Karnatak Law Society's
GOGTE INSTITUTE OF TECHNOLOGY
Udyambag Belagavi -590008
Karnataka, India.

A Course Project Report on

Compiler Design

Submitted for the requirements of 5th semester B.E. in CSE

for **"Compiler Design (18CS62)"**

Submitted by

NAME	USN
1)Nimisha G J	2GI20CS074
2)Pranav D	2GI20CS091
3)Pratik D	2GI20CS093
4)Prathamesh B	2GI20CS094

Under the guidance of

Dr. Mallikarjun Math

Associate Professor, Dept. of CS

Academic Year 2022-2023 (Odd semester)

Karnatak Law Society's
GOGTE INSTITUTE OF TECHNOLOGY
Udyambag Belagavi -590008
Karnataka, India.

Department of Computer Science and Engineering

Certificate

This is to certify that the Course Project work titled **"Compiler Design"** carried out by **Nimisha G J, Pranav D,Pratik D,Prathamesh B** bearing **USNs:2GI20CS074, 2GI20CS091, 2GI20CS093, 2GI20CS094** for **Compiler Design (18CS62)** course is submitted in partial fulfilment of the requirements for 5th semester B.E. in **COMPUTER SCIENCE AND ENGINEERING,** Visvesvaraya Technological University, Belagavi. It is certified that all corrections/ suggestions indicated have been incorporated in the report. The course project report has been approved as it satisfies the academic requirements prescribed for the said degree.

Date: Signature of Guide

Place: Belagavi Dr. Mallikarjun Math

Asso., Prof., Dept. of CSE

KLS Gogte Institute of Technology, Belagavi

Karnatak Law Society's
GOGTE INSTITUTE OF TECHNOLOGY
Udyambag Belagavi -590008

Academic Year 2022-23 (Odd Semester)

Semester: V

Course:Compiler Design (18CS562)

Rubrics for evaluation of Course Project

S.No	Project / Seminar	Name→	Nimisha g j	Pranav D	Pratik D	Prathamesh B
		USN →	2GI20CS074	2GI20CS091	2GI20CS093	2GI20CS094
		Max. Marks				
1	Relevance of the project and its objectives	02				
3	Demonstration / Presentation	03				
4	Q and A	02				
5	Project Report	03				
	Total	**10**				

1.Engineering Knowledge: Apply the knowledge of mathematics, science, engineering fundamentals and an engineering specialization to the solution of complex engineering problems.
2.Problem Analysis: Identify, formulate, review research literature, and analyze complex engineering problems reaching substantiated conclusions using first principles of mathematics, natural sciences and Engineering sciences.
3.Design/Development of solutions: Design solutions for complex engineering problems and design system components or processes that meet the specified needs with appropriate consideration for the public health and safety, and the cultural, societal, and environmental considerations.

4.Conduct investigations of complex problems: Use research-based knowledge and research methods including design of experiments, analysis and interpretation of data, and synthesis of the information to provide valid conclusions.
5.Modern tool usage: Create, select, and apply appropriate techniques, resources, and modern engineering and IT tools including prediction and modeling to complex engineering activities with an understanding of the limitations.
6.The engineer and society: Apply reasoning informed by the contextual knowledge to assess societal, health, safety, legal and cultural issues and the consequent responsibilities relevant to the professional engineering practice.
7.Environment and sustainability: Understand the impact of the professional engineering solutions in societal and environmental contexts, and demonstrate the knowledge of, and need for sustainable development.
8.Ethics: Apply ethical principles and commit to professional ethics and responsibilities and norms of the engineering practice.
9.Individual and team work: Function effectively as an individual and as a member or leader in diverse teams, and in multidisciplinary settings.
10.Communication: Communicate effectively on complex engineering activities with the engineering community and with society at large, such as, being able to comprehend and write effective reports and design documentation, make effective presentations, and give and receive clear instructions.
11. Project management and finance: Demonstrate knowledge and understanding of the engineering management principles and apply these to one's own work, as a member and leader in a team, to manage projects and in multidisciplinary environments.
12. Life-long learning: Recognize the need for and have the preparation and ability to engage in independent and lifelong learning in the broadest context of technological channel

Index

PROBLEM STATEMENT

Design and develop a simple lexical analyser for c++ language using transition diagram.

INTRODUCTION

A lexical analyzer, also known as a lexer or scanner, is a crucial component in the compilation or interpretation of programming languages. Its primary purpose is to break down the input source code into a stream of tokens, which are meaningful units of the language. This report outlines the process of designing a lexical analyzer for the C++ programming language using a transition diagram. The transition diagram visually represents the states and transitions involved in tokenizing the input code.

I. Token Definition:

To design a lexical analyzer for C++, we need to identify the tokens present in the language. Some common tokens in C++ include:

Keywords (e.g., if, for, while)

Identifiers (variable names)

Literals (numbers, strings)

Operators (e.g., +, -, *, /)

Punctuation symbols (e.g., ,, ;, (,))

II. State Identification:

Based on the tokens identified, we determine the different states that the lexical analyzer will go through during the tokenization process. Each state represents a specific phase or aspect of lexical analysis. The following states are commonly encountered:

Initial state: Represents the starting point of the tokenization process.

Identifier state: Represents the recognition of identifiers (variable names).

Number state: Represents the recognition of number literals.

String state: Represents the recognition of string literals.

Operator state: Represents the recognition of operators.

Punctuation state: Represents the recognition of punctuation symbols.

Error state: Represents the detection of invalid or unrecognized input.

III. Transition Definition:

Once the states are identified, we establish the transitions between them based on the input characters. The transition diagram illustrates the flow between states triggered by specific input characters. Some examples of transitions include:

Transition from the initial state to the identifier state when encountering a letter.

Transition from the initial state to the number state when encountering a digit.

Transition from the initial state to the string state when encountering a quotation mark.

Transition from the initial state to the operator state when encountering an operator character.

Transition from the initial state to the punctuation state when encountering a punctuation symbol.

IV. Token Recognition:

In the transition diagram, specific states indicate the recognition of tokens. For example:

When the lexer reaches the identifier state, it recognizes that an identifier token has been formed.

Similarly, when the lexer reaches the number state, it recognizes that a number literal token has been formed.

This recognition allows the lexer to generate the appropriate token output for further processing or analysis.

V. Error Handling:

To handle invalid or unrecognized input, error states or transitions can be incorporated into the transition diagram. These error states provide the ability to detect and report lexical errors encountered during the tokenization process. Error handling can include transitions to the error state when unexpected or invalid input is encountered.

METHODOLOGY

The methodology for a lexical analyzer, also known as a lexer or scanner, involves a series of steps to tokenize the input source code into meaningful tokens. Here is a general methodology for implementing a lexical analyzer:

Define Token Types: Identify the different types of tokens in the C++ programming language, such as identifiers, literals (integers, floating-point numbers, strings), keywords, operators, punctuations, etc. Define these token types as constants or enumerations.

Create Lexer Class: Create a lexer class specific to C++ that will perform the tokenization process. The class should have a constructor that takes the source code as input and initializes the necessary variables.

Implement Tokenize Method: Implement a tokenize method in the lexer class. This method will scan the source code and generate tokens.

Define State Transitions: Define the state transitions for the lexer. This can be done using a transition table, state diagram, or a series of if-else conditions. Each state represents a stage in the tokenization process, and the transitions determine how the lexer moves from one state to another based on the current character.

Implement State Processing Methods: Implement methods corresponding to each state in the lexer. These methods will handle the processing logic for each state, such as building identifiers, literals, or detecting keywords/operators/punctuations.

Scan the Source Code: In the tokenize method, iterate over the characters of the source code and follow the state transitions to tokenize the input. Depending on the current state and the character being processed, invoke the corresponding state processing method to generate tokens or perform necessary actions.

Store Tokens: As tokens are generated, store them in a data structure (e.g., a list) within the lexer class.

Return Tokens: After the tokenization process is complete, return the generated tokens from the tokenize method.

Test and Refine: Test the lexer with different C++ source code samples, including edge cases, to ensure correct tokenization. Refine and optimize the implementation as needed.

By following this methodology, you can implement a lexical analyzer specifically for C++ to tokenize the source code accurately and generate tokens that can be utilized by a parser or other language processing tasks.

Transition diagram:

For identifiers:

For Identifiers :-

State 0:

```
C=GETCHAR();
if LETTER(c) then goto State 1
else FAIL()
```

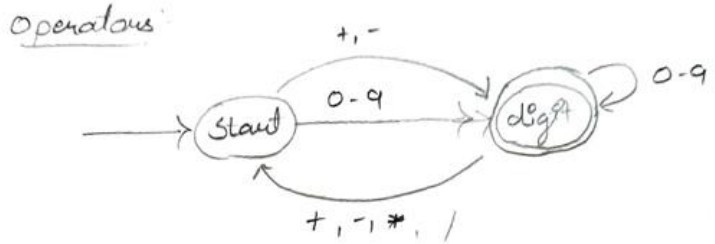
State 1:

```
C=GETCHAR();
if LETTER(c) OR DIGIT(c) then goto State 1
else if DELIMITER(c) then goto State 2
else FAIL()
```

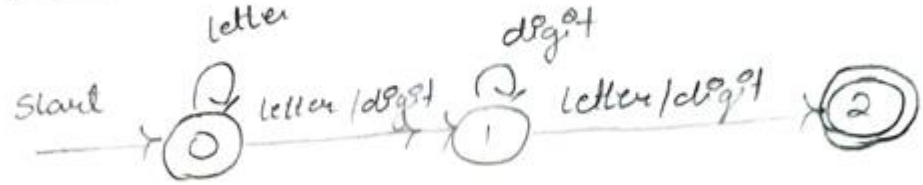
State 2:

```
RETRACT();
RETURN(ID,INSTALL())
```

For Operators:



For Literals:



Source code:

```
#include <fstream>
#include <iostream>
#include <stdlib.h>
#include <string.h>
#include <ctype.h>


using namespace std;

bool isPunctuator(char ch)                                  //check if the given character is a
punctuator or not
{
   if (ch == ' ' || ch == '+' || ch == '-' || ch == '*' ||
      ch == '/' || ch == ',' || ch == ';' || ch == '>' ||
      ch == '<' || ch == '=' || ch == '(' || ch == ')' ||
      ch == '[' || ch == ']' || ch == '{' || ch == '}' ||
      ch == '&' || ch == '|')
      {
         return true;
      }
   return false;
}

bool validIdentifier(char* str)                                 //check if the given
identifier is valid or not
{
   if (str[0] == '0' || str[0] == '1' || str[0] == '2' ||
      str[0] == '3' || str[0] == '4' || str[0] == '5' ||
      str[0] == '6' || str[0] == '7' || str[0] == '8' ||
      str[0] == '9' || isPunctuator(str[0]) == true)
      {
         return false;
      }                                                         //if first character of string is
a digit or a special character, identifier is not valid
   int i,len = strlen(str);
   if (len == 1)
   {
      return true;
   }                                                                  //if length is one,
validation is already completed, hence return true
   else
   {
   for (i = 1 ; i < len ; i++)                                  //identifier cannot contain
special characters
   {
      if (isPunctuator(str[i]) == true)
      {
         return false;
      }
   }
   }
   return true;
}
```

```
bool isOperator(char ch)                                    //check if the given
character is an operator or not
{
   if (ch == '+' || ch == '-' || ch == '*' ||
      ch == '/' || ch == '>' || ch == '<' ||
      ch == '=' || ch == '|' || ch == '&')
   {
     return true;
   }
   return false;
}

bool isKeyword(char *str)                                   //check if the given
substring is a keyword or not
{
   if (!strcmp(str, "if") || !strcmp(str, "else") ||
      !strcmp(str, "while") || !strcmp(str, "do") ||
      !strcmp(str, "break") ||  !strcmp(str, "continue")
      || !strcmp(str, "int") || !strcmp(str, "double")
      || !strcmp(str, "float") || !strcmp(str, "return")
      || !strcmp(str, "char") || !strcmp(str, "case")
      || !strcmp(str, "long") || !strcmp(str, "short")
      || !strcmp(str, "typedef") || !strcmp(str, "switch")
      || !strcmp(str, "unsigned") || !strcmp(str, "void")
      || !strcmp(str, "static") || !strcmp(str, "struct")
      || !strcmp(str, "sizeof") || !strcmp(str,"long")
      || !strcmp(str, "volatile") || !strcmp(str, "typedef")
      || !strcmp(str, "enum") || !strcmp(str, "const")
      || !strcmp(str, "union") || !strcmp(str, "extern")
      || !strcmp(str,"bool"))
      {
         return true;
      }
   else
   {
     return false;
   }
}

bool isNumber(char* str)                                    //check if the given
substring is a number or not
{
   int i, len = strlen(str),numOfDecimal = 0;
   if (len == 0)
   {
      return false;
   }
   for (i = 0 ; i < len ; i++)
   {
      if (numOfDecimal > 1 && str[i] == '.')
      {
         return false;
      } else if (numOfDecimal <= 1)
      {
```

```
            numOfDecimal++;
        }
        if (str[i] != '0' && str[i] != '1' && str[i] != '2'
            && str[i] != '3' && str[i] != '4' && str[i] != '5'
            && str[i] != '6' && str[i] != '7' && str[i] != '8'
            && str[i] != '9' || (str[i] == '-' && i > 0))
            {
                return false;
            }
    }
    return true;
}

char* subString(char* realStr, int l, int r)                    //extract the required substring from
the main string
{
    int i;

    char* str = (char*) malloc(sizeof(char) * (r - l + 2));

    for (i = l; i <= r; i++)
    {
        str[i - l] = realStr[i];
        str[r - l + 1] = '\0';
    }
    return str;
}


void parse(char* str)                                           //parse the expression
{
    int left = 0, right = 0;
    int len = strlen(str);
    while (right <= len && left <= right) {
        if (isPunctuator(str[right]) == false)                  //if character is a digit or an alphabet
            {
                right++;
            }

        if (isPunctuator(str[right]) == true && left == right)  //if character is a punctuator
            {
            if (isOperator(str[right]) == true)
            {
                std::cout<< str[right] <<" IS AN OPERATOR\n";
            }
            right++;
            left = right;
            } else if (isPunctuator(str[right]) == true && left != right
                || (right == len && left != right))             //check if parsed substring is a
keyword or identifier or number
            {
            char* sub = subString(str, left, right - 1);   //extract substring

            if (isKeyword(sub) == true)
                    {
```

```
                cout<< sub <<" IS A KEYWORD\n";
            }
        else if (isNumber(sub) == true)
            {
                cout<< sub <<" IS A NUMBER\n";
            }
        else if (validIdentifier(sub) == true
            && isPunctuator(str[right - 1]) == false)
            {
                cout<< sub <<" IS A VALID IDENTIFIER\n";
            }
        else if (validIdentifier(sub) == false
            && isPunctuator(str[right - 1]) == false)
            {
                cout<< sub <<" IS NOT A VALID IDENTIFIER\n";
            }

        left = right;
        }
    }
    return;
}

int main()
{
    char c[] = "#include <stdio.h>int find_largest_number(int a, int b, int c) {if (a >= b && a >= c)
{return a;} else if (b >= a && b >= c) {return b;} else {return c;}}int main(){int num1 = 10;int num2
= 20;int num3 = 15;int largest_number = find_largest_number(num1, num2, num3);printf(\"The
largest number is: %d\n\", largest_number);return 0;}";
    parse(c);
    return 0;
}
```

Input:

```
#include <stdio.h>

int find_largest_number(int a, int b, int c) {
   if (a >= b && a >= c) {
      return a;
   } else if (b >= a && b >= c) {
      return b;
   } else {
      return c;
   }
}

int main() {
   int num1 = 10;
   int num2 = 20;
   int num3 = 15;

   int largest_number = find_largest_number(num1, num2, num3);
   printf(\"The largest number is: %d\n\", largest_number);

   return 0;
}
```

Output:

```
#include IS A VALID IDENTIFIER
< IS AN OPERATOR
stdio.h IS A VALID IDENTIFIER
> IS AN OPERATOR
int IS A KEYWORD
find_largest_number IS A VALID IDENTIFIER
int IS A KEYWORD
a IS A VALID IDENTIFIER
int IS A KEYWORD
b IS A VALID IDENTIFIER
int IS A KEYWORD
c IS A VALID IDENTIFIER
if IS A KEYWORD
a IS A VALID IDENTIFIER
> IS AN OPERATOR
= IS AN OPERATOR
b IS A VALID IDENTIFIER
& IS AN OPERATOR
& IS AN OPERATOR
a IS A VALID IDENTIFIER
> IS AN OPERATOR
= IS AN OPERATOR
c IS A VALID IDENTIFIER
return IS A KEYWORD
a IS A VALID IDENTIFIER
else IS A KEYWORD
if IS A KEYWORD
b IS A VALID IDENTIFIER
> IS AN OPERATOR
= IS AN OPERATOR
a IS A VALID IDENTIFIER
& IS AN OPERATOR
& IS AN OPERATOR
b IS A VALID IDENTIFIER
> IS AN OPERATOR
```

```
= IS AN OPERATOR
c IS A VALID IDENTIFIER
return IS A KEYWORD
b IS A VALID IDENTIFIER
else IS A KEYWORD
return IS A KEYWORD
c IS A VALID IDENTIFIER
int IS A KEYWORD
main IS A VALID IDENTIFIER
int IS A KEYWORD
num1 IS A VALID IDENTIFIER
= IS AN OPERATOR
10 IS A NUMBER
int IS A KEYWORD
num2 IS A VALID IDENTIFIER
= IS AN OPERATOR
20 IS A NUMBER
int IS A KEYWORD
num3 IS A VALID IDENTIFIER
= IS AN OPERATOR
15 IS A NUMBER
int IS A KEYWORD
largest_number IS A VALID IDENTIFIER
= IS AN OPERATOR
find_largest_number IS A VALID IDENTIFIER
num1 IS A VALID IDENTIFIER
num2 IS A VALID IDENTIFIER
num3 IS A VALID IDENTIFIER
printf IS A VALID IDENTIFIER
"The IS A VALID IDENTIFIER
largest IS A VALID IDENTIFIER
number IS A VALID IDENTIFIER
is: IS A VALID IDENTIFIER
%d
" IS A VALID IDENTIFIER
```

Conclusion:

In conclusion, implementing a lexical analyzer for the C++ programming language requires a well-defined methodology and careful consideration of the language's syntax and token types. By following the outlined steps of defining token types, creating a lexer class, implementing the tokenize method, defining state transitions, implementing state processing methods, scanning the source code, storing and returning tokens, and thorough testing, a robust and efficient lexical analyzer can be developed.

The lexical analyzer plays a crucial role in the compilation or interpretation process by breaking down the source code into meaningful tokens. These tokens serve as the building blocks for subsequent stages of language processing, such as parsing and semantic analysis. A properly implemented lexer simplifies the subsequent stages by transforming the source code into a token stream that can be easily processed and analyzed.

During the implementation process, it is essential to handle various token types accurately, including keywords, identifiers, literals, operators, and punctuations. Designing appropriate state transitions and state processing methods ensures that the lexer can recognize and generate tokens correctly. Additionally, incorporating error handling mechanisms allows for the identification and reporting of lexical errors, enhancing the overall reliability of the lexer.

Thorough testing is crucial to ensure the correctness and robustness of the lexer implementation. Testing should encompass a wide range of source code samples, including edge cases and complex scenarios, to validate the lexer's ability to tokenize the code accurately under various conditions.

By adhering to this methodology and giving careful consideration to the specific requirements of the C++ language, developers can build a reliable and efficient lexical analyzer. This foundational component sets the stage for further language processing tasks and the development of compilers, interpreters, and other language-related tools in the C++ ecosystem.