QUICK COMPILER(Semantic Analyzer)

18CSC304J COMPLIER DESIGN

MINI PROJECT REPORT

Submitted by :
AVANITH K.(RA2011026010073)
UPAMANYU GHOSH(RA2011026010076)

Under the guidance of

Dr. J. Jeyasudha
Assistant Professor, CINTEL

In partial satisfaction of the requirements for the degree of

BACHELOR OF TECHNOLOGY in

COMPUTER SCIENCE & ENGINEERING
With specialization in AI and Machine Learning

SRM INSTITUTE OF SCIENCE AND TECHNOLOGY

Kattankulathur, Chengalpattu District - 603203

MAY 2023

COLLEGE OF ENGINEERING & TECHNOLOGY
SRM INSTITUTE OF SCIENCE & TECHNOLOGY
S.R.M. NAGAR, KATTANKULATHUR – 603 203

BONAFIDE CERTIFICATE

Certified that this project report **" Quick compiler – Semantic Analyser"** is the bonafide work of **"Avanith.K (RA2011026010073), Upamanyu Ghosh (RA2011026010076)"** who carried out under my supervision. Certified further, that to the best of my knowledge the work reported herein does not form part of any other work.

Signature	**Signature**
Dr. J. Jeyasudha	Dr. Annie Uthra
Assistant Professor	Head of Department
Department of Computational Intelligence	Department of Computational Intelligence

ABSTRACT

A Java compiler is a software tool that translates Java source code into machineexecutable bytecode. The compilation process typically consists of several phases that are performed sequentially. The first phase is lexical analysis, which involves breaking the source code into tokens. The second phase is syntax analysis, where the compiler checks the program's structure and creates a syntax tree. The third phase is semantic analysis, where the compiler checks the program's meaning, including type checking and scope rules. The fourth phase is intermediate code generation, where the syntax tree is transformed into an intermediate representation. The fifth phase is code optimization, which applies various optimization techniques to the intermediate code to improve efficiency. The sixth phase is code generation, where the final machine code is produced. Throughout the compilation process, a symbol table is maintained to store information about the program's symbols. Understanding the phases of a Java compiler is essential for efficient and effective Java programming.

TABLE OF CONTENTS

CHAPTER 1

INTRODUCTION

1.1 Introduction

A compiler is a program that translates source code written in a high-level programming language into machine code that can be executed by a computer. The compilation process involves several phases, each of which performs a specific task.

Lexical Analysis: In this phase, the source code is broken down into tokens, which are the smallest units of meaning in a programming language. This phase is also known as scanning.

Syntax Analysis: In this phase, the tokens are analyzed to ensure that they conform to the rules of the programming language's grammar. This phase is also known as parsing. Semantic Analysis: In this phase, the compiler checks the meaning of the source code to ensure that it is semantically correct. This includes checking for type errors, undeclared variables, and other semantic errors.

Intermediate Code Generation: In this phase, the compiler generates an intermediate representation of the source code that is easier to work with than the source code itself. This intermediate representation is often in the form of a low-level language, such as assembly language.

Code Optimization: In this phase, the intermediate code is analyzed and optimized to make it faster and more efficient.

Code Generation: In this final phase, the optimized intermediate code is translated into machine code that can be executed by a computer.

1.2 Problem Statement

To implement the phases of compiler explores the key stages involved in transforming source code into an executable form, highlighting the purpose and significance of each phase. Develop a compiler for a new programming language that includes all the necessary phases of compilation, including lexical analysis, syntax analysis, semantic analysis, intermediate code generation, code optimization, and code generation. The compiler should be able to translate the source code of the programming language into efficient machine code that can be executed on a specific target architecture. The compiler should also be able to handle common programming constructs such as loops, conditional statements, and functions, and should provide informative error messages in case of syntax or semantic errors in the source code. Finally, the compiler should be designed to handle large and complex programs with reasonable efficiency and scalability.

1.3 Objectives

To break down the source code into tokens and remove any irrelevant or unnecessary characters.

Goals:

Identify and categorize the lexemes into tokens based on language-specific rules.

Remove comments, whitespace, and other irrelevant characters.

Generate an organized token stream for further processing.

1.4 Need for Semantic Analyser

Semantic Analysis is the third phase of Compiler. Semantic Analysis makes sure that declarations and statements of program are semantically correct. It is a collection of procedures which is called by parser as and when required by grammar. Both syntax tree of previous phase and symbol table are used to check the consistency of the given code. **Type checking** is an important part of semantic analysis where compiler makes sure that each operator has matching operands.

It uses syntax tree and symbol table to check whether the given program is semantically consistent with language definition. It gathers type information and stores it in either syntax tree or symbol table. This type information is subsequently used by compiler during intermediate-code generation.

1.5 Requirement Specification

Lexical Analysis:

Tokenization:

Define the rules for recognizing and categorizing tokens in the source code.Specify the supported token types, such as keywords, identifiers, operators, literals, and punctuation symbols.Ensure accurate handling of language-specific features and syntax rules.Handle error detection and reporting, including illegal characters and unrecognized lexemes.

Syntax Analysis:

Parsing:

Define the grammar rules for the programming language being compiled.Specify the supported language constructs, including statements, expressions, and declarations.Ensure accurate construction of parse trees or abstract syntax trees (AST) representing the code's structure.Detect and report syntax errors, such as mismatched parentheses or incorrect use of language constructs.

Semantic Analysis:

Type Checking:

Define the type system rules for the programming language.Perform type inference and checking to ensure compatibility and consistency of data types.Validate variable declarations, scoping rules, and identifier usage.Detect and report semantic errors, such as undeclared variables or type mismatches.

Intermediate Code Generation:

Intermediate Representation:

Define the intermediate representation to be used (e.g., three-address code, bytecode). Specify the transformation rules for generating the intermediate code from the source code. Preserve the semantics of the original code while simplifying it for optimization.Ensure platform independence and compatibility with subsequent compilation phases.

Optimization:

Optimization Techniques:

Specify the optimization techniques to be applied, such as constant folding, loop unrolling, or dead code elimination.Define the rules and algorithms for identifying optimization opportunities in the intermediate representation.Optimize the code for factors like execution time, memory usage, and code size.Ensure the preservation of correctness and adherence to the language semantics during optimization.

Code Generation:

Target Machine Code:

Specify the target platform or architecture for code generation.Define the translation rules and algorithms for generating machine code or assembly language instructions.Handle platform-specific considerations, such as register allocation, memory management, and calling conventions.Ensure the generated code is efficient, correct, and compatible with the target platform.

CHAPTER 2

NEEDS

2.1 Need of Compiler Design

- Language Portability: A compiler enables the execution of a high-level programming language on different platforms and architectures. By translating source code into machine code or an intermediate representation, compilers allow programs to run on diverse systems without requiring manual rewriting or modification.
- Efficiency and Optimization: Compilers play a crucial role in optimizing the performance of programs. They analyze the source code and apply various optimization techniques, such as constant folding, loop unrolling, and dead code elimination, to generate efficient machine code. By optimizing the code, compilers enhance execution speed, reduce memory usage, and improve overall program efficiency.
- Error Detection and Reporting: Compilers perform thorough analysis of the source code, detecting and reporting syntax errors, semantic errors, and other programming mistakes. By identifying errors early in the development process, compilers help programmers locate and fix issues before the program is executed. This results in more robust and reliable software.
- Language Features and Abstractions: High-level programming languages provide powerful features and abstractions that simplify complex tasks. Compilers are responsible for translating these language features into low-level instructions that can be executed by the underlying hardware. Compiler design ensures accurate implementation of language constructs, such as control flow statements, data structures, and object-oriented concepts.
- Standardization and Consistency: Compilers play a vital role in enforcing language standards and ensuring consistent behavior across different implementations. They adhere to the defined syntax, semantics, and rules of a programming language, promoting interoperability and reducing language-specific inconsistencies. Compiler design ensures that programs written in the same language produce the same results, regardless of the compiler used.

- Productivity and Development Speed: Compilers enable developers to write programs in high-level languages, which are more expressive and easier to understand compared to low-level languages. This improves productivity and reduces development time, as programmers can focus on the logical structure of their code rather than low-level implementation details.
- Code Reusability and Modularity: Compilers facilitate code reusability by providing mechanisms such as libraries and modules. They enable the separation of code into reusable components, promoting modularity and maintainability. By supporting modular design, compilers help create scalable and maintainable software systems.
- Language Innovation and Evolution: Compiler design allows for the introduction of new language features and enhancements. As programming languages evolve, compilers play a key role in implementing these changes and ensuring backward compatibility. Compiler designers continuously improve and adapt their tools to support new language standards and emerging technologies.

2.2 Limitations of CFGs

- Inability to handle context-sensitive languages: CFGs are unable to generate or recognize languages that require context-sensitive rules. These languages have complex grammatical structures that depend on the context in which a particular symbol appears. Examples of context-sensitive languages include nested parentheses, balanced brackets, and palindrome strings.
- Lack of semantic information: CFGs do not capture the semantic information of a language, such as the meaning of words, the context of use, or the intent of the speaker. CFGs focus only on the syntactic structure of a language and are incapable of representing the complex relationships between words, phrases, and concepts.
- Limited expressiveness: CFGs have limited expressiveness when it comes to describing complex grammars. Some grammatical structures, such as recursion, left-recursion, and indirect left-recursion, cannot be captured by CFGs. Additionally, CFGs are unable to describe the constraints on the order or occurrence of symbols in a language.
- Inability to handle ambiguous languages: CFGs can produce ambiguous grammars, where a single string of symbols can be generated by multiple derivations. These ambiguities can lead to parsing errors, where the same input string can be interpreted in

different ways. CFGs are unable to resolve such ambiguities, and additional techniques such as disambiguation algorithms may be needed.

- Computational complexity: The parsing of CFGs can be computationally expensive, particularly for languages with complex grammars. The worst-case time complexity for parsing a CFG is O(n^3), where n is the length of the input string. This complexity can become a significant bottleneck for large input strings or when parsing is required in real-time applications.

2.3 Types of Attributes

- Synthesized Attributes:

Synthesized attributes are computed or derived from the attributes of a node's children in a parse tree or syntax tree.They carry information that is determined during the bottom-up construction of the tree.Synthesized attributes flow from the child nodes to the parent node.These attributes are typically used to propagate information about expressions, types, values, and intermediate computation results.

- Inherited Attributes:

Inherited attributes are passed from the parent node to its child nodes in a parse tree or syntax tree.They carry information that is needed by a node's children to compute their own attributes.Inherited attributes are typically used to propagate information about the context or environment in which the node appears.

CHAPTER 3

SYSTEM & ARCHITECTURE DESIGN

3.1 System Architecture Components

- Source Code Input:

  This component handles the input of the source code to be analyzed. It can take input from various sources, such as files, user input, or network streams.

- Lexical Analysis Engine:

  The lexical analysis engine is the core component responsible for performing the tokenization of the source code.It scans the input source code character by character and groups them into tokens based on predefined lexical rules and regular expressions.The engine identifies keywords, identifiers, literals, operators, and other language-specific tokens.It may use techniques like finite automata, regular expressions, or lexemes to efficiently tokenize the input.

- Token Buffer:

  The token buffer is a data structure used to store the tokens generated by the lexical analysis engine.It provides a temporary storage for tokens before they are consumed by the parser or other components in the compilation process.The buffer can be implemented as a queue or a list, allowing tokens to be retrieved in the order they were generated.

- Symbol Table:

  The symbol table is a data structure that keeps track of identifiers encountered during the lexical analysis.It stores information about each identifier, such as its name, data type, scope, and memory location.The symbol table is used by subsequent phases of the compiler, such as the semantic analysis and code generation stages.

- Error Handling and Reporting:

  This component detects and handles lexical errors, such as invalid characters, unrecognized tokens, or syntax violations.It generates informative error messages, highlighting the location of the error in the source code.The error handling and reporting component may include mechanisms for error recovery and resynchronization to continue lexical analysis after encountering an error.

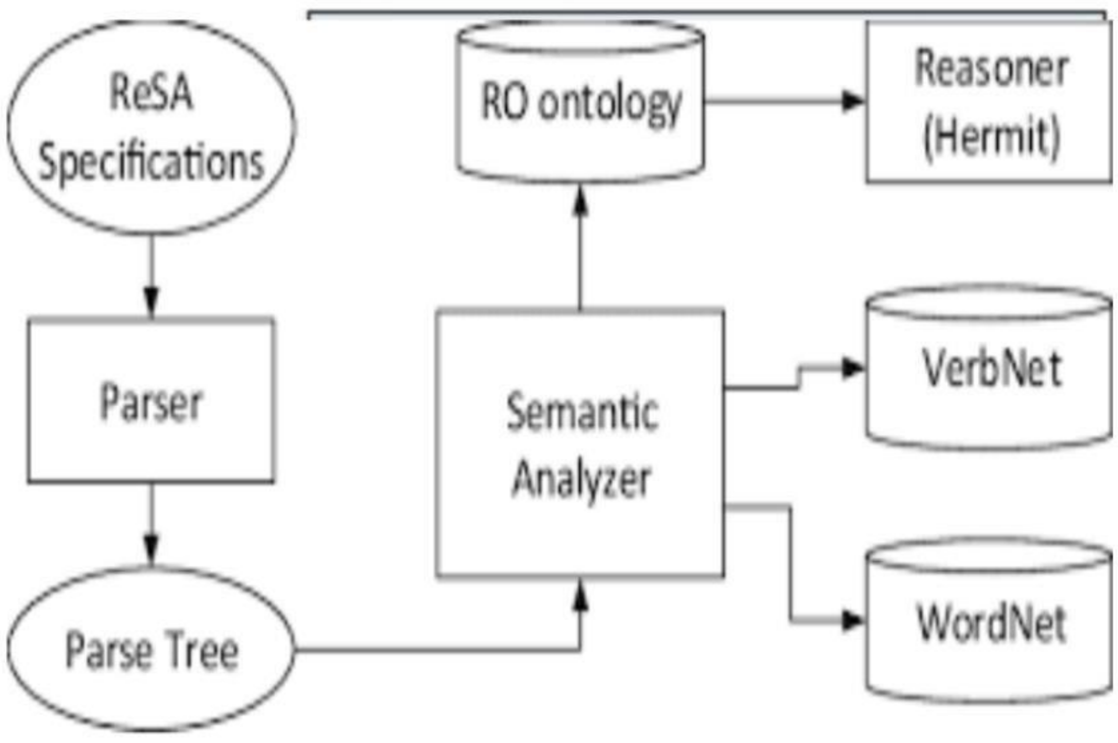
  Output:

  The output component of the lexical analyzer system architecture delivers the generated tokens and any associated information to the next phase of the compiler.It may provide an interface or API for accessing the tokens and their attributes.The output can be consumed by the parser, semantic analyzer, or other components involved in the compilation process.

- Integration with Compiler Pipeline:

  The lexical analyzer is typically one of the initial stages in the compiler pipeline.It interfaces with other components, such as the parser, semantic analyzer, and code generator, to provide them with the token stream and necessary lexical information.

3.2 Architecture Diagram



CHAPTER 4

REQUIREMENTS

4.1 Technologies Used

The technologies used for the different phases of a compiler may vary depending on the design and implementation of the compiler. Here are some common technologies used for each phase of a compiler:

- Lexical Analysis:

  Regular expressions for defining the lexical rules of the programming languageFinite Automata (FA) or Deterministic Finite Automata (DFA) for recognizing and tokenizing the input stream.Lexical Analyzer Generators such as Flex or JFlex for automatically generating the lexical analyzer code based on the lexical rules.

- Syntax Analysis:

  Context-Free Grammars (CFGs) for defining the syntax rules of the programming language Parsing techniques such as LL(k), LR(k), or Recursive Descent for constructing a parse tree or an Abstract Syntax Tree (AST) from the token stream Parser Generators such as Bison or ANTLR for automatically generating the parser code based on the syntax rules

- Semantic Analysis:

  Symbol table data structures for storing and managing the program's symbols and their propertiesType checking algorithms for verifying the compatibility of the types used in the program.Intermediate Code Generation techniques such as Three-Address Code or Quadruple Code for generating an intermediate representation of the program.

- Optimization:

Control Flow Graph (CFG) for analyzing the control flow of the program.Data Flow Analysis techniques such as Constant Propagation, Dead Code Elimination, and Loop Optimization for optimizing the intermediate code.Machine Code Generation techniques such as Register Allocation, Instruction Selection, and Peephole Optimization for generating efficient machine code.

- Code Generation:

Code Generation algorithms that translate the intermediate code into the target machine code.Assembly Language or Machine Language programming for generating low-level code that can be executed by the target hardware.Code Generation Tools such as LLVM or GCC for generating machine code from the intermediate representation.

- Regular Expressions:

Regular expressions are used to define the lexical rules of a programming language. A regular expression specifies a pattern that matches a set of strings, which can be used to identify tokens in the input program.

- Finite Automata (FA) or Deterministic Finite Automata (DFA):

Finite automata are used to recognize and tokenize the input stream. A finite automaton is a mathematical model of a machine that can recognize a regular language. DFA is a type of FA that has a unique transition for each symbol in the input alphabet.

- Lexical Analyzer Generators:

Lexical analyzer generators are software tools that automate the generation of lexical analyzers from a set of regular expressions. Some common lexical analyzer generators used in compiler design include:

  - Flex:

    Flex is a lexical analyzer generator that generates C code for the lexical analyzer based on a set of regular expressions.

- JFlex:

  JFlex is a lexical analyzer generator that generates Java code for the lexical analyzer based on a set of regular expressions.

- Symbol Tables:

  Symbol tables are data structures that are used to store information about identifiers (e.g., variable names, function names) encountered in the input program. The symbol table is typically populated during the lexical analysis phase.

In summary, the technologies used for lexical analysis in compiler design include Regular Expressions, Finite Automata, Lexical Analyzer Generators, and Symbol Tables.

In summary, the technologies used for different phases of a compiler can include Regular Expressions, Finite Automata, Context-Free Grammars, Symbol Tables, Type Checking, Intermediate Code Generation, Control Flow Analysis, Data Flow Analysis, Machine Code Generation, Assembly Language, and Code Generation Tools.

4.2 Requirements to run the script

The requirements to run a script for compiler design can vary depending on the specific script and the language being used. However, in general, you will need the following:

- A compatible operating system: Make sure the operating system you are using is compatible with the language and tools you will be using. For example, if you are using a script written in Python, make sure you have a version of Python installed that is compatible with your operating system.
- Required software and libraries: Check if the script has any specific software or library dependencies that need to be installed before running it.
- Input files: Prepare any input files that the script requires for testing or running. These may include source code files, configuration files, or input data files.
- Command-line interface (CLI) or Integrated Development Environment (IDE): Decide whether you will be running the script from the command line or using an IDE. Make sure you have the necessary tools installed and configured properly.

- Access to the source code: If the script is open-source, make sure you have access to the source code and any documentation that may be necessary to understand how to use the script.
- System resources: Depending on the size and complexity of the script, you may need sufficient system resources such as memory, processing power, and storage space to run it efficiently.

Overall, the requirements to run a script for compiler design will depend on the specific script and language being used, so it is important to carefully read the documentation and instructions provided with the script before attempting to run it.

CHAPTER 5

CODING & TESTING

5.1 Coding

```
import parse from "./parser.js";
import { tokenize, lex } from "./tokenizer.js";

function analyze(input, isFile) {   if
(parse(tokenize(lex(input, isFile)))) {
input = lex(input);     if (input.length
=== 3) return true;     try {       const
dataType = input[0];
      const value = input[3];
       if
(
        dataType           ===           "int"           &&
!value.includes(".") &&
        !value.includes("'") &&
        !value.includes("")
      )         return
true;       else if
(
        (dataType     ===     "double"     ||     dataType     ===     "float")     &&
value.includes(".") &&
        !value.includes("'") &&
        !value.includes("")
      )         return true;       else if (dataType === "String" &&
value.includes("")) return true;       else if (
        dataType === "boolean" &&
        (value.includes("true") || value.includes("false")) &&
        !value.includes(".") &&
        !value.includes("'") &&
        !value.includes('"')
      )         return true;       else if (dataType === "char" && value.length === 3
&& value.includes("'"))         return true;     } catch (e) {       return false;
    }
}
  return false;
}

export default analyze;
parser.js function
parse(tokens) {   const
correctSyntax = [
    [
      "<data_type>",
      "<identifier>",
      "<assignment_operator>",
      "<value>",
      "<delimiter>",
    ],
    ["<data_type>", "<identifier>", "<delimiter>"],
  ];

  let state = false;

  if (tokens.length > correctSyntax[0].length) return false;
  for (let syntax of correctSyntax) {
for (let j = 0; j < syntax.length; j++) {
try {         state = tokens[j] ===
syntax[j];         if (!state) break;       }
catch (e) {         state = false;
```

```
      }     }
if (state) {
break;
    }   }
return state;
}

export default parse;
```

tokenizer.js

```
function tokenize(lexemes) {
  const dataTypes = ["int", "double", "char", "String", "float", "boolean"];   const
tokens = [];

  for (const lexeme of lexemes) {     if
(dataTypes.includes(lexeme)) {
tokens.push("<data_type>");     } else if
(lexeme.includes("=")) {
tokens.push("<assignment_operator>");
    } else if (
lexeme.includes("") ||
lexeme.includes(""") ||
!isNaN(lexeme.charAt(0)) ||
lexeme.includes(".") ||
lexeme === "true" ||       lexeme
=== "false"
    ) {
      tokens.push("<value>");     }
else if (lexeme.includes(";")) {
tokens.push("<delimiter>");
    }                    else                    {
tokens.push("<identifier>");
```

```
    }
  }
   return tokens;
  }
  function lex(input, isFile) {
   const individualChars = input.split("");

   const lexemes = [];

   let temp = "",
     quotedString = "";

   let isQuote = false;

   for (const c of individualChars) {
if (c === "=" && !isQuote) {
lexemes.push(temp);
lexemes.push(c);       temp = "";
    } else if (c === ";" && !isQuote) {
lexemes.push(temp);
lexemes.push(c);       temp = "";
    } else if (c === " " && !isQuote) {
lexemes.push(temp);       temp = "";
} else if (c === '"') {
quotedString += c;       if (isQuote) {
lexemes.push(temp);         temp = "";
lexemes.push(quotedString);
quotedString = "";         isQuote =
false;       } else {         isQuote =
true;
      }
    } else if (isQuote) {
quotedString += c;
} else {       temp +=
c;
    }
  }
  lexemes.push(temp);   if (isFile)
lexemes.pop(); // remove /n   return
lexemes.filter((n) => n !== "");
}

export { tokenize, lex };
```

App.js

```
import Footer from "./components/Footer.jsx"; import
Home from "./components/Home.jsx";
import Navbar from "./components/Navbar.jsx";



function      App()      {
return (
    <>
      <Navbar />
      <Home />
      <Footer />
    </>
  );
}

export default App;
```

index.js

```
import React from "react"; import
ReactDOM from "react-dom/client"; import
App from "./App";

const root = ReactDOM.createRoot(document.getElementById("root")); root.render(
  <React.StrictMode>
    <App />
  </React.StrictMode>
);
```

Information.jsx

```
import { Alert } from "@mui/material";

function      Information()      {
return (
    <Alert icon={false} severity="info">
      {"CD project Q1"}
    </Alert>
  );
}

export default Information;
```

IOBox.js

```
import React from "react";
import { Card, Typography } from "@mui/material";

export default function IOBox(props) {
  return      (
<>
      <center>
```

```
        <Typography variant="h5" fontWeight="600" sx={{ marginTop: "2%" }}>
          {props.title}
        </Typography>
        <Card variant="outlined" sx={{ padding: "3%", margin: "1% 0% 2%" }}>
          <Typography fontFamily="roboto" fontSize="1.5rem" className="ioBox">
            {props.text}
          </Typography>
        </Card>
      </center>
    </>
  );
}
```

Navbar.js

```
import { Container, Typography } from "@mui/material";

function Navbar() {
  return (     <Container
sx={{         margin: "4% auto
2% auto",
      }}
    >
      <Typography textAlign="center" fontSize={"4.5rem"} fontWeight="600">
        {"Phases of Compiler"}
      </Typography>
    </Container>
  );
}

export default Navbar;
```

Home.jsx

```
import { React, Component } from "react";
import {   Button,
  Input,
  Container,
  Paper,
  Grid,
  TextField,
} from "@mui/material/"; import ArrowRight from
"@mui/icons-material/ArrowRightAlt"; import ClearIcon from
"@mui/icons-material/Clear"; import AnalyzerIcon from
"@mui/icons-material/Spellcheck"; import TokenizerIcon from
"@mui/icons-material/Toll"; import Information from
"./Information"; import IOBox from "./IOBox"; import { lex,
tokenize } from "../utils/tokenizer"; import ParserIcon from
"@mui/icons-material/ManageSearch";
import parse from "../utils/parser";
import analyze from "../utils/analyze";

class Home extends Component {
constructor(props) {     super();
this.state = {       inputText: "",
outputText: "",
toggleTokenizer: "disabled",
toggleParser: "disabled",
toggleAnalyzer: "disabled",
toggleInput: true,
toggleClear: "disabled",
isFile: false,       toggleTextField:
false,
    };
}
```

```
  readFile = async (e) => {
e.preventDefault();     const reader
= new FileReader();
reader.onload = async (e) => {
const text = e.target.result;
this.setState({         inputText: text,
toggleTokenizer: "contained",
toggleInput: false,
toggleClear: "contained",
isFile: true,         toggleTextField:
true,
      });
    };
    reader.readAsText(e.target.files[0]);
  };

  render    =    ()    =>    {
return (
      <>
        <Container>
          <Paper elevation={10} sx={{ padding: "6%", margin: "2% 0% 2%" }}>
            <Information />             <TextField
variant="outlined"               placeholder={'String str
= "Hello World !";'}
disabled={this.state.toggleTextField}
fullWidth               label="Enter single line code"
margin="normal"               onChange={(e) => {
this.setState({                   inputText: e.target.value,
toggleTokenizer: "contained",
toggleClear: "contained",                   toggleInput:
false,                   toggleParser: "disabled",
toggleAnalyzer: "disabled",
                });
              }}
/>
            <Grid               container
spacing={1}
justifyContent="center"
alignItems="center"
            >
              <Grid item>
                {this.state.toggleInput ? (
                  <Input type="file" onChange={(e) => this.readFile(e)} />
                ) : (
                  <Input type="file" disabled />
                )}
</Grid>

              <Grid item>                 <Button
variant={this.state.toggleClear}
color="error"                   onClick={() =>
window.location.reload()}
                >
                  <ClearIcon /> Clear
                </Button>
              </Grid>
</Grid>

            <Grid               container
spacing={1}
justifyContent="center"
alignItems="center"               sx={{
marginTop: "1%" }}
            >
```

```
          <Grid item>            <Button
variant={this.state.toggleTokenizer}
endIcon={<TokenizerIcon />}              onClick={()
=>                this.setState({
outputText: tokenize(
lex(this.state.inputText, this.state.isFile)
                ),
                toggleParser:                     "contained",
toggleTokenizer: "disabled",
               })
             }
           >
             Lexical Analysis
           </Button>
</Grid>

          <Grid item>
           <ArrowRight sx={{ padding: "1%" }} />
          </Grid>

          <Grid item>            <Button
variant={this.state.toggleParser}
endIcon={<ParserIcon />}              onClick={() => {
this.setState({                 outputText: String(
parse(                    tokenize(lex(this.state.inputText,
this.state.isFile))                  )
                   ? "The syntax is correct!"
                   : "The syntax is incorrect!"
                ),                 });                 parse(
tokenize(lex(this.state.inputText, this.state.isFile))
               ) === true
```

```
                    ? this.setState({
toggleAnalyzer: "contained",
toggleParser: "disabled",
                      })
                    : this.setState({ toggleAnalyzer: "disabled" });
                }}
              >
                Syntax Analysis
              </Button>
</Grid>

             <Grid item>
               <ArrowRight sx={{ padding: "1%" }} />
             </Grid>

             <Grid item>                <Button
variant={this.state.toggleAnalyzer}
endIcon={<AnalyzerIcon />}
onClick={() => {                    this.setState({
outputText: analyze(
this.state.inputText,
this.state.isFile
                    )
                      ? "This is semantically correct!"
: "This is semantically incorrect!",
toggleAnalyzer: analyze(
this.state.inputText,
this.state.isFile
                    )
                      ? "disabled"
                      : "contained",
                  });
                  }}
                >
                  Semantic Analysis
                </Button>
              </Grid>
</Grid>

            <IOBox title="Input" text={this.state.inputText} />
            <IOBox      title="Output"      text={this.state.outputText}      />
</Paper>
        </Container>
      </>
    );
  };
}

export default Home; Footer.jsx

import { Stack, Link, Container, Button, Typography } from "@mui/material";

function      Footer()      {
return (
    <Container>
      <Stack direction="row" justifyContent="center">

        <Typography  color="gray"  sx={{  margin:  "2%"  }}>
Reg No:
          RA2011026010060
          RA2011026010076
          RA2011026010082
          RA2011026010099
          RA2011026010113
        </Typography>
      </Stack>
    </Container>
  );
}

export default Footer;
```

5.2 Testing

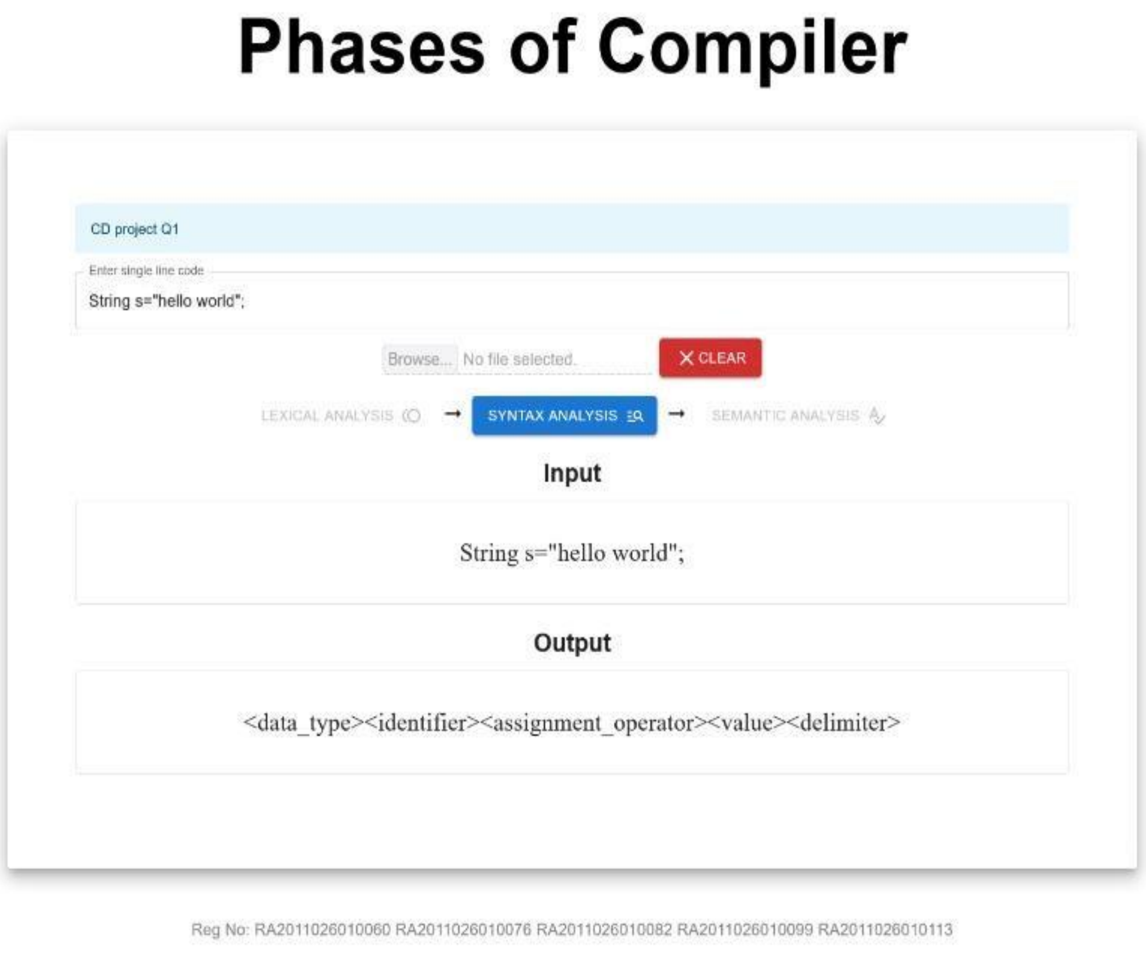
Input code: class

```
Main {

  public static void main(String[] args) {

    int first = 10;
    int second = 20;

    // add two numbers     int
sum = first + second;
    System.out.println(first + " + " + second + " = "  + sum);
  }
}
```

CHAPTER 6

OUTPUT &RESULT

6.1 Output


Phases of Compiler

CD project Q1

Enter single line code

String s="hello world";

Input

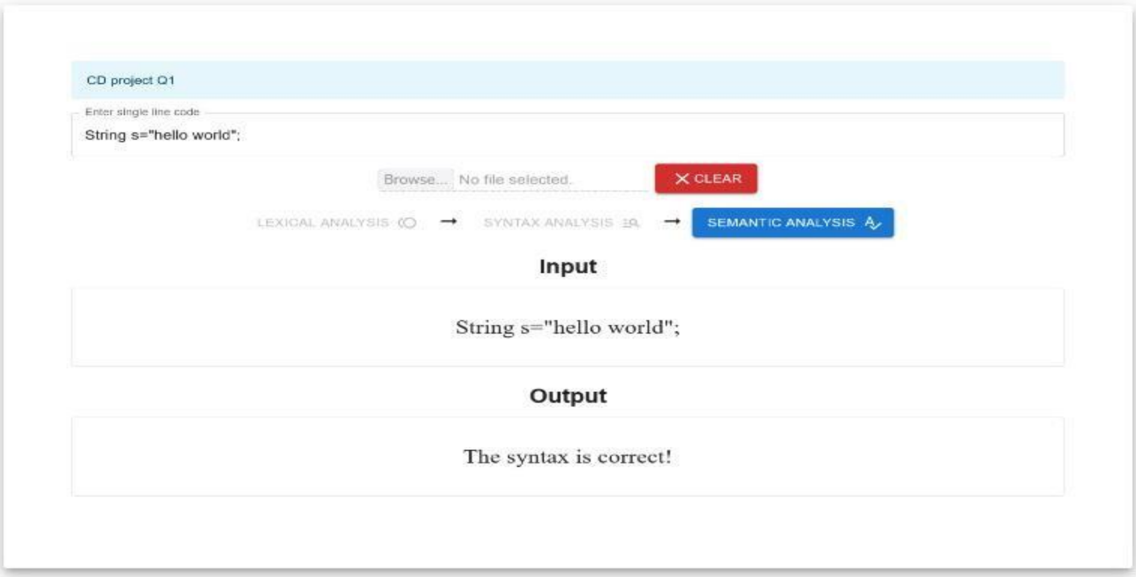
String s="hello world";

Output

<data_type><identifier><assignment_operator><value><delimiter>

Reg No: RA2011026010060 RA2011026010076 RA2011026010082 RA2011026010099 RA2011026010113

Phases of Compiler



String s="hello world";

Input

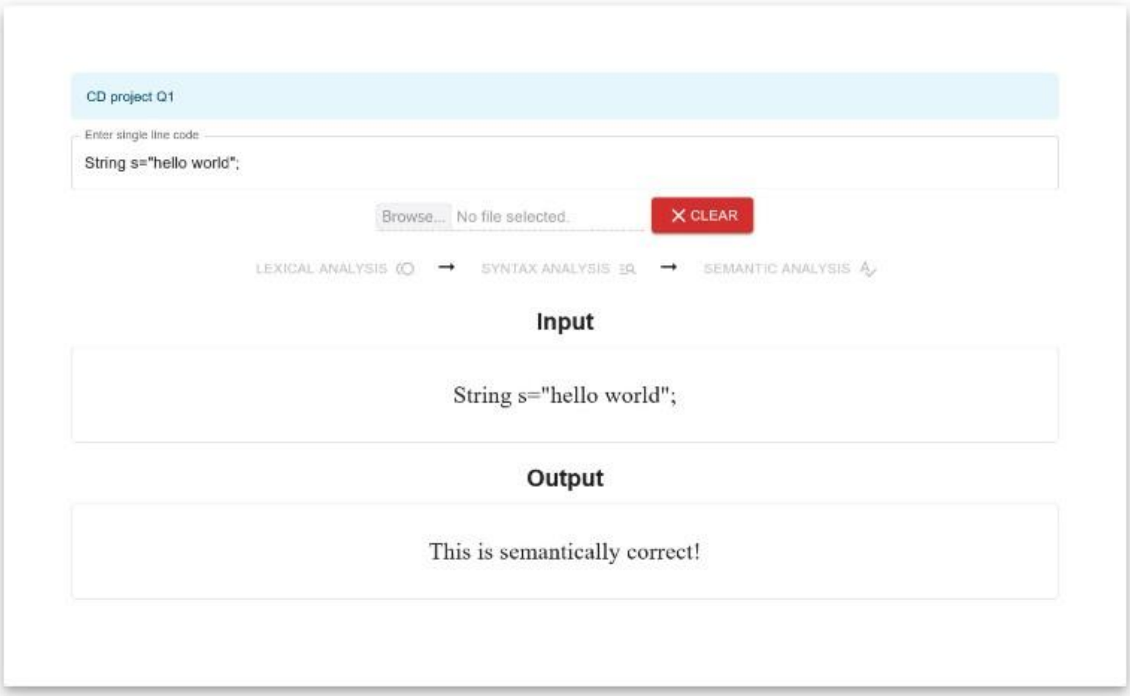
String s="hello world";

Output

The syntax is correct!

Reg No: RA2011026010060 RA2011026010076 RA2011026010082 RA2011026010099 RA2011026010113

Phases of Compiler



String s="hello world";

Input

String s="hello world";

Output

This is semantically correct!

Reg No: RA2011026010060 RA2011026010076 RA2011026010082 RA2011026010099 RA2011026010113

6.2 Results

The result analysis for a lexical analyzer in a compiler design project involves evaluating the accuracy, efficiency, and completeness of the lexical analysis phase. Here are some key factors to consider:

- Accuracy: The lexical analyzer should accurately recognize all valid tokens in the input source code and flag any errors or invalid tokens. To evaluate accuracy, the project team can use a set of test cases that cover all the language constructs and edge cases. The lexical analyzer should be able to handle different input formats, such as source code files and input from the command line.
- Efficiency: The lexical analyzer should be efficient in terms of time and memory usage. It should be able to process large input files without slowing down or running out of memory. To evaluate efficiency, the project team can measure the time and memory usage of the lexical analyzer on a set of large input files.
- Completeness: The lexical analyzer should recognize all language constructs defined in the language specification. The project team should check that all the keywords, operators, and punctuation symbols are recognized correctly by the lexical analyzer.
- Error handling: The lexical analyzer should provide meaningful error messages when it encounters invalid tokens or errors in the input source code. The project team should check that the error messages are informative and help the programmer to locate and fix the error.
- Scalability: The lexical analyzer should be designed to scale to different programming languages and language versions. It should be easy to extend and modify to handle new language constructs and syntax rules. The project team should evaluate the design and modularity of the lexical analyzer to ensure it can be extended easily.

Overall, the result analysis for a lexical analyzer in a compiler design project should focus on the accuracy, efficiency, completeness, error handling, and scalability of the lexical analysis phase. The team should use a set of test cases and performance metrics to evaluate the quality and effectiveness of the lexical analyzer.

CHAPTER 7

CONCLUSIONS

In conclusion, a compiler is a complex software system that converts source code written in a high-level programming language into machine code that can be executed by a computer. The compilation process is typically divided into several phases, each of which performs a specific task.

The main phases of a typical compiler project include lexical analysis, syntax analysis, sema ntic analysis, intermediate code generation, code optimization, and code generation.

Lexical analysis involves breaking down the source code into a sequence of tokens, while syntax analysis verifies that the sequence of tokens conforms to the syntax rules of the language. Semantic analysis checks for semantic correctness and generates an abstract syntax tree that represents the program's meaning.

Intermediate code generation produces an intermediate representation of the program that can be optimized before being translated into machine code. Code optimization is the process of improving the efficiency of the generated code. Finally, code generation generates machine code that can be executed by the target machine.

Each of these phases is critical to the successful compilation of a program, and they build on each other to produce an efficient and correct program. A well-designed compiler must be able to handle various programming language constructs and produce efficient and optimized code.

CHAPTER 8

REFERENCES

1. **https://llvm.org/**
2. **https://www.antlr.org/**
3. **https://github.com/westes/flex**
4. **https://github.com/westes/flex**
5. **https://www.gnu.org/software/bison/**