**SIMATS SCHOOL OF ENGINEERING**

**SAVEETHA INSTITUTE OF MEDICAL AND TECHNICAL SCIENCES**

**CHENNAI-602105**

**Building Compiler with Code Auto-Completion and Suggestions**

**A CAPSTONE PROJECT REPORT**

*Submitted in the partial fulfillment for the award of the degree of*

**BACHELOR OF ENGINEERING**

**IN**

**COMPUTER SCIENCE AND ENGINEERING**

**Submitted by**

**HARI SRINIVAS T (192111083)**

**CHARAN TEJA M (192111606)**

**LAKSHAN J (192110484)**

**Under the Supervision of**

**Dr. LATHA**

**FEBRUARY 2024**

**DECLARATION**

We, **Hari Srinivas T, Charan Teja M , Lakshan J,** students of **‘Bachelor of Engineering in Computer Science and Engineering**, Department of Computer Science and Engineering, Saveetha Institute of Medical and Technical Sciences, Saveetha University, Chennai, hereby declare that the work presented in this Capstone Project Work entitled **Building Compiler with code auto-completion and suggestions** is the outcome of our own bonafide work and is correct to the best of our knowledge and this work has been undertaken taking care of Engineering Ethics.

(Hari Srinivas T 192111083)

(Charan Teja M 192111606)

(Lakshan J 192110484)

Date:

Place:

**CERTIFICATE**

This is to certify that the project entitled **“Building Compiler with code auto-completion and suggestions”** submitted by **Hari Srinivas T, Charan Teja M, Lakshan J** has been carried out under our supervision. The project has been submitted as per the requirements in the current semester of B.E Computer Science.

Faculty-in-charge

Dr. Latha

**Table of Contents**

|  |  |
| --- | --- |
| **S.NO** | **TOPICS** |
| 1 | **Abstract** |
| 2 | **Introduction** |
| 3 | **Problem Statement** |
| 4 | **Proposed Design**  1. Requirement Gathering and Analysis  2. Tool selection criteria  3. Scanning and Testing Methodologies |
| 5. | **Functionality**  1.Code Completion and Error Correction.  2. Tool Inventory and Management  3. Security and Compliance Control |
| 6 | **Conclusion** |

**ABSTRACT:**

This project focuses on creating a compiler with advanced code auto-completion and suggestion capabilities. By integrating these features into the compiler, developers can benefit from real-time assistance while writing code, leading to increased productivity and reduced debugging time. The system will utilize lexical and syntactical analysis techniques to understand the code context and provide relevant suggestions. Additionally, machine learning algorithms will be employed to enhance the accuracy and effectiveness of the suggestions over time.

The compiler's auto-completion feature will suggest code snippets, function signatures, variable names, and other relevant elements based on the current context. It will also provide suggestions for potential errors or improvements in the code, helping developers write cleaner and more efficient code. By combining these capabilities, the compiler will act as an intelligent assistant, guiding developers through the coding process and improving the overall development experience.

Overall, this project aims to develop a compiler that not only translates code into executable programs but also actively assists developers in writing high-quality code. By integrating advanced code auto-completion and suggestion features, the compiler will empower developers to write code more efficiently and effectively, ultimately leading to improved software development outcomes.

**Introduction:**

In the realm of software development, efficient coding practices are paramount to ensuring the smooth and timely creation of robust applications. One critical aspect of this efficiency is the ability to quickly and accurately write code without getting bogged down in syntax errors or repetitive typing. This project seeks to address this challenge by developing a compiler with advanced code auto-completion and suggestion capabilities, aimed at streamlining the coding process and enhancing developer productivity.

By integrating intelligent code auto-completion and suggestion features directly into the compiler, developers can benefit from real-time assistance while writing code. This assistance can range from suggesting relevant code snippets and variable names to identifying potential errors and offering solutions. Leveraging lexical and syntactical analysis techniques, the compiler will be able to understand the context of the code being written and provide tailored suggestions, making it easier for developers to write clean, error-free code.

Furthermore, the compiler will employ machine learning algorithms to continually improve the relevance and accuracy of its suggestions over time. This adaptive approach ensures that the compiler evolves alongside developers' coding practices, becoming more effective at providing timely and useful suggestions. Ultimately, the goal of this project is to create a developer-friendly compiler that not only translates code into executable programs but also acts as an intelligent assistant, enhancing the overall coding experience.

**Problem Statement:**

Developers often struggle with writing code efficiently due to manual errors and the time-consuming nature of searching for correct syntax. Existing code editors provide limited auto-completion features that lack context awareness. To address these issues, this project aims to create a compiler with advanced code auto-completion and suggestion capabilities. By analyzing code in real-time and offering context-aware suggestions, the compiler will help developers write code faster and with fewer errors, ultimately improving productivity and code quality.

**Proposed Design:**

**Requirements Gathering and Analysis:** The compiler should offer real-time code analysis to understand the context and structure of the code. It should provide context-aware auto-completion suggestions for code snippets, function signatures, and variable names. Additionally, the compiler should detect potential errors in the code and offer relevant solutions. Integration of machine learning algorithms should be employed to enhance the relevance and accuracy of suggestions over time. Efficient performance is crucial, ensuring that suggestions are provided quickly and seamlessly to improve the overall coding experience.

**Tool Selection Criteria:** Tool selection criteria include compatibility with the programming language and development environment, a comprehensive feature set encompassing code analysis, auto-completion, and error detection, and user-friendly aspects like interface and workflow integration.

**Scanning and Testing Methodology:** Scanning and testing methodologies for the compiler include static analysis to identify vulnerabilities without executing the code, dynamic analysis for runtime testing, and fuzz testing to assess robustness against unexpected inputs.

**Functionality:**

**Code Completion and Error Correction:**

● Once the compiler has analyzed the code, it can provide suggestions for completing the current line of code based on context. For example, if the developer starts typing a function name, the compiler could suggest completing it with the function's parameters or provide a list of available functions.

● In addition to auto-completion, the compiler can suggest corrections for common errors in the code. For example, if the developer misspells a variable name or forgets to import a required module, the compiler can suggest fixes to resolve these issues.

**Tool Inventory and Management:**

● Maintain a comprehensive inventory of tools used in the compiler development process, including compilers, libraries, and development environments.

● Use version control systems like Git to manage changes to tools, ensuring that the most up-to-date and compatible versions are used throughout the development lifecycle.

**Security and Compliance Controls:**

● Conduct regular security testing, including penetration testing and vulnerability scanning, to identify and address security weaknesses in the compiler.

**Conclusion:**

In conclusion, building a compiler with advanced code auto-completion and suggestion capabilities requires a thoughtful approach to tool selection, security, and access control. Selecting tools that integrate seamlessly with the development environment and provide robust scanning and testing methodologies is crucial. Additionally, implementing strong user authentication and role-based access control ensures that only authorized users can access and modify the compiler code.

Maintaining a detailed inventory of tools and versions used in the development process is essential for managing the development environment effectively. Version control mechanisms, such as Git, help track changes to tools and ensure that the most up-to-date and compatible versions are used. Security and compliance controls, including regular code reviews and security testing, are vital for identifying and mitigating vulnerabilities in the compiler code, ensuring compliance with security standards and regulations.

**Code:-**

from ply import lex, yacc

# Define tokens

tokens = (

    'ID',

    'NUMBER',

    'PLUS',

    'MINUS',

    'TIMES',

    'DIVIDE',

    'LPAREN',

    'RPAREN',

)

# Define token regex

t\_PLUS = r'\+'

t\_MINUS = r'-'

t\_TIMES = r'\\*'

t\_DIVIDE = r'/'

t\_LPAREN = r'\('

t\_RPAREN = r'\)'

# Define a rule to handle identifiers

def t\_ID(t):

    r'[a-zA-Z\_][a-zA-Z\_0-9]\*'

    return t

# Define a rule to handle numbers

def t\_NUMBER(t):

    r'\d+'

    t.value = int(t.value)

    return t

# Ignore whitespace

t\_ignore = ' \t'

# Error handling

def t\_error(t):

    print(f"Illegal character '{t.value[0]}'")

    t.lexer.skip(1)

# Build the lexer

lexer = lex.lex()

# Define the grammar

def p\_expression\_binop(p):

    '''

    expression : expression PLUS expression

               | expression MINUS expression

               | expression TIMES expression

               | expression DIVIDE expression

    '''

    # Do something with the parsed expression

def p\_expression\_group(p):

    'expression : LPAREN expression RPAREN'

    p[0] = p[2]

def p\_expression\_number(p):

    'expression : NUMBER'

    p[0] = p[1]

def p\_expression\_id(p):

    'expression : ID'

    p[0] = p[1]

def p\_error(p):

    print("Syntax error in input!")

# Build the parser

parser = yacc.yacc()

# Main function for input and parsing

def main():

    while True:

        try:

            s = input('>>> ')

        except EOFError:

            break

        if not s:

            continue

        result = parser.parse(s)

        print(result)

if \_\_name\_\_ == '\_\_main\_\_':

    main()