ALL IN ONE COMPILER(Semantic Analyser)

A MINI PROJECT REPORT

Submitted by

S P JAGRIT [RA2011033010121]
ANKIT MURARKA [RA2011033010122]

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

SRM

INSTITUTE OF SCIENCE & TECHNOLOGY

BONAFIDE CERTIFICATE

Certified that this project report "**ALL IN ONE COMPILER(Semantic Analyzer**" is the bonafide work of "**S P JAGRIT [RA2011033010121], ANKIT MURARKA [RA2011033010122]**," 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).

9/5/23

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

A semantic analyzer is a software tool used in computer programming that analyzes the meaning and context of programming code. Its primary goal is to ensure that the code is semantically correct and follows the rules and constraints of the programming language.

The semantic analyzer performs a deep analysis of the code, checking for inconsistencies in types, declarations, and usage of variables, and verifying that functions and procedures are used correctly. It goes beyond simple syntax checks and analyzes the intent behind the code and the context in which it is being used.

The output of a semantic analyzer is typically a high-level abstract representation of the code that is easier to understand and manipulate by other software tools, such as compilers, interpreters, or code generators. This helps developers catch errors and bugs early in the development process, saving time and effort in debugging later on.

Overall, the semantic analyzer is a crucial component of the overall software development process, ensuring that the code is semantically correct and will behave as intended when executed.

TABLE OF CONTENTS

CHAPTER 1

INTRODUCTION

1.1 INTRODUCTION

In computer programming, a semantic analyzer (a semantic checker or semantic parser) is a software tool that analyzes the meaning and context of programming code.

The role of a semantic analyzer is to ensure that the code follows the rules and constraints of the programming language and to detect any semantic errors that the syntax checker might not catch. It performs a deep analysis of the code, looking for inconsistencies in types, declarations, and usage of variables, and checking that functions and procedures are used correctly.

The output of a semantic analyzer is usually a high-level abstract representation of the code that is easier to understand and manipulate by other software tools, such as compilers, interpreters, or code generators. The semantic analyzer is a crucial component of the compilation process, translating human-readable code into machine-executable code.

One of the primary tasks of a semantic analyzer is to verify that the types of all variables and expressions are correct. For example, if a variable is declared to be an integer, the semantic analyzer will check that all operations performed on that variable are compatible with the integer type. If a type mismatch is detected, the analyzer will flag it as an error.

Overall, the role of a semantic analyzer is to ensure that the code is semantically correct and will behave as intended when executed. This helps developers catch errors and bugs early in the development process, saving time and effort in debugging later on.

1.2 PROBLEM STATEMENT

The problem statement of this project is to develop a robust and efficient semantic analyzer for a given programming language. The semantic analyzer should be capable of performing the following tasks:

1. **Type Checking:** The semantic analyzer should enforce type safety rules and verify the compatibility of data types used in expressions, assignments, and function calls. It should detect type errors such as assigning a value of an incompatible type, mismatched function arguments, or incompatible operands in arithmetic or logical operations.

2. **Scope Analysis:** The semantic analyzer should track variable declarations and their scopes within the program. It should detect and report errors such as undeclared variables, redeclared variables in the same scope, and access to variables outside their defined scope.

3. **Function and Procedure Validation:** The semantic analyzer should validate the usage of functions and procedures, ensuring that they are correctly declared, called with the appropriate number and types of arguments, and used in a syntactically and semantically correct manner.

4. **Semantic Constraints:** The semantic analyzer should enforce other semantic constraints imposed by the programming language, such as the correct usage of keywords, reserved identifiers, and language-specific rules related to control flow, looping, and conditional statements.

5. **Error Reporting:** The semantic analyzer should provide informative error messages that clearly identify the nature and location of semantic errors in the source code.

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. **Type Checking:** The semantic analyzer verifies the compatibility and correctness of data types used in expressions, assignments, and function calls. It ensures that the operations and manipulations performed on variables and expressions are semantically valid, preventing type-related errors during runtime.

2. **Scope Analysis:** The semantic analyzer tracks variable declarations and their scopes within the program. It ensures that variables are properly declared before their usage, detecting undeclared variables, redeclaration of variables in the same scope, and access to variables outside their defined scope. This helps in maintaining proper scoping and avoiding conflicts or inconsistencies.

3. **Symbol Table Management:** The semantic analyzer builds and maintains a symbol table, which stores information about identifiers (variables, functions, procedures, etc.) and their associated attributes (data type, scope, memory location, etc.). The symbol table serves as a reference for the compiler to resolve symbol references, detect redeclarations, and perform other semantic analyses.

4. **Function and Procedure Validation:** The semantic analyzer validates the usage of functions and procedures, ensuring that they are correctly declared, called with the appropriate number and types of arguments, and used in a syntactically and semantically correct manner. It helps in detecting errors such as calling undefined functions, mismatched function signatures, or incorrect parameter passing.

5. **Array and Pointer Checks:** If the programming language supports arrays and pointers, the semantic analyzer performs checks related to array bounds, pointer arithmetic, and the appropriate usage of array indices and pointer dereferencing. It ensures that array accesses are within bounds, pointer operations are valid, and memory accesses are correct.

6. **Semantic Constraints:** The semantic analyzer enforces other semantic constraints imposed by the programming language. This includes language-specific rules related to control flow, looping, conditional statements, and other language constructs. It ensures that the program adheres to the language's specified semantics, preventing semantic violations and ensuring reliable program execution.

7. **Error Reporting:** The semantic analyzer provides informative error messages that clearly identify the nature and location of semantic errors in the source code. It helps programmers understand the issues and assists in resolving them by providing meaningful feedback and suggestions for corrective actions.

By achieving these objectives, the semantic analyzer contributes to the overall quality, reliability, and correctness of the compiled program. It acts as a crucial component in the compiler pipeline, bridging the gap between the syntactic analysis and the subsequent code generation phase.

1.4 Need for Semantic Analysis

Semantic analysis is a pass by a compiler that adds semantic information to the parse tree and performs certain checks based on this information. It logically follows the parsing phase, in which the parse tree is generated, and logically precedes the code generation phase, in which (intermediate/target) code is generated. (In a compiler implementation, it may be possible to fold different phases into one pass.) Typical examples of semantic information that is added and checked is typing information (type checking) and the binding of variables and function names to their definitions (object binding). Sometimes also some early code optimization is done in this phase. For this phase the compiler usually maintains symbol tables in which it stores what each symbol (variable names, function names, etc.) refers to.

Following things are done in Semantic Analysis:

1. **Disambiguate Overloaded operators :** If an operator is overloaded, one would like to specify the meaning of that particular operator because from one will go into code generation phase next.

2. **Type checking :** The process of verifying and enforcing the constraints of types is called type checking. This may occur either at compile-time (a static check) or run-time (a dynamic check). Static type checking is a primary task of the semantic analysis carried out by a compiler. If type rules are enforced strongly (that is, generally allowing only those automatic type conversions which do not lose information), the process is called strongly typed, if not, weakly typed.

3. **Uniqueness checking :** Whether a variable name is unique or not, in the its scope. Syntax Directed Translation: Syntax Directed definitions, Bottom up Evaluation of S attributed definitions, L attributed definitions, Top Down translation, Bottom up evaluation of Inherited attributes Type Checking: Type Systems, Specification of a simple type checker 2

4. **Type coercion :** If some kind of mixing of types is allowed. Done in languages which are not strongly typed. This can be done dynamically as well as statically.

5. **Name Checks :** Check whether any variable has a name which is not allowed. Ex. Name is same as an identifier(Ex. int in java).

A parser has its own limitations in catching program errors related to semantics, something that is deeper than syntax analysis. Typical features of semantic analysis cannot be modeled using context free grammar formalism. If one tries to incorporate those features in the definition of a language then that language doesn't remain context free anymore. These are a couple of examples which tell us that typically what a compiler has to do beyond syntax analysis. An identifier x can be declared in two separate functions in the program, once of the type int and then of the type char. Hence the same identifier will have to be bound to these two different properties in the two different contexts.

1.5 REQUIREMENTS SPECIFICATION

Hardware Requirements:

Processor: A modern multi-core processor (e.g., Intel Core i5 or higher) to handle the compilation process efficiently.
Memory (RAM): A minimum of 8 GB is recommended
Storage: Adequate storage space for the source code, compiler tools, libraries, and any additional resources.(A minimum of 128 GB is recommended)

Operating System: Windows / linux distributions / macOS

Development Environment:
Integrated Development Environment (IDE): Visual Studio Code, Eclipse, or JetBrains IntelliJ IDEA
Version Control: Git to manage source code, track changes, and collaborate with other developers if applicable

Programming Languages and Tools:

Compiler Design Language: Python
Back-end Framework: Flask
Front-end Framework: Boostrap

Documentation and Reporting:

Document Preparation Software: Used word processing software like Microsoft Word for creating the compiler design report.

CHAPTER 2

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

Whether a variable has been declared? Are there variables which have not been declared? What is the type of the variable? Whether a variable is a scalar, an array, or a function? What declaration of the variable does each reference use?If an expression is type consistent?If an array use like A[i,j,k] is consistent with the declaration? Does it have three dimensions?

For example, we have the third question from the above list, i.e., what is the type of a variable and we have a statement like int a, b , c;

Then we see that syntax analyzer cannot alone handle this situation. We actually need to traverse the parse trees to find out the type of identifier and this is all done in semantic analysis phase. Purpose of listing out the questions is that unless we have answers to these questions we will not be able to write a semantic analyzer. This becomes a feedback 3 mechanism. If the compiler has the answers to all these questions only then will it be able to successfully do a semantic analysis by using the generated parse tree. These questions give a feedback to what is to be done in the semantic analysis. These questions help in outlining the work of the semantic analyzer. In order to answer the previous questions the compiler will have to keep information about the type of variables, number of parameters in a particular function etc. It will have to do some sort of computation in order to gain this information. Most compilers keep a structure called symbol table to store this information. At times the information required is not available locally, but in a different scope altogether. In syntax analysis we used context free grammar. Here we put lot of attributes around it. So it consists of context sensitive grammars along with extended attribute grammars. Ad-hoc methods also good as there is no structure in it and the formal method is simply just too tough. So we would like to use something in between. Formalism may be so difficult that writing specifications itself may become tougher than writing compiler itself. So we do use attributes but we do analysis along with parse tree itself instead of using context sensitive grammars.

2.2 Limitations of CFGs.

Context free grammars deal with syntactic categories rather than specific words. The declare before use rule requires knowledge which cannot be encoded in a CFG and thus CFGs cannot match an instance of a variable name with another. Hence the we introduce the attribute grammar framework.

Syntax directed definition

This is a context free grammar with rules and attributes. It specifies values of attributes by associating semantic rules with grammar productions.

An example

PRODUCTION	SEMANTIC RULE
E → E1 + T	E.code = E1.code

Syntax Directed Translation

This is a compiler implementation method whereby the source language translation is completely driven by the parser.
The parsing process and parse tree are used to direct semantic analysis and translation of the source program.
Here we augment conventional grammar with information to control semantic analysis and translation. This grammar is referred to as attribute grammar.
The two main methods for SDT are Attribute grammars and syntax directed translation scheme

Attributes

An attribute is a property whose value gets assigned to a grammar symbol. Attribute computation functions, also known as semantic functions are functions associated with productions of a grammar and are used to compute the values of an attribute. Predicate functions are functions that state some syntax and the static semantic rules of a particular grammar.

Types of Attributes

Type -These associate data objects with the allowed set of values.
Location - May be changed by the memory management routine of the operating system.
Value -These are the result of an assignment operation.

Name-These can be changed when a sub-program is called and returns. **Component** - Data objects comprised of other data objects. This binding is represented by a pointer and is subsequently changed.

Synthesized Attributes.

These attributes get values from the attribute values of their child nodes. They are defined by a semantic rule associated with the production at a node such that the production has the non-terminal as its head.

An example

S → ABC
S is said to be a synthesized attribute if it takes values from its child node (A, B, C).

An example
E → E + T { E.value = E.value + T.value }
Parent node E gets its value from its child node.

Inherited Attributes.

These attributes take values from their parent and/or siblings.
They are defined by a semantic rule associated with the production at the parent such that the production has the non-terminal in its body.
They are useful when the structure of the parse tree does not match the abstract syntax tree of the source program.
They cannot be evaluated by a pre-order traversal of the parse tree since they depend on both left and right siblings.

An example;
S → ABC

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

Expansion

This is when a non-terminal is expanded to terminals as per the provided grammar.

Reduction

This is when a terminal is reduced to its corresponding non-terminal as per the grammar rules. Note that syntax trees are parsed top, down and left to right

Attribute grammar

This is a special case of context free grammar where additional information is appended to one or more non-terminals in-order to provide context-sensitive information.
We can also define it as SDDs without side-effects.
It is the medium to provide semantics to a context free grammar and it helps with the specification of syntax and semantics of a programming language.
When viewed as a parse tree, it can pass information among nodes of a tree.

An Example

Given the CFG below;
$E \rightarrow E + T$ { E.value = E.value + T.value }
The right side contains semantic rules that specify how the grammar should be interpreted.
The non-terminal values of E and T are added and their result copied to the non-terminal E.

An Example

Consider the grammar for signed binary numbers
number → signlist
sign → + | −
list → listbit | bit
bit → 0 | 1

We want to build an attribute grammar that annotates Number with the value it represents.
First we associate attributes with grammar symbols

SYMBOL	ATTRIBUTES
number	val
sign	neg
list	pos, val
bit	pos, val

The attribute grammar

Production	Attribute Rule

$number \rightarrow sign\ list$	$list.pos = 0$
	if $sign.neg$:
	$number.val = -list.val$
	else:
	$number.val = -list.val$

$sign \rightarrow +$	$sign.neg =$ false
$sign \rightarrow -$	$sign.neg =$ true
$list \rightarrow bit$	$bit.pos = list.pos$ $list.val = bit.val$
$list_0 \rightarrow list_1 bit$	$list_1.pos = list_0.pos + 1$ $bit.pos = list_0.pos$ $list_0.val = list_1.val + bit.val$
$bit \rightarrow 0$	$bit.val = 0$
$bit \rightarrow 1$	$bit.val = 2^{bit.pos}$

Defining an Attribute Grammar

Attribute grammar will consist of the following features;

- Each symbol X will have a set of attributes A(X)
- A(X) can be;
    - Extrinsic attributes obtained outside the grammar, notable the symbol table
    - Synthesized attributes passed up the parse tree
    - Inherited attributes passed down the parse tree.
- Each production of the grammar will have a set of semantic functions and predicate functions(may be an empty set)

- Based on the way an attribute gets its value, attributes can be divided into two categories; these are, Synthesized or inherited attributes.

Abstract Syntax Trees(ASTs)

These are a reduced form of a parse tree.

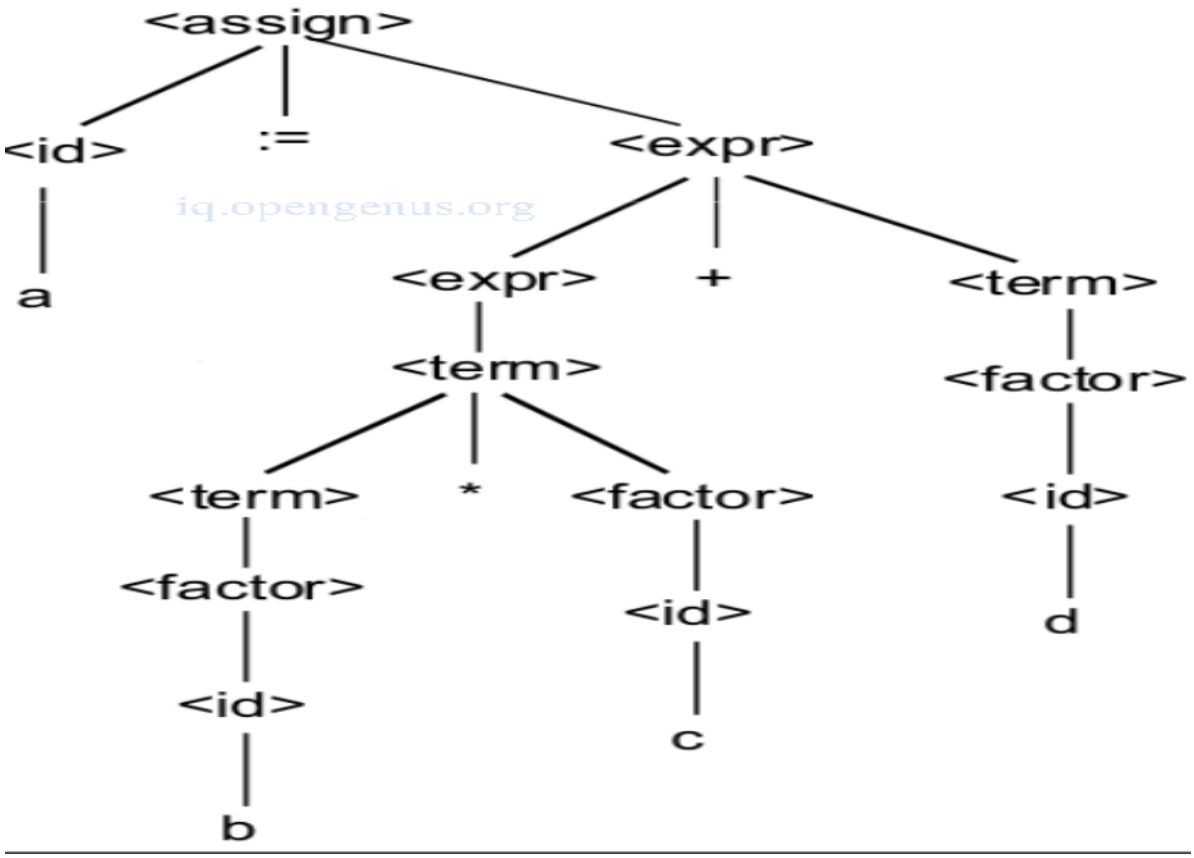
They don't check for string membership in the language of the grammar.

They represent relationships between language constructs and avoid derivations.

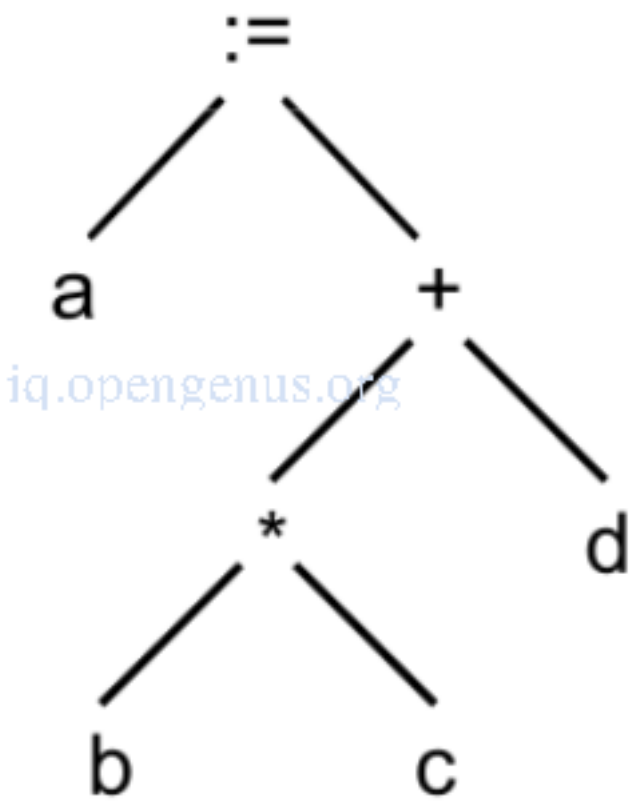
An example

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:-

For the Front-End Framework we have use Bootstrap, overview of Bootstrap:-

1. **Bootstrap Framework:** Bootstrap is also the name of a popular front-end development framework used to build responsive and mobile-first websites. The Bootstrap framework consists of various components and tools that aid in web development, including:

2. **HTML/CSS Components:** Bootstrap provides a collection of pre-designed HTML and CSS components, such as buttons, forms, navigation bars, and grids. These components can be easily integrated into web pages to ensure consistency and responsiveness.

3. **JavaScript Plugins:** Bootstrap offers a set of JavaScript plugins that add functionality and interactivity to web pages. These plugins include features like carousels, modals, tooltips, and dropdown menus.

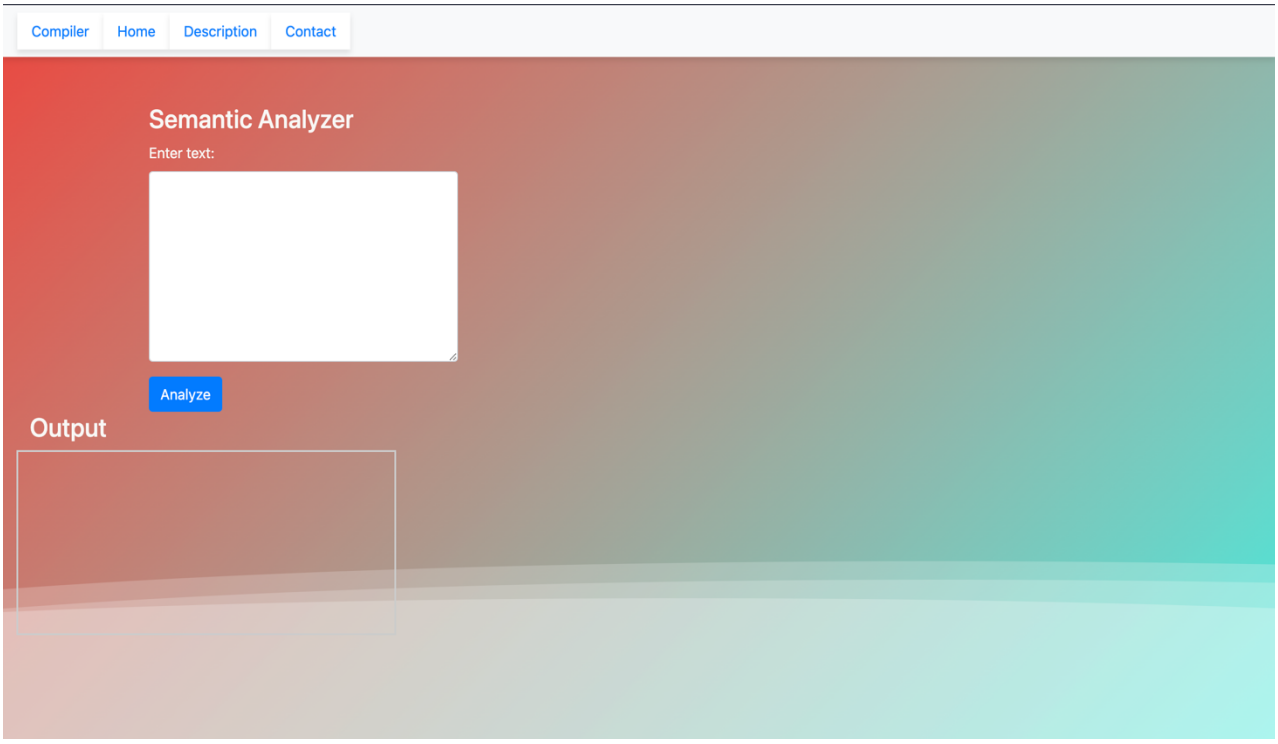
4. **Responsive Grid System:** Bootstrap includes a responsive grid system that enables developers to create flexible and responsive layouts for web pages. The grid system helps in achieving a consistent look and feel across different devices and screen sizes.

5. **Bootstrapping a Compiler or Interpreter:** In the context of compiler or interpreter design, bootstrapping refers to the process of implementing a compiler or interpreter for a programming language using the same language itself. The bootstrapping process typically involves the following stages:

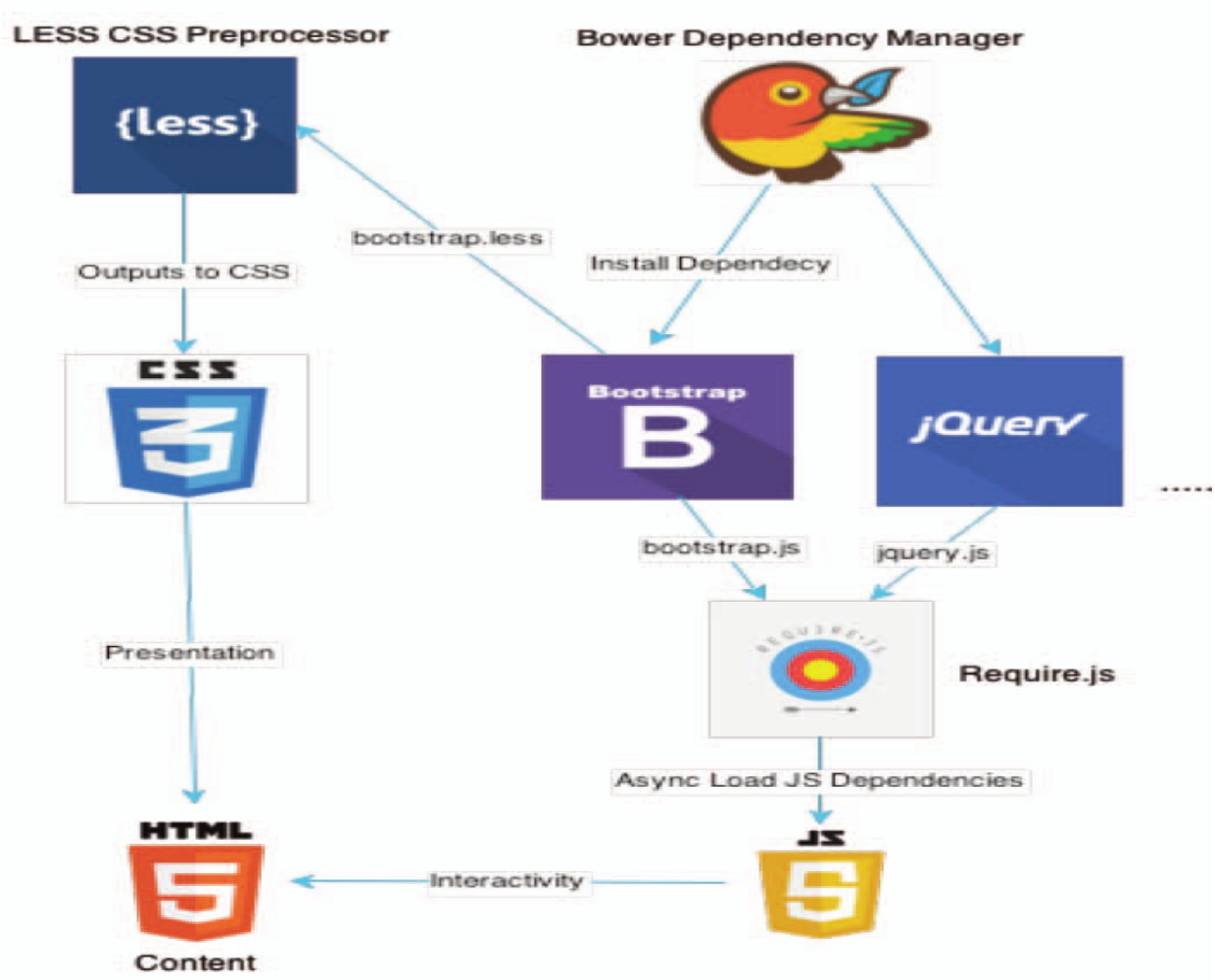
6. **Initial Compiler/Interpreter:** A basic version of the compiler or interpreter is written in a different language (often a lower-level language or an existing language). It is used to compile or interpret the subsequent versions of the compiler or interpreter.

7. **Self-Compilation:** The initial compiler or interpreter is used to compile or interpret an updated version of itself written in the target language. Iterative Refinement: The process is repeated, using each new version to compile or interpret a more advanced version until the final compiler or interpreter is achieved.

Overall, the components of bootstrap vary depending on the specific context, such as the system or software application being bootstrapped. It can involve components like the bootloader, operating system kernel, application initialization, HTML/CSS components, JavaScript plugins, and self-compilation in the case of compiler or interpreter design.



FRONT-END ARCHITECTURE DESIGN:-



3.2 BACK-END DESIGN:-

Flask is a popular back-end web framework for Python. It provides a lightweight and flexible approach to building web applications. When using Flask as a back-end framework, it typically consists of the following key components:

1. **Routing:** Flask allows you to define URL routes and associate them with specific functions or methods. These routes determine how the application responds to different URLs or HTTP methods (GET, POST, etc.). By using decorators or URL patterns, you can map routes to corresponding view functions or class methods.

2. **Views:** Views in Flask are Python functions or class methods that handle requests and generate responses. They receive data from requests, process it, and return appropriate responses. Views can render templates, return JSON data, redirect to other URLs, or perform other actions based on the application's requirements.

3. **Templates:** Flask integrates with template engines (such as Jinja2) to separate the presentation logic from the application logic. Templates allow you to dynamically generate HTML pages by incorporating data and logic. Flask provides support for template inheritance, variable substitution, control structures, and other template features.

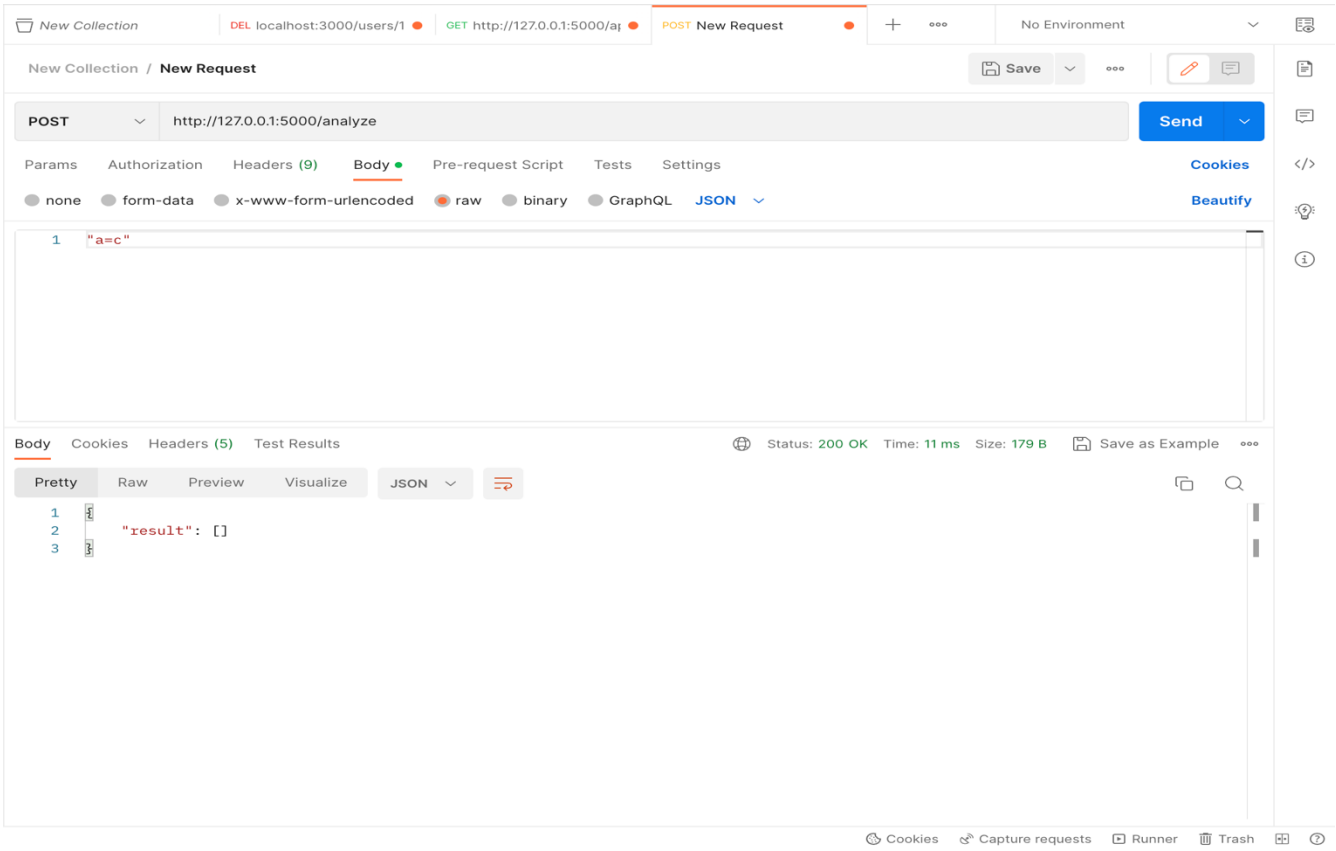
4. **Forms:** Flask provides utilities for handling HTML forms, including form validation, data retrieval, and rendering. It allows you to define form classes, specify validation rules, and generate HTML form elements. Flask also supports form submission handling, including processing form data and handling validation errors.

5. **Middleware:** Flask allows the use of middleware, which are components that intercept and process requests and responses before they reach the view functions. Middleware can perform tasks such as authentication, logging, error handling, or modifying the request/response objects.

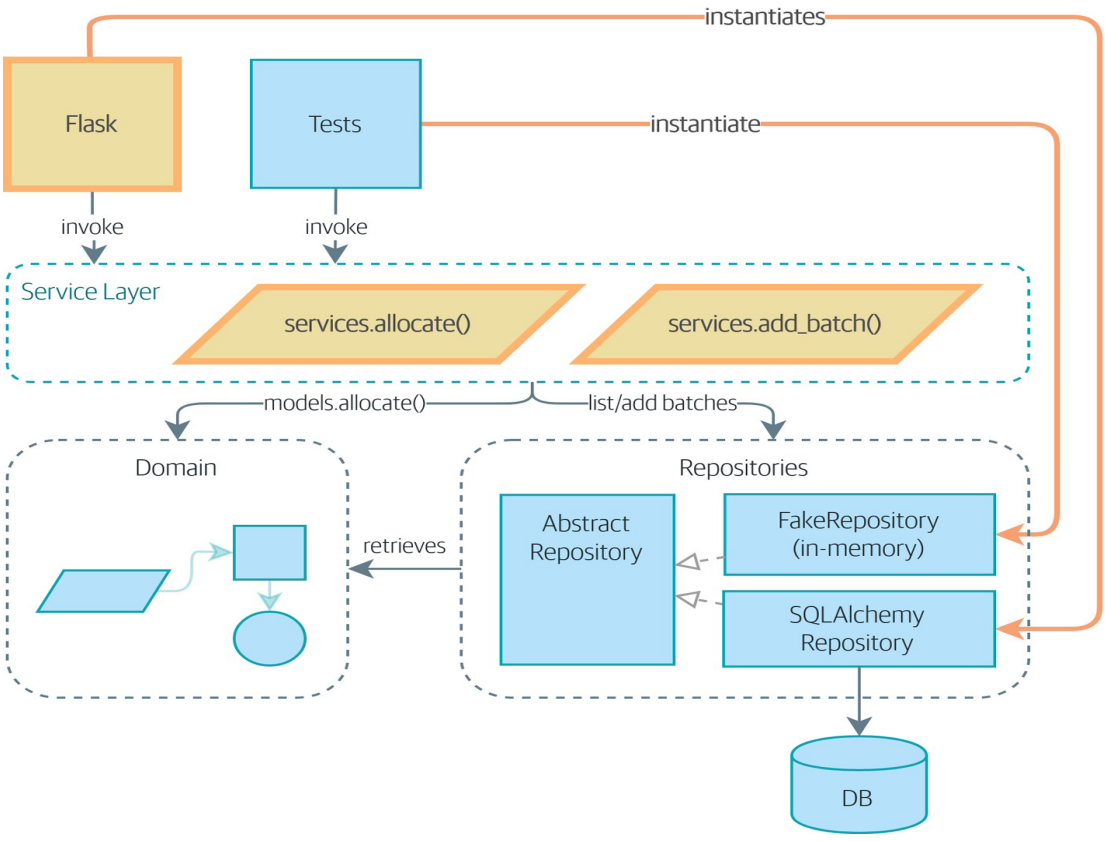
6. **Extensions:** Flask has a rich ecosystem of extensions that provide additional functionality and integrate with various services. These extensions cover areas such as database integration, authentication, session management, caching, API development, and more. Extensions can be easily integrated into Flask applications to enhance their capabilities.

7. **Configuration:** Flask allows you to configure various aspects of the application, such as database connections, debugging options, logging settings, and more. Configuration can be done through environment variables, configuration files, or programmatically in the application code.

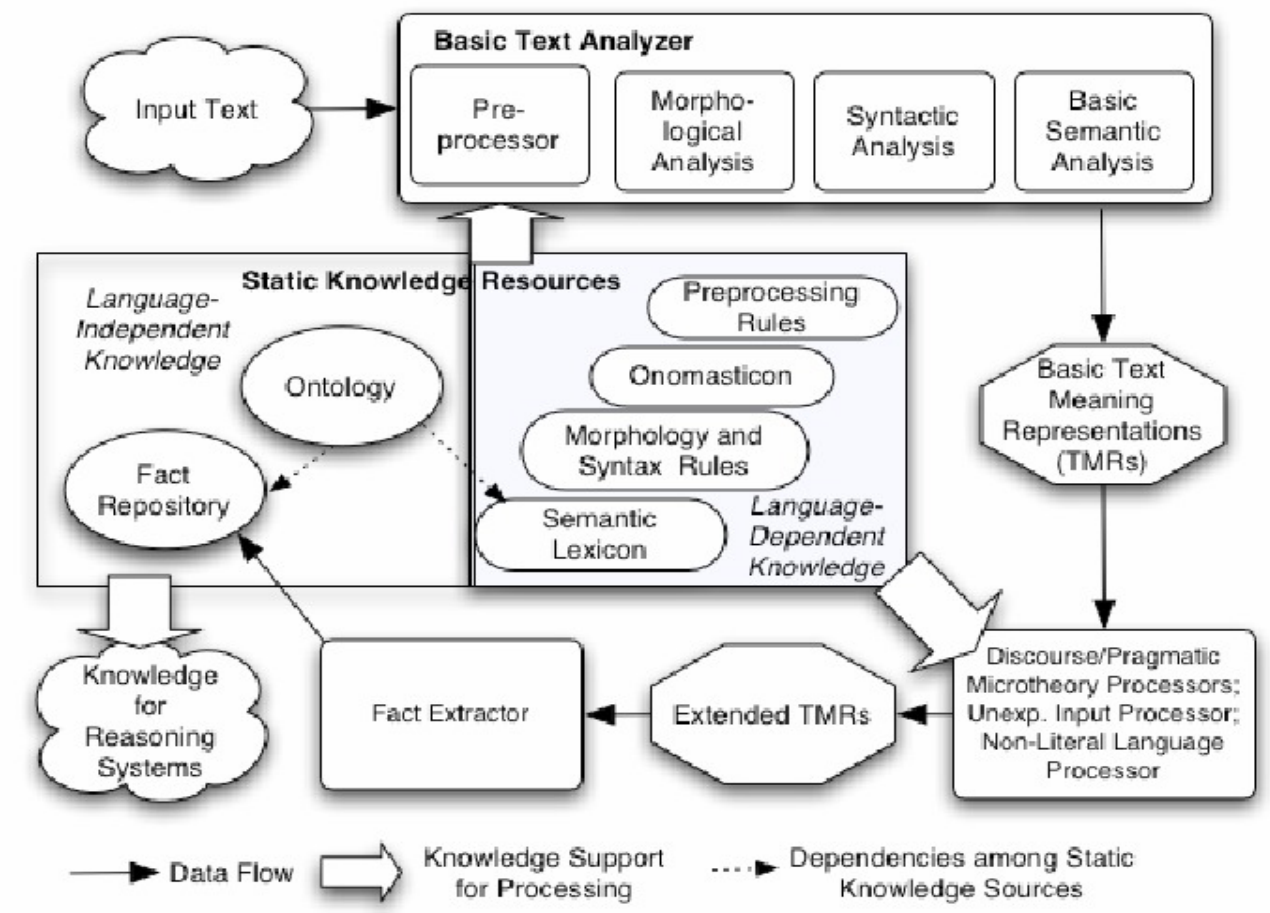
These components of Flask form the foundation for developing back-end web applications. They provide the necessary tools and abstractions to handle routing, views, templates, database interactions, form handling, middleware, and configuration. Flask's simplicity and flexibility make it a popular choice for building web applications of varying sizes and complexities.



BACK END ARCHITECTURE DESIGN:-



SEMANTIC ANALYZER ARCHITECTURE DESIGN:-



CHAPTER 4

The requirement to run the script -

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

1. **A compatible version of Python:** Flask is a web framework for Python, so you will need to have Python installed on your computer. The specific version of Python required may vary depending on the version of Flask and other dependencies used in the code.

2. **Flask and its dependencies:** Flask is a third-party library for Python, and it has several dependencies that must be installed to use it. These dependencies may include Werkzeug, Jinja2, and others. You can use a package manager like pip to install these dependencies.

3. **A web server:** Flask is a web framework, and it requires a web server to run. You can use the built-in development server that comes with Flask, or you can use a production-ready web server like Apache or Nginx.

4. **A browser:** The JavaScript code used in the semantic analyzer will be executed in the user's browser, so you will need a modern web browser like Chrome, Firefox, or Safari to view and interact with the analyzer.

5. **Code editor or IDE:** To edit the Flask and JavaScript code, you will need a code editor or integrated development environment (IDE) that supports Python and JavaScript.

6. **Semantic analysis libraries:** Depending on the specific requirements of your semantic analyzer, you may need to install additional libraries or tools for semantic analysis, such as Natural Language Processing (NLP) libraries or machine learning frameworks.

Overall, running a semantic analyzer built using Flask and JavaScript requires a working environment with the appropriate dependencies, a web server, and a compatible browser. With these requirements met, you can run and interact with the semantic analyzer to analyze and understand the meaning and context of programming code.

CHAPTER 5

CODING AND TESTING

CODING:-

1. APP.PY:-

```
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
```

```
# Traverse the AST and check for semantic errors
    for node in ast.walk(parsed_program):
        if isinstance(node, ast.Call):
            if isinstance(node.func, ast.Attribute):

if node.func.attr == 'append':
                    if isinstance(node.func.value, ast.Name) and node.func.value.id == 'list':
                        errors.append(f"Using 'list.append' is not recommended, use the '+' operator
instead. Line {node.lineno}")
            elif isinstance(node.func, ast.Name):
                if node.func.id == 'print':
                    errors.append(f"Using 'print' is not recommended, use logging instead. Line
{node.lineno}")

    return errors

@app.route('/analyze', methods=['POST'])
def analyze():
    app.logger.info("Got req")
    input_data = request.json
```

2. MAIN.PY:-

```
import ast

def analyze_python_program(program):
    """
    Analyze a Python program and return a list of semantic errors.

    :param program: str, the Python program to analyze
    :return: list of str, the semantic errors found in the program
    """
    errors = []

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

    # Traverse the AST and check for semantic errors
    for node in ast.walk(parsed_program):
        if isinstance(node, ast.Call):
            if isinstance(node.func, ast.Attribute):
                if node.func.attr == 'append':
                    if isinstance(node.func.value, ast.Name) and node.func.value.id == 'list':
                        errors.append(f"Using 'list.append' is not recommended, use the '+' operator
instead. Line {node.lineno}")
            elif isinstance(node.func, ast.Name):
                if node.func.id == 'print':
                    errors.append(f"Using 'print' is not recommended, use logging instead. Line
{node.lineno}")

    return errors
program = """
import cmath

else:
    print("No semantic errors were found.")
```

3. **PROTOTYPE.HTML:-**(contains the structure and the JS code that analyses the code)

```
<!DOCTYPE html>
<html lang="en">
  <head>
    <meta charset="UTF-8" />
    <meta http-equiv="X-UA-Compatible" content="IE=edge" />
    <meta name="viewport" content="width=device-width, initial-scale=1.0" />
    <link rel="stylesheet" href="../static/css/main.css" />
    <title>Document</title>
    <link
      rel="stylesheet"
      href="https://cdn.jsdelivr.net/npm/bootstrap@4.0.0/dist/css/bootstrap.min.css"
      integrity="sha384-
Gn5384xqQ1aoWXA+058RXPxPg6fy4IWvTNh0E263XmFcJlSAwiGgFAW/dAiS6JXm"
      crossorigin="anonymous"
    />
  </head>
  <body>
    <div>
      <div class="wave"></div>
      <div class="wave"></div>
      <div class="wave"></div>
    </div>
    <div class="bg-image">
      <nav class="navbar navbar-expand-lg navbar-light bg-light">
        <a class="navbar" href="#">Compiler</a>
        <button
          class="navbar-toggler"
          type="button"
          data-toggle="collapse"
          data-target="#navbarNav"
          aria-controls="navbarNav"
          aria-expanded="false"
          aria-label="Toggle navigation"
        >
          <span class="navbar-toggler-icon"></span>
        </button>
        <div class="collapse navbar-collapse" id="navbarNav">
          <ul class="navbar-nav">
            <li class="nav-item active">
              <a class="navbar" href="#"
                >Home <span class="sr-only">(current)</span></a
              >
            </li>
            <li class="nav-item">
              <a class="navbar" href="/description">Description</a>
            </li>
```

```
<a class="navbar" href="#"
            >Home <span class="sr-only">(current)</span></a
          >
        </li>
        <li class="nav-item">
          <a class="navbar" href="/description">Description</a>
        </li>
        <li class="nav-item">
          <a class="navbar" href="/contact">Contact</a>
        </li>
      </ul>
    </div>
  </nav>
  <div class="container">
    <div class="row mt-3">
      <div class="col-md-4">
        <h3>Semantic Analyzer</h3>
        <form onsubmit="event.preventDefault()">
          <div class="form-group">
            <label for="inputText">Enter text:</label>
            <textarea
              class="form-control"
              id="inputText"
              rows="8"
            ></textarea>
          </div>
          <button onclick="submitCode()" class="btn btn-primary">Analyze</button>
        </form>
      </div>
    </div>
  </div>
  <div class="col-md-8">
    <h3 class="col-md-4">Output</h3>
    <div id="output"></div>
  </div>
 </div>
 <script>

   function submitCode(e){

     var inputData = document.getElementById('inputText').value;
     console.log(inputData)
fetch("http://127.0.0.1:5000/analyze", {
     method: "POST",
     headers: {
       "Content-Type": "application/json",
     },
     body: JSON.stringify(`${inputData}`),
```

```
},
      body: JSON.stringify(`${inputData}`),
    })
      .then((response) => response.json())
      .then((data) => {
        const outputData = data.result;
        if(outputData.length == 0 ){
          document.getElementById('output').innerText = 'No semantic errors';
        }
        if(outputData.length == 1 ){
          document.getElementById('output').innerText = outputData[0];
        }
        console.log(outputData.length)
        // Code to process outputData
      });
    }
  </script>

  <script
    src="https://code.jquery.com/jquery-3.2.1.slim.min.js"
    integrity="sha384-
KJ3o2DKtIkvYIK3UENzmM7KCkRr/rE9/Qpg6aAZGJwFDMVNA/GpGFF93hXpG
5KkN"
    crossorigin="anonymous"
  ></script>
  <script
    src="https://cdn.jsdelivr.net/npm/popper.js@1.12.9/dist/umd/popper.min.js"
    integrity="sha384-
ApNbgh9B+Y1QKtv3Rn7W3mgPxhU9K/ScQsAP7hUibX39j7fakFPskvXusvfa0b4
Q"
    crossorigin="anonymous"
  ></script>
  <script
    src="https://cdn.jsdelivr.net/npm/bootstrap@4.0.0/dist/js/bootstrap.min.js"
    integrity="sha384-
JZR6Spejh4U02d8jOt6vLEHfe/JQGiRRSQQxSfFWpi1MquVdAyjUar5+76PVCmYl
"
    crossorigin="anonymous"
  ></script>
 </body>
</html>
```

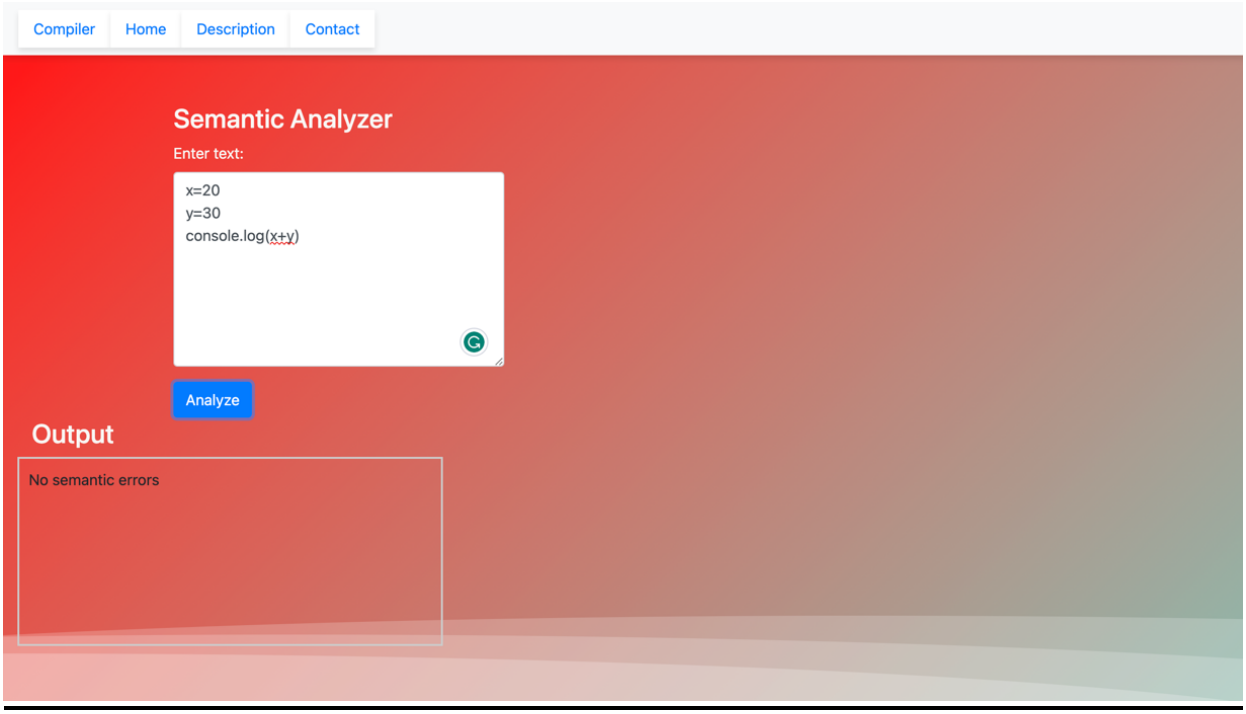
4. MAIN.CSS:-

```
body {
  margin: auto;
  font-family: -apple-system, BlinkMacSystemFont, sans-serif;
  overflow: auto;
  background: linear-gradient(315deg, rgba(101,0,94,1) 3%, rgba(60,132,206,1) 38%,
rgba(48,238,226,1) 68%, rgba(255,25,25,1) 98%);
  animation: gradient 15s ease infinite;
  background-size: 400% 400%;
  background-attachment: fixed;
}

@keyframes gradient {
  0% {
     background-position: 0% 0%;
  }
  50% {
     background-position: 100% 100%;
  }
  100% {
     background-position: 0% 0%;
  }
}

.wave {
  background: rgb(255 255 255 / 25%);
  border-radius: 1000% 1000% 0 0;
  position: fixed;
  width: 200%;
  height: 12em;
  animation: wave 10s -3s linear infinite;
  transform: translate3d(0, 0, 0);
  opacity: 0.8;
  bottom: 0;
  left: 0;
  z-index: -1;
}

.wave:nth-of-type(2) {
  bottom: -1.25em;
  animation: wave 18s linear reverse infinite;
  opacity: 0.8;
}
```

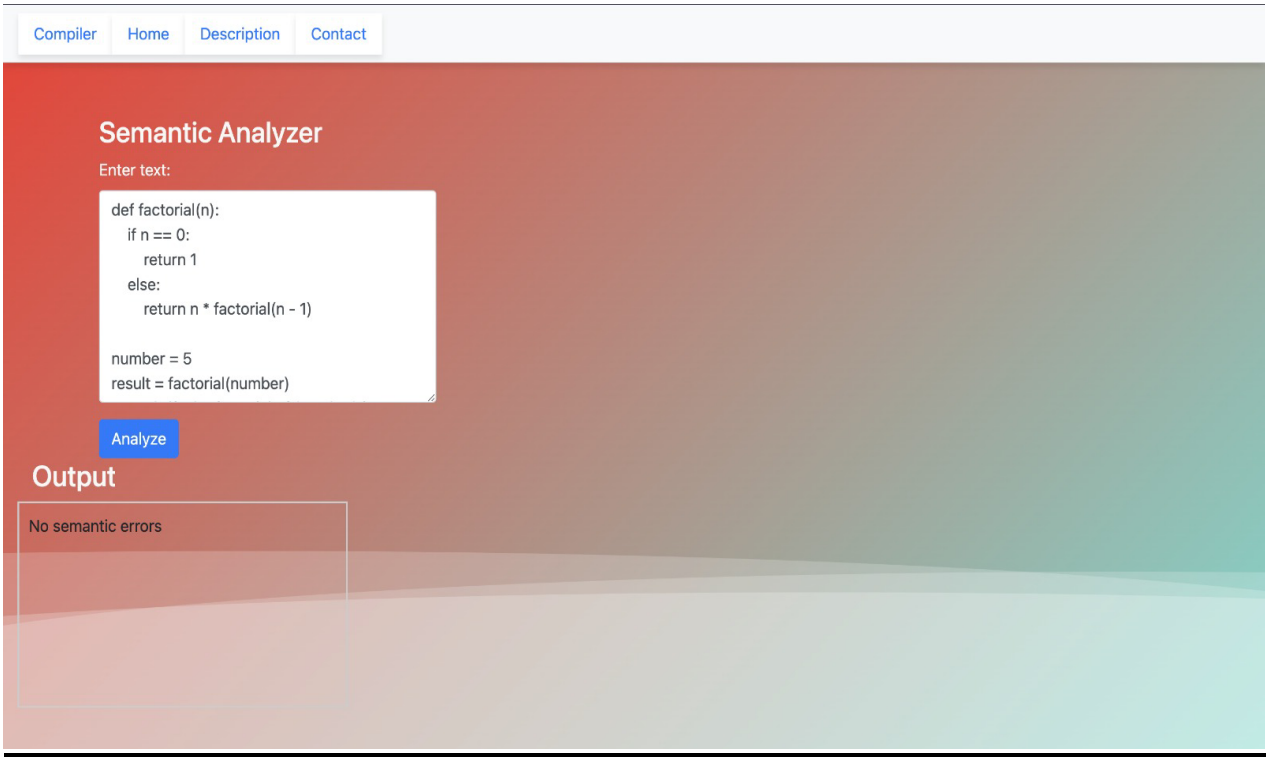
```
.wave:nth-of-type(3) {
  bottom: -2.5em;
  animation: wave 20s -1s reverse infinite;
  opacity: 0.9;
}

@keyframes wave {
  2% {
     transform: translateX(1);
  }

  25% {
     transform: translateX(-25%);
  }

  50% {
     transform: translateX(-50%);
  }

  75% {
     transform: translateX(-25%);
  }

  100% {
     transform: translateX(1);
  }
}
.navbar {
  background-color: #fff;
  box-shadow: 0 4px 6px rgba(0, 0, 0, 0.1);
}
.form-group {
  margin-bottom: 20px;
}

#output {
  height: 200px;
  border: 2px solid #ccc;
  padding: 10px;
  margin-right: 500px;
  overflow-y: auto;
}
.container {
  margin-top: 50px;
}
.col-md-4{
  color:rgb(249, 248, 246);
```

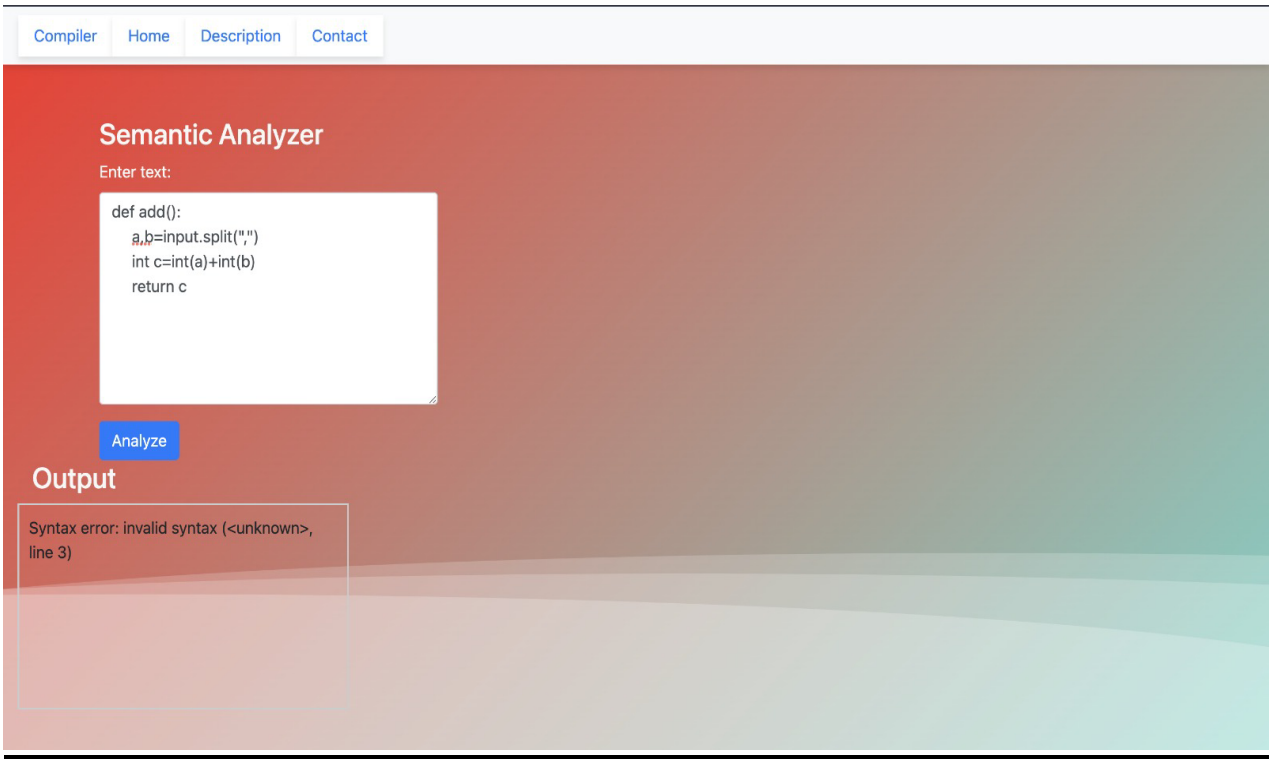
TESTING

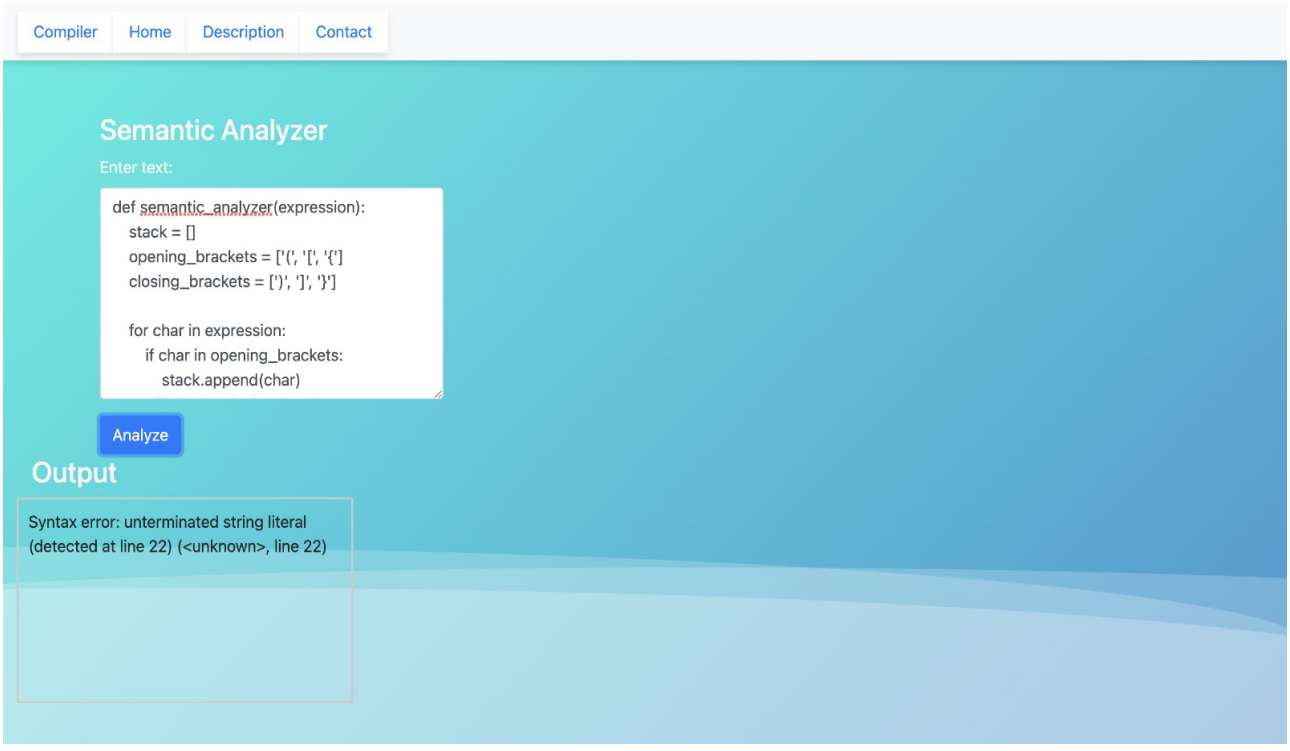
Testing the analyzer -



Testing with some different inputs -




Compiler
Home
Description
Contact
Semantic Analyzer
Enter text:
def add():
a,b=input.split(",")
int c=int(a)+int(b)
return c
Analyze
Output
Syntax error: invalid syntax (<unknown>, line 3)

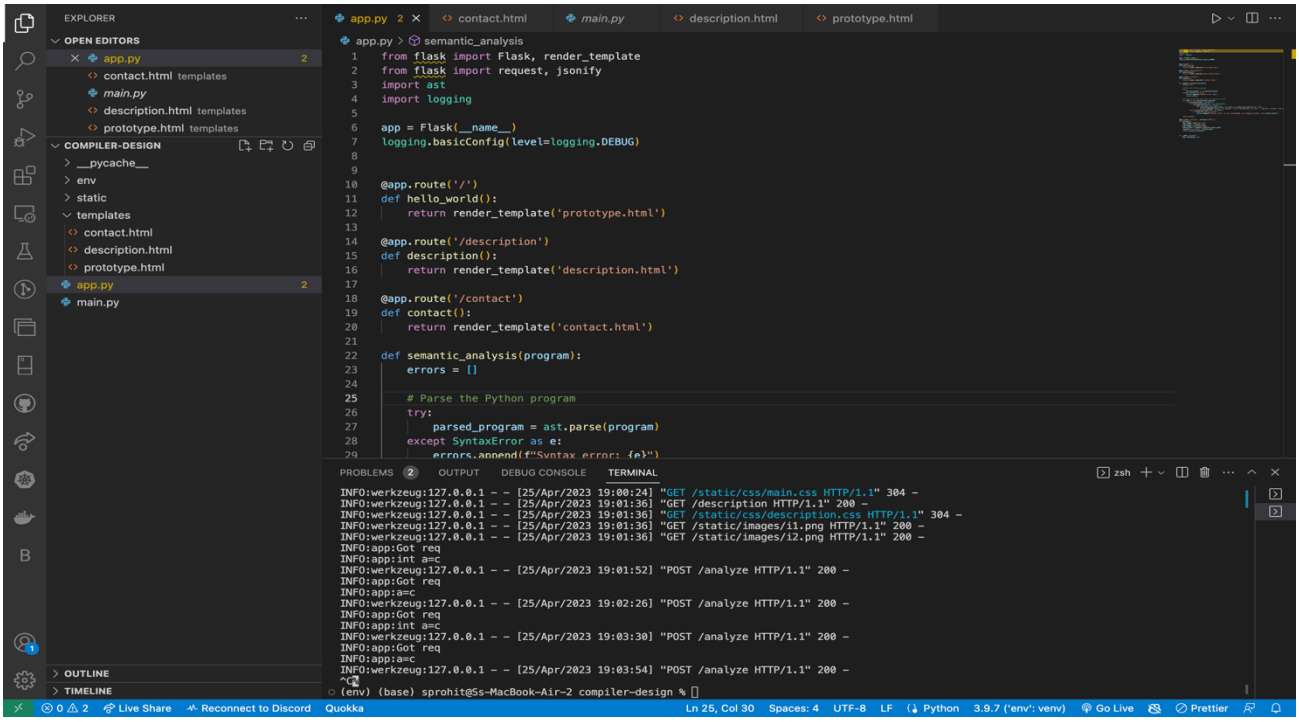

Compiler
Home
Description
Contact
Semantic Analyzer
Enter text:
def semantic_analyzer(expression):
stack = []
opening_brackets = ['(', '[', '{']
closing_brackets = [')', ']', '}']
for char in expression:
if char in opening_brackets:
stack.append(char)
Analyze
Output
Syntax error: unterminated string literal (detected at line 22) (<unknown>, line 22)

CHAPTER 6

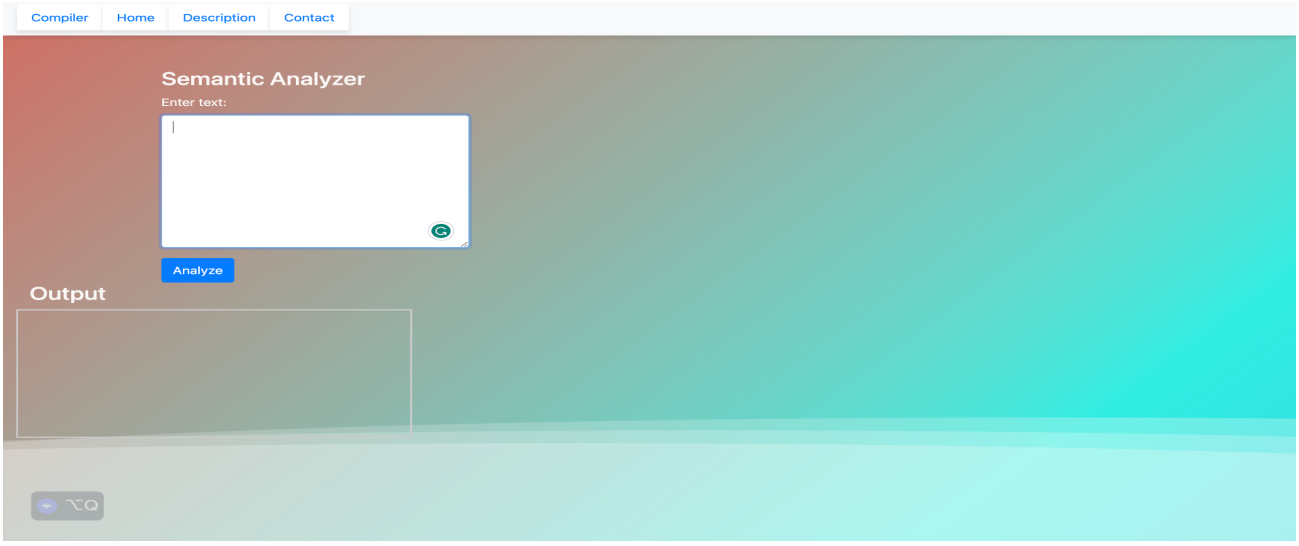
OUTPUT AND RESULTS

5.1 OUTPUT:-

Terminal -



Home Page -



Description Page about the Analyzer -

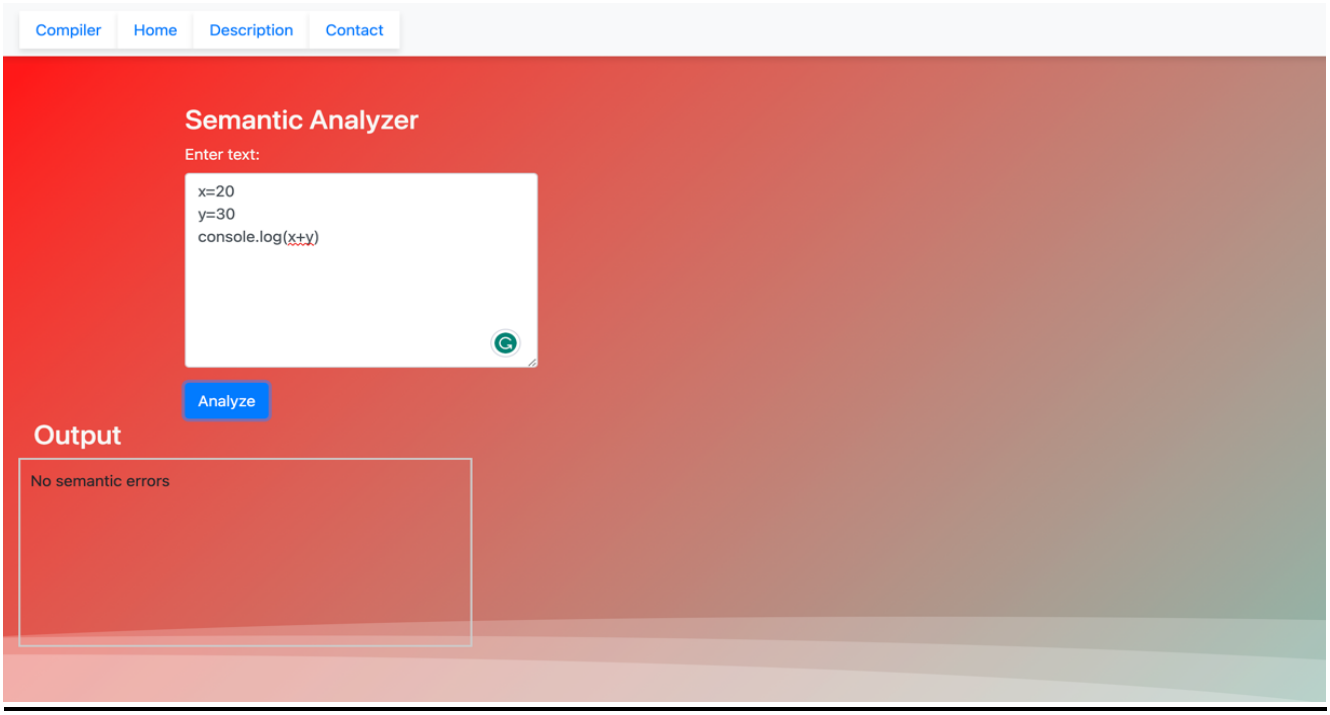
Semantic Analyzer

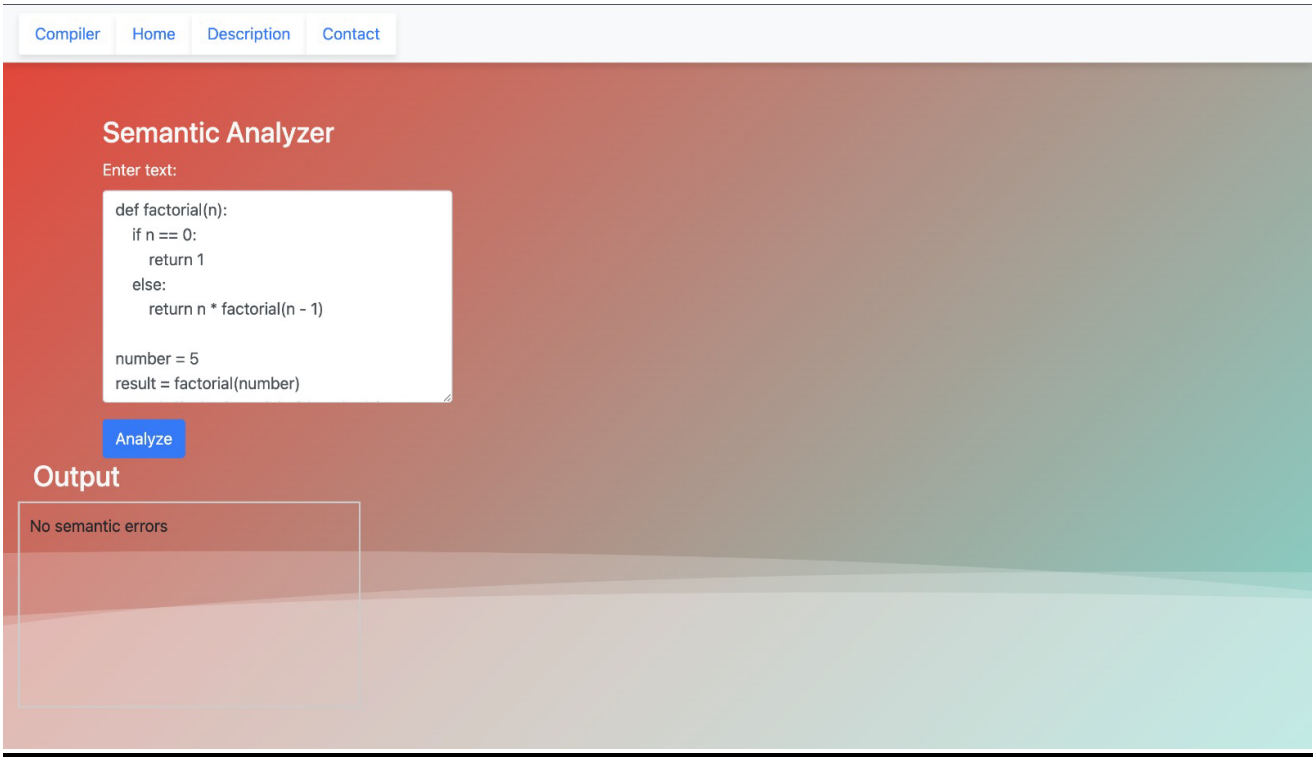
A semantic analyzer is an essential component of a compiler that checks the syntax and structure of a program's source code to ensure that it follows the rules of the programming language. Its primary function is to ensure that the program is semantically correct, which means that it will behave as intended when executed. The semantic analyzer performs a deeper analysis of the program's code than a syntax checker, which only checks for correct spelling and punctuation. The semantic analyzer checks for correct usage of language constructs, such as correct data types, variable declarations, and function calls. It also checks for compatibility between different parts of the program, such as the return type of a function and the type of variable that stores its return value.

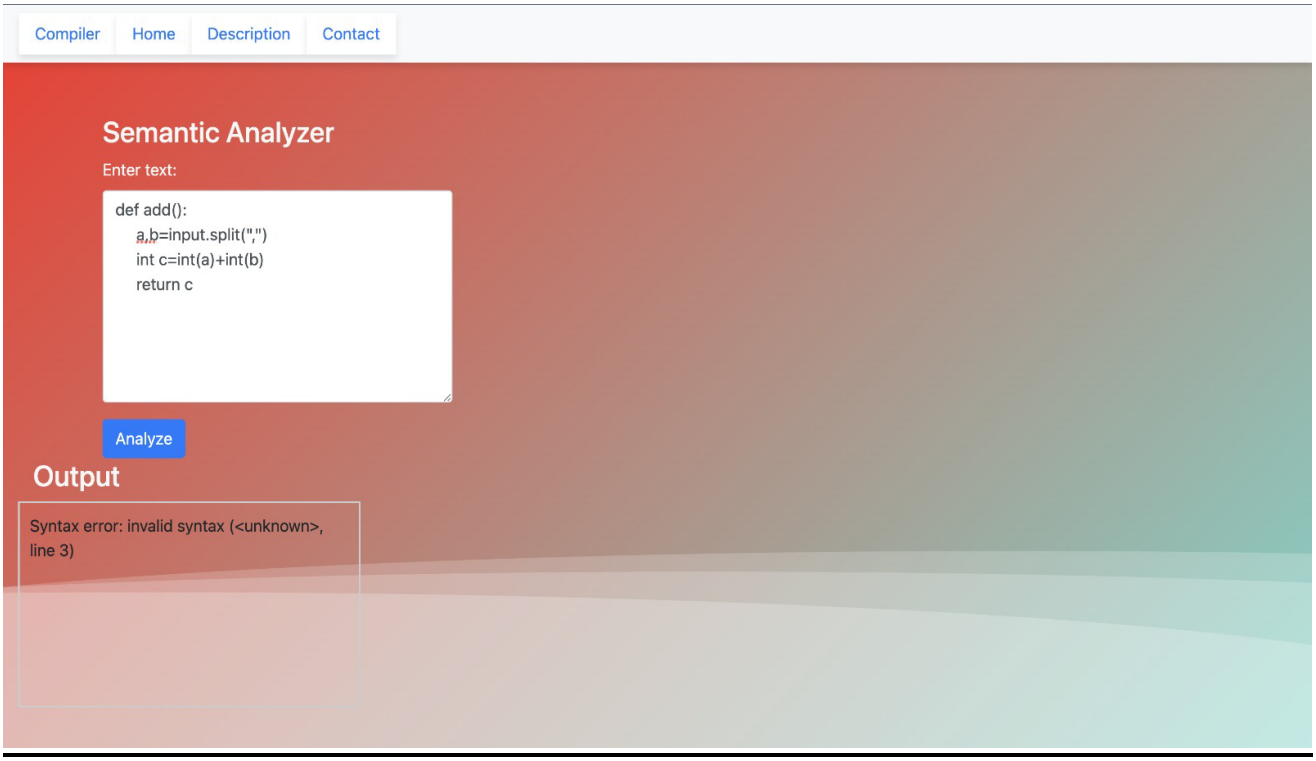
- Type checking: The semantic analyzer checks that the data types used in the program are compatible with the operations performed on them. For example, it checks that you are not trying to add a string to an integer.
- Scope checking: The semantic analyzer checks that the variables and functions used in the program are declared in the correct scope. It ensures that you are not trying to use a variable or function that is not defined or is out of scope.
- Declaration checking: The semantic analyzer checks that the variables and functions are declared before they are used in the program. It ensures that you are not trying to use a variable or function before it is defined.
- Function signature checking: The semantic analyzer checks that the arguments passed to a function match the function signature. It ensures that you are not passing the wrong number or type of arguments to a function.
- Return type checking: The semantic analyzer checks that the return type of a function matches the type of variable that stores its return value. It ensures that you are not trying to assign the return value of a function to a variable of a different type.

In short, the semantic analyzer performs a deeper analysis of the program's code to ensure that it is semantically correct and will behave as intended when executed.

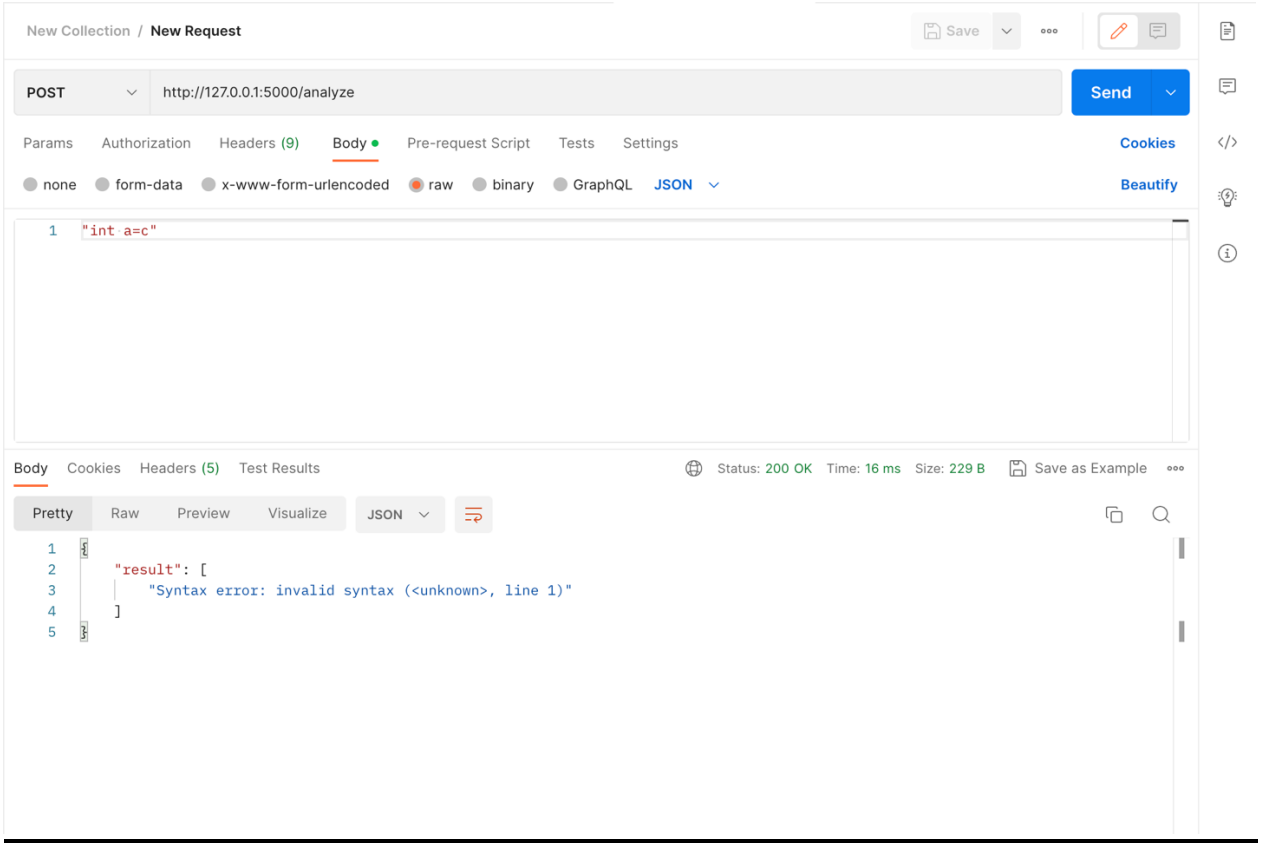
Testing the analyzer -

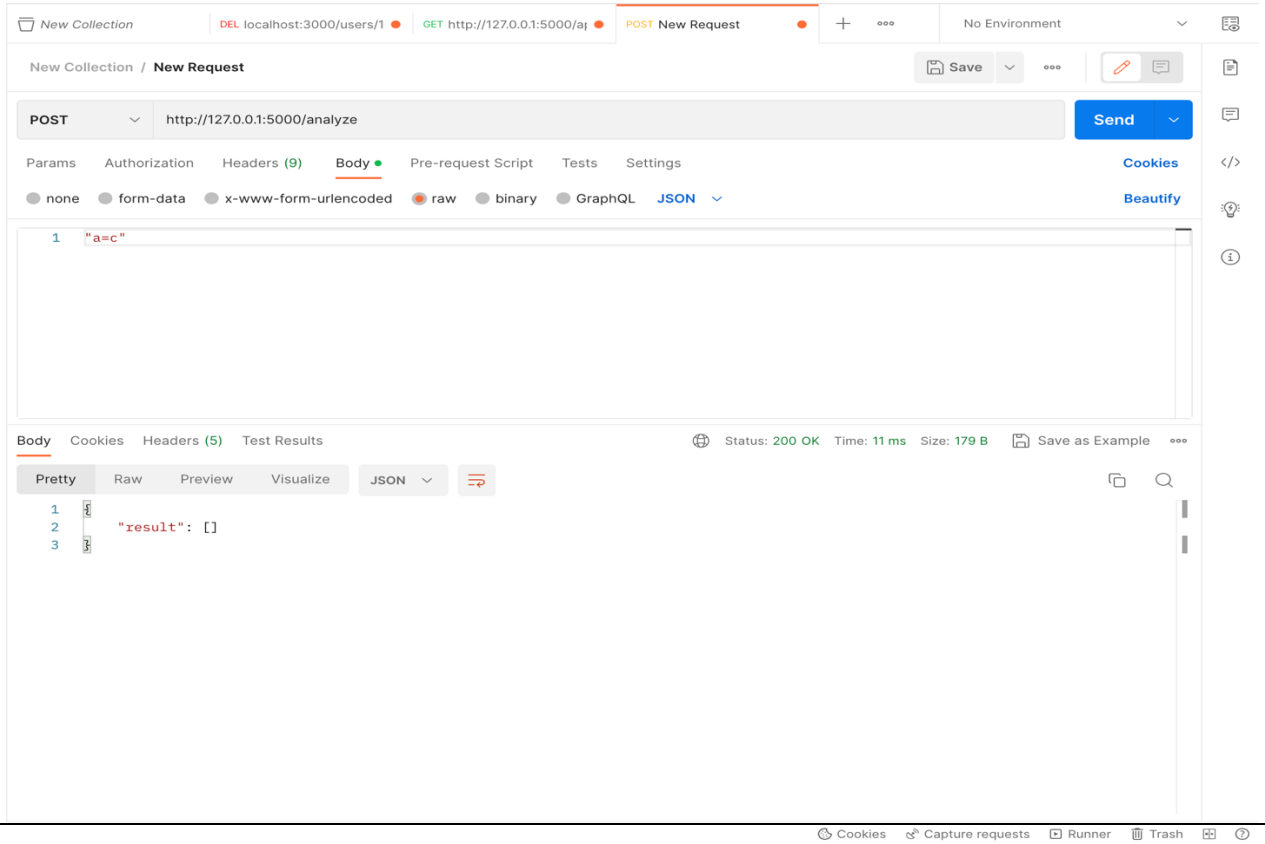



Compiler
Home
Description
Contact
Semantic Analyzer
Enter text:
def factorial(n):
if n == 0:
return 1
else:
return n * factorial(n - 1)
number = 5
result = factorial(number)
Analyze
Output
No semantic errors


Compiler
Home
Description
Contact
Semantic Analyzer
Enter text:
def add():
a,b=input.split(",")
int c=int(a)+int(b)
return c
Analyze
Output
Syntax error: invalid syntax (<unknown>, line 3)

Post request (Backend Implementation) -





5.2 RESULT

The results of a semantic analyzer can vary depending on the specific implementation and use case. However, some common results of a semantic analyzer could include:

1. **Improved code quality:** A semantic analyzer can help developers identify and fix semantic errors in their code, improving its overall quality and reducing the likelihood of bugs and errors.

2. **Faster development time:** By catching errors early in the development process, a semantic analyzer can save time and effort that would otherwise be spent debugging code later.

3. **Better performance:** A semantic analyzer can help optimize code by identifying inefficient or redundant code and suggesting improvements.

4. **Enhanced security:** A semantic analyzer can detect potential security vulnerabilities in code, such as buffer overflows or SQL injection attacks, and suggest ways to mitigate these risks.

5. **More accurate program understanding:** A semantic analyzer can help developers understand the meaning and context of programming code more accurately, leading to better design decisions and more effective program execution.

6. **Greater consistency and adherence to standards:** By enforcing rules and constraints of the programming language, a semantic analyzer can help ensure greater consistency and adherence to standards across an organization's codebase.

CHAPTER 7

7.1 CONCLUSION:-

In conclusion, the semantic analyzer is a crucial component in the field of compiler design. It plays a vital role in ensuring the correctness, consistency, and adherence to the semantic rules of a programming language. By performing various analyses and checks on the source code, the semantic analyzer detects and reports semantic errors that cannot be captured during the earlier phases of lexical and syntactic analysis.

The semantic analyzer performs tasks such as type checking, scope analysis, symbol table management, function and procedure validation, array and pointer checks, and enforcement of language-specific semantic rules. It detects and reports errors related to type mismatches, undeclared variables, incompatible assignments, invalid function calls, and other semantic violations. It provides informative error messages that help programmers identify and resolve these issues effectively.

Additionally, the semantic analyzer contributes to optimization opportunities by analyzing the semantic structure of the code. It identifies potential optimizations such as constant expressions, dead code, and unreachable code, which can be leveraged in subsequent compiler phases to improve program efficiency.

Overall, the semantic analyzer ensures the correctness, reliability, and efficiency of the compiled program. It bridges the gap between syntactic analysis and code generation, playing a vital role in the compilation process. Its applications encompass error detection and reporting, type checking, scope analysis, symbol management, semantic rule enforcement, and optimization opportunities, making it an indispensable component in compiler design.

REFERENCES

1. https://iq.opengenus.org/semantic-analysis-in-compiler-design/

2. http://www.icet.ac.in/Uploads/Downloads/M4.pdf

3. https://www.tutorialspoint.com/compiler_design/compiler_design_semantic_analysis.htm#:~:text=Semantics,derives%20any%20meaning%20or%20not.