

# **PARSER (SYNTAX ANALYSER)**

*Submitted by*

**Mahi Prasad (RA2011033010112)**

*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 AND TECHNOLOGY KATTANKULATHUR-603203

### BONAFIDE CERTIFICATE

Certified that this Course Project Report titled “**PARSER - SYNTAX ANALYSER**” is the bonafide work done by **Mahi Prasad (RA2011033010112)** 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

The syntax analyzer, or parser, is a fundamental component of a compiler or interpreter responsible for analyzing the syntactic structure of source code written in a programming language. Its main objective is to ensure that the code conforms to the grammar rules specified by the language and construct a structural representation of its syntax. This abstract provides an overview of the syntax analyzer's role, highlighting its key features and contributions. It begins by tokenizing the source code, recognizing individual tokens based on lexical rules. It then applies parsing algorithms to validate the token stream against the grammar rules, constructing a parse tree or abstract syntax tree (AST) that captures the hierarchical relationships between language constructs. The syntax analyzer excels in error detection and reporting, promptly identifying syntax errors and providing developers with meaningful error messages to aid in debugging. Additionally, the syntax analyzer plays a crucial role in enforcing compliance with the language specification, ensuring syntactic validity and promoting language interoperability. It employs efficient parsing techniques and may implement error recovery mechanisms to handle syntax errors gracefully and continue analyzing the code. The resulting parse tree or AST serves as an intermediate representation for subsequent compilation phases, enabling further analysis, transformation, and code optimization.

In summary, the syntax analyzer serves as a vital component in the compilation process, guaranteeing syntactic correctness, constructing a structural representation of code syntax, detecting and reporting errors, enforcing language specifications, and providing an intermediate representation for subsequent phases. Its contributions facilitate reliable and accurate processing of source code, ultimately supporting the development of robust and efficient software systems.

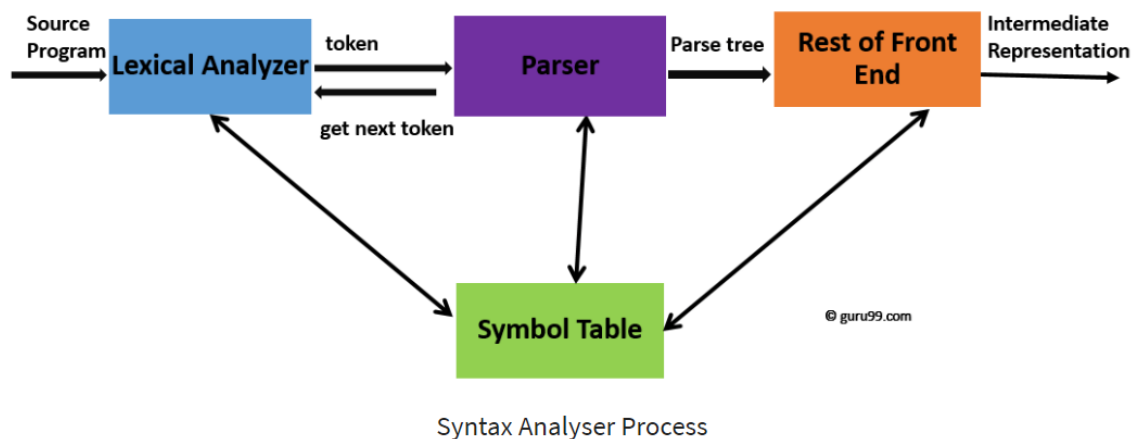
<b>CHAPTER NO.</b>	<b>TABLE OF CONTENTS</b>	<b>PAGE NUMBER</b>
	<b>ABSTRACT</b>	3
<b>1</b>	<b>INTRODUCTION</b>	
	1.1 Introduction	5
	1.2 Problem Statement	7
	1.3 Objectives	9
	1.4 Need for Syntax Analyser	10
<b>2</b>	<b>REQUIREMENTS</b>	
	2.1 Why does Compiler needs it	11
	2.2 Need for DFA in Syntax Analyser	12
<b>3</b>	<b>SYSTEM AND ARCHITECTURE DESIGN</b>	
	3.1 Front End Design	13
	3.2 Architecture Design of Front End	14
	3.3 Back End Design	15
	3.4 Architecture Design of Back End	16
	3.5 System Architecture Design	17
<b>4</b>	<b>SYSTEM REQUIREMENTS</b>	
	4.1 Requirements to Run the Script	18
<b>5</b>	<b>CODING &amp; TESTING</b>	
	5.1 Index	20
	5.2 Syntax Analyser	26
<b>6</b>	<b>OUTPUT &amp; RESULTS</b>	
	6.1 Testcase I	39
	6.2 Testcase II	40
	6.3 Testcase III	42
<b>7</b>	<b>CONCLUSION</b>	44
<b>8</b>	<b>REFERENCES</b>	45

## CHAPTER 1

### INTRODUCTION

#### 1.1 Introduction

The syntax analyzer, also known as the parser, is a vital component in the field of compiler design. It plays a critical role in the compilation process by analyzing the syntactic structure of source code written in a programming language. Its main objective is to ensure that the code adheres to the grammar rules specified by the language and to construct a structural representation of its syntax.



**Fig. 1.1** Syntax Analysis in compiler Design process comes after the lexical phase and then the generated parse tree is utilized by rest of the Front End of compiler.

The syntax analyzer serves as a language expert that scrutinizes the code's syntax, checking the arrangement and combination of language constructs, expressions, and statements. It ensures that the code is well-formed and follows the specified syntax, preventing the use of undefined constructs or deviations from the language's grammar.

In the early stages of the compilation process, the syntax analyzer receives a stream of tokens generated by the lexical analyzer. It uses these tokens as input and applies parsing algorithms to validate their sequence against the grammar rules. By employing efficient parsing techniques,

such as top-down or bottom-up parsing, the syntax analyzer constructs a parse tree or abstract syntax tree (AST) that represents the hierarchical relationships between language elements.

One of the critical aspects of the syntax analyzer is error detection and reporting. It promptly identifies syntax errors in the code and provides meaningful error messages to aid developers in locating and resolving issues. Error recovery mechanisms may also be employed to handle syntax errors gracefully, allowing the parser to continue analyzing the code despite encountering errors.

The output of the syntax analyzer, the parse tree or AST, serves as an intermediate representation for subsequent compiler phases. It provides a structured basis for performing further analysis, transformations, optimizations, and code generation.

Overall, the syntax analyzer plays a vital role in ensuring the syntactic correctness and structural understanding of the source code. It guarantees compliance with the language's grammar rules, aids in error detection and reporting, and provides a foundation for subsequent compilation phases, contributing to the efficient and accurate processing of code.

## 1.2 Problem Statement

The problem statement of a syntax analyzer, or parser, is to analyze the syntactic structure of the source code and determine whether it conforms to the grammar rules defined by the programming language. The primary goal is to verify the syntactic validity of the code and construct a structural representation of its syntax.

1. **Token Recognition:** The parser needs to recognize individual tokens in the input source code. This involves identifying keywords, identifiers, operators, literals, and other language-specific symbols. Tokens are typically generated by the lexical analyzer (lexer) and passed as input to the syntax analyzer.
2. **Grammar Compliance:** The parser checks whether the sequence of tokens adheres to the grammar rules defined for the programming language. It verifies that the code follows the correct arrangement and combination of language constructs, expressions, and statements. The parser ensures that the code is well-formed and complies with the syntax specified by the language specification.
3. **Error Detection and Reporting:** The syntax analyzer is responsible for identifying syntax errors in the code. It detects situations where the input code violates the grammar rules or contains syntax-related issues. When an error is encountered, the parser generates meaningful error messages that pinpoint the location and nature of the error. This helps developers understand and correct their code.
4. **Parse Tree or Abstract Syntax Tree (AST) Construction:** As part of the analysis, the parser constructs a parse tree or an abstract syntax tree (AST) that represents the hierarchical structure of the code. The tree shows how language constructs are organized, nested, and related to each other. The construction of the tree allows for further analysis, transformations, or optimizations in subsequent compiler phases.

5. Error Recovery: In the case of encountering syntax errors, the syntax analyzer may attempt to recover from those errors and continue parsing the remaining code. Error recovery techniques are employed to minimize the impact of errors and enable the parser to make progress despite encountering syntax violations.

Overall, the problem statement of a syntax analyzer involves recognizing tokens, enforcing grammar rules, detecting and reporting syntax errors, constructing a parse tree or AST, and potentially implementing error recovery mechanisms. By solving this problem, the syntax analyzer ensures the syntactic correctness and structural understanding of the source code, facilitating subsequent phases in the compilation process.



### 1.3 Objectives

The objectives of a syntax analyzer, or parser, can be summarized as follows:

1. **Syntactic Validity:** The primary objective of a syntax analyzer is to ensure that the source code adheres to the grammar rules defined by the programming language. It verifies the syntactic correctness of the code, ensuring that it is well-formed and follows the specified syntax.
2. **Error Detection and Reporting:** The syntax analyzer aims to detect and report syntax errors in the code. It identifies violations of the grammar rules and generates meaningful error messages that provide developers with information about the location and nature of the errors.
3. **Structural Representation:** Another objective is to construct a structural representation of the code's syntax. The parse tree/AST captures the hierarchical relationships between language constructs, expressions, and statements, providing a structural understanding of the code.
4. **Language Specification Compliance:** The syntax analyzer enforces compliance with the language specification or standard. It ensures that the code follows the specified grammar rules and adheres to the syntax defined by the language.
5. **Parsing Efficiency:** The syntax analyzer aims to parse the code efficiently. It employs parsing algorithms and techniques that optimize the parsing process, minimizing time and resources required for analysis.
6. **Error Recovery:** In the event of encountering syntax errors, the syntax analyzer may employ error recovery mechanisms to handle errors gracefully.
7. **Intermediate Representation:** The syntax analyzer provides an intermediate representation, such as the parse tree or AST, that serves as a basis for subsequent compiler phases. This objective facilitates further analysis, transformations, optimizations, and code generation.

## 1.4 Need for Syntax Analyser

1. **Syntax Verification:** A syntax analyzer is essential for verifying the syntactic correctness of the source code. It checks whether the code conforms to the grammar rules defined by the programming language.

2. **Language Specification Compliance:** Programming languages have well-defined syntax rules and grammar specifications. The syntax analyzer ensures that the code adheres to these specifications with the language standard.

3. **Structural Understanding:** The syntax analyzer constructs a structural representation of the code's syntax, typically in the form of a parse tree or abstract syntax tree (AST). This representation captures the hierarchical relationships between language constructs, expressions, and statements.

4. **Error Detection and Reporting:** The syntax analyzer plays a crucial role in detecting syntax errors in the code. It identifies violations of the grammar rules and generates meaningful error messages that highlight the location and nature of the errors.

5. **Efficiency in Parsing:** The syntax analyzer employs parsing algorithms and techniques that optimize the parsing process. Efficient parsing techniques, such as LL(1), LR(1), or LALR(1), enable fast and accurate analysis of the code.

Overall, the syntax analyzer serves as a crucial component in the compilation process, ensuring syntactic validity, enforcing compliance with language specifications, providing a structural understanding of the code, detecting and reporting errors, promoting compiler consistency, and optimizing parsing efficiency. It contributes to the development of reliable and efficient software systems by facilitating the correct interpretation of programming language syntax.

## **CHAPTER 2**

### **REQUIREMENT**

#### **2.1 Why does Compiler needs Syntax Analyser**

A compiler requires a syntax analyzer, also known as a parser, for several critical reasons. The syntax analyzer plays a fundamental role in the compilation process by ensuring that the source code conforms to the grammar rules specified by the programming language. It verifies the syntactic correctness of the code by checking the proper arrangement and combination of language constructs, expressions, and statements. This step is essential to ensure that the code is well-formed and follows the specified syntax.

Furthermore, the syntax analyzer constructs a structural representation of the code's syntax, typically in the form of a parse tree or abstract syntax tree (AST). This representation captures the hierarchical relationships between language elements, providing a structured understanding of the code's organization. The parse tree/AST serves as a foundation for subsequent compiler phases, enabling further analysis, transformations, and optimizations.

Another crucial function of the syntax analyzer is error detection and reporting. It examines the code for violations of the grammar rules and generates meaningful error messages to assist developers in locating and fixing syntax-related issues. The syntax analyzer's ability to provide detailed error reporting is invaluable for efficient debugging and code correction.

Moreover, the syntax analyzer enforces compliance with the language specification. By verifying that the code follows the specified grammar rules, it promotes language consistency, improves code readability, and facilitates code maintenance and collaboration among developers.

The output of the syntax analyzer, the parse tree or AST, is utilized by subsequent compiler phases such as semantic analysis, optimization, and code generation. The accurate construction of the parse tree/AST is crucial for correct semantic analysis, identifying semantic errors, and performing type checking.

## 2.2 Need for DFA in Syntax Analyser

Deterministic Finite Automaton (DFA) is a critical component in the syntax analyzer of a compiler, serving a significant need for efficient and accurate analysis of the source code's syntax. The DFA plays a crucial role in tokenization, which is the process of recognizing and categorizing individual tokens such as keywords, identifiers, operators, and literals. By constructing a DFA for each token type, the syntax analyzer can efficiently match the input characters against the DFA states, determining the token boundaries and enabling the extraction of meaningful language elements.

The need for DFA arises due to its ability to provide efficient lexical scanning. Once constructed, the DFA allows for the quick scanning of input characters in a linear fashion, transitioning between states based on the current input. This enables fast recognition of tokens and reduces the time complexity of the tokenization process. DFA-based lexical scanning is particularly advantageous for handling large codebases or complex programs, where efficient and optimized processing of tokens is crucial.

Additionally, DFA assists in error detection within the lexical analysis phase. If the DFA encounters an invalid state or reaches a dead-end state, it indicates that the input does not match any valid token, leading to the detection of lexical errors.

DFA is also employed in resolving lexical ambiguities that may arise in the source code. DFA helps in resolving such ambiguities by prioritizing and selecting the correct interpretation based on the defined rules, ensuring unambiguous tokenization and accurate syntax analysis.

In summary, the need for DFA in the syntax analyzer arises from its efficiency in tokenization, efficient lexical scanning, error detection, resolution of lexical ambiguities, and performance optimization. Utilizing DFA enhances the accuracy, speed, and resource efficiency of the syntax analysis phase in a compiler, ultimately contributing to the overall effectiveness of the compilation process.

## **CHAPTER 3**

### **SYSTEM AND ARCHITECTURE DESIGN**

#### **3.1 Front End Design**

##### **Technologies Used for designing Front End**

- **HTML**

HTML, which stands for HyperText Markup Language, is the standard markup language used for creating and structuring web pages. It provides the basic building blocks and elements for organizing and presenting content on the internet.

- **CSS**

CSS, short for Cascading Style Sheets, is a language used to define the presentation and layout of a web page. It works in conjunction with HTML to enhance the visual appearance and formatting of web content

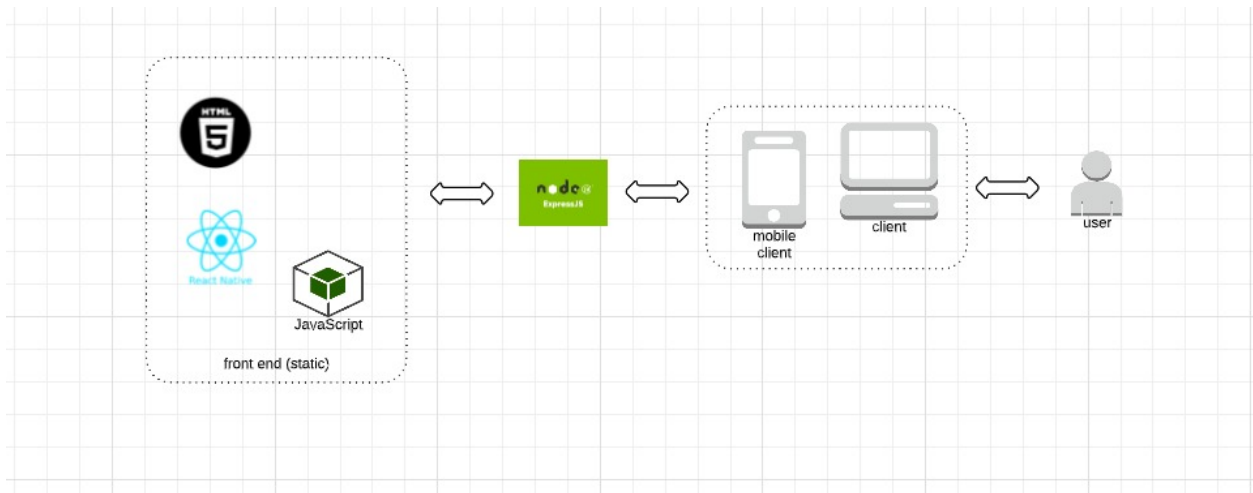
- **JAVA SCRIPT**

JavaScript is a high-level programming language primarily used for web development. It is a versatile language that allows developers to create dynamic and interactive web pages, build web applications, and even develop server-side and desktop applications.

- **REACTJS**

ReactJS, often referred to as React, is an open-source JavaScript library for building user interfaces. Developed by Facebook, ReactJS has gained significant popularity among developers for its efficient and declarative approach to building interactive and reusable UI components.

### 3.2 Architecture Design of Front End



**Fig. 3.1** The user interacts with the GUI to enter the source code through their respective device. The frontend architecture is based on HTML, CSS and Javascript build using ReactJs.

### 3.3 Back End Design

#### Technologies Used for designing Back End

- **JAVASCRIPT**

JavaScript is a high-level programming language primarily used for web development. It is a versatile language that allows developers to create dynamic and interactive web pages, build web applications, and even develop server-side and desktop applications.

- **PYTHON**

Python is a versatile programming language that can be used for backend development to create server-side applications, APIs (Application Programming Interfaces), and handle data processing tasks.

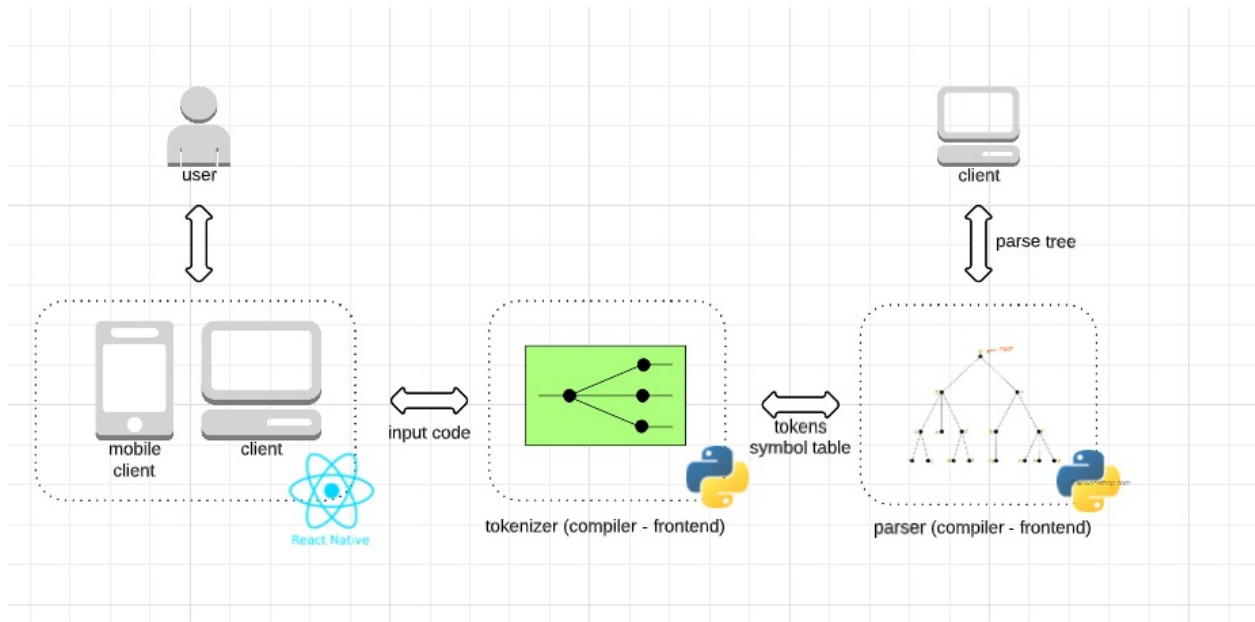
- **MONGODB**

MongoDB is a popular NoSQL database that can be used as a backend data store for various types of applications. It offers flexibility, scalability, and high performance, making it suitable for many backend development scenarios.

- **EXPRESS JS + NODE JS**

Express.js is a popular web application framework for Node.js that simplifies backend development by providing a minimal and flexible set of features. Node.js, on the other hand, is a JavaScript runtime environment that allows you to run JavaScript code on the server side.

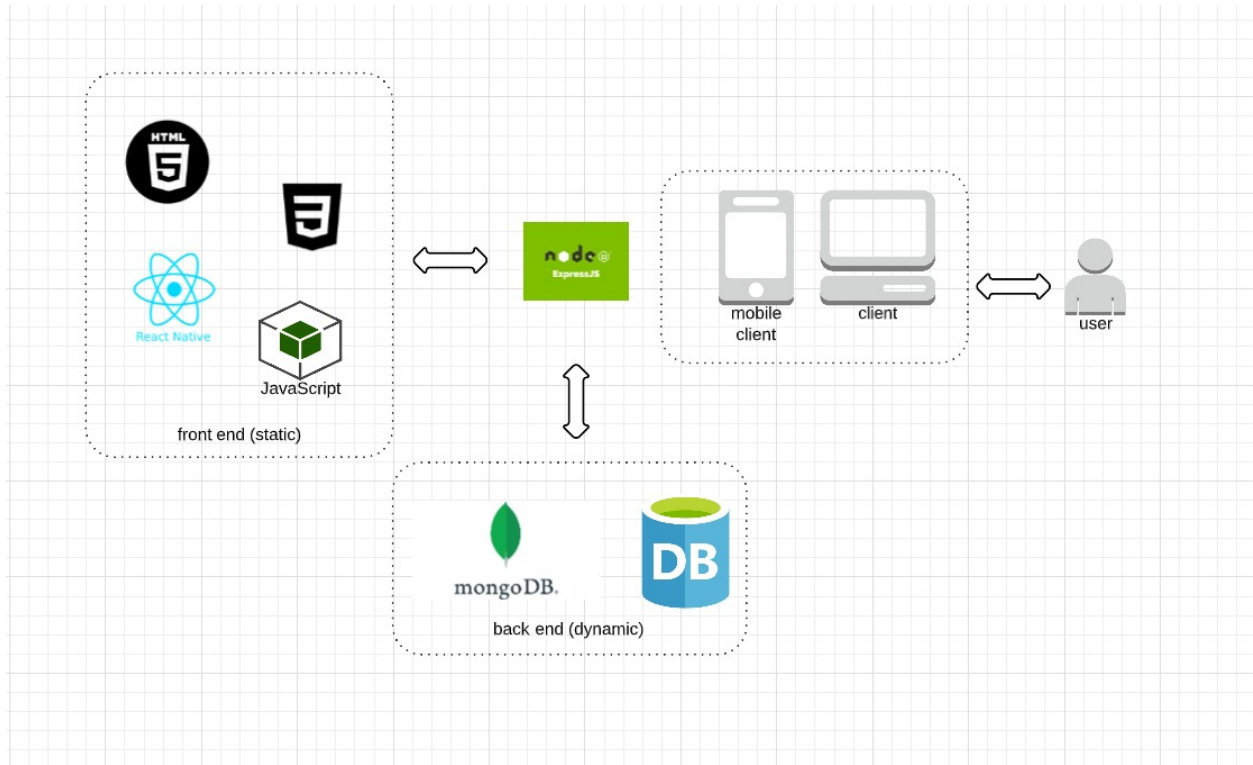
### 3.4 Architecture design of Back End



**Fig. 3.2** Once the user enters the source code, the code is passed through frontend to the middleware. The middleware send the input as a parameter for processing through lexical analyzer and syntatic analyzer. The backend architecture is based on compiler stages running on python3.



### 3.5 System Architecture Design



**Fig. 3.3** The overall architecture of entire system with skeletal structure based on ReactJs along with Python 3 code running in backend for source code analysis. Database in effect to store intermediate values and variables generated throughout the process.

## CHAPTER 4

### SYSTEM REQUIREMENTS

#### 4.1 Requirements to run the Script

The system requirements for running code related to syntax analysis depend on the specific programming language, tools, and frameworks you are using. Syntax analysis, also known as parsing, is a common step in the compilation process of programming languages. Here are some general requirements to consider:

1. **Hardware:** Syntax analysis itself does not have significant hardware requirements. As long as you have a computer that meets the minimum specifications for running your chosen programming language and development tools, you should be able to perform syntax analysis. The hardware requirements may vary based on the size and complexity of the code you're analyzing.
2. **Operating System:** Syntax analysis can be performed on different operating systems, including Windows, macOS, and various Linux distributions. Choose an operating system that is compatible with your programming language and tools. Most commonly used programming languages and development tools are available for multiple operating systems.
3. **Programming Language:** The specific programming language you are using for syntax analysis may have its own system requirements. Refer to the documentation or official website of the programming language to determine its specific requirements.
4. **Development Tools:** Depending on the programming language, you may need specific development tools or frameworks to perform syntax analysis effectively. These tools can include integrated development environments (IDEs), text editors, compilers, or parser generators. Check the system requirements of the tools or frameworks you intend to use.

5. **Memory and Storage:** Syntax analysis typically does not require significant memory or storage resources unless you are working with extremely large codebases. Ensure that your computer has enough RAM to handle the size of the code you are analyzing, and have sufficient storage space to store the code and any necessary dependencies or libraries.

6. **Dependencies and Libraries:** If your syntax analysis involves using external libraries or dependencies, make sure your system meets the requirements for those dependencies. Check the documentation of the libraries or frameworks you are using to determine their specific requirements.

## CHAPTER 5

### CODING AND TESTING

#### 5.1 Index.js

```
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 TokenizerIcon from "@mui/icons-material/Toll";
import IOBox from "./IOBox";
import { lex, tokenize } from "../utils/tokenizer";
import ParserIcon from "@mui/icons-material/ManageSearch";
import parse from "../utils/parser";

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%" }}
    >

```

```

style={{
  backgroundColor: "#2b4c61",
  color: "#fff",
}}
>
<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>

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

);
};
}

export default Home;

```

## 5.2 Syntax\_Analyser.py

```
from lxml import etree

# For compatibility with Py2.7
try:
    from io import StringIO
except ImportError:
    from StringIO import StringIO

def get_record(filehandle):
    """Iteratively go through file and get text of each WoS record"""
    record = ""
    flag = False
    for line in filehandle:
        if not flag and not line.startswith('<REC'):
            continue
        flag = True
        record = record + line
        if line.strip().endswith('</REC>'):
            return record
    return None

def parse_records(file, verbose, n_records):
    records = []
    count = 0
    while True:
        record = get_record(file)
        count += 1
        try:
            rec = etree.fromstring(record)
            records.append(rec)
```

```

except:
    pass

if verbose:
    if count % 5000 == 0: print('read total %i records' % count)
    if record is None:
        break
    if n_records is not None:
        if count >= n_records:
            break

return records

def read_xml(path_to_xml, verbose=True, n_records=None):
    """
    Read XML file and return full list of records in element tree

    Parameters
    =====
    path_to_xml: str, full path to WoS XML file
    verbose: (optional) boolean, True if we want to print number of records parsed
    n_records: (optional) int > 1, read specified number of records only
    """
    with open(path_to_xml, 'r') as file:
        records = parse_records(file, verbose, n_records)

    return records

def read_xml_string(xml_string, verbose=True, n_records=None):
    """
    Parse XML string and return list of records in element tree.

```

See Also

=====

- \* ``read_xml``
- \* ``get_record``

Parameters

=====

- \* `xml_string`: str, XML string
- \* `verbose`: (optional) boolean, True if we want to print number of records parsed
- \* `n_records`: (optional) int > 1, read specified number of records only

"""

with StringIO(xml\_string) as file:

    records = parse\_records(file, verbose, n\_records)

return records

def extract\_wos\_id(elem):

    """Return WoS id from given element tree"""

    if elem.find('UID') is not None:

        wos\_id = elem.find('UID').text

    else:

        wos\_id = "

    return wos\_id

def extract\_authors(elem):

    """Extract list of authors from given element tree"""

    wos\_id = extract\_wos\_id(elem)

    authors = list()

    names = elem.findall('./static\_data/summary/names/')

    for name in names:

```

dais_id = name.attrib.get('dais_id', "")
seq_no = name.attrib.get('seq_no', "")
role = name.attrib.get('role', "")
addr_no = name.attrib.get('addr_no', "")
if name.find('full_name') is not None:
    full_name = name.find('full_name').text
else:
    full_name = ""
if name.find('first_name') is not None:
    first_name = name.find('first_name').text
else:
    first_name = ""
if name.find('last_name') is not None:
    last_name = name.find('last_name').text
else:
    last_name = ""
author = {'dais_id': dais_id,
          'seq_no': seq_no,
          'addr_no': addr_no,
          'role': role,
          'full_name': full_name,
          'first_name': first_name,
          'last_name': last_name}
author.update({'wos_id': wos_id})
authors.append(author)
return authors

def extract_keywords(elem):
    """Extract keywords and keywords plus each separated by semicolon"""
    keywords = elem.findall('./static_data/fullrecord_metadata/keywords/keyword')
    keywords_plus = elem.findall('./static_data/item/keywords_plus/keyword')

```

```

if keywords:
    keywords_text = '; '.join([keyword.text for keyword in keywords])
else:
    keywords_text = ""
if keywords_plus:
    keywords_plus_text = '; '.join([keyword.text for keyword in keywords_plus])
else:
    keywords_plus_text = ""
return keywords_text, keywords_plus_text

def extract_addresses(elem):
    """Give element tree of WoS, return list of addresses"""
    address_dict_all = list()
    wos_id = extract_wos_id(elem)
    addresses = elem.findall('./static_data/fullrecord_metadata/addresses/address_name')
    for address in addresses:
        address_dict = dict()
        address_spec = address.find('address_spec')
        addr_no = address_spec.attrib.get('addr_no', "")
        for tag in ['city', 'state', 'country', 'zip', 'full_address']:
            if address_spec.find(tag) is not None:
                address_dict[tag] = address_spec.find(tag).text
            else:
                address_dict[tag] = ""
        if address_spec.find('organizations') is not None:
            organizations = '; '.join([oraginzation.text for oraginzation in
address_spec.find('organizations')])
        else:
            organizations = ""
        if address_spec.find('suborganizations') is not None:
            suborganizations = '; '.join([s.text for s in address_spec.find('suborganizations')])

```

```

else:
    suborganizations = "
address_dict.update({'wos_id': wos_id,
                    'addr_no': addr_no,
                    'organizations': organizations,
                    'suborganizations': suborganizations})
    address_dict_all.append(address_dict)
return address_dict_all

def extract_publisher(elem):
    """Extract publisher details"""
    wos_id = extract_wos_id(elem)
    publisher_list = list()
    publishers = elem.findall('./static_data/summary/publishers/publisher')
    for publisher in publishers:
        publisher_dict = dict()
        name = publisher.find('names/name')
        for tag in ['display_name', 'full_name']:
            if name.find(tag) is not None:
                publisher_dict[tag] = name.find(tag).text
            else:
                publisher_dict[tag] = "
        addr = publisher.find('address_spec')
        for tag in ['full_address', 'city']:
            if addr.find(tag) is not None:
                publisher_dict[tag] = addr.find(tag).text
            else:
                publisher_dict[tag] = "
        publisher_dict.update({'wos_id': wos_id})
        publisher_list.append(publisher_dict)
    return publisher_list

```

```

def extract_pub_info(elem):
    """
    Extract publication information from WoS

    See Also
    =====
    * `read_xml`
    * `get_record`

    Parameters
    =====
    * elem: object, XML etree element object

    Returns
    =====
    * dict, of publication information
    """
    pub_info_dict = dict()
    pub_info_dict.update({'wos_id': extract_wos_id(elem)})

    pub_info = elem.find('.static_data/summary/pub_info').attrib
    for key in ['sortdate', 'has_abstract', 'pubtype', 'pubyear', 'pubmonth', 'issue']:
        if key in pub_info.keys():
            pub_info_dict.update({key: pub_info[key]})
        else:
            pub_info_dict.update({key: ""})

    for title in elem.findall('./static_data/summary/titles/title'):
        if title.attrib['type'] in ['source', 'item']:

```



```

# more attribute includes source_abbrev, abbrev_iso, abbrev_11, abbrev_29
title_dict = {title.attrib['type']: title.text}
pub_info_dict.update(title_dict)

language = elem.find('./static_data/fullrecord_metadata/languages/language')
if language.tag is not None:
    pub_info_dict.update({'language': language.text})
else:
    pub_info_dict.update({'language': ""})

heading_tag = elem.find('./static_data/fullrecord_metadata/category_info/headings/heading')
if heading_tag is not None:
    heading = heading_tag.text
else:
    heading = ""
pub_info_dict.update({'heading': heading})

subject_tr = []
subject_ext = []

for subject_tag in
elem.findall('./static_data/fullrecord_metadata/category_info/subjects/subject'):
    if subject_tag is not None:
        if subject_tag.attrib["ascatype"] == "traditional":
            subject_tr.append(subject_tag.text)
        if subject_tag.attrib["ascatype"] == "extended":
            subject_ext.append(subject_tag.text)

pub_info_dict.update({'subject_traditional': subject_tr})
pub_info_dict.update({'subject_extended': subject_ext})

```

```

subheading_tag =
elem.find('./static_data/fullrecord_metadata/category_info/subheadings/subheading')
if subheading_tag is not None:
    subheading = subheading_tag.text
else:
    subheading = "
pub_info_dict.update({'subheading': subheading})

doctype_tag = elem.find('./static_data/summary/doctypes/doctype')
if doctype_tag is not None:
    doctype = doctype_tag.text
else:
    doctype = "
pub_info_dict.update({doctype_tag.tag: doctype})

abstract_tag =
elem.findall('./static_data/fullrecord_metadata/abstracts/abstract/abstract_text/p')
if len(abstract_tag) > 0:
    abstract = ''.join([p.text for p in abstract_tag])
else:
    abstract = "
pub_info_dict.update({'abstract': abstract})

keywords, keywords_plus = extract_keywords(elem)
pub_info_dict.update({'keywords': keywords,
                      'keywords_plus': keywords_plus})

identifiers = extract_identifiers(elem)
for k, v in identifiers.items():
    pub_info_dict.update({k: v})
# End for

```

```
return pub_info_dict
```

```
def extract_funding(elem):
```

```
    """Extract funding text and funding agency separated by semicolon from WoS
    if see no funding, it will return just Web of Science id and empty string
    """
```

```
    was_id = extract_was_id(elem)
```

```
    grants = elem.findall('./static_data/fullrecord_metadata/fund_ack/grants/grant')
```

```
    fund_text_tag = elem.find('./static_data/fullrecord_metadata/fund_ack/fund_text')
```

```
    if fund_text_tag is not None:
```

```
        fund_text = ' '.join([p_.text for p_ in fund_text_tag.findall('p')])
```

```
    else:
```

```
        fund_text = "
```

```
    grant_list = list()
```

```
    for grant in grants:
```

```
        if grant.find('grant_agency') is not None:
```

```
            grant_list.append(grant.find('grant_agency').text)
```

```
    return {'was_id': was_id,
```

```
            'funding_text': fund_text,
```

```
            'funding_agency': ' '.join(grant_list)}
```

```
def extract_conferences(elem):
```

```
    """Extract list of conferences from given WoS element tree
```

```
    if no conferences exist, return None"""
```

```
    conferences_list = list()
```

```
    was_id = extract_was_id(elem)
```

```
    conferences = elem.findall('./static_data/summary/conferences/conference')
```

```

for conference in conferences:
    conference_dict = dict()
    conf_title_tag = conference.find('conf_titles/conf_title')
    if conf_title_tag is not None:
        conf_title = conf_title_tag.text
    else:
        conf_title = ""

    conf_date_tag = conference.find('conf_dates/conf_date')
    if conf_date_tag is not None:
        conf_date = conf_date_tag.text
    else:
        conf_date = ""
    for key in ['conf_start', 'conf_end']:
        if key in conf_date_tag.attrib.keys():
            conference_dict.update({key: conf_date_tag.attrib[key]})
        else:
            conference_dict.update({key: ""})

    conf_city_tag = conference.find('conf_locations/conf_location/conf_city')
    conf_city = conf_city_tag.text if conf_city_tag is not None else ""

    conf_state_tag = conference.find('conf_locations/conf_location/conf_state')
    conf_state = conf_state_tag.text if conf_state_tag is not None else ""

    conf_sponsor_tag = conference.findall('sponsors/sponsor')
    if len(conf_sponsor_tag) > 0:
        conf_sponsor = '; '.join([s.text for s in conf_sponsor_tag])
    else:
        conf_sponsor = ""

```

```

conf_host_tag = conference.find('./conf_locations/conf_location/conf_host')
conf_host = conf_host_tag.text if conf_host_tag is not None else "

conference_dict.update({'wos_id': wos_id,
                        'conf_title': conf_title,
                        'conf_date': conf_date,
                        'conf_city': conf_city,
                        'conf_state': conf_state,
                        'conf_sponsor': conf_sponsor,
                        'conf_host': conf_host})

conferences_list.append(conference_dict)
if not conferences_list:
    conferences_list = None
return conferences_list

def extract_references(elem):
    """Extract references from given WoS element tree"""
    wos_id = extract_wos_id(elem)
    references = elem.findall('./static_data/fullrecord_metadata/references/reference')
    ref_list = list()
    for reference in references:
        ref_dict = dict()
        for tag in ['uid', 'citedAuthor', 'year', 'page',
                    'volume', 'citedTitle', 'citedWork', 'doi']:
            ref_tag = reference.find(tag)
            if ref_tag is not None:
                ref_dict[tag] = ref_tag.text
            else:
                ref_dict[tag] = "
        ref_dict.update({'wos_id': wos_id})

```

```

    ref_list.append(ref_dict)
return ref_list

```

```
def extract_identifiers(elem):
```

```
    """Extract document identifiers from WoS element tree
```

```

    Parameters

```

```

    =====

```

```

    elem: etree.Element object, WoS element

```

```

    Returns

```

```

    =====

```

```

    dict {identifier type: value} or empty dict if none found. Identifier types may be DOI, ISSN,
etc.

```

```

    """

```

```

    idents = elem.findall('./dynamic_data/cluster_related/identifiers')

```

```

    id_dict = {}

```

```

    for ident in idents:

```

```

        for child in ident.getchildren():

```

```

            id_dict.update({child.get('type'): child.get('value')})

```

```

    # End for

```

```

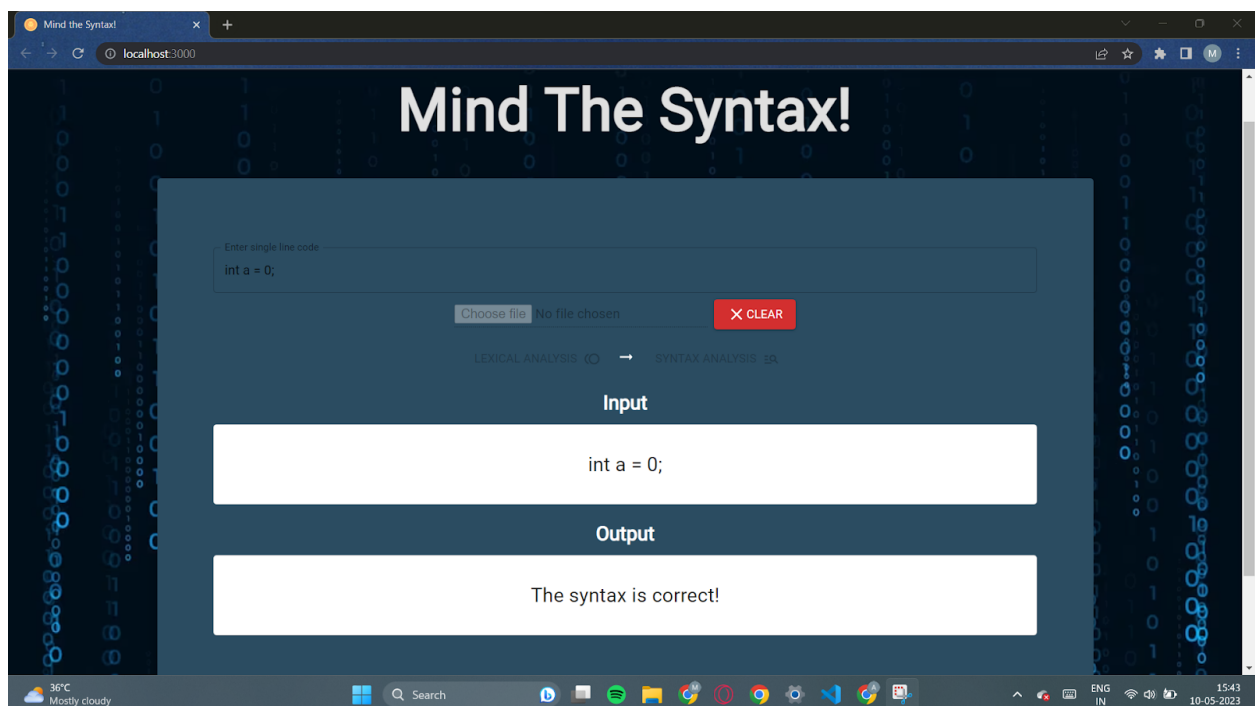
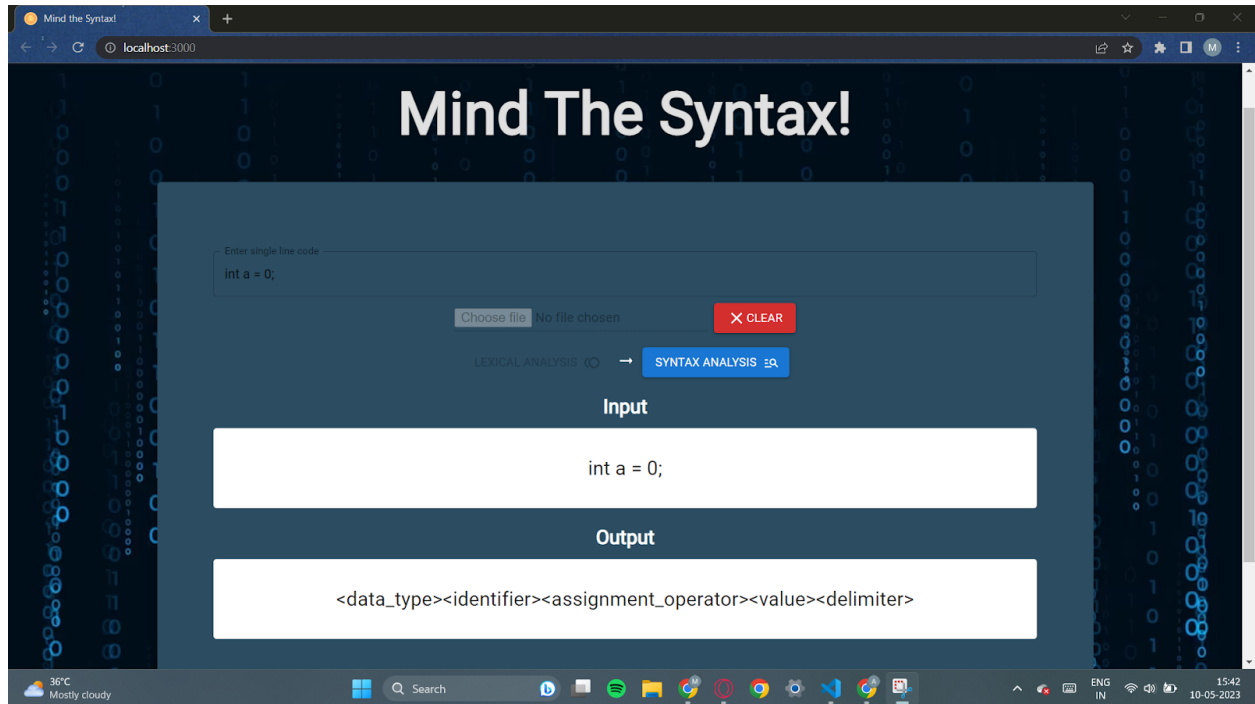
    return id_dict

```

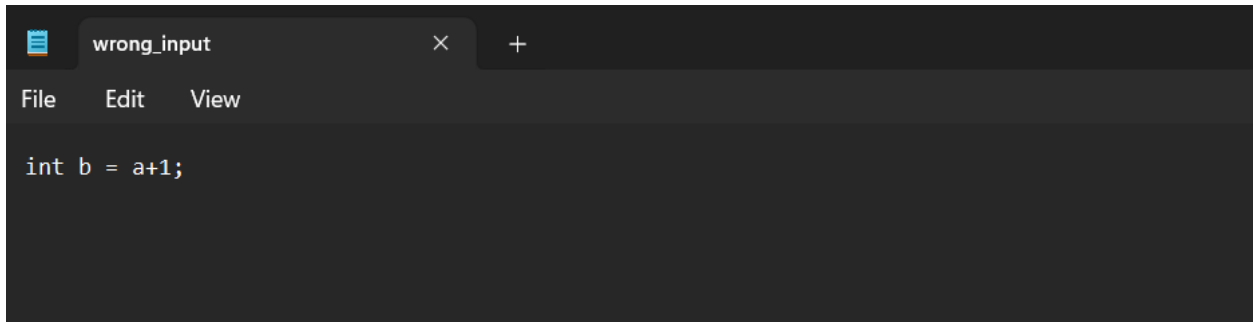
## CHAPTER 6

### OUTPUT AND RESULTS

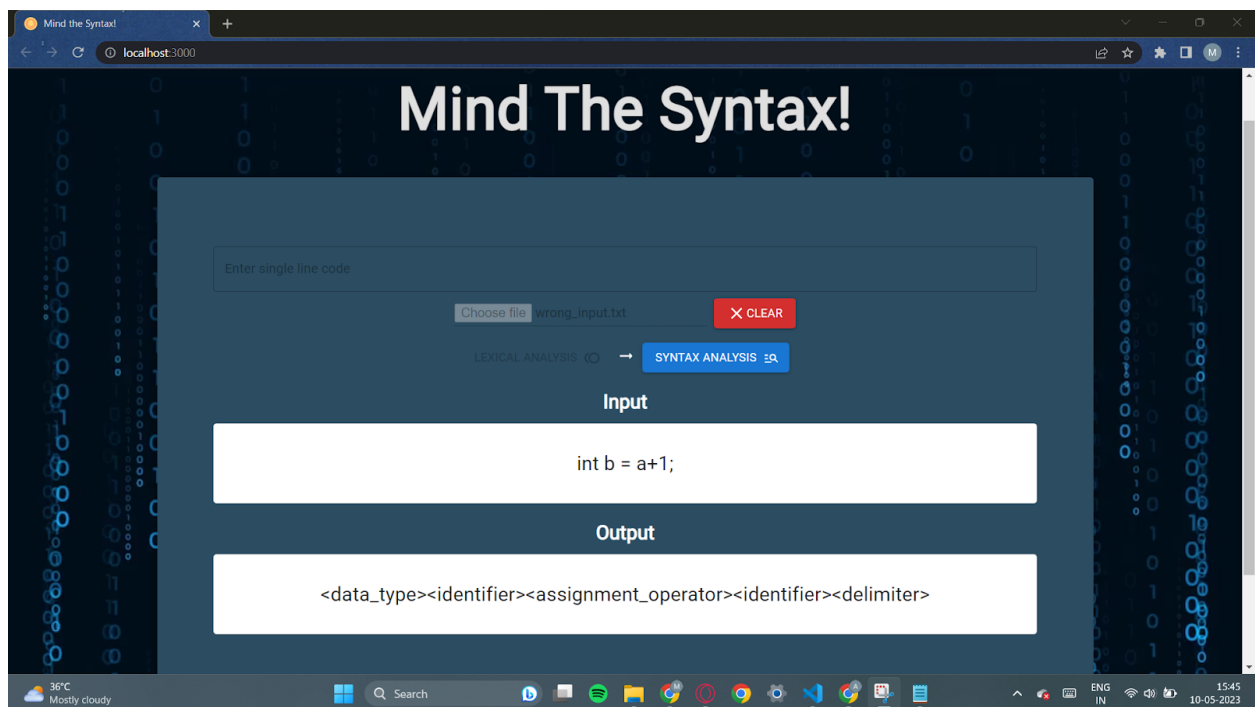
#### 6.1 Testcase I



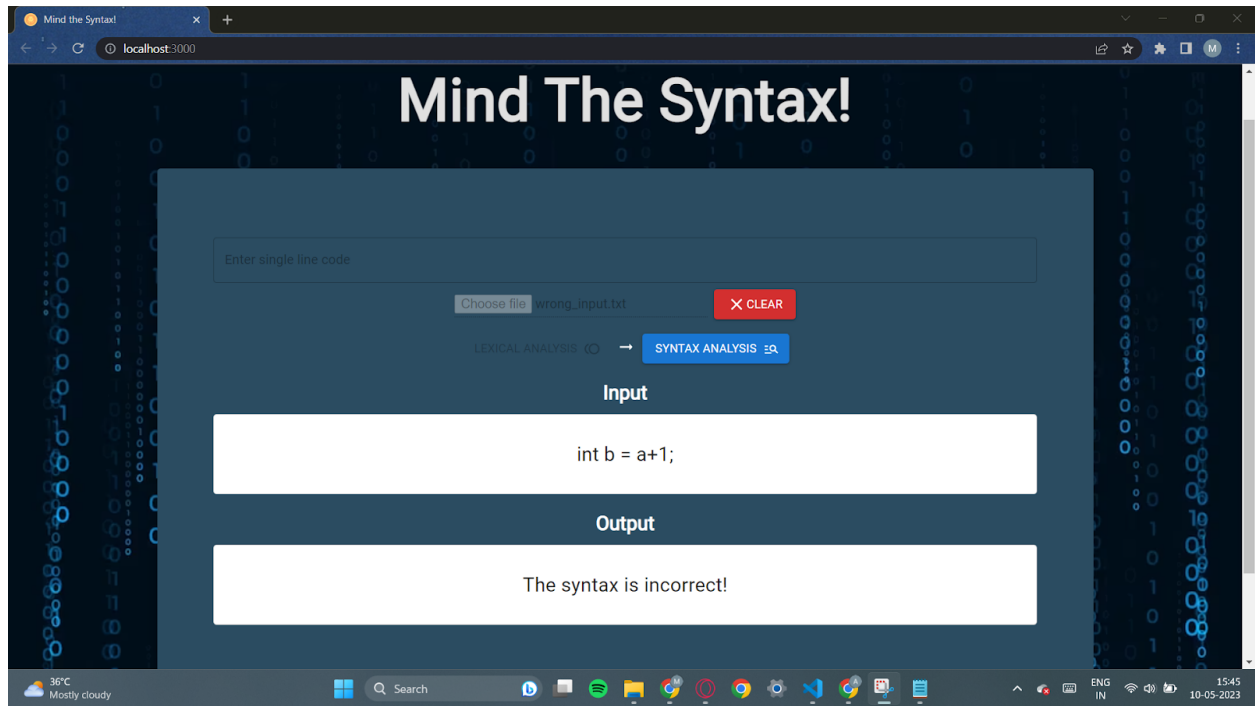
## 6.2 Testcase II



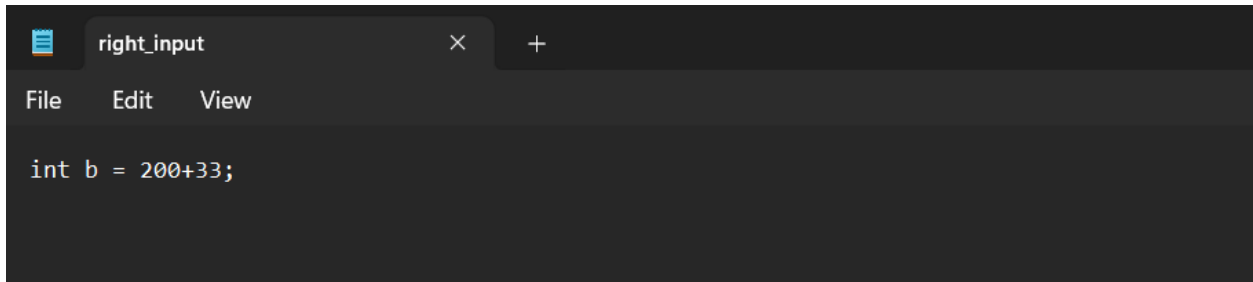
A screenshot of a code editor window with a dark theme. The window title is "wrong\_input". The menu bar shows "File", "Edit", and "View". The code being edited is a single line: `int b = a+1;`.



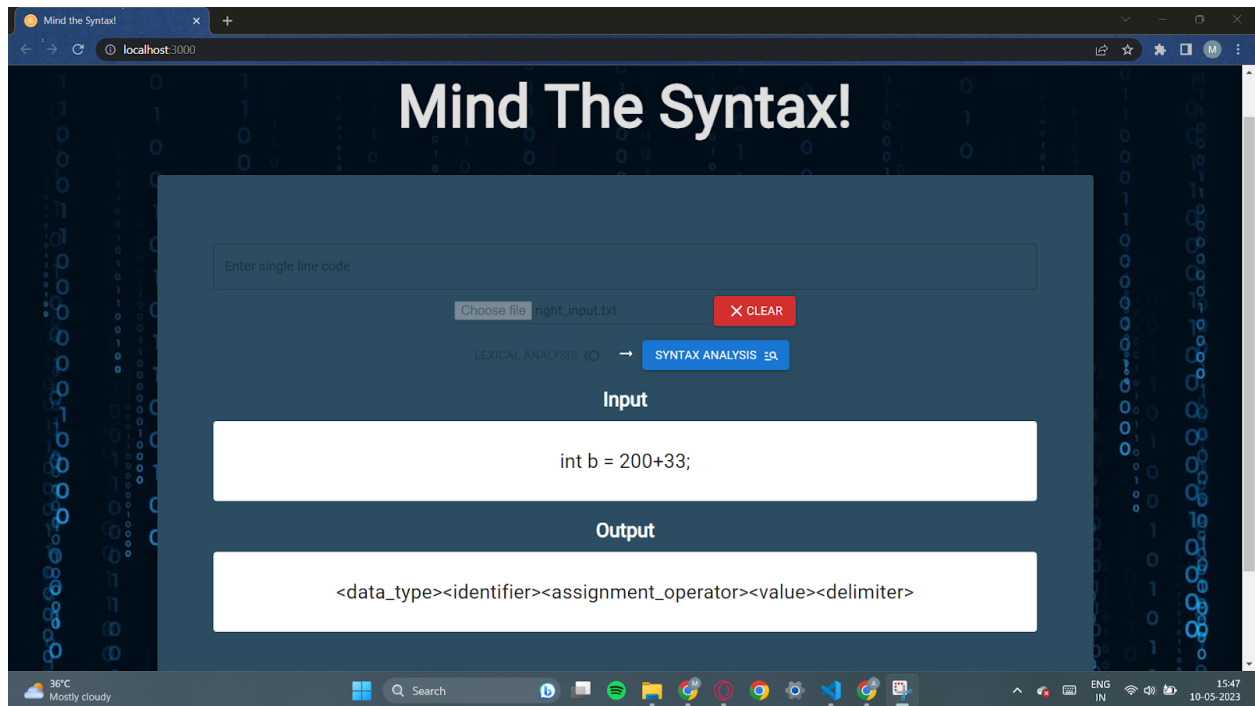


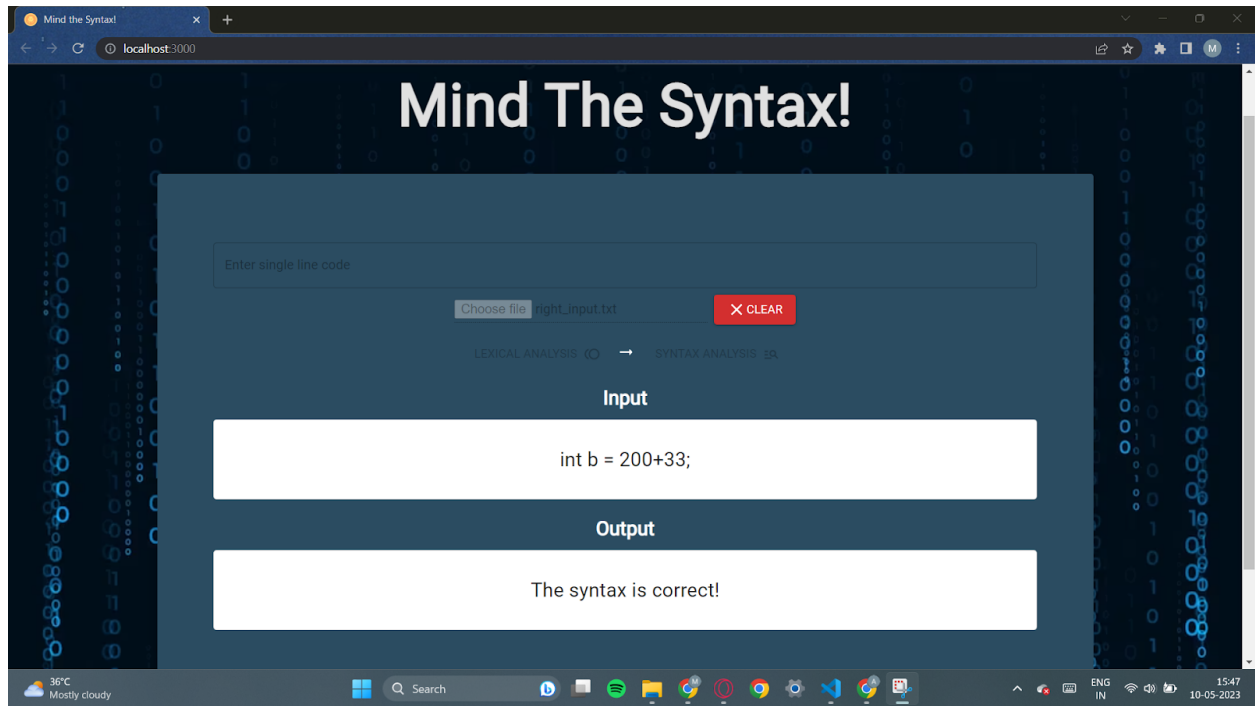


### 6.3 Testcase III



A screenshot of a code editor window with the title 'right\_input'. The editor contains a single line of code: `int b = 200+33;`. The window has a menu bar with 'File', 'Edit', and 'View' options.





## **CHAPTER 7**

### **CONCLUSION**

In conclusion, syntax analysis, or parsing, is a fundamental step in the compilation process of programming languages. Its purpose is to analyze the structure of the source code and ensure its adherence to the rules defined by the language's grammar. By breaking down the code into tokens and constructing a parse tree or abstract syntax tree (AST), the syntax analyzer verifies the syntactic correctness of the code and detects any syntax errors.

Through lexical analysis, the syntax analyzer tokenizes the source code, categorizing each element into keywords, identifiers, operators, and literals. The parser then utilizes the grammar rules of the programming language to validate the code's syntactic structure. It employs various parsing techniques to match tokens against the grammar rules and identifies any syntax errors encountered.

The proper working of a syntax analyzer is crucial in providing developers with timely feedback on potential syntax errors in their code. By detecting and reporting errors, it helps in debugging and ensuring the code's correctness before further stages of compilation or execution. Additionally, the syntax analyzer constructs an abstract syntax tree (AST) that captures the code's structural semantics, facilitating subsequent phases of compilation and analysis.

In summary, the syntax analysis phase plays a vital role in the compilation process by verifying the syntactic correctness of the source code and providing a foundation for subsequent analysis and transformation steps. It enables developers to write code that adheres to the language's grammar, promotes code reliability, and contributes to the overall efficiency of software development.

## **CHAPTER 8**

### **REFERENCES**

- [1] <https://www.techtarget.com/whatis/definition/compiler>
- [2] <https://www.geeksforgeeks.org/introduction-to-syntax-analysis-in-compiler-design/>
- [3 ]<https://www.javatpoint.com/parser>
- [4] <https://www.guru99.com/syntax-analysis-parsing-types.html>