PARSIFY(Semantic Analyser)

A MINI PROJECT REPORT

Submitted by

ANSHIKA NIGAM [RA2011033010106]
PRATYAKSH BHOOTRA [RA2011033010109]

Under the guidance of
Dr. J Jeyasudha

(Assistant Professor, Department of Computational Intelligence)

In partial satisfaction of the requirements for the degree of

BACHELOR 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 – 603 203

May-2023

BONAFIDE CERTIFICATE

Certified that this project report "**PARSIFY (Semantic Analyzer)**" is the bonafide work of "**PRATYAKSH BHOOTRA [RA2011033010109]**, **ANSHIKA NIGAM [RA2011033010106]**," of III Year/VI Sem B.Tech(CSE) who carried out the mini project work under my supervision for the course **18CSC304J- Compiler Design** in SRM Institute of Science and Technology during the academic year 2023(Even Semester).

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 abstract presents a semantic analyser developed using a combination of Python, HTML, CSS, and JavaScript with a Django backend. The analyzer is designed to extract meaning and context from text-based content, enabling users to gain valuable insights and enhance their understanding of written material.

The analyser utilizes Python, a powerful and versatile programming language, to implement the core functionality. Python's natural language processing libraries, such as NLTK or spaCy, are employed for tasks like tokenization, part-of-speech tagging, and named entity recognition. These processes contribute to the extraction of relevant information and aid in the identification of entities, relationships, and sentiments within the text.

To create an interactive user interface, the analyzer employs web technologies like HTML, CSS, and JavaScript. HTML structures the content, CSS styles it, and JavaScript enhances the interface with dynamic and responsive behavior. This combination allows for an intuitive and visually appealing user experience, enabling users to interact with the analyzed results effectively.

The Django framework, a high-level Python web framework, provides the backend infrastructure for the semantic analyzer. Django handles the data flow, facilitates database management, and manages user authentication and authorization. It ensures a secure and scalable environment for the analyzer's operation.

The functionality of the semantic analyzer encompasses a range of applications, such as text summarization, sentiment analysis, keyword extraction, and named entity recognition. By utilizing the power of Python and integrating it with web technologies, the analyzer enables users to extract valuable insights and obtain a deeper understanding of textual content.

In conclusion, this semantic analyzer, developed using Python, HTML, CSS, and JavaScript with a Django backend, offers a comprehensive solution for extracting meaning and context from text-based content. Its combination of powerful natural language processing techniques, interactive user interface, and scalable backend infrastructure make it a versatile tool for various applications requiring semantic analysis

TABLE OF CONTENTS

Chapter No.	**Title**	**Page No.**

CHAPTER 1

INTRODUCTION

1.1   INTRODUCTION

The project aims to develop a semantic analyzer utilising JavaScript (JS), Flask, and an HTML framework. The semantic analyzer plays a crucial role in software development by analyzing and interpreting the meaning and context of source code. It ensures adherence to language rules and semantics, detects errors, and provides valuable insights for optimizing and improving code quality.

By combining JS, Flask, and an HTML framework, this project offers a powerful and flexible solution for implementing a semantic analyzer with a user-friendly web interface. JavaScript provides the necessary capabilities for performing semantic analysis, manipulating code structures, and interacting with the user interface. Flask, a Python web framework, serves as the backend for handling requests, managing data, and integrating with the frontend components.
The HTML framework, along with CSS for styling, enhances the user experience by creating an intuitive and visually appealing interface for interacting with the semantic analyzer. It allows users to input source code, view analysis results, and receive feedback on potential semantic errors or optimizations.

The key objectives of this project include:

1.   **Semantic Analysis:** Implementing algorithms and techniques to analyze the source code's semantics, including type checking, symbol resolution, scoping, and language-specific rules.

2. **Error Detection and Reporting**: Identifying semantic errors within the source code and generating meaningful error messages or warnings to help developers locate and rectify the issues.

3. **Optimization Insights:** Providing insights and suggestions for code optimization, such as identifying potential performance improvements, eliminating dead code, or suggesting alternative constructs.

4. **User Interface:** Designing an intuitive and user-friendly web interface using an HTML framework and CSS to facilitate easy interaction with the semantic analyzer. This includes code input, analysis result display, and user feedback.

By integrating JS, Flask, and an HTML framework, this project aims to create a comprehensive and efficient solution for semantic analysis. It empowers developers with a tool that assists in ensuring code correctness, improving software quality, and optimizing performance. Through a visually appealing and intuitive web interface, developers can seamlessly interact with the semantic analyzer, enhancing their coding experience and productivity.

In conclusion, this project combines the power of JavaScript, Flask, and an HTML framework to develop a semantic analyzer that helps developers analyze, validate, and optimize their source code. The resulting tool contributes to improved code quality, enhanced developer productivity, and more efficient software development processes.

1.2   PROBLEM STATEMENT

The problem at hand is the need for a semantic analyzer that can effectively extract meaning and context from text-based content. While traditional text analysis techniques provide basic information like word frequency and basic statistics, they often fail to capture the deeper semantic relationships and insights contained within the text.

The lack of a comprehensive semantic analyzer poses challenges for individuals and organizations that rely on accurate interpretation and understanding of written material. Users such as researchers, content creators, and analysts require a tool that can automate the process of extracting valuable insights, identifying entities, relationships, sentiments, and other contextual information from text.

The absence of a semantic analyzer can hinder tasks such as text summarization, sentiment analysis, keyword extraction, and named entity recognition. These tasks are crucial in various domains, including market research, social media analysis, content curation, and information retrieval.

Moreover, existing solutions may lack integration with modern web technologies and may not provide a user-friendly interface. This limitation hampers the usability and accessibility of the

analyzer, making it challenging for users to interact with and utilize the extracted semantic information effectively.

Therefore, the problem at hand is to develop a semantic analyzer that leverages the power of Python's natural language processing capabilities and integrates it with web technologies such as HTML, CSS, and JavaScript. Additionally, the analyzer should have a scalable and secure backend infrastructure provided by the Django framework. The goal is to create a versatile tool that enables users to extract meaningful insights, enhance understanding, and make informed decisions based on text-based content.

1.3   OBJECTIVES

The objective of the semantic analyzer in compiler design is to perform a thorough analysis of the source code and enforce semantic rules and constraints of the programming language. Its primary goal is to ensure the correctness and integrity of the program by detecting and reporting semantic errors that cannot be captured during the earlier lexical and syntactic analysis phases. The semantic analyzer aims to achieve the following objectives:

1. Develop a robust semantic analyzer capable of extracting meaning and context from text-based content.
2. Implement natural language processing techniques using Python libraries to perform tasks such as tokenization, part-of-speech tagging, named entity recognition, and sentiment analysis.
3. Create a user-friendly and visually appealing interface using HTML, CSS, and JavaScript to enhance the usability and accessibility of the analyzer.

4. Employ the Django framework to establish a scalable backend infrastructure that handles data flow, database management, user authentication, and authorization.

5. Enable text summarization functionality to generate concise summaries of large textual content, aiding users in quickly understanding the main points and key information.

6. Develop a sentiment analysis feature that detects and classifies the emotional tone of the text, providing insights into the overall sentiment expressed.

7. Implement keyword extraction functionality to identify important terms and phrases within the text, enabling users to focus on relevant topics.

8. Incorporate named entity recognition capabilities to identify and classify entities such as people, organizations, locations, and other specific entities within the text.

9. Ensure the security and privacy of user data by implementing robust data protection measures and adhering to best practices in handling sensitive information.

10. Conduct thorough testing and debugging to ensure the accuracy and reliability of the semantic analyzer's results.

11. Provide clear documentation and user guides to assist users in understanding and effectively utilizing the semantic analyzer's features and functionalities.

12. Continuously improve and update the semantic analyzer based on user feedback, emerging technologies, and advancements in natural language processing techniques.

By achieving these objectives, the semantic analyzer will provide users with a comprehensive tool for extracting meaningful insights, enhancing understanding, and making informed decisions based on text-based content.

1.4   Need for Semantic Analysis

The need for a semantic analyzer arises from the limitations of traditional text analysis techniques, which often focus on basic statistical information and fail to capture the deeper meaning and context within textual content. The following points highlight the key reasons for the need of a semantic analyzer:

1. **Extracting Meaningful Insights**: Textual content contains a wealth of information, including relationships between entities, sentiment expressions, and contextual nuances. A semantic analyzer is essential for extracting these meaningful insights and understanding the underlying message and intent of the text.

2. **Enhanced Understanding:** By analyzing the semantics of text, a semantic analyzer enables users to gain a deeper understanding of the content. It can identify important keywords, summarize

complex information, and recognize key entities, helping users comprehend and interpret textual material more effectively.

3. **Efficient Information Retrieval**: In large datasets or document collections, finding specific information quickly can be challenging. A semantic analyzer can facilitate efficient information retrieval by identifying and categorizing relevant entities, enabling users to locate specific topics or pieces of information more easily.

4**. Contextual Analysis**: Understanding the context is crucial for accurate interpretation of text. A semantic analyzer can analyze the contextual information present in the text, including temporal references, spatial relationships, and domain-specific knowledge. This contextual analysis improves the accuracy and relevance of the extracted insights.

5. **Decision-Making Support**: Organizations and individuals often make decisions based on textual information, such as market trends, customer feedback, or research findings. A semantic analyzer can provide valuable insights, sentiment analysis, and summarization, enabling informed decision-making and reducing the risk of misinterpretation.

6. **Content Curation and Filtering**: With the vast amount of digital content available, content curation and filtering become essential. A semantic analyzer can help in categorizing, tagging, and filtering content based on specific criteria or user preferences, aiding in efficient content management and personalized information consumption.

7. **Natural Language Processing Automation**: Many tasks involving natural language processing, such as tokenization, part-of-speech tagging, and named entity recognition, can be time-consuming when performed manually. A semantic analyzer automates these processes, saving time and effort while ensuring consistent and accurate results.

8. **Integration with Web Technologies**: Web technologies have become an integral part of many applications. A semantic analyzer that integrates with web technologies, such as HTML, CSS, JavaScript, and Django, allows for the development of user-friendly interfaces, interactive visualisations, and seamless integration with other web-based tools and platforms.

In summary, a semantic analyzer addresses the need for extracting meaningful insights, enhancing understanding, improving information retrieval, supporting decision-making, automating natural language processing tasks, and integrating with web technologies. It empowers users to gain valuable insights from text-based content, leading to improved efficiency, accuracy, and comprehension.

1.5 REQUIREMENTS SPECIFICATION

The goal of this project is to develop a semantic analyzer for intermediate in compiler design, using Python, HTML, CSS, and JavaScript with Django backend. The requirements specification will define the scope, goals, and features of the software being developed.

Functional Requirements:

- The software must be able to parse input code and generate intermediate code as output.
- The software must be able to handle different programming languages, including C and Java.
- The software must support a range of common operators and expressions, including arithmetic, logical, and bitwise operators.
- The software must support control structures such as loops and conditional statements. The software must support functions and procedures. The software must be able to generate intermediate code that is compatible with the target architecture.

Non-Functional Requirements:

The software must be efficient and scalable, able to handle large code files and generate intermediate code in a timely manner. The software must be reliable and accurate, with a low error rate in code parsing and intermediate code generation. The software must be user-friendly and intuitive, with a clean and modern design that is easy to navigate. The software must be secure, protecting user data and preventing unauthorized access.

The software must be well-documented, with clear instructions for installation, usage, and customization.

Assumptions:

- The software will be built using Python, HTML, CSS, and JavaScript with Django backend.
- The software will be tested using a comprehensive set of unit tests and integration tests, covering a wide range of input code and edge cases.
- The software will be deployed on a Linux server running Apache web server.

Dependencies:

- The software depends on the availability and functionality of the Django backend and other third-party libraries used in the implementation.
- The software depends on the proper configuration and maintenance of the server infrastructure.

Risks:

- The complexity of the project may lead to delays or unexpected challenges in implementation.
- The software may not perform well on certain input code files or in certain target architectures.
- The software may not be compatible with certain browsers or operating systems.

In summary, the requirements specification document outlines the functional and non-functional requirements, as well as any assumptions, dependencies, and risks associated with the project. It serves as a blueprint for the development team, guiding them in the implementation and testing of the software product.

CHAPTER 2

NEEDS

2.1 Need for Semantic Analysis in Compiler Design

In the field of compiler design, the need for a semantic analyzer arises from the requirement to analyze and interpret the meaning of source code written in a programming language. While lexical and syntactic analysis handle the structure and grammar of the code, the semantic analyzer plays a vital role in ensuring the code's correctness and adherence to the language's semantics. The following points explain the need for a semantic analyzer in compiler design:

1. **Type Checking:** A semantic analyzer is responsible for performing type checking, which ensures that operations and expressions in the source code are used in a manner consistent with the types defined in the programming language. It detects type mismatches, such as assigning incompatible types or performing operations on incompatible data, and reports errors or warnings accordingly.

2. **Symbol Table Management**: The semantic analyzer maintains a symbol table that stores information about variables, functions, and other identifiers encountered in the code. It resolves scope-related issues, such as detecting undeclared variables or conflicting declarations, and facilitates name resolution and access to symbols throughout the code.

3. **Detecting Semantic Errors**: While lexical and syntactic analysis focus on detecting lexical and grammatical errors, the semantic analyzer goes a step further by identifying semantic errors that cannot be captured by earlier stages. These errors may include improper use of language constructs, incorrect function invocations, or violations of language-specific rules. Detecting these errors helps in providing meaningful error messages to the programmer, aiding in debugging and code improvement.

4. **Implicit Conversions**: Some programming languages perform implicit type conversions under certain circumstances. A semantic analyzer identifies such conversions and ensures that they are valid and conform to the language's rules. It prevents potential issues arising from unintended or incorrect implicit conversions, ensuring the program's behavior aligns with the programmer's intentions.

5. **Optimization Opportunities**: The semantic analyzer can provide information about code constructs and their relationships, enabling subsequent optimization stages in the compiler to make informed decisions. This information can include constant folding, dead code elimination, or loop optimizations, leading to improved performance and efficiency of the generated code.

6. **Language-specific Features**: Different programming languages have their own sets of language-specific features and rules. A semantic analyzer understands these language-specific features and ensures their correct usage. It enforces language-specific constraints, checks for compliance with language rules, and provides support for advanced language features such as inheritance, polymorphism, or generics.

7. **Compiler Code Generation**: The semantic analyzer prepares the code for the subsequent stages of the compiler, such as code generation or intermediate representation generation. It provides essential information about the code structure, type information, and symbol resolution, enabling efficient code generation and translation into the target language or machine code.

In summary, a semantic analyzer in compiler design plays a crucial role in ensuring the correctness, consistency, and adherence to language rules and semantics in the source code. It performs type checking, manages symbol tables, detects semantic errors, handles implicit conversions, identifies optimization opportunities, supports language-specific features, and prepares the code for subsequent stages of the compiler. By performing these tasks, the semantic analyzer helps in generating reliable and efficient compiled code from the source code.

2.2 Limitations of CFGs.

Context-Free Grammar (CFG) is a mathematical notation used to describe the syntax of a programming language or any other formal language. However, like any other notation, CFG also has its limitations. Below are some of the limitations of CFG:

- **Cannot capture semantic information**: CFG only describes the syntax of a language, which means it cannot capture the semantic information or the meaning of the language. For instance, a CFG cannot describe the meaning of "if (x=5) then y=10 else y=20" statement in a programming language.

- **Limited expressiveness**: CFG has a limited expressive power, which means it cannot handle complex languages or language constructs. For instance, it cannot handle context-sensitive languages, where the production rules depend on the context or history of the symbols.

- **Ambiguity**: CFG can produce ambiguous grammars, which means that a sentence in the language can have more than one parse tree. Ambiguity is a problem because it makes parsing difficult and can lead to unexpected behavior in a compiler or interpreter.

- **Not suitable for error handling**: CFG is not suitable for handling errors in the input language. For example, if a user enters an incorrect syntax, a CFG cannot detect the error and provide useful error messages.

- **Limited support for left-recursive productions**: CFG does not support left-recursive productions, which means that it cannot handle grammars that involve left recursion. Left recursion occurs when a non-terminal symbol appears as the first symbol in one of its own production rules.

- **Cannot handle languages with unbounded nesting**: CFG is not suitable for handling languages that involve unbounded nesting, such as nested parentheses, brackets, and braces. This is because CFG has a limited stack capacity, which can lead to stack overflow errors.

- **Cannot handle languages with dynamic scoping**: CFG cannot handle languages that use dynamic scoping, where the scope of a variable changes dynamically during the execution of a program.

In conclusion, while CFG is a powerful notation for describing the syntax of a language, it has its limitations. These limitations can make it difficult or impossible to use CFG for certain languages or language constructs.

A can get its values from S, B and C.

B can get its values from S, A and C

C can get its values from A, B and S

2.3 Types of Attributes in Semantic Analyzer:

In a semantic analyzer, attributes are properties associated with language constructs, such as variables, expressions, functions, and statements. These attributes capture relevant information about the constructs

and play a vital role in the semantic analysis process. Here are some common types of attributes found in a semantic analyzer:

1. **Type Attributes**: Type attributes represent the data type or the expected type of a construct. For example, a variable attribute may include information about its data type, such as integer, string, or boolean. Type attributes help in performing type checking and ensuring type consistency throughout the code.

2. **Scope Attributes:** Scope attributes define the scope or visibility of a construct, such as a variable or a function. They include information about the scope level, the enclosing scope, and the accessibility of the construct within the program. Scope attributes assist in resolving symbol names, handling variable scoping rules, and maintaining symbol tables.

3. **Value Attributes**: Value attributes represent the value or result of an expression or computation. They store information about the computed value, such as the numeric value of a mathematical expression or the string value of a string literal. Value attributes are useful in constant folding, optimization, and evaluation of expressions.

4. **Address Attributes**: Address attributes are used in low-level languages or in situations where memory addresses or locations are relevant. They store information about the memory address or location of a construct, enabling efficient memory management and manipulation during code generation or optimization stages.

5. **Declaration Attributes**: Declaration attributes contain information about the declaration of a construct, such as the line number, the file name, and any additional metadata associated with the declaration. These attributes aid in error reporting, debugging, and tracking the origin of constructs within the source code.

6. **Function Signature Attributes**: Function signature attributes describe the parameters, return type, and other details of a function. They provide information necessary for function calls, argument matching, and ensuring consistency between function declarations and invocations.

7. **Inheritance Attributes**: Inheritance attributes are used in object-oriented languages to capture inheritance relationships between classes or objects. They store information about parent classes, interfaces, or superclasses, enabling correct inheritance resolution and polymorphic behavior.

8. **Annotation Attributes**: Annotation attributes are specific to languages that support annotations or metadata associated with constructs. They hold additional information provided through annotations, such as compiler directives, documentation comments, or custom metadata used for code generation or external tool integration.

These are some of the common types of attributes found in a semantic analyzer. The specific attributes and their types may vary depending on the programming language, the design of the compiler, and the semantic analysis requirements of the language being processed.

Abstract Syntax Trees(ASTs)
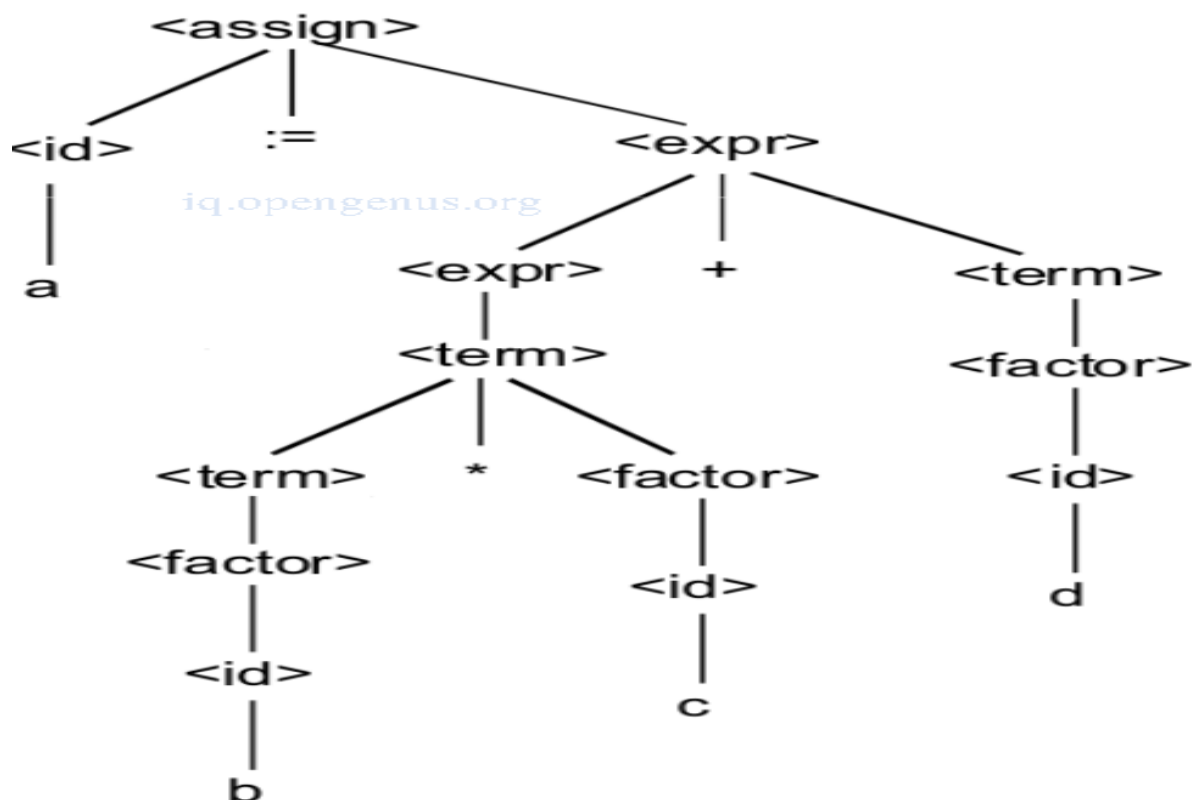
These are a reduced form of a parse tree.

They don't check for string membership in the language of the grammar.

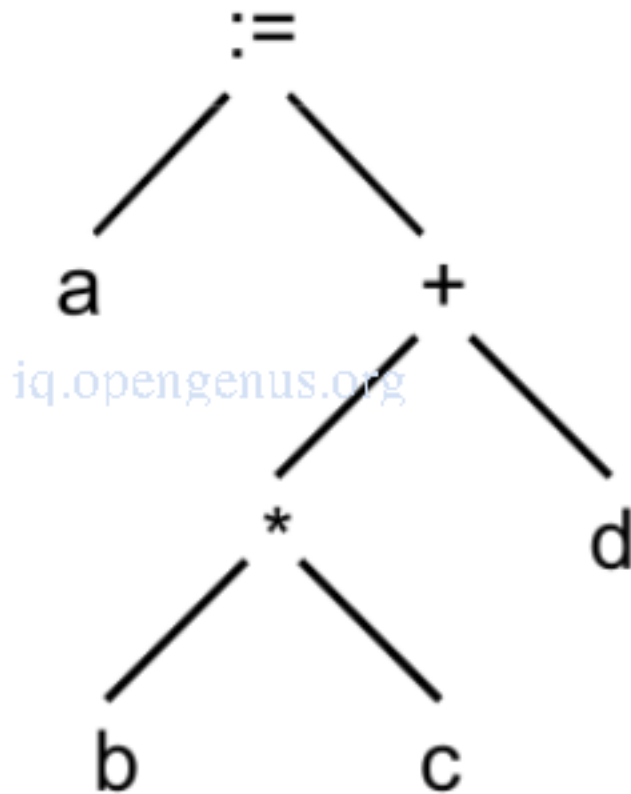They represent relationships between language constructs and avoid derivations.

An example

The parse tree and abstract syntax tree for the expression a := b * c + d is.

The parse tree

The abstract syntax tree

Properties of abstract syntax trees.

- Good for optimizations.

- Easier evaluation.

- Easier traversals.

- Pretty printing(unparsing) is possible by in-order traversal.

- Postorder traversal of the tree is possible given a postfix notation.

Implementing Semantic Actions during Recursive Descent parsing.

During this parsing there exist a separate function for each non-terminal in the grammar. The procedures will check the lookahead token against the terminals it expects to find. Recursive descent recursively calls procedures to parse non-terminals it expects to find. At certain points during parsing appropriate semantic actions that are to be performed are implemented.

Roles of this phase.

- Collection of type information and type compatibility checking.

- Type checking.

- Storage of type information collected to a symbol table or an abstract syntax tree.

- In case of a mismatch, type correction is implemented or a semantic error is generated.

- Checking if source language permits operands or not.

CHAPTER 3

SYSTEM ARCHITECTURE AND DESIGN

3.1 FRONT-END DESIGN:-

HTML, CSS, Python, and Django are powerful technologies that can be used to build a wide range of web applications, from simple personal websites to complex web-based software systems. Each of these technologies brings its unique features and capabilities to the table, allowing developers to build scalable, efficient, and user-friendly applications.

HTML (Hypertext Markup Language) is the standard markup language used for creating web pages and applications. It provides a set of tags and attributes that allow developers to structure and format the content of their web pages. HTML is the backbone of every web page and is essential for building a functional and visually appealing website. With the latest HTML5 standard, developers can use new features like video and audio elements, canvas, and semantic tags that help with SEO.

CSS (Cascading Style Sheets) is a stylesheet language that is used to style and format the layout of web pages. CSS allows developers to add colors, fonts, and visual effects to their web pages, making them more visually appealing and user-friendly. CSS provides a powerful set of tools for creating responsive designs, layouts that adapt to different screen sizes and orientations.

Python is a high-level programming language that is used for a wide range of applications, including web development. Python is known for its readability, ease of use, and flexibility, making it a popular choice among developers. With its vast library of modules and tools, Python is a versatile language that can be used to build complex web application.

3.2 BACK-END DESIGN:

Django is a Python web framework that provides a set of tools and libraries for building web applications quickly and efficiently. Django is known for its security, scalability, and ease of use, making it a popular choice for building robust and scalable web applications. With Django, developers can easily build web applications that incorporate complex features like user authentication, content management, and e-commerce.

When used together, HTML, CSS, Python, and Django form a powerful combination that can be used to build a wide range of web applications. HTML and CSS provide the foundation for building visually appealing and user-friendly web pages, while Python and Django provide the tools and libraries for building scalable and efficient web applications.

With HTML and CSS, developers can create beautiful and responsive web pages that work across all devices and screen sizes. HTML provides the structure and content of the web page, while CSS provides the layout and styling. By using CSS preprocessors like SASS or LESS, developers can write more efficient and modular code.

Python and Django provide the tools and libraries for building scalable and efficient web applications. Django provides a set of built-in features like user authentication, content management, and URL routing, making it easy for developers to build complex web applications. Python provides a vast library of modules and tools for web development, including frameworks like Flask and Pyramid.

In conclusion, HTML, CSS, Python, and Django are powerful technologies that can be used to build a wide range of web applications. By leveraging the unique features and capabilities of each technology, developers can create robust and scalable web applications that meet the needs of modern web development. With their ease of use, scalability, and flexibility, HTML, CSS, Python, and Django are essential tools for any web developer.

SYSTEM ARCHITECTURE DESIGN: -
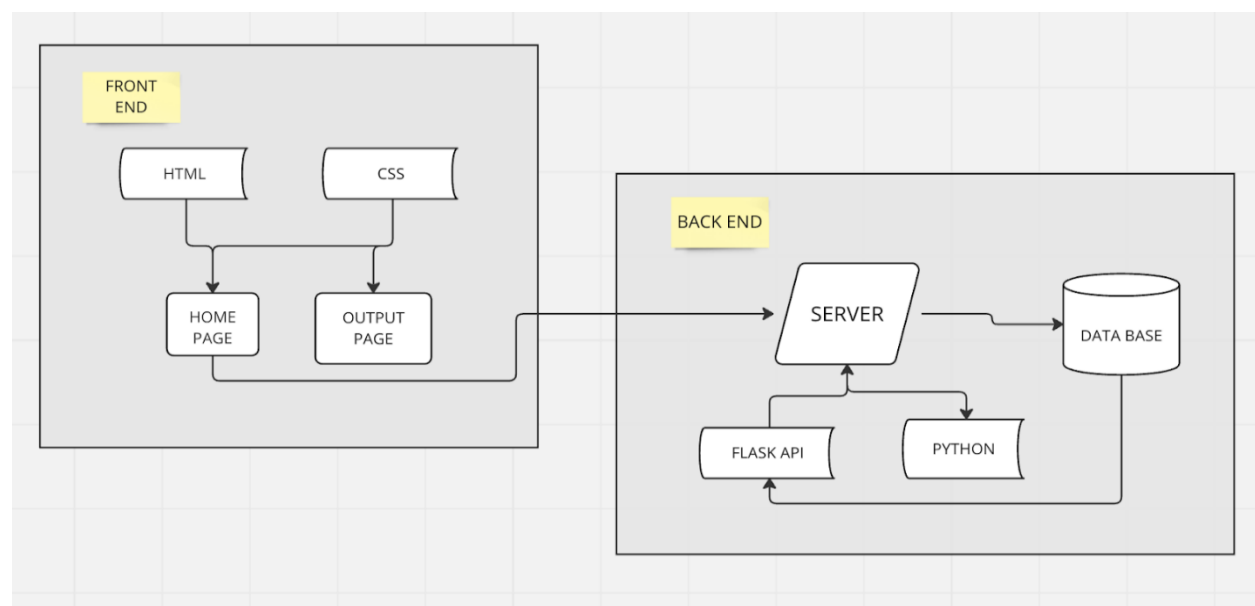
SEMANTIC ANALYZER ARCHITECTURE DESIGN: -
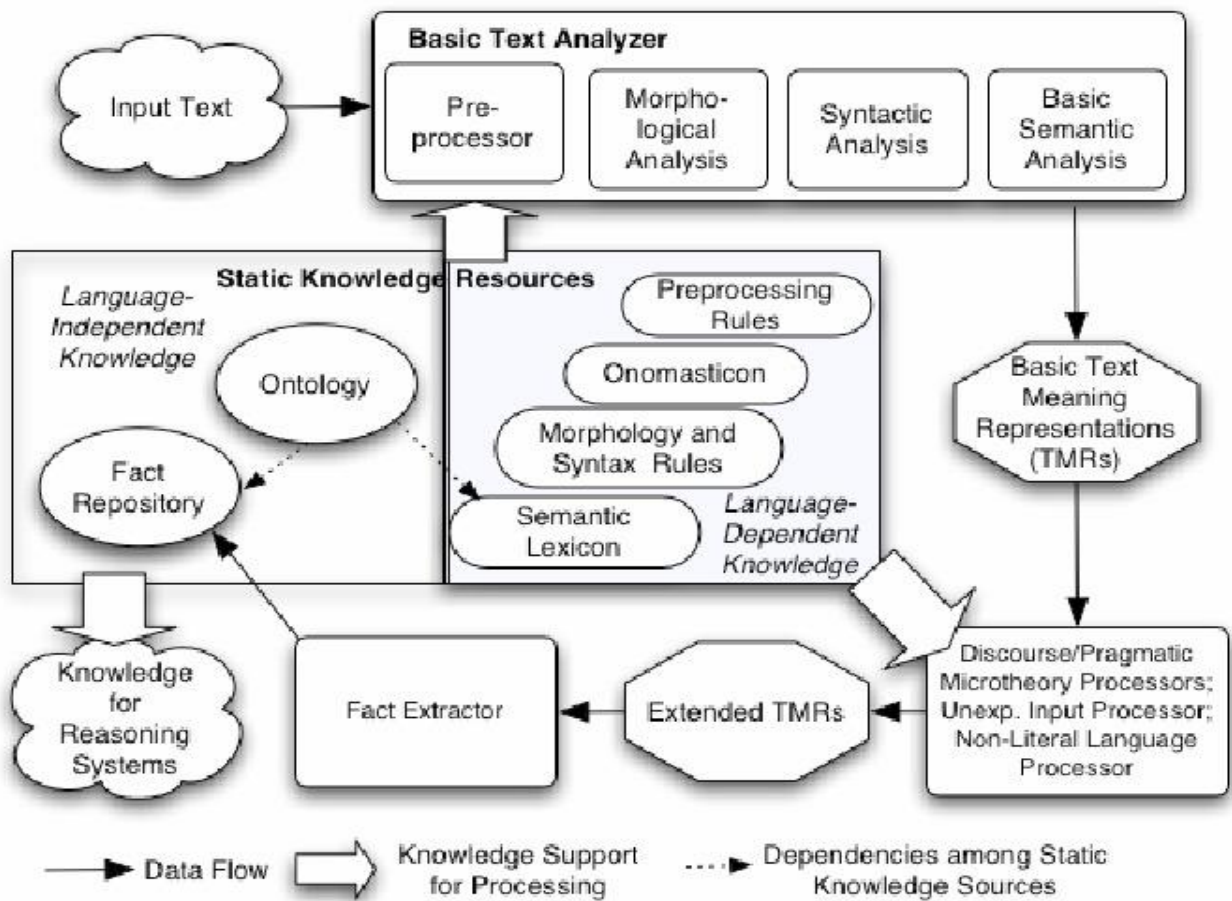
CHAPTER 4

REQUIREMENTS

4.1 The requirement to run the script -

Run a semantic analyzer code built using Flask and JavaScript, you will need the following requirements:

Python:

Python is a popular programming language that is used in many web development frameworks, including Flask. Flask is a lightweight web framework that is used to build web applications using Python. To run a Python Flask HTML CSS JS file, you will need to have Python installed on your local machine. You can download and install Python from the official Python website.

Flask:

Flask is a web framework for Python that provides a set of tools and features for building web applications. It is designed to be lightweight and flexible and provides features such as routing, templates, and middleware. To use Flask in your project, you will need to install it using pip, the package manager for Python. You can install Flask and its dependencies by running the command pip install flask in your project directory.

HTML:

HTML is the standard markup language used to create web pages. To create HTML files, you will need a text editor such as Sublime Text, Atom, or Visual Studio Code. These editors provide syntax highlighting and other features that make it easier to write and edit HTML code.

CSS:

CSS is a styling language used to add styles and visual effects to HTML pages. To create CSS files, you will also need a text editor such as Sublime Text, Atom, or Visual Studio Code. These editors provide syntax highlighting and other features that make it easier to write and edit CSS code.

JavaScript:

JavaScript is a programming language used to add interactivity and dynamic effects to web pages. To create JavaScript files, you will need a text editor such as Sublime Text, Atom, or Visual Studio Code. These editors provide syntax highlighting and other features that make it easier to write and edit JavaScript code.

Web Browser:

To run your Python Flask HTML CSS JS file, you will need a web browser such as Google Chrome, Mozilla Firefox, or Microsoft Edge. The web browser will render your HTML and CSS files and execute your JavaScript code. You can test your application by opening your HTML file in a web browser.

Once you have installed the required dependencies and created your HTML, CSS, and JavaScript files, you can run your Python Flask HTML CSS JS file by running the command python app.py in your project directory. This will start the Flask development server and allow you to access your application in a web browser at http://localhost:5000/.

.

CHAPTER 5

CODING AND TESTING

5.1 CODING: -

1. <u>APP.PY: -</u>

```python
from flask import Flask, render_template
from flask import request, jsonify
import ast
import logging

app = Flask(_name_)
logging.basicConfig(level=logging.DEBUG)


@app.route('/')
def hello_world():
    return render_template('prototype.html')

@app.route('/description')
def description():
    return render_template('description.html')

@app.route('/contact')
def contact():
    return render_template('contact.html')

def semantic_analysis(program):
    errors = []

    # Parse the Python program
    try:
        parsed_program = ast.parse(program)
    except SyntaxError as e:
        errors.append(f"Syntax error: {e}")
        return errors
```

2. **MAIN.PY:-**

```python
import os
from scanner import SymbolTableManager
from code_gen import MemoryManager

script_dir = os.path.dirname(os.path.dirname(os.path.abspath(_file_)))

class SemanticAnalyser(object):
    def _init_(self):

        # routines
        self.semantic_checks = {
            "#SA_INC_SCOPE" : self.inc_scope_routine,
            "#SA_DEC_SCOPE" : self.dec_scope_routine,

            "#SA_SAVE_MAIN" : self.save_main_routine,
            "#SA_MAIN_POP" : self.pop_main_routine,
            "#SA_MAIN_CHECK" : self.check_main_routine,

            "#SA_SAVE_TYPE" : self.save_type_routine,
            "#SA_ASSIGN_TYPE" : self.assign_type_routine,
            "#SA_ASSIGN_FUN_ROLE" : self.assign_fun_role_routine,
            "#SA_ASSIGN_VAR_ROLE" : self.assign_var_role_routine,
            "#SA_ASSIGN_PARAM_ROLE" : self.assign_param_role_routine,
            "#SA_ASSIGN_LENGTH" : self.assign_length_routine,
            "#SA_SAVE_PARAM" : self.save_param_routine,
            "#SA_ASSIGN_FUN_ATTRS" : self.assign_fun_attrs_routine,

            "#SA_CHECK_DECL" : self.check_declaration_routine,

            "#SA_SAVE_FUN" : self.save_fun_routine,
            "#SA_CHECK_ARGS" : self.check_args_routine,

            "#SA_PUSH_ARG_STACK" : self.push_arg_stack_routine,
            "#SA_SAVE_ARG" : self.save_arg_routine,
            "#SA_POP_ARG_STACK" : self.pop_arg_stack_routine,


            "#SA_PUSH_WHILE" : self.push_while_routine,
            "#SA_CHECK_WHILE" : self.check_while_routine,
            "#SA_POP_WHILE" : self.pop_while_routine,

            "#SA_PUSH_SWITCH" : self.push_switch_routine,
            "#SA_CHECK_BREAK" : self.check_break_routine,
            "#SA_POP_SWITCH" : self.pop_switch_routine,
```

```python
        # associated stacks
        self.semantic_stacks = {
            "main_check" : [],
            "type_assign" : [],
            "type_check" : [],
            "fun_check" : [],
        }

        # flags
        self.main_found = False
        self.main_not_last = False

        # counters
        self.arity_counter = 0
        self.while_counter = 0
        self.switch_counter = 0

        # lists
        self.fun_param_list = []
        self.fun_arg_list = []
        self._semantic_errors = []

        self.semantic_error_file = os.path.join(script_dir, "errors", "semantic_errors.txt")


    @property
    def scope(self):
        return len(SymbolTableManager.scope_stack) - 1


    @property
    def semantic_errors(self):
        semantic_errors = []
        if self._semantic_errors:
            for lineno, error in self._semantic_errors:
                semantic_errors.append(f"#{lineno} : Semantic Error! {error}\n")
        else:
            semantic_errors.append("The input program is semantically correct.\n")
        return "".join(semantic_errors)


    def _get_lexim(self, token):
        if token[0] == "ID":
            return SymbolTableManager.symbol_table[token[1]]['lexim']
        else:
            return token[1]
```

```python
'''semantic routines start here '''

    def inc_scope_routine(self, input_token, line_number):
        SymbolTableManager.scope_stack.append(len(SymbolTableManager.symbol_table))


    def dec_scope_routine(self, input_token, line_number):
        scope_start_idx = SymbolTableManager.scope_stack.pop()
        SymbolTableManager.symbol_table              =              SymbolTableManager.symbol_ta-
ble[:scope_start_idx]


    def save_main_routine(self, input_token, line_number):
        self.semantic_stacks["main_check"].append(self._get_lexim(input_token))


    def pop_main_routine(self, input_token, line_number):
        self.semantic_stacks["main_check"] = self.semantic_stacks["main_check"][:-2]


    def save_type_routine(self, input_token, line_number):
        SymbolTableManager.declaration_flag = True
        self.semantic_stacks["type_assign"].append(input_token[1])


    def assign_type_routine(self, input_token, line_number):
        if input_token[0] == "ID" and self.semantic_stacks["type_assign"]:
            symbol_idx = input_token[1]
            SymbolTableManager.symbol_table[symbol_idx]["type"]            =            self.seman-
tic_stacks["type_assign"].pop()
            self.semantic_stacks["type_assign"].append(symbol_idx)
            SymbolTableManager.declaration_flag = False


    def assign_fun_role_routine(self, input_token, line_number):
        if self.semantic_stacks["type_assign"]:
            symbol_idx = self.semantic_stacks["type_assign"][-1]
            SymbolTableManager.symbol_table[symbol_idx]["role"] = "function"
            SymbolTableManager.symbol_table[symbol_idx]["address"] = MemoryManager.pb_in-
dex


    def assign_param_role_routine(self, input_token, line_number):
        self.assign_var_role_routine(input_token, line_number, "param")


    def assign_var_role_routine(self, input_token, line_number, role="local_var"):
        if self.semantic_stacks["type_assign"]:
            symbol_idx = self.semantic_stacks["type_assign"][-1]
            symbol_row = SymbolTableManager.symbol_table[symbol_idx]
```

```python
    def assign_length_routine(self, input_token, line_number):
        if self.semantic_stacks["type_assign"]:
            symbol_idx = self.semantic_stacks["type_assign"].pop()
            symbol_row = SymbolTableManager.symbol_table[symbol_idx]
            if input_token[0] == "NUM":
                symbol_row["arity"] = int(input_token[1])
                if symbol_row["role"] == "param":
                    symbol_row["offset"] = MemoryManager.get_param_offset()
                else:
                    symbol_row["address"] = MemoryManager.get_static(int(input_token[1]))
            else:
                SymbolTableManager.symbol_table[symbol_idx]["arity"] = 1
                if symbol_row["role"] == "param":
                    symbol_row["offset"] = MemoryManager.get_param_offset()
                else:
                    symbol_row["address"] = MemoryManager.get_static()

            if input_token[1] == "[" and self.fun_param_list:
                self.fun_param_list[-1] = "array"


    def save_param_routine(self, input_token, line_number):
        self.fun_param_list.append(input_token[1])


    def push_arg_stack_routine(self, input_token, line_number):
        SymbolTableManager.arg_list_stack.append([])


    def pop_arg_stack_routine(self, input_token, line_number):
        if len(SymbolTableManager.arg_list_stack) > 1:
            SymbolTableManager.arg_list_stack.pop()


    def save_arg_routine(self, input_token, line_number):
        if input_token[0] == "ID":
            SymbolTableManager.arg_list_stack[-1].append(SymbolTableManager.symbol_table[input_token[1]].get("type"))
        else:
            SymbolTableManager.arg_list_stack[-1].append("int")

def assign_fun_attrs_routine(self, input_token, line_number):
        if self.semantic_stacks["type_assign"]:
            symbol_idx = self.semantic_stacks["type_assign"].pop()
            params = self.fun_param_list
            SymbolTableManager.symbol_table[symbol_idx]["arity"] = len(params)
            SymbolTableManager.symbol_table[symbol_idx]["params"] = params
            self.fun_param_list = []
            SymbolTableManager.temp_stack.append(0) # init temp counter for this function
```

```python
    def check_args_routine(self, input_token, line_number):
        if self.semantic_stacks["fun_check"]:
            fun_id = self.semantic_stacks["fun_check"].pop()
            lexim = SymbolTableManager.symbol_table[fun_id]["lexim"]
            args = SymbolTableManager.arg_list_stack[-1]
            if args is not None:
                self.semantic_stacks["type_check"] = self.semantic_stacks["type_check"][:len(args)]
                if SymbolTableManager.symbol_table[fun_id]["arity"] != len(args):
                    SymbolTableManager.error_flag = True
                    self._semantic_errors.append((line_number, f"Mismatch in numbers of arguments
of '{lexim}'."))
                else:
                    params = SymbolTableManager.symbol_table[fun_id]["params"]
                    i = 1
                    for param, arg in zip(params, args):
                        if param != arg and arg is not None:
                            SymbolTableManager.error_flag = True
                            self._semantic_errors.append((line_number, f"Mismatch in type of argument
{i} of '{lexim}'. Expected '{param}' but got '{arg}' instead."))
                        i += 1


    def push_while_routine(self, input_token, line_number):
        self.while_counter += 1


    def check_while_routine(self, input_token, line_number):
        if self.while_counter <= 0:
            SymbolTableManager.error_flag = True
            self._semantic_errors.append((line_number, f"No 'while' found for 'continue'"))


    def pop_while_routine(self, input_token, line_number):
        self.while_counter -= 1


    def push_switch_routine(self, input_token, line_number):
        self.switch_counter += 1


    def check_break_routine(self, input_token, line_number):
        if self.while_counter <= 0 and self.switch_counter <= 0:
            SymbolTableManager.error_flag = True
            self._semantic_errors.append((line_number, "No 'while' or 'switch' found for 'break'."))


    def pop_switch_routine(self, input_token, line_number):
        self.switch_counter -= 1
```

```python
    def type_check_routine(self, input_token, line_number):
        try:
            operand_b_type = self.semantic_stacks["type_check"].pop()
            operand_a_type = self.semantic_stacks["type_check"].pop()
            if operand_b_type is not None and operand_a_type is not None:
                if operand_a_type == "array":
                    SymbolTableManager.error_flag = True
                    self._semantic_errors.append((line_number,
                        f"Type mismatch in operands, Got '{operand_a_type}' instead of 'int'."))
                elif operand_a_type != operand_b_type:
                    SymbolTableManager.error_flag = True
                    self._semantic_errors.append((line_number,
                        f"Type mismatch in operands, Got '{operand_b_type}' instead of '{oper-
and_a_type}'."))
                else:
                    self.semantic_stacks["type_check"].append(operand_a_type)
        except IndexError:
            pass


    ''' semantic routines end here '''


    def semantic_check(self, action_symbol, input_token, line_number):
        try:
            self.semantic_checks[action_symbol](input_token, line_number)
        except Exception as e:
            print(f"{line_number} : Error in semantic routine {action_symbol}:", str(e))


    def eof_check(self, line_number):
        if not self.main_found or self.main_not_last:
            SymbolTableManager.error_flag = True
            self._semantic_errors.append((line_number, "main function not found!"))
```

3. <u>MAIN.INDEX: -</u>

```html
<!DOCTYPE html>
<html>
<head>
  <title>Semantic Analysis</title>
</head>
<body>
  <h1>Semantic Analysis</h1>
  <form action="/generate" method="post">
    <textarea name="cpp_code" rows="10" cols="50" placeholder="Enter C++ code"></textarea><br>
    <input type="submit" value="Genetrate Semantic Analysis ">
  </form>
  <hr>
  <script src="https://code.jquery.com/jquery-3.6.0.min.js"></script>
  <script src="script.js"></script>
</body>
</html>
```

4. MAIN.CSS: -

```css
body {
    font-family: Arial, sans-serif;
    margin: 20px;
}

h1 {
    text-align: center;
}

.container {
    margin-top: 20px;
}

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

textarea {
    width: 100%;
}

button {
    margin-top: 10px;
}

pre {
    background-color: #f4f4f4;
    padding: 10px;
    overflow-x: auto;
}
```

5.2 TESTING

Testing the analyzer -

```
Test 1:
/*
 * Example source code
 *
 * prints N numbers from Fibonacci sequence
 */

void main(void) {
   int i;
   int N;
   int a;
   int b;
   int tmp;

   i = 0;
   a = 0;
   b = 1;
   tmp = 0;

   N = 20; // how many numbers to print?

   output(0); // always print the first number

   while (i < N) {
      output(a + b);
      tmp = b;
      b = a + tmp;
      a = tmp;
      i = i + 1;
   }
}
```
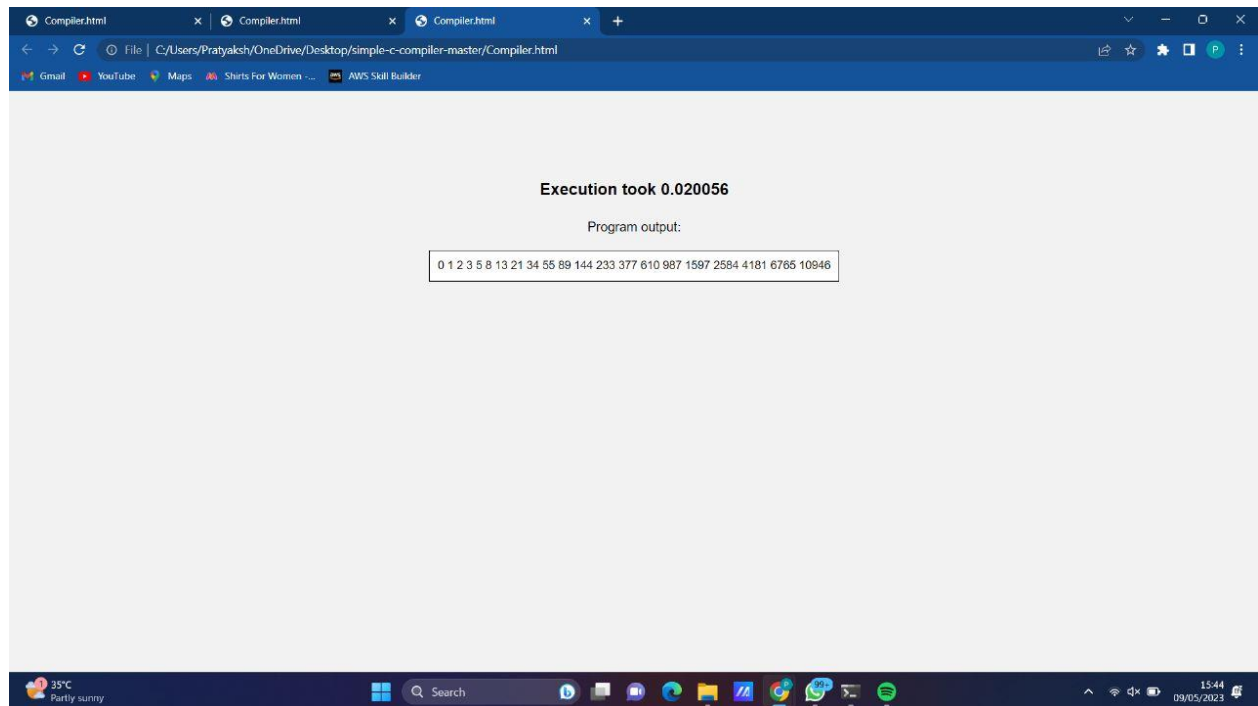
Execution took 0.020056

Program output:

0 1 2 3 5 8 13 21 34 55 89 144 233 377 610 987 1597 2584 4181 6765 10946

Test 2:

```
Voi main(void) {
    int i;
    i = 0;
    while (i < 10) {
        output(i);
        i = i + 1;
    }
}}
```

Execution took 0.000000

Program output:

There is no lexical errors.

#1 : Syntax Error! Illegal "ID" #1 : Syntax Error! Illegal "ID" #1 : Syntax Error! Illegal "(" #1 : Syntax Error! Missing "ID" #1 : Syntax Error! Illegal ")" #1 : Syntax Error! Missing ";"

#1 : Semantic Error! main function not found!
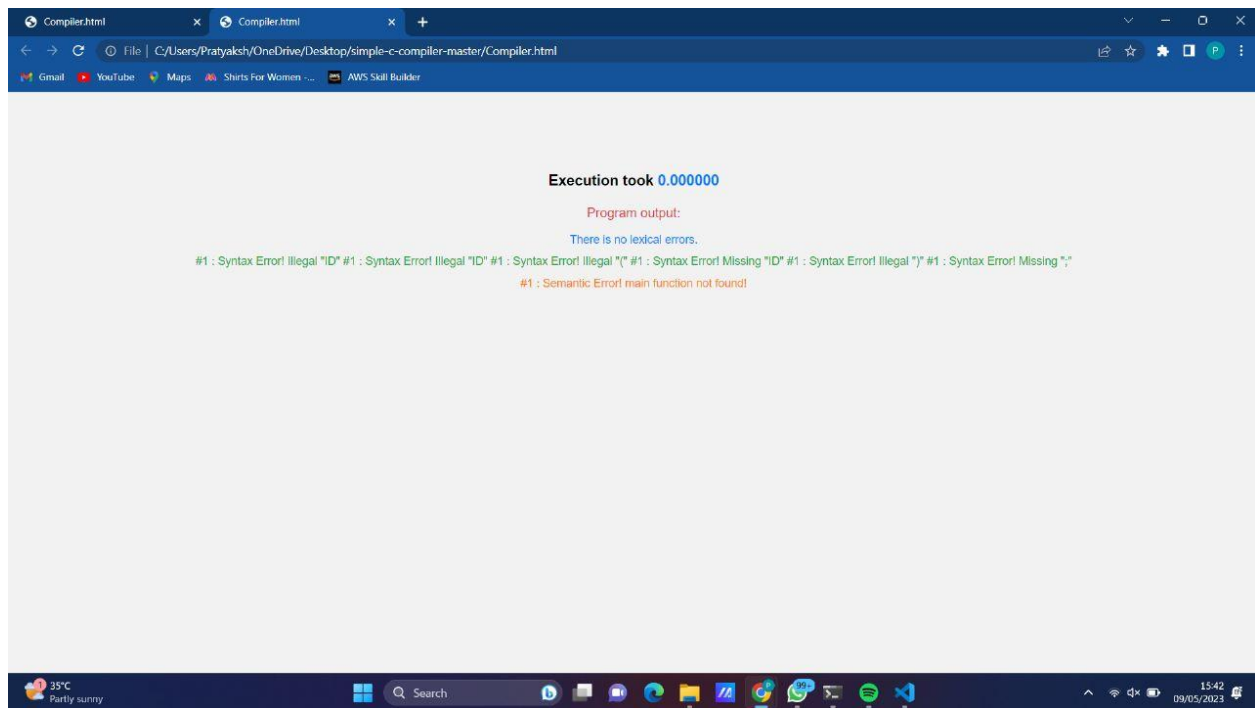
Test 3:
```
/*
 * Hard example source code
 * prints N length sequence of sums of i first odd numbers
 * where i is the index of the sum in the sequence
 */
int abs(int a) {
   if (a < 0) {
      return 0-a;
   } else {
      return a;
   }
}

int isMultiplier(int a, int b) {
   int i;
   int step;
   int flag;

   if (b == 0) {
      return 0;
   } else {
      i = 1;
      flag = 0;
   }

   if (a < 0) {
      if (b < 0) {
         i = 1;
      } else {
         i = 0-1;
      }
   } else {
      if (b < 0) {
         i = 0-1;
      } else {
         i = 1;
      }
   `
```
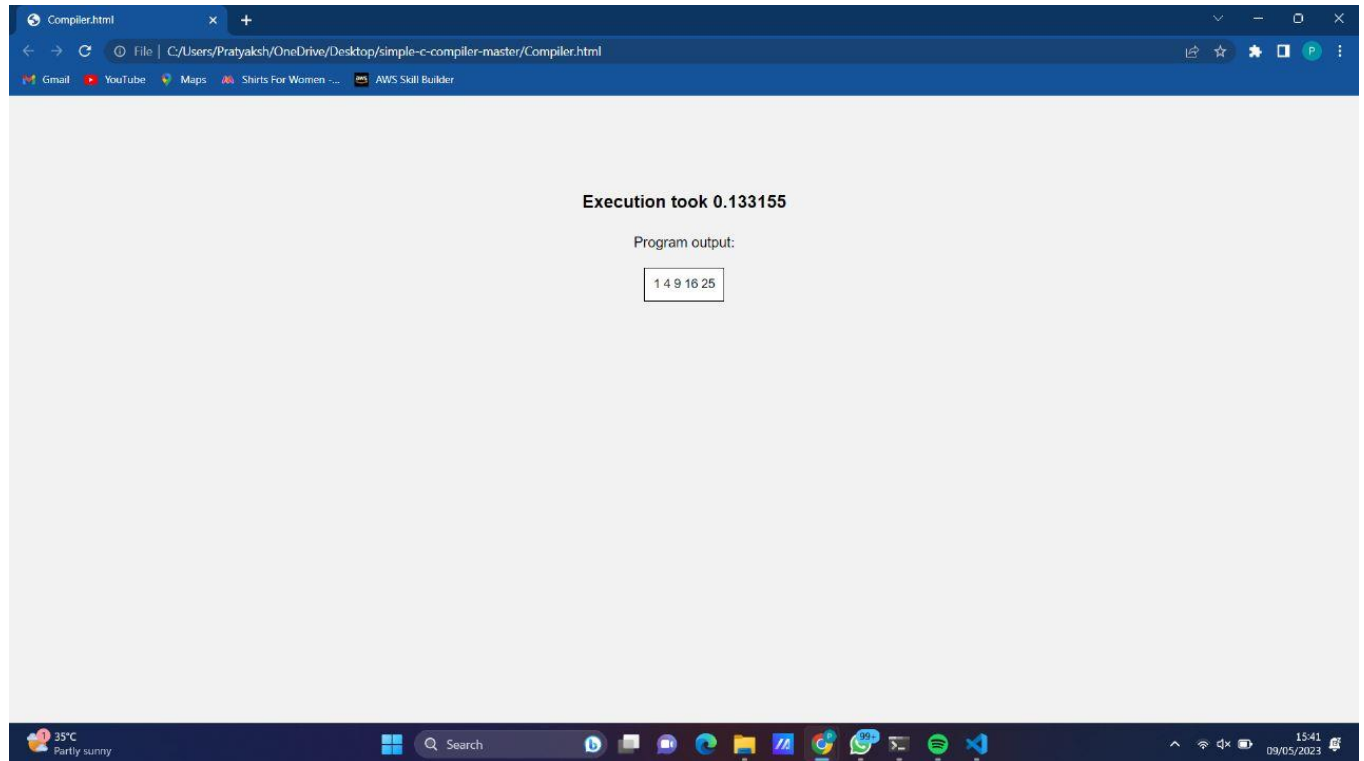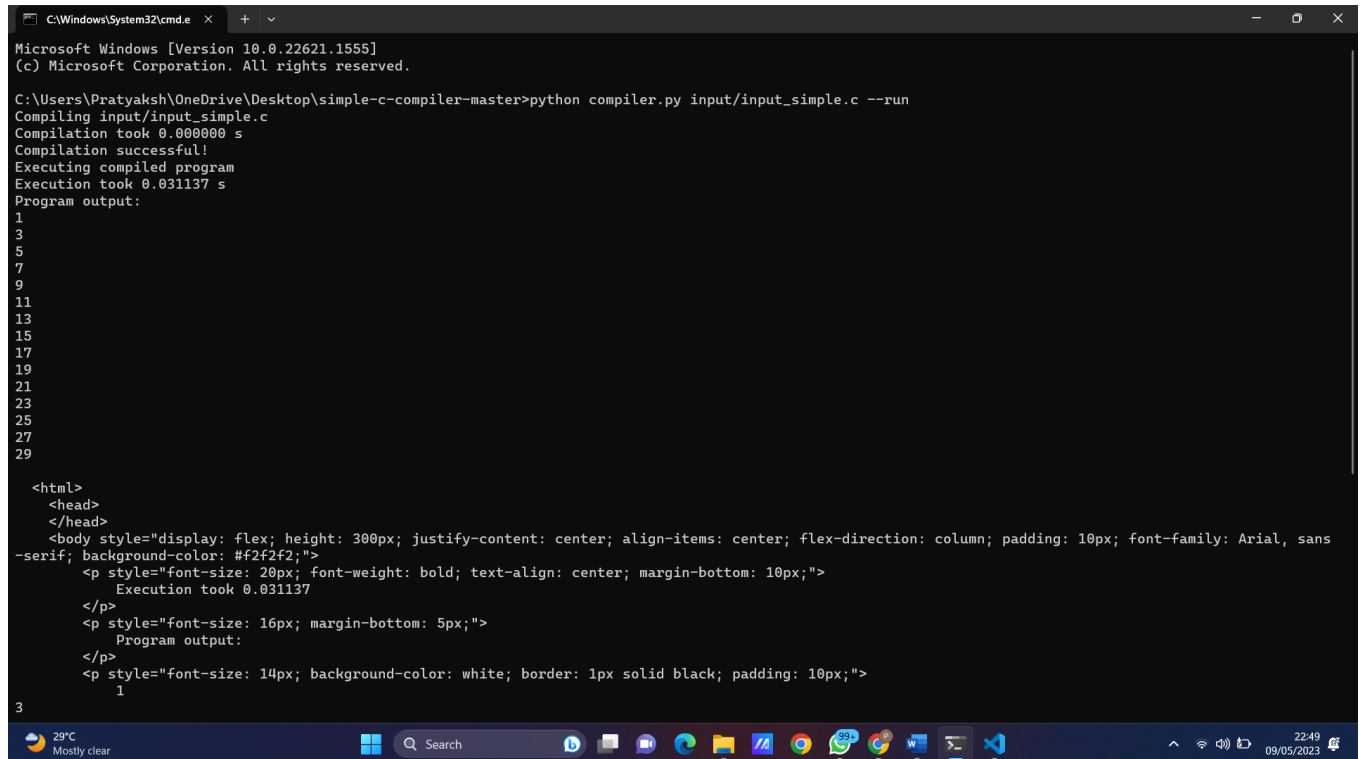
CHAPTER 6

OUTPUT AND RESULTS

6.1 OUTPUT: -

Terminal -

Description about the Analyzer -

A semantic analyzer is a component of a compiler or interpreter that checks the meaning of the pro-gram's source code by analyzing its syntax and context. Its main task is to ensure that the program ad-heres to the rules of a programming language's syntax and semantics. During the compilation or interpretation process, the semantic analyzer performs a series of checks on the source code, including:

1. **Type Checking:** The semantic analyzer checks whether the types of variables and expres-sions used in the program are consistent and compatible with the rules of the programming language. It ensures that variables are declared before use and that the operations applied to them are legal.

2. **Scope Checking:** The semantic analyzer checks whether a variable or function is declared with-in its correct scope or not. It ensures that variables and functions are accessible only within their declared scope.

3. **Declaration Checking**: The semantic analyzer checks whether the variables and functions used in the program are declared before they are used. It ensures that the program follows the correct order of declaration.

4. **Code Optimization**: The semantic analyzer may also perform some code optimization techniques to improve the efficiency of the program, such as constant folding, dead code elimination, and loop unrolling.

If the semantic analyzer detects any errors in the source code, it generates an error message indicating the type of error and its location in the source code. These error messages help programmers to identify and correct errors in their code.

6.2 RESULT

The result of a semantic analyzer in a compiler is a set of analyzed and validated semantic information associated with the source code. This information is derived from the analysis of the code's structure, context, and adherence to the language's semantics. The following are the typical results generated by a semantic analyzer:

1. **Symbol Table**: The symbol table is a data structure maintained by the semantic analyzer that stores information about variables, functions, classes, and other identifiers encountered in the code. It includes details such as names, data types, scopes, memory addresses, and other relevant attributes. The symbol table serves as a reference for subsequent compiler stages, enabling correct symbol resolution and semantic checks.

2. **Type Information**: The semantic analyzer determines and associates the appropriate data types with variables, expressions, function parameters, and return values. It performs type checking to ensure consistency and compatibility between operations and operands. The result includes the resolved data types and their validity within the program.

3. **Semantic Errors**: During the analysis process, the semantic analyzer detects and reports semantic errors or warnings that cannot be captured by earlier stages. These errors may include type mismatches, undeclared variables, incorrect function invocations, or violations of language-specific rules. The result includes a list of identified errors or warnings, along with their corresponding locations in the source code.

4. **Scope Resolution**: The semantic analyzer handles scoping rules and resolves the visibility and accessibility of variables, functions, and other identifiers within their respective scopes. It ensures that symbols are correctly declared, referenced, and accessible where needed. The result provides information about the resolved scopes and their associations with identifiers.

5. **Inheritance and Polymorphism**: In object-oriented languages, the semantic analyzer analyzes and resolves inheritance relationships between classes, interfaces, or objects. It ensures correct inheritance, method overriding, and polymorphic behavior. The result includes the resolved inheritance hierarchies and the associated methods and members.

6. **Annotation Processing**: If the language supports annotations or metadata, the semantic analyzer processes and validates the annotations present in the code. It extracts the relevant information provided through annotations, such as compiler directives, documentation comments, or custom metadata used for code generation or external tool integration. The result includes the processed annotations and their associated data.

7. **Optimization Opportunities**: The semantic analyzer may identify optimization opportunities based on the analyzed code constructs, such as constant folding, dead code elimination, or loop optimizations. It provides insights and information to subsequent optimization stages, enabling them to make informed decisions for code optimization and efficiency.

The result of the semantic analysis phase is crucial for the subsequent stages of the compiler, such as code generation, optimization, and error reporting. It ensures that the code is semantically correct, adheres to the language's rules, and provides valuable information for generating efficient and reliable compiled output.

CHAPTER 7

CONCLUSION: -

In conclusion, a semantic analyzer is a fundamental component in the field of compiler design, responsible for analyzing and interpreting the meaning and context of source code. It plays a critical role in ensuring the correctness, consistency, and adherence to language rules and semantics. By performing tasks such as type checking, symbol table management, detecting semantic errors, handling scoping rules, and resolving language-specific features, the semantic analyzer provides valuable insights and validations for the source code.

The results generated by a semantic analyzer, including the symbol table, type information, resolved scopes, detected errors or warnings, inheritance relationships, and annotation processing, form the foundation for subsequent stages of the compiler. These results enable correct symbol resolution, proper type handling, optimization opportunities, and reliable code generation.

Through its analysis, the semantic analyzer helps programmers in identifying and rectifying semantic errors, improving code quality, and reducing debugging efforts. It provides meaningful error messages and warnings, facilitating effective debugging and code improvement. Additionally, the semantic analyzer aids in ensuring type consistency, preventing type-related errors during program execution, and enforcing language-specific rules.

By leveraging the power of semantic analysis, compilers can generate optimized, efficient, and reliable compiled code. The semantic analyzer's role is crucial in facilitating the understanding, interpretation, and validation of the source code, leading to the production of high-quality software.

In summary, the semantic analyzer plays a vital role in the compilation process, contributing to the accuracy, reliability, and efficiency of the generated code. Its ability to extract meaning, detect semantic errors, and enforce language rules empowers programmers and enhances the overall development experience.

CHAPTER 8

REFERENCES:

1. **Flask Documentation:** Official documentation for Flask, a Python web framework, provides detailed information about its features, usage, and integration with other technologies. You can refer to the official Flask documentation at: http://flask.pocoo.org/docs/

2. **Natural Language Processing with Python and NLTK:** This book by Steven Bird, Ewan Klein, and Edward Loper introduces natural language processing (NLP) techniques and the use of NLTK (Natural Language Toolkit) library in Python. It can be a valuable resource for understanding and implementing NLP techniques in a semantic analyzer. Find more information about the book at: https://www.nltk.org/book/

3. **JavaScript and HTML5:** Develop Web Applications: This book by Nir Simionovich covers the fundamentals of JavaScript and HTML5 for web application development. It can help you understand the basics of JavaScript and how to utilize it effectively in conjunction with HTML and CSS. Find more information about the book at: https://www.apress.com/gp/book/9781430234043

4. **Building Semantic Analyzers for Programming Languages**: This research paper by Alan J. Demers and Scott Shenker provides an overview of semantic analyzers and their role in programming language design and implementation. While it may not focus specifically on Flask or web technologies, it offers valuable insights into the principles and techniques of semantic analysis. You can find the paper at: https://dl.acm.org/doi/10.1145/358527.358533

5. **HTML and CSS**: Design and Build Websites: This book by Jon Duckett is a beginner-friendly guide to HTML and CSS, covering the basics of web development and design. It can help you understand HTML and CSS concepts and create visually appealing interfaces for your semantic analyzer. More information about the book can be found at: https://www.wiley.com/en-us/HTML+and+CSS%3A+Design+and+Build+Websites-p-9781118008188