

    while (i < input.size() || !st.empty())
    {
        // Print Current Stack
```

```
cout << "Stack: ";
stack<string> temp = st;
vector<string> stackContent;
while (!temp.empty())
{
    stackContent.push_back(temp.top());
    temp.pop();
}
for (int j = stackContent.size() - 1; j >= 0; --j)
{
    cout << stackContent[j] << " ";
}
cout << endl;

// Shift Operation
if (i < input.size())
{
    st.push(input[i]);
    cout << "Shift: " << input[i] << endl;
    i++;
}

// Apply Reductions Until no Further Reduction is Possible
while (reduce(st)) {}

// Check if only Start Symbol Remains
if (st.size() == 1 && isNonTerminal(st.top()) && i == input.size())
{
    cout << "\nParsing Successful. The Input is Accepted.\n";
    return;
```

```
        }
    }

    cout << "\nParsing Failed. The Input is Rejected.\n";
}

int main()
{
    vector<string> input = {"id", "+", "id", "*", "id"};
    cout << "Input Expression: ";
    for (const string& token : input)
    {
        cout << token << " ";
    }
    cout << endl;

    shiftReduceParsing(input);

    return 0;
}
```

OUTPUT FILE

```
PS C:\Users\Acer\Desktop\Practicals\Compiler Design\Experiments> cd "c:\Users\Acer\Desktop\Practicals\Compiler Design\Experiment
s\" ; if ($?) { g++ Experiment_5.cpp -o Experiment_5 } ; if ($?) { .\Experiment_5 }
Input Expression: id + id * id

Parsing Steps:
Stack:
Shift: id
Reduce: id -> E
Stack: E
Shift: +
Stack: E +
Shift: id
Reduce: id -> E
Reduce: E + E -> E
Stack: E
Shift: *
Stack: E *
Shift: id
Reduce: id -> E
Reduce: E * E -> E

Parsing Successful. The Input is Accepted.
```

FINDINGS AND LEARNINGS

1. Learned how Shift-Reduce Parsing works by performing shift and reduce operations iteratively.
2. Implemented a simple parser that recognizes arithmetic expressions based on pre-defined grammar.
3. Used a stack to maintain the parsing process, mimicking bottom-up parsing.
4. Understood how reductions work by replacing RHS with the corresponding LHS based on grammar rules.

EXPERIMENT 6

AIM

Implementation of OPERATOR PRECEDENCE PARSER.

THEORY

Operator Precedence Parsing is a bottom-up parsing technique used for grammars where no production has:

- Two adjacent non-terminals
- An ε-production (empty production)

Operator Precedence Relations:

Three relations define the parsing process:

1. > (Greater Than): The top of the stack has higher precedence, so reduce.
2. < (Less Than): The input symbol has higher precedence, so shift.
3. = (Equal): Used for handling parentheses.

Steps in Operator Precedence Parsing:

1. Initialize an empty stack and push a special start symbol ($).
2. Shift Step: Compare the top of the stack with the input symbol using precedence rules. If '<' or '=', push the symbol onto the stack and advance the input pointer.
3. Reduce Step: If '>' is encountered, pop symbols from the stack until a '<' is found, replacing them with a non-terminal.
4. Repeat shifting and reducing until:
    - The stack contains only the start symbol, and the input is processed (Accepted).
    - No valid shift or reduction is possible (Error and reject the input).

This parsing method is efficient for handling expressions with operators of different precedence levels.

Example Grammar with Precedence Rules:

$E \rightarrow E + E$

$E \rightarrow E * E$

$E \rightarrow (E)$

$E \rightarrow id$

Precedence Order:

$* > +$

(has the lowest precedence and) has the highest for grouping.

PROGRAM CODE

```
#include <iostream>
#include <stack>
#include <map>
#include <vector>

using namespace std;

// Operator Precedence Table
map<string, map<string, char>> precedence = {
    {"+", {{"+", '>'}, {"*", '<'}, {"(", '<'}, {")", '>'}, {"id", '<'}, {"$", '>'}}},
    {"*", {{"+", '>'}, {"*", '>'}, {"(", '<'}, {")", '>'}, {"id", '<'}, {"$", '>'}}},
    {"(", {{"+", '<'}, {"*", '<'}, {"(", '<'}, {")", '='}, {"id", '<'}, {"$", 'E'}}},
    {")", {{"+", '>'}, {"*", '>'}, {"(", 'E'}, {")", '>'}, {"id", 'E'}, {"$", '>'}}},
    {"id", {{"+", '>'}, {"*", '>'}, {"(", 'E'}, {")", '>'}, {"id", 'E'}, {"$", '>'}}},
    {"$", {{"+", '<'}, {"*", '<'}, {"(", '<'}, {")", 'E'}, {"id", '<'}, {"$", 'A'}}}
};

// Function to Get Precedence Relation
char getPrecedence(string stackTop, string input)
{
    if (precedence[stackTop].find(input) != precedence[stackTop].end())
        return precedence[stackTop][input];
    return 'E';         // Error Case
}

// Function to Perform Operator Precedence Parsing
void operatorPrecedenceParsing(vector<string> input)
{
    stack<string> st;
```

```
st.push("$");        // Bottom of Stack Symbol

int i = 0;
input.push_back("$"); // End of Input Marker

cout << "\nParsing Steps:\n";
cout << "Stack\t\tInput\t\tAction\n";
cout << "----------------------------------------\n";

while (!st.empty())
{
    // Print Current Stack
    string stackContent;
    stack<string> temp = st;
    vector<string> revStack;
    while (!temp.empty()) {
        revStack.push_back(temp.top());
        temp.pop();
    }
    for (int j = revStack.size() - 1; j >= 0; --j)
    {
        stackContent += revStack[j] + " ";
    }

    // Print Input Remaining
    string remainingInput;
    for (int j = i; j < input.size(); j++) {
        remainingInput += input[j] + " ";
    }
```

```
// Get Top of Stack
string stackTop = st.top();
string currentSymbol = input[i];

// Get Precedence Relation
char relation = getPrecedence(stackTop, currentSymbol);

cout << stackContent << "\t\t" << remainingInput << "\t\t";

if (relation == '<' || relation == '=')
{
    // Shift
    st.push(currentSymbol);
    cout << "Shift: " << currentSymbol << endl;
    i++;
}
else if (relation == '>')
{
    // Reduce
    string popped;
    do {
        popped = st.top();
        st.pop();
    } while (!st.empty() && getPrecedence(st.top(), popped) != '<');

    cout << "Reduce" << endl;
}
else if (relation == 'A')
{
    // Accept
```

```
            cout << "Parsing Successful!\n";
            return;
        }
        else
        {
            // Error
            cout << "Error: Invalid Syntax!\n";
            return;
        }
    }

    cout << "Parsing Failed!\n";
}

int main()
{
    vector<string> input = {"id", "+", "id", "*", "id"};
    cout << "Input Expression: ";
    for (const string& token : input)
    {
        cout << token << " ";
    }
    cout << endl;

    operatorPrecedenceParsing(input);

    return 0;
}
```

OUTPUT FILE

```
PS C:\Users\Acer\Desktop\Practicals\Compiler Design\Experiments> cd "c:\Users\Acer\Desktop\Practicals\Compiler Design\Experiment
s\" ; if ($?) { g++ Experiment_6.cpp -o Experiment_6 } ; if ($?) { .\Experiment_6 }
Input Expression: id + id * id

Parsing Steps:
Stack           Input           Action
--------------------------------------
$               id + id * id $          Shift: id
$ id            + id * id $             Reduce
$               + id * id $             Shift: +
$ +             id * id $               Shift: id
$ + id          * id $          Reduce
$ +             * id $          Shift: *
$ + *           id $            Shift: id
$ + * id                $               Reduce
$ + *           $               Reduce
$ +             $               Reduce
$               $               Parsing Successful!
```

FINDINGS AND LEARNINGS

1. Understood Operator Precedence Parsing and its rules.
2. Implemented shift and reduce operations based on operator precedence.
3. Used a stack to process input symbols and maintain parsing state.
4. Applied an operator precedence table to determine relations between symbols.

EXPERIMENT 7

AIM

Implementation of RECURSIVE DESCENT PARSER.

THEORY

A Recursive Descent Parser is a top-down parsing technique that processes input using recursive functions based on a given grammar. Each non-terminal in the grammar has a corresponding function that attempts to match and consume input symbols. This method is commonly used for parsing arithmetic expressions and programming languages.

Steps in Recursive Descent Parsing:

1. Read the first token from the input.
2. Call the function corresponding to the start symbol of the grammar.
3. Each function:
   - Tries to match the current token with expected terminals.
   - If a non-terminal is encountered, it recursively calls the respective function.
   - If a match is found, the token is consumed, and the next token is read.
   - If no match is found, an error is reported.
4. The process continues until the input is fully parsed or an error occurs.
5. If all tokens are successfully consumed, the input is valid; otherwise, it's rejected.

Example Grammar (Arithmetic Expressions):

$E \rightarrow T\ E'$

$E' \rightarrow +\ T\ E' \mid \varepsilon$

$T \rightarrow F\ T'$

$T' \rightarrow *\ F\ T' \mid \varepsilon$

$F \rightarrow (E) \mid id$

Where:

- E (Expression)
- E' (Expression Prime, for handling +)
- T (Term)
- T' (Term Prime, for handling *)
- F (Factor: (E) or id)

Parsing Example:

For input (id + id * id), the parser follows these derivations:

E → T E'

T → F T'

F → id

T' → ε

E' → + T E'

T → F T'

F → id

T' → * F T'

F → id

T' → ε

E' → ε

This method efficiently processes nested expressions and operator precedence using recursive function calls.

PROGRAM CODE

```
#include <iostream>
#include <string>

using namespace std;

string input;
int pos = 0;   // Position in Input String

// Function Prototypes
bool E();      // Expression
bool EPrime(); // Expression Prime
bool T();      // Term
bool TPrime(); // Term Prime
bool F();      // Factor
```

```
// Function to Match Expected Character
bool match(string expected)
{
    if (input.substr(pos, expected.length()) == expected)
    {
        pos += expected.length();
        return true;
    }
    return false;
}

// Function for E → T E'
bool E()
{
    cout << "E -> T E'\n";
    if (T())
    {
        return EPrime();
    }
    return false;
}

// Function for E' → + T E' | Epsilon
bool EPrime()
{
    if (match("+"))
    {
        cout << "E' -> + T E'\n";
        if (T())
```

```
        {
            return EPrime();
        }
        return false;
    }
    cout << "E' -> Epsilon\n";          // Prints ε
    return true;
}

// Function for T → F T'
bool T()
{
    cout << "T -> F T'\n";
    if (F())
    {
        return TPrime();
    }
    return false;
}

// Function for T' → * F T' | Epsilon
bool TPrime()
{
    if (match("*"))
    {
        cout << "T' -> * F T'\n";
        if (F())
        {
            return TPrime();
        }
```

```
        return false;
    }
    cout << "T' -> Epsilon\n";          // Prints ε
    return true;
}

// Function for F → (E) | id
bool F()
{
    if (match("("))
    {
        cout << "F -> (E)\n";
        if (E() && match(")"))
        {
            return true;
        }
        return false;
    }
    else if (match("id"))
    {
        cout << "F -> id\n";
        return true;
    }
    return false;
}

int main()
{
    cout << "Enter Expression (Use 'id' for identifier, E.g., id+id*id): ";
    cin >> input;
```

```
    if (E() && pos == input.length())
    {
        cout << "Parsing Successful! The Input is Valid.\n";
    }
    else
    {
        cout << "Parsing Failed! The Input is Invalid.\n";
    }

    return 0;
}
```

OUTPUT FILE

```
PS C:\Users\Acer\Desktop\Practicals\Compiler Design\Experiments> cd "c:\Users\Acer\Desktop\Practicals\Compiler Design\Experiment
s\" ; if ($?) { g++ Experiment_7.cpp -o Experiment_7 } ; if ($?) { .\Experiment_7 }
Enter Expression (Use 'id' for identifier, E.g., id+id*id): id+id*id
E -> T E'
T -> F T'
F -> id
T' -> Epsilon
E' -> + T E'
T -> F T'
F -> id
T' -> * F T'
F -> id
T' -> Epsilon
E' -> Epsilon
Parsing Successful! The Input is Valid.
```

FINDINGS AND LEARNINGS

1. Learned how Recursive Descent Parsing works by implementing a recursive function for each non-terminal.
2. Used match() function to validate input characters.
3. Implemented ε-productions by allowing optional function calls.
4. Understood how top-down parsing constructs a parse tree step by step.

EXPERIMENT 8

AIM

Implementation of CODE OPTIMIZATION TECHNIQUES.

THEORY

Code Optimization is a process used in Compiler Design to improve the efficiency of the generated code while maintaining its functionality.

Optimizations can be performed at various levels:

- Local Optimization: Optimizes small sections of code (basic blocks).
- Global Optimization: Optimizes across multiple blocks in a function.

Common Optimization Techniques:

- Constant Folding: Evaluate expressions with constants at compile time.
- Common Subexpression Elimination (CSE): Remove redundant computations.
- Dead Code Elimination: Remove unused or unnecessary code that doesn't affect output.
- Loop Invariant Code Motion: Move loop-invariant code outside of loops to reduce redundancy.
- Strength Reduction: Replace expensive operations with cheaper ones (E.g., x * 2 → x << 1).

Steps in Code Optimization:

1. Identify redundant operations in the intermediate representation (IR) of the code.
2. Apply local optimizations within basic blocks, such as constant folding and dead code elimination.
3. Perform global optimizations by analyzing relationships between multiple basic blocks.
4. Optimize loops using techniques like loop invariant code motion and strength reduction.
5. Generate optimized machine code while ensuring the semantic correctness of the program.

These techniques help reduce execution time, optimize memory usage, and improve overall performance.

PROGRAM CODE

```
#include <iostream>
#include <vector>
#include <string>
#include <map>

using namespace std;

// Function to Perform Constant Folding
string constantFolding(string expression)
{
    if (expression == "3 + 4") return "7";
    if (expression == "10 - 2") return "8";
    if (expression == "6 * 2") return "12";
    if (expression == "8 / 2") return "4";
    return expression;                         // Return Unchanged if no Folding Applied
}

// Function to Perform Common Subexpression Elimination (CSE)
vector<string> commonSubexpressionElimination(vector<string> &expressions)
{
    map<string, string> computedValues;
    vector<string> optimizedCode;

    for (string expr : expressions)
    {
        if (computedValues.find(expr) != computedValues.end())
        {
            optimizedCode.push_back(computedValues[expr] + " (CSE Applied)");
        }
```

```
        else
        {
            computedValues[expr] = expr;
            optimizedCode.push_back(expr);
        }
    }
    return optimizedCode;
}

// Function to Perform Dead Code Elimination
vector<string> deadCodeElimination(vector<string> &code)
{
    vector<string> optimizedCode;
    map<string, bool> usedVariables;

    // Simulating a Scenario where Some Variables are not Used
    usedVariables["x"] = true;
    usedVariables["y"] = false;                    // 'y' is Never Used

    for (string line : code)
    {
        if (line.find("y =") != string::npos && !usedVariables["y"])
        {
            continue;                              // Remove Dead Code
        }
        optimizedCode.push_back(line);
    }
    return optimizedCode;
}
```

```
// Function to Apply Strength Reduction (e.g., Replacing Multiplication with Shift)
string strengthReduction(string expr)
{
    if (expr == "x * 2") return "x << 1";
    if (expr == "y * 4") return "y << 2";
    return expr;
}

// Function to Apply Optimizations to a Given Code Segment
void optimizeCode(vector<string> code)
{
    cout << "\nOriginal Code:\n";
    for (string line : code) cout << line << endl;

    // Apply Constant Folding
    for (string &line : code)
    {
        line = constantFolding(line);
    }

    // Apply CSE
    code = commonSubexpressionElimination(code);

    // Apply Dead Code Elimination
    code = deadCodeElimination(code);

    // Apply Strength Reduction
    for (string &line : code)
    {
        line = strengthReduction(line);
```

```
    }

    cout << "\nOptimized Code:\n";
    for (string line : code) cout << line << endl;
}

int main()
{
    vector<string> code = {
        "x = 3 + 4",     // Constant Folding
        "y = 6 * 2",     // Constant Folding
        "z = x + y",
        "w = 6 * 2",     // Common Subexpression
        "x = x * 2",     // Strength Reduction
        "y = 10 - 2",    // Dead Code Elimination (y is Not Used)
    };

    optimizeCode(code);

    return 0;
}
```

OUTPUT FILE

```
PS C:\Users\Acer\Desktop\Practicals\Compiler Design\Experiments> cd "c:\Users\Acer\Desktop\Practicals\Compiler Design\Experiments\" ; if ($?) { g++ Experiment_8.cpp -o Experiment_8 } ; if ($?) { .\Experiment_8 }

Original Code:
x = 3 + 4
y = 6 * 2
z = x + y
w = 6 * 2
x = x * 2
y = 10 - 2

Optimized Code:
x = 3 + 4
z = x + y
w = 6 * 2
x = x * 2
```

FINDINGS AND LEARNINGS

1. Implemented Constant Folding to evaluate expressions at compile time.
2. Used Common Subexpression Elimination (CSE) to avoid redundant calculations.
3. Applied Dead Code Elimination to remove unnecessary assignments.
4. Replaced expensive operations using Strength Reduction.
5. Understood how optimizations enhance execution efficiency in compilers.

EXPERIMENT 9

AIM

Implementation of CODE GENERATOR.

THEORY

Code Generation is the final phase of a compiler, where high-level language instructions are converted into low-level assembly or machine code. The generated code must be optimized, efficient, and executable by the target machine.

Key Tasks in Code Generation:

- Instruction Selection: Convert IR operations into machine instructions.
- Register Allocation: Assign variables to CPU registers for faster execution
- Addressing Mode Selection: Choose efficient memory access techniques.
- Instruction Ordering: Arrange instructions to improve execution speed.
- Optimization: Apply techniques like constant folding and peephole optimization.

Steps in Code Generation:

1. Convert expressions from three-address code (TAC) to assembly-like instructions.
2. Allocate registers to reduce memory accesses and enhance performance.
3. Generate optimized instructions based on target architecture constraints.
4. Reorder instructions for efficient execution, reducing delays and stalls.
5. Output the final code in an assembly-like format for execution.

PROGRAM CODE

```
#include <iostream>

#include <vector>

#include <map>


using namespace std;


// Function to Generate Simple Assembly-like Code

void generateCode(vector<vector<string>> &TAC)

{

    map<string, string> registers;              // To Store Register Allocations
```

```
    int regCount = 0;

    cout << "Generated Code:\n";
    for (auto &instruction : TAC)
    {
        string op = instruction[0];
        string arg1 = instruction[1];
        string arg2 = instruction.size() > 2 ? instruction[2] : "";
        string result = instruction.back();

        if (op == "=")
        {
            cout << "MOV " << result << ", " << arg1 << endl;
        }
        else
        {
            string reg1 = (registers.count(arg1)) ? registers[arg1] : "R" + to_string(regCount++);
            string reg2 = (registers.count(arg2)) ? registers[arg2] : "R" + to_string(regCount++);
            registers[result] = "R" + to_string(regCount++);

            cout << op << " " << registers[result] << ", " << reg1 << ", " << reg2 << endl;
        }
    }
}

int main()
{
    // Example Three-Address Code (TAC)
    vector<vector<string>> TAC = {
        {"=", "5", "", "x"},
```

```
        {"=", "10", "", "y"},
        {"+", "x", "y", "z"},
        {"*", "z", "y", "w"}
    };

    generateCode(TAC);

    return 0;
}
```

OUTPUT FILE

```
PS C:\Users\Acer\Desktop\Practicals\Compiler Design\Experiments> cd "c:\Users\Acer\Desktop\Practicals\Compiler Design\Experiment
s\" ; if ($?) { g++ Experiment_9.cpp -o Experiment_9 } ; if ($?) { .\Experiment_9 }
Generated Code:
MOV x, 5
MOV y, 10
+ R2, R0, R1
* R4, R2, R3
```

FINDINGS AND LEARNINGS

1. Understood how high-level expressions are translated into low-level machine instructions.
2. Learned how to allocate registers efficiently to minimize memory access delays.
3. Practiced implementing basic code generation techniques for instruction selection and ordering.
4. Explored how three-address code (TAC) simplifies the transition from IR to machine code.
5. Developed an understanding of how compiler backends generate efficient executable code.