

LexiPy(Lexical Analyzer)

Submitted by

Jaspreet Singh [RA2011033010092]

Vama Pachori [RA2011033010115]

Under the Guidance of

Dr. J. Jeyasudha

Assistant Professor, Department of Computational Intelligence

In partial satisfaction of the requirements for the degree of

**BACHELORS OF TECHNOLOGY
in
COMPUTER SCIENCE ENGINEERING
with specialization in Software Engineering**



**SCHOOL OF COMPUTING
COLLEGE OF ENGINEERING AND TECHNOLOGY
SRM INSTITUTE OF SCIENCE AND TECHNOLOGY
KATTANKULATHUR - 603203**

May 2023



SRM
INSTITUTE OF SCIENCE & TECHNOLOGY
Deemed to be University u/s 3 of UGC Act, 1956

SRM INSTITUTION OF SCIENCE AND TECHNOLOGY
KATTANKULATHUR-603203

BONAFIDE CERTIFICATE

Certified that this Course Project Report titled "**Lexical Analyzer**" is the bonafide work done by **Jaspreet Singh [RA2011033010092]** and **Vama Pachori [RA2011033010115]** 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

Faculty In-Charge

Dr. J Jeyasudha

Assistant Professor

Department of Computational Intelligence

SRM Institute of Science and Technology

Kattankulathur Campus, Chennai

HEAD OF THE DEPARTMENT

Dr. R Annie Uthra

Professor and Head ,

Department of Computational Intelligence,

SRM Institute of Science and Technology

Kattankulathur Campus, Chennai

Abstract

This project presents the design and implementation of a lexical analyzer for parsing and analyzing input text using Node.js. The analyzer is built using JavaScript, Node.js and its built-in APIs to define a set of regular expressions and rules for recognizing different types of tokens, including keywords, operators, and identifiers. We demonstrate the effectiveness of our analyzer through a series of experiments, showing that it is capable of accurately and efficiently tokenizing a wide range of input text. For lexical analyzer, we utilized the built-in Node.js APIs like `JSON.parse` to convert each word into lexemes and `JSON.stringify` for whole backend. We evaluated the effectiveness of our lexical analyzer through a series of experiments, showing that it is capable of accurately and efficiently tokenizing a wide range of input text.

TABLE OF CONTENTS

Chapter No.	Title	Page No.
	ABSTRACT	3
	TABLE OF CONTENTS	4
1.	INTRODUCTION	5
1.1	Introduction	5
1.2	Problem Statement	6
1.3	Objectives	7
1.4	Need for Lexical Analyzer	8
1.5	Requirements Specification	10
2.	NEEDS	11
2.1	Needs of compiler during Lexical Analyzer	11
2.2	Working of Lexical Analyzer	11
3.	SYSTEM ARCHITECTURE AND DESIGN	13
3.1	Frontend	13
3.2	Backend	15
3.3	Architecture Design	17
4.	REQUIREMENTS	19
5.	CODING AND TESTING	20
5.1	Coding	20
5.2	Testing	29
5.3	Result	36
6.	CONCLUSION	38
7.	REFERENCES	39

Chapter 1

Introduction

1.1 Introduction

A lexical analyzer, also known as a lexer or scanner, is an important component of a compiler that performs the first phase of the compilation process. The purpose of the lexical analyzer is to take the source code of a program as input and produce a stream of tokens, which are the basic building blocks of the program.

The input to the lexer is typically a sequence of characters, and its output is a sequence of tokens that represent the different elements of the program, such as keywords, identifiers, literals, and operators. The lexer breaks down the input into individual tokens based on a set of rules that define the syntax of the programming language being compiled.

The lexer uses regular expressions or other similar techniques to define patterns that match the various types of tokens in the source code. It then scans the input character by character and matches the input against the defined patterns. When a match is found, the lexer generates a token and moves on to the next character.

The tokens generated by the lexer are passed to the next phase of the compiler, the syntax analyzer, which uses the tokens to build a parse tree that represents the syntactic structure of the program. The syntax analyzer checks that the program follows the rules of the programming language's syntax.

1.2 Problem Statement

The problem that the lexical analyzer aims to solve is to accurately and efficiently break down input text into a sequence of tokens, each representing a meaningful unit of the text. The lexical analyzer plays a vital role in this process by identifying and categorizing different types of tokens in the input text, such as keywords, operators, and identifiers.

The problem with manual tokenization is that it is time-consuming, error-prone, and not scalable for large datasets. Therefore, automated lexical analyzers are essential for processing text in a more efficient and accurate manner. The challenge in building a lexical analyzer is to create a flexible and efficient system that can handle a wide range of input text and accurately identify and categorize the tokens. Additionally, the system must be optimized for performance, able to handle large volumes of data, and provide meaningful insights into the analyzed text.

1.3 Objectives

- To design and implement a flexible and efficient system for breaking down input text into tokens using Node.js.
- To define a set of regular expressions and rules for recognizing and categorizing different types of tokens, including keywords, operators, and identifiers.
- To evaluate the effectiveness of the system through a series of experiments, demonstrating its ability to accurately and efficiently tokenize a wide range of input text.
- To optimize the performance of the system by implementing caching techniques and other optimizations.
- To develop a user-friendly command-line interface for the system, allowing users to easily input and analyze text.
- To explore the potential applications of the system, including natural language processing, information retrieval, and data mining.
- To provide clear and concise documentation for the system, enabling others to understand and utilize its capabilities.
- To contribute to the open-source community by sharing the code and making it available for others to use and build upon.

1.4 Need for Lexical Analyzer

The need for a lexical analyzer arises from the fact that text data is often unstructured and difficult to analyze. For example, in natural language processing, text data can come from a variety of sources, such as social media, news articles, and scientific papers. In order to extract meaningful insights from this data, it is necessary to first break it down into a series of meaningful units, or tokens, such as words, phrases, and punctuation marks. This is where the lexical analyzer comes in.

A lexical analyzer is a tool that automates the process of tokenizing input text, making it faster, more accurate, and scalable. By using a set of regular expressions and rules to recognize different types of tokens, a lexical analyzer can accurately identify and categorize different units of text. This is important for a variety of applications, including information retrieval, data mining, and natural language processing.

Without a lexical analyzer, tokenizing text would require manual effort, making it time-consuming and error-prone. Furthermore, manual tokenization would not be scalable for large datasets. By using a lexical analyzer, text data can be quickly and accurately tokenized, making it easier to analyze and extract insights from. This can lead to better decision-making and improved performance in a variety of industries, including finance, marketing, and healthcare.

Functional requirements:

- The system should be able to tokenize input text accurately and efficiently using regular expressions and rules.
- The system should be able to recognize and categorize different types of tokens, such as keywords, identifiers, and operators.
- The system should be able to handle a wide range of input text, including text in different languages, and special characters.
- The system should be able to handle large volumes of data and be optimized for performance.
- The system should have a user-friendly command-line interface for inputting and analyzing text.
- The system should provide meaningful insights into the analyzed text, such as the frequency of different types of tokens.

Non-functional requirements:

- The system should be built using Node.js and relevant Node packages.
- The system should be easy to install and use, with clear and concise documentation.
- The system should be scalable, allowing for the analysis of large datasets.
- The system should be secure and protect against potential security threats, such as SQL injection and cross-site scripting attacks.
- The system should be maintainable, with well-organized code and proper documentation.
- The system should be compatible with different operating systems and browsers.

Performance requirements:

- The system should be able to tokenize input text in real-time, with minimal lag.
- The system should be able to handle large volumes of data efficiently, without crashing or becoming unresponsive.
- The system should be optimized for performance, using caching techniques and other optimizations.

1.5 Requirements Specification

Hardware Requirements:

Processor: A multi-core processor with a clock speed of at least 2 GHz or higher is recommended for faster processing of data.

RAM: The amount of RAM required depends on the size of the input text data. A minimum of 8 GB RAM is recommended, but for processing large data sets, 16 GB or higher may be required.

Storage: Sufficient storage space should be available for storing the input text data and the results of the analysis. SSD storage is preferred over HDD for faster read/write speeds.

Operating System: The system should run on any modern operating system, such as Windows, macOS, or Linux.

Internet Connection: If the system requires external dependencies or packages, an internet connection may be required for installing them.

Graphics card: A dedicated graphics card is not required for a lexical analyzer project, as it does not involve heavy graphics processing.

Programming Language: Node JS

Frontend Framework: HTML,CSS,Bootstrap

Backend Framework: Node JS

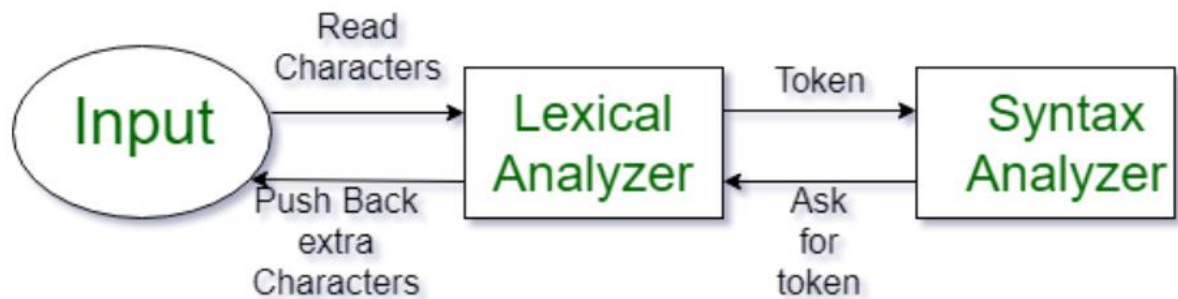
Chapter 2

2.1 What does a compiler need to know during Lexical analysis?

During lexical analysis, a compiler needs to know the following:

- Programming language syntax: The compiler needs to know the syntax of the programming language being compiled. It needs to know the keywords, operators, and other language constructs that can appear in the program.
- Regular expressions: The compiler uses regular expressions to define the patterns that match the various types of tokens in the source code.
- Token types: The compiler needs to know the different types of tokens that can appear in the program, such as keywords, identifiers, literals, and operators.
- Source code input: The compiler needs to take the source code as input, which can be in the form of a file or a string.
- Source code location: The compiler needs to keep track of the location of the tokens in the source code. This information is useful for error reporting, debugging, and generating meaningful error messages.
- Reserved words: The compiler needs to know which words in the programming language are reserved and cannot be used as identifiers.
- Preprocessing directives: The compiler needs to handle any preprocessing directives, such as `#include` or `#define`, which are used to manipulate the source code before compilation.

2.2 Working of a Lexical Analyzer



Following are the some steps that how lexical analyzer work:

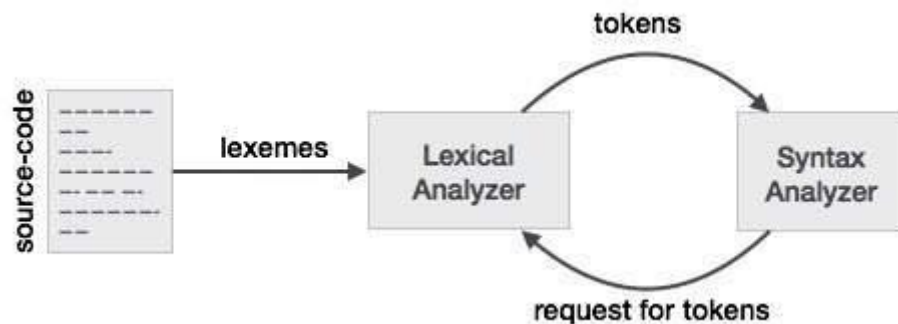
- 1. Input pre-processing:** In this stage involves cleaning up, input takes and preparing lexical analysis this may include removing comments, white space and other non-input text from input text.
- 2. Tokenization:** This is a process of breaking the input text into sequence of a tokens.
- 3. Token classification:** Lexeme determines type of each token, it can be classified keyword, identifier, numbers, operators and separator.
- 4. Token validation:** Lexeme checks each token with valid according to rule of programming language.
- 5. Output Generation:** It is a final stage lexeme generate the outputs of the lexical analysis process, which is typically list of tokens.

- We can represent in the form of lexemes and tokens as under

Lexemes	Tokens	Lexemes	Tokens	
while	WHILE	a	IDENTIFIER	
(LPAREN	=	ASSIGNMENT	
a	IDENTIFIER	a	IDENTIFIER	
>=	COMPARISON	–	ARITHMETIC	
b	IDENTIFIER	2	INTEGER	
)	RPAREN	;	SEMICOLON	

Chapter 3

SYSTEM ARCHITECTURE AND DESIGN



3.1 FrontEnd:

Bootstrap: Bootstrap is a popular front-end framework that provides a set of tools and components for building responsive web applications.

Bootstrap includes pre-built CSS styles and JavaScript components, such as forms, buttons, navigation bars, modal windows, and carousels

HTML/CSS: HTML is used to define the structure and content of web pages. It uses tags to define different elements of a web page, such as headings, paragraphs, images, links, and forms. HTML is a markup language, meaning it is used to annotate text to give it semantic meaning, rather than being a programming language.

CSS is used to define the presentation and layout of web pages. It is used to control the appearance of HTML elements, such as font size, color, spacing, and position. CSS allows developers to separate the presentation layer from the content layer, making it easier to maintain and update web pages.

Javascript: JavaScript is used to add interactivity to web pages, such as animations, pop-ups, and form validation. It can also be used to manipulate the Document Object Model (DOM),

which represents the structure of an HTML document, allowing developers to dynamically update the content and appearance of web pages without having to reload the entire page.

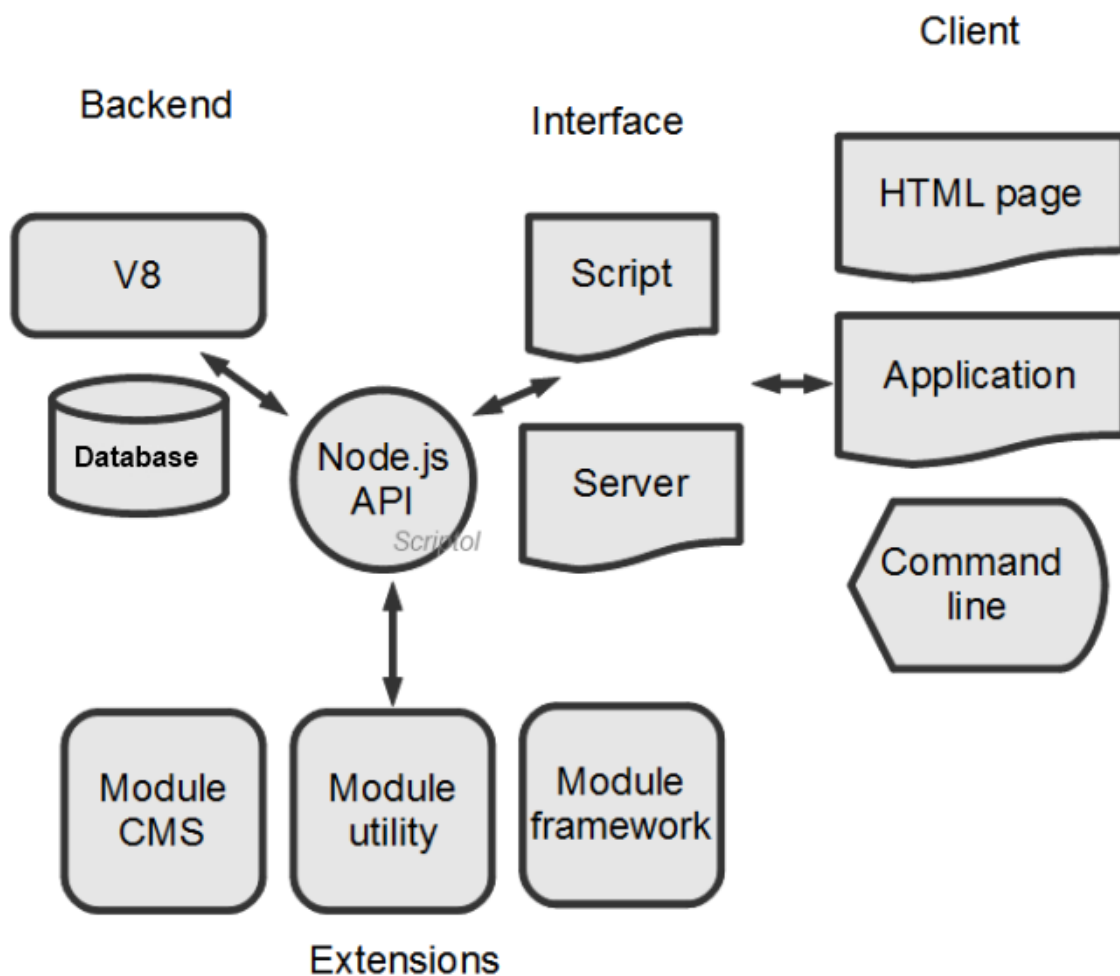
Code:

```
def maximum(a, b):  
    if a >= b:  
        return a  
    else:  
        return b  
a = 2  
b = 4  
print(maximum(a, b))
```

Submit

Responsive Table Example with Animations

SrNO.	Token	Value
1	IDENTIFIER	def
2	WHITESPACE	
3	IDENTIFIER	maximum
4	L_PAREN	(
5	IDENTIFIER	a
6	COMMA	,
7	WHITESPACE	
8	IDENTIFIER	b
9	R_PAREN)
10	TERN_ELSE	:
11	WHITESPACE	
12	NEW_LINE	
13	WHITESPACE	
14	WHITESPACE	
15	WHITESPACE	
16	WHITESPACE	
17	WHITESPACE	
18	IF	if
19	WHITESPACE	
20	IDENTIFIER	a



Node.js environment - (c) 2012 Scriptol.com

3.2 Backend Design:

Nodejs: Node.js is a server-side JavaScript runtime environment built on the V8 engine, which is the same engine used by Google Chrome. It allows developers to run JavaScript on the server, allowing them to write both the front-end and back-end code in the same language.

Node.js is designed to be lightweight and efficient, making it an ideal platform for building scalable and high-performance web applications. It provides a non-blocking I/O model, which means that multiple requests can be handled simultaneously without blocking the execution of other requests.

Node.js also includes a built-in package manager, npm, which provides access to a large and growing ecosystem of open-source packages and modules that can be easily integrated into

Node.js applications. This makes it easier for developers to add functionality to their applications without having to write all the code from scratch.

Lexer: The system should include a lexer component that tokenizes the Bash script and generates a stream of tokens that can be processed by the parser.

Install Node packages: Before you can begin building your backend, you will need to install Node.js and any necessary packages or modules. You can do this using a package manager such as npm.

Server Setup: To serve your Python code and perform lexical analysis on it, you will need to set up a server using Node.js. You can use a framework like Express.js to simplify the process of creating a server.

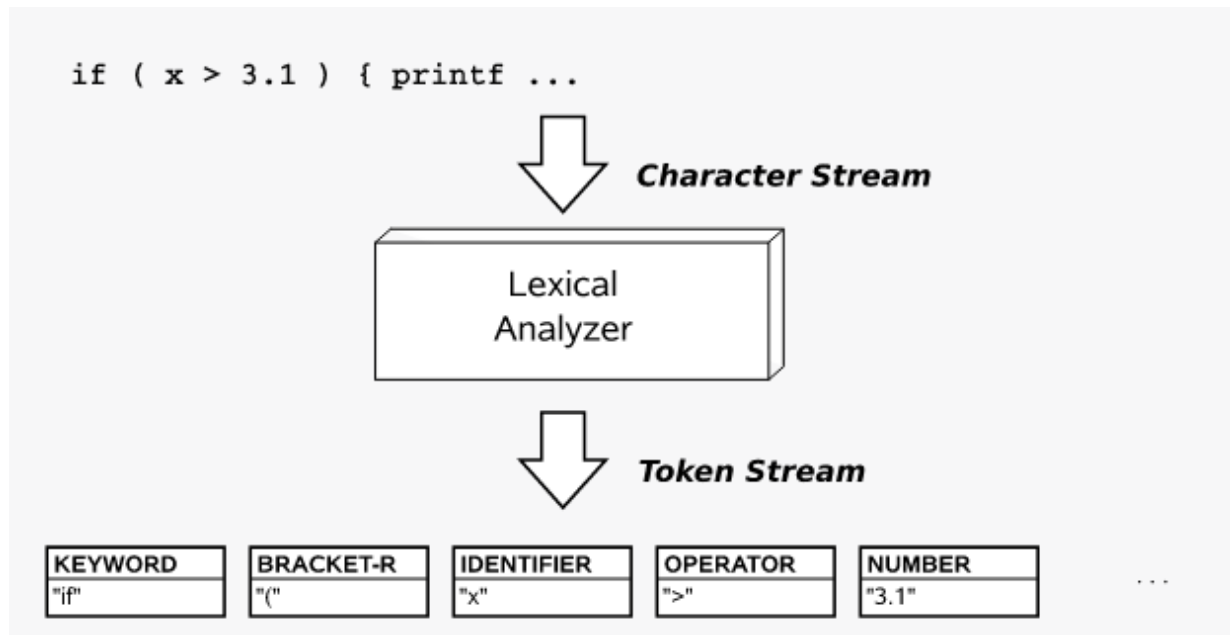
Route Creating: Once your server is set up, you can create a route that will allow you to send your Python code to the server. This can be done using the HTTP POST method.

Processing python code: When the Python code is sent to the server, you can use a module like Child Process to execute the code and capture its output.

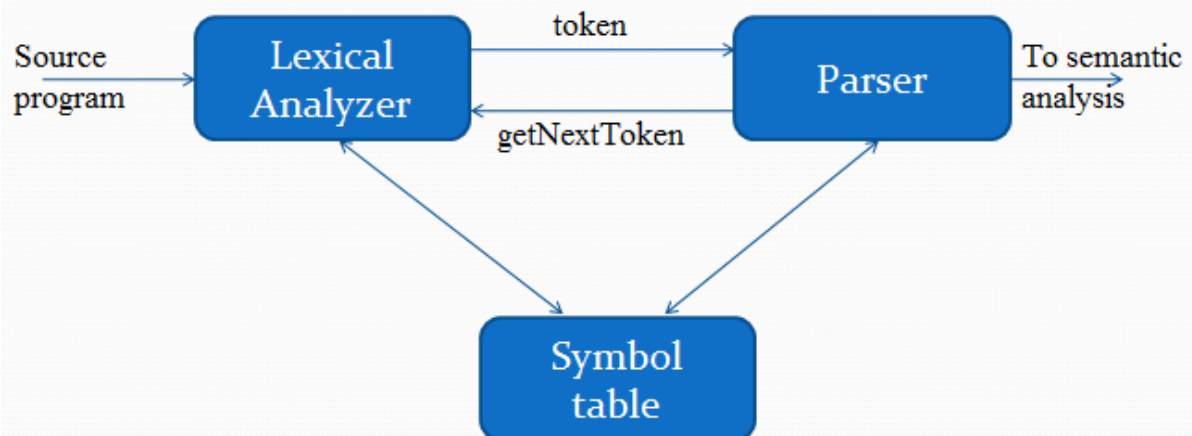
Performing lexical analysis: Once you have captured the output from the Python code, you can perform lexical analysis on it using a lexical analysis package or module named lexr.

Return the lexical analysis results: Finally, you can return the results of the lexical analysis to the client-side application that made the request. This can be done using the HTTP response object.

3.3 Architecture Design



The role of lexical analyzer



Chapter 4

The requirements to run the scripts:-

- **Node.js:** You will need to have Node.js installed on your computer to run Node.js scripts. You can download Node.js from the official website: <https://nodejs.org/>
- **Required Node packages:** Depending on the specific Node.js script you are running, you may need to install additional packages or modules. You can do this using a package manager like npm.
- **Text editor:** You will need a text editor to write and edit the Node.js script. There are many text editors available, such as Visual Studio Code, Sublime Text, and Atom.
- **Python:** If the Node.js script is executing Python code, you will need to have Python installed on your computer. You can download Python from the official website: <https://www.python.org/>
- **Required Python packages:** Depending on the Python code you are running, you may need to install additional packages or modules. You can do this using pip, which is the package installer for Python.

Chapter 5

Coding and Testing

5.1 Coding:-

1. home.ejs

```
<!DOCTYPE html>
<html>
<head>
  <meta charset="UTF-8">
  <title>Code Input Form</title>
  <link
href="https://cdn.jsdelivr.net/npm/bootstrap@5.0.2/dist/css/bootstrap.min.css"
rel="stylesheet" integrity="sha384-
EVSTQN3/azprG1Anm3QDgpJLIm9Nao0Yz1ztcQTwFspd3yD65VohhpuuCO
mLASjC" crossorigin="anonymous">

  <style>
    * {
      box-sizing: border-box;
      font-family: Arial, sans-serif;
    }

    body {
      margin: 0;
      padding: 0;
    }

    .container {
      max-width: 800px;
      margin: 0 auto;
      padding: 20px;
    }

    .form-group {
      margin-bottom: 20px;
    }

    label {
      display: block;
      margin-bottom: 10px;
      font-weight: bold;
    }

    input[type="text"], textarea {
      width: 100%;
```

```

padding: 10px;
border: 1px solid #ccc;
border-radius: 5px;
resize: none;
}

@media screen and (max-width: 600px) {
  .container {
    padding: 10px;
  }
}
</style>
</head>
<body>
  <div class="container">
    <h1>Code Input Form</h1>
    <form method="post" action="/output">
      <div class="form-group">
        <label for="code">Code:</label>
        <textarea id="code" name="code" rows="10"
placeholder="Enter your code here"></textarea>
      </div>
      <input type="submit"></input>
    </form>
  </div>
  <hr>
  <h1>History</h1>
  <% if(data.length >0){%>
    <% data.forEach(data => {%>
  <div>
    <div class="card" style="width: 18rem;">

      <div class="card-body">

        <p class="card-text"><%= data.data %></p>
        <a href="#" class="btn btn-primary">Go somewhere</a>
      </div>
    </div>

  </div>
  <% }%>
  <% } %>
</body>
</html>

```

2. login.ejs

```

<!DOCTYPE html>
<html lang="en">
<head>

```

```
<meta charset="UTF-8">
<meta name="viewport" content="width=device-width, initial-scale=1.0">
<title>Sign Up Page</title>
<link rel="stylesheet" href="style.css">
</head>
<body>
  <div class="container">
    <form method="post" action="/login">
      <h1>Log In</h1>

      <label for="email">Email</label>
      <input type="email" id="email" name="email" required>
      <label for="password">Password</label>
      <input type="password" id="password" name="password" required>
      <input type="submit" value="Log In">
    </form>

  </div>
</body>
<style>
  * {
    box-sizing: border-box;
  }

  body {
    font-family: Arial, sans-serif;
    margin: 0;
  }

  .container {
    display: flex;
    flex-direction: column;
    justify-content: center;
    align-items: center;
    height: 100vh;
    background-color: #f2f2f2;
  }

  form {
    width: 90%;
    max-width: 400px;
    padding: 20px;
    border: 1px solid #ccc;
    border-radius: 5px;
    background-color: #fff;
    animation: slide-up 0.5s ease;
  }

  h1 {
    text-align: center;
```

```
    margin-top: 0;
}

label {
    display: block;
    margin-bottom: 5px;
}

input[type="text"],
input[type="email"],
input[type="password"] {
    width: 100%;
    padding: 10px;
    margin-bottom: 15px;
    border: 1px solid #ccc;
    border-radius: 5px;
}

input[type="submit"] {
    background-color: #4CAF50;
    color: #fff;
    border: none;
    padding: 10px 20px;
    border-radius: 5px;
    cursor: pointer;
    transition: background-color 0.3s ease;
}

input[type="submit"]:hover {
    background-color: #3e8e41;
}

.login-button {
    background-color: #4CAF50;
    border: none;
    color: #fff;
    border-radius: 5px;
    font-size: 18px;
    margin-top: 20px;
    cursor: pointer;
    padding: 10px 20px;
    animation: fade-in 1s ease 0.5s forwards;
}

.login-button:hover {
    text-decoration: underline;
}

@keyframes slide-up {
    from {
```



```
        transform: translateY(100px);
        opacity: 0;
    }
    to {
        transform: translateY(0);
        opacity: 1;
    }
}

@keyframes fade-in {
    from {
        opacity: 0;
    }
    to {
        opacity: 1;
    }
}

</style>
</html>
```

3. output.ejs

```
<!DOCTYPE html>
<html>
<head>
    <meta charset="UTF-8">
    <title>Responsive Table Example with Animations</title>
    <style>
        * {
            box-sizing: border-box;
            font-family: Arial, sans-serif;
        }

        body {
            margin: 0;
            padding: 0;
        }

        .container {
            max-width: 800px;
            margin: 0 auto;
            padding: 20px;
        }
    </style>
</head>
<body>
    <div class="container">
        <table>
            <tr>
                <th>Name</th>
                <th>Age</th>
                <th>Gender</th>
            </tr>
            <tr>
                <td>John</td>
                <td>25</td>
                <td>Male</td>
            </tr>
            <tr>
                <td>Jane</td>
                <td>30</td>
                <td>Female</td>
            </tr>
            <tr>
                <td>Mike</td>
                <td>35</td>
                <td>Male</td>
            </tr>
            <tr>
                <td>Emily</td>
                <td>28</td>
                <td>Female</td>
            </tr>
        </table>
    </div>
</body>
</html>
```

```

    table {
        width: 100%;
        border-collapse: collapse;
        position: relative;
        animation: fadeIn 1s;
        animation-fill-mode: forwards;
        opacity: 0;
    }

    th, td {
        padding: 10px;
        border: 1px solid #ccc;
    }

    th {
        background-color: #f2f2f2;
        font-weight: bold;
    }

    @media screen and (max-width: 600px) {
        table {
            display: block;
            overflow-x: auto;
        }
        th, td {
            display: block;
        }
        th {
            text-align: left;
        }
    }

    @keyframes fadeIn {
        from {
            opacity: 0;
            transform: translateY(-20px);
        }
        to {
            opacity: 1;
            transform: translateY(0);
        }
    }
</style>
</head>
<body>
    <div class="container">
        <h1>Responsive Table Example with Animations</h1>
        <table>
            <thead>

```

```

        <tr>
            <th>SrNO.</th>
            <th>Token</th>
            <th>Value</th>

        </tr>
    </thead>
    <tbody>
        <% if(data.length >=0){%>
            <% for(var i=1; i<=(data.length); i++){%>
                <tr>
                    <td><%= i %></td>
                    <td><%= data[i-1].token %></td>
                    <td><%= data[i-1].value %></td>

                </tr>
            <% }} %>
        </tbody>
    </table>
</div>
</body>
</html>
```

4. server.js

```
const express=require('express')
const connectDb = require('./config/dbConnection');
const bodyParser=require('body-parser')
const bcrypt = require('bcrypt');
const jwt=require('jsonwebtoken')
const { default: mongoose } = require('mongoose')
const { findByIdAndUpdate } = require('./models/user')

const user = require('./models/user')
const code = require('./models/code');
const { resolveSoa } = require('dns/promises');
connectDb();
const app = express();

const port = process.env.PORT || 5000;

app.use(express.json());
app.use(express.urlencoded({extended:true}));
app.use(bodyParser.json());
app.set('view engine','ejs');
app.get('/test',(req,res)=>{
    res.render('home');
})
app.get('/',(req,res)=>{
    res.render('singup');
})
app.post('/singup',async(req,res)=>{
    const password=req.body.password;
    const hashedPassword = await bcrypt.hash(password, 10);
    const u = new user(
    {
        email:req.body.email,
        password:hashedPassword
    });
    const email=req.body.email;

    const k = await user.findOne({email});

    if(k){
        res.send('user already exist')
        return;
    }
    else{
        u.save();

        const ress=await code.find();
```

```

    res.render('home',{data:ress});
  }
})
app.post('/login',async (req,res)=>{

  const password=req.body.password;

  const email=req.body.email;

  const k = await user.findOne({email});

  if(k && await bcrypt.compare(password, k.password)){
    const ress=await code.find();
    res.render('home',{data:ress});
  }
  else{

    res.send('nohehe')
  }
})
app.get('/login',(req,res)=>{
  res.render('login');
})
app.post('/output',async(req,res)=>{
  let lexr = require('lexr');
  let tokenizer = new lexr.Tokenizer("Javascript");
  tokenizer.setErrTok("DIFF_ERROR");
  let input = req.body.code;
  const k = new code(
    {
      data:req.body.code

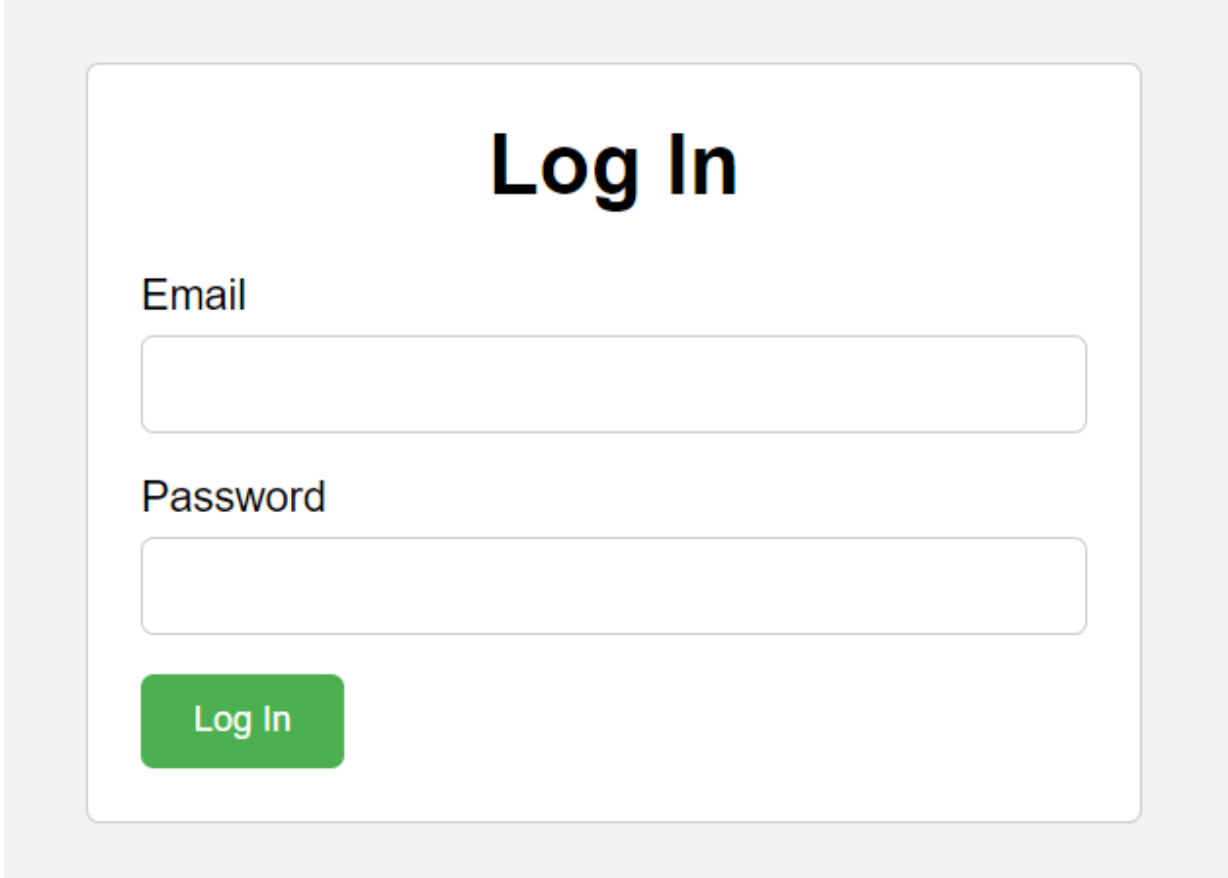
    });
  k.save();
  let output = await tokenizer.tokenize(input);
  const {parse,generate}= require('abstract-syntax-tree')
  res.render('output',{data:output});
})

app.listen(port, () => {
  console.log(`Server running on port ${port}`);
});

```

5.2 Testing:-

Login Page



A login form titled "Log In" is displayed within a light gray rounded rectangle. The form itself is a white rounded rectangle containing the title "Log In" in a large, bold, black font. Below the title, there are two input fields: one labeled "Email" and another labeled "Password", both in a standard black font. Each label is positioned to the left of its corresponding input field. The "Email" input field is a simple white rectangle with a thin gray border. The "Password" input field is a white rectangle with a thin gray border and a small gray eye icon on the right side, indicating a toggle for password visibility. Below the password field is a green rectangular button with rounded corners and the text "Log In" in white.

Test Case 1:

Input Page

Code Input Form

Code:

```
c = 'g'  
  
print("The ASCII value of '" + c + "' is", ord(c))
```

Submit

Output Page

Responsive Table Example with Animations

SrNO.	Token	Value
1	IDENTIFIER	c
2	WHITESPACE	
3	ASSIGN	=
4	WHITESPACE	
5	STRING_LIT	'g'
6	NEW_LINE	
7	NEW_LINE	
8	IDENTIFIER	print
9	L_PAREN	(
10	STRING_LIT	"The ASCII value of "
11	WHITESPACE	
12	ADD	+
13	WHITESPACE	
14	IDENTIFIER	c
15	WHITESPACE	
16	ADD	+
17	WHITESPACE	
18	STRING_LIT	" is"
19	COMMA	,
20	WHITESPACE	
21	IDENTIFIER	ord
22	L_PAREN	(
23	IDENTIFIER	c
24	R_PAREN)
25	R_PAREN)

Test Case 2:

Input

Code Input Form

Code:

```
def maximum(a, b):  
  
    if a >= b:  
        return a  
    else:  
        return b  
  
a = 2  
b = 4  
print(maximum(a, b))
```

Submit

Output

Responsive Table Example with Animations

SrNO.	Token	Value
1	IDENTIFIER	def
2	WHITESPACE	
3	IDENTIFIER	maximum
4	L_PAREN	(
5	IDENTIFIER	a
6	COMMA	,
7	WHITESPACE	
8	IDENTIFIER	b
9	R_PAREN)
10	TERN_ELSE	:
11	NEW_LINE	
12	WHITESPACE	
13	WHITESPACE	
14	WHITESPACE	
15	WHITESPACE	
16	WHITESPACE	
17	WHITESPACE	
18	WHITESPACE	
19	WHITESPACE	
20	IF	if
21	WHITESPACE	
22	IDENTIFIER	a
23	WHITESPACE	
24	G_THAN_EQ	>=
25	WHITESPACE	

37	RETURN	return
38	WHITESPACE	
39	IDENTIFIER	a
40	NEW_LINE	
41	WHITESPACE	
42	WHITESPACE	
43	WHITESPACE	
44	WHITESPACE	
45	ELSE	else
46	TERN_ELSE	:
47	NEW_LINE	
48	WHITESPACE	
49	WHITESPACE	
50	WHITESPACE	
51	WHITESPACE	
52	WHITESPACE	
53	WHITESPACE	
54	WHITESPACE	
55	WHITESPACE	
56	RETURN	return
57	WHITESPACE	
58	IDENTIFIER	b
59	NEW_LINE	
60	IDENTIFIER	a
61	WHITESPACE	
62	ASSIGN	=
63	WHITESPACE	
64	NUM_LIT	2

56	RETURN	return
57	WHITESPACE	
58	IDENTIFIER	b
59	NEW_LINE	
60	IDENTIFIER	a
61	WHITESPACE	
62	ASSIGN	=
63	WHITESPACE	
64	NUM_LIT	2
65	NEW_LINE	
66	IDENTIFIER	b
67	WHITESPACE	
68	ASSIGN	=
69	WHITESPACE	
70	NUM_LIT	4
71	NEW_LINE	
72	IDENTIFIER	print
73	L_PAREN	(
74	IDENTIFIER	maximum
75	L_PAREN	(
76	IDENTIFIER	a
77	COMMA	,
78	WHITESPACE	
79	IDENTIFIER	b
80	R_PAREN)
81	R_PAREN)

Test Case 3:

Input

Code Input Form

Code:

```
num1 = 15
num2 = 12
sum = num1 + num2
print("Sum of", num1, "and", num2 , "is", sum)
```

Submit

Output

Responsive Table Example with Animations

SrNO.	Token	Value
1	IDENTIFIER	num1
2	WHITESPACE	
3	ASSIGN	=
4	WHITESPACE	
5	NUM_LIT	15
6	NEW_LINE	
7	IDENTIFIER	num2
8	WHITESPACE	
9	ASSIGN	=
10	WHITESPACE	
11	NUM_LIT	12
12	NEW_LINE	
13	IDENTIFIER	sum
14	WHITESPACE	
15	ASSIGN	=
16	WHITESPACE	
17	IDENTIFIER	num1
18	WHITESPACE	

17	IDENTIFIER	num1
18	WHITESPACE	
19	ADD	+
20	WHITESPACE	
21	IDENTIFIER	num2
22	NEW_LINE	
23	IDENTIFIER	print
24	L_PAREN	(
25	STRING_LIT	"Sum of"
26	COMMA	,
27	WHITESPACE	
28	IDENTIFIER	num1
29	COMMA	,
30	WHITESPACE	
31	STRING_LIT	"and"
32	COMMA	,
33	WHITESPACE	
34	IDENTIFIER	num2
35	WHITESPACE	
36	COMMA	,
37	WHITESPACE	
38	STRING_LIT	"is"
39	COMMA	,
40	WHITESPACE	
41	IDENTIFIER	sum
42	R_PAREN)
43	NEW_LINE	

5.3 Result:

The result of a lexical analysis is a set of tokens that represent the lexical structure of the input code. These tokens can be used to build a parse tree, which represents the syntactic structure of the code.

The tokens themselves are typically defined by regular expressions and correspond to the different parts of the code, such as keywords, operators, identifiers, and literals. For example, in Python code, the tokens might include keywords like "if" and "else", operators like "+", "-", and "*", and identifiers like variable names.

The parse tree that is built from the tokens can be used for further analysis, such as semantic analysis, optimization, and code generation. The parse tree is a hierarchical structure that represents the syntactic structure of the code and allows for more advanced analysis and transformation.

Advantages:-

There are several advantages to performing lexical analysis as part of the compilation process:

- **Improved error handling:** By breaking the input code into tokens, the lexical analyzer can provide more informative error messages when syntax errors occur. This makes it easier for developers to identify and fix issues in their code.
- **Improved program comprehension:** By analyzing the lexical structure of the code, the lexical analyzer can provide useful information to developers about the structure and organization of the code. This can make it easier for developers to understand how the code works and modify it as needed.

- **Improved program optimization:** By identifying the different parts of the code, such as keywords, operators, and identifiers, the lexical analyzer can help identify opportunities for optimization. For example, the analyzer can identify repeated code patterns and suggest ways to refactor the code to make it more efficient.
- **Improved portability:** By separating the lexical analysis stage from other stages of the compilation process, the compiler can be made more portable. This allows the compiler to be used on different platforms and with different programming languages, which can be useful for developers who work on multiple projects.
- **Improved performance:** By breaking the code into tokens before performing further analysis, the compiler can save time and resources by avoiding unnecessary computations. This can lead to faster compile times and improved overall performance.

Chapter 6

Conclusion

Lexical analysis is a crucial part of the compilation process for any programming language. It helps to identify the basic elements of the code, such as keywords, operators, and identifiers, which are then used to build a parse tree that represents the syntactic structure of the code. This parse tree can then be used for further analysis and optimization, including semantic analysis and code generation.

Node.js is a popular platform for implementing lexical analysis in web applications. It provides a range of powerful tools and packages for processing code efficiently and accurately. Node.js is built on the V8 JavaScript engine and provides a non-blocking I/O model that makes it ideal for handling I/O-heavy tasks such as lexical analysis.

One of the key advantages of using Node.js for lexical analysis is its ability to handle large volumes of data quickly and efficiently. This is particularly important for web applications that need to process large amounts of code in real-time. Node.js also provides a range of libraries and packages for working with regular expressions, which are a fundamental tool for defining tokens during lexical analysis.

In addition to its efficiency and versatility, Node.js is also highly customizable, allowing developers to build and configure their lexical analysis tools to suit their specific needs. This flexibility makes it a popular choice for developers who want to build robust and efficient tools for processing code.

Overall, lexical analysis is an essential part of the compilation process, and Node.js provides a powerful platform for implementing this process in web applications. By leveraging the power and flexibility of Node.js, developers can build efficient and accurate tools for analyzing and processing code that can help improve the overall quality of software development.

Chapter 7

References

- <https://www.geeksforgeeks.org/compiler-design-lexical-analysis/>
- <https://nodejs.org/>
- <https://youtu.be/GGWtkQXJKUs>
- <https://github.com/365bytes/LexicalAnalyzer>
- https://www.w3schools.com/nodejs/nodejs_regex.asp
- <https://codeforgeek.com/node-js-lexical-analysis-flex/>
- <https://www.geeksforgeeks.org/introduction-of-lexical-analysis/>