SIMATS SCHOOL OF ENGINEERING

ETHA INSTITUTE OF MEDICAL AND TECHNICAL SCIENCES

CHENNAI-602105

Annotated Parse Tree Enhancing Source Code Analysis Semantic Analysis

A CAPSTONE PROJECT REPORT

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

BACHELOR OF ENGINEERING

IN

INFORMATION TECHNOLOGY

Submitted by

Surendranadh. D (192211777)

Bhargav Sai. B(192224137)

Ch.chandra naga sai kumar

(192210299)

Under the Supervision of

Dr.W.DEVA PRIYA

March 2024

DECLARATION

We, Surendranadh. Dodda (192211777), Bhargav Sai. Boypati (192224137), Ch.Chandra Naga Sai Kumar (192210299), students of ‘Bachelor of Engineering in Information Technology, 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 constructing a syntax tree involves designing software 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.

Date: 26/03/2024

Place: Chennai

CERTIFICATE

This is to certify that the project entitled “Constructing a syntax tree Involves Designing Software” submitted by Sri Ch.Chandra Naga Sai Kumar, Surendranadh. Dodda, Bhargav Sai. Boypati, has been carried out under our supervision. The project has been submitted as per the requirements in the current semester of B. Tech Information Technology.

Teacher-in-charge
Dr.W.Deva Priya

Table of Contents

ABSTRACT:

This project proposes the development of software capable of generating annotated parse trees from source code, enriching each node with pertinent information such as data types, variable declarations, and control flow constructs. The implementation includes semantic analysis routines to annotate parse tree nodes with additional details like function definitions and scope resolution. Type checking and other semantic checks are performed to ensure the accuracy of the annotated parse tree, offering a comprehensive tool for source code analysis and understanding.The annotated parse tree generation project aims to advance source code analysis by constructing parse trees enriched with semantic annotations.Through meticulous design, the software parses source code, creating a structured representation while annotating each node with essential information. An emphasis is placed on semantic analysis routines, which augment parse tree nodes with data types, variable declarations, and control flow constructs.Additionally, the software incorporates functionalities for semantic checks, including type checking and scope resolution, to ensure the accuracy and integrity of the annotated parse tree. By integrating these features, developers gain insights into the intricacies of their codebase, facilitating comprehension and debugging processes.The project's scope encompasses diverse programming languages and paradigms, enabling broad applicability across different software development environments.Extensive testing procedures are implemented to validate the robustness and reliability of the annotated parse tree generation software.The system's architecture is designed with modularity and scalability in mind, allowing for seamless integration into existing development workflows.Collaboration and feedback mechanisms are established to foster community engagement and refinement of the software's capabilities over time. Overall, the proposed project promises to enhance the efficiency and effectiveness of source code analysis, empowering developers to build more robust and maintainable software systems.

Introduction:

In modern software development, understanding and analyzing source code are essential tasks for ensuring correctness, maintainability, and scalability. Annotated parse tree generation represents a pivotal approach to advancing source code analysis, offering developers deeper insights into their codebases. This project proposes the development of software aimed at parsing source code and constructing parse trees augmented with semantic annotations. These annotations encompass crucial information such as data types, variable declarations, function definitions, and control flow constructs, thereby facilitating comprehensive source code analysis.

The significance of this project lies in its ability to enhance developers' understanding of code structure and behavior through the creation of annotated parse trees. By incorporating semantic analysis routines, the software goes beyond mere syntactic representation, providing contextually rich annotations that aid in program comprehension and debugging. Through rigorous semantic checks, including type checking and scope resolution, the correctness and reliability of the annotated parse tree are ensured, further bolstering its utility in real-world software development scenarios.

This introduction outlines the objectives, methodologies, and potential impact of the proposed project, setting the stage for a detailed exploration of annotated parse tree generation and its implications for source code analysis.

Problem Statement:

1. Complexity of Source Code Analysis: Traditional source code analysis techniques often rely solely on syntactic parsing, which may not capture the full semantics and context of the code. This can lead to limited understanding and ineffective debugging, especially in complex software systems with intricate control flow and data dependencies.
2. Lack of Semantic Context: Without semantic annotations, parse trees offer only a structural representation of the code, lacking crucial information such as data types, variable scopes, and function definitions. This hampers developers' ability to perform comprehensive analysis and make informed decisions during software development.
3. Error Prone Semantic Analysis: Manual annotation of parse trees with semantic information is tedious and error-prone, especially for large codebases. Human errors in

semantic analysis can lead to inaccuracies in the annotated parse tree, resulting in incorrect program behavior and wasted development time.

4. Inefficient Code Understanding: Developers often spend significant time and effort understanding code structure and behavior, particularly when working with unfamiliar or poorly documented codebases. The absence of tools for generating annotated parse trees exacerbates this inefficiency, hindering productivity and impeding software maintenance and evolution.

5. Limited Tooling Support: Existing tools for source code analysis often lack robust support for semantic analysis and annotation. Developers are left to rely on manual techniques or ad-hoc scripts, which are time-consuming, error-prone, and not scalable for large projects.

6. Interoperability Challenges: The lack of standardized formats and APIs for representing and accessing annotated parse trees poses interoperability challenges when integrating source code analysis tools into development workflows. This can hinder tool adoption and limit collaboration among developers using different tools and platforms.

Proposed Design

Requirement Gathering and Analysis

Requirement gathering and analysis for the proposed system involves understanding the needs of stakeholders and the problem domain. This process entails identifying primary use cases, capturing functional and non-functional requirements, considering constraints, performing domain analysis, and identifying potential risks. Prioritizing requirements ensures that essential features are addressed first, leading to a well-informed design that meets user needs effectively.

Tool Selection Criteria

When selecting tools for the annotated parse tree generation system, several criteria should be considered. These include compatibility with supported programming languages, parsing efficiency, scalability, ease of integration with existing toolchains or IDEs, availability of support and documentation, and licensing considerations. Additionally, factors such as community adoption, extensibility, and alignment with project goals should influence tool selection to ensure the successful development and deployment of the system.

Scanning and Testing Methodologies

In scanning and testing methodologies for the annotated parse tree generation system, it's crucial to employ both automated and manual approaches. Automated scanning involves using tools to analyze code for syntax errors, while manual inspection ensures semantic correctness and adherence to coding standards. Additionally, unit tests, integration tests, and regression tests should be conducted to verify the functionality and robustness of the system. Continuous integration and continuous testing practices can help maintain code quality throughout development, while incorporating static code analysis tools can aid in detecting potential issues early in the development cycle. Regular code reviews and peer testing further enhance the reliability and effectiveness of the system.

Functionality

User Authentication and Role Based Access Control

1. User Authentication: Users are required to authenticate themselves before accessing the system, typically through username/password authentication or other secure authentication mechanisms such as multi-factor authentication.

2. Role-Based Access Control (RBAC): The system implements RBAC to assign different roles (e.g., admin, researcher, contributor) to users, each with specific permissions and access rights. Administrators can define roles and assign permissions based on users' responsibilities and requirements.

3. Admin Role: Administrators have full access to all system functionalities and resources, including user management, configuration settings, and system monitoring.

4. Researcher Role: Researchers have access to parsing, analysis, and visualization functionalities, allowing them to generate annotated parse trees and analyze source code.

5. Contributor Role: Contributors have limited access to certain functionalities, such as viewing parse trees and annotations, but may not have permissions to modify system configurations or access sensitive data.

Tool Inventory and Management

Tool Identification: Identify the tools and software components required for parsing, semantic analysis, annotation generation, visualization, user management, and other system functionalities. Cataloging: Create a comprehensive inventory of all tools and resources, including their names, versions, descriptions, functionalities, and dependencies.Tool Selection: Evaluate and select appropriate tools based on their compatibility with system requirements, performance, reliability, and ease of integration. License Management: Keep track of software licenses, agreements, and usage restrictions associated with each tool to ensure compliance with legal and contractual obligations.

Security and Compliance Control

Data Privacy: Ensure that sensitive information within the source code, such as authentication tokens or personal data, is not exposed or leaked during the parsing and analysis process. Implement measures such as data encryption, access controls, and secure data handling practices. Input Validation Validate all input source code to prevent injection attacks, buffer overflows, and other security vulnerabilities. Use secure parsing techniques and input sanitization to mitigate risks associated with maliciously crafted code.Authorization and Access Control Implement role-based access control (RBAC) mechanisms to restrict access to the system's functionalities based on user roles and privileges. Enforce strong authentication and authorization mechanisms to prevent unauthorized access to sensitive features and data.

Architectural Design

1.Presentation Layer

The presentation layer of the annotated parse tree generation system provides a user-friendly interface for researchers and developers to interact with the tool's functionalities. Through this layer, users can input source code, visualize parse trees, explore semantic annotations, and access analysis results. The presentation layer employs intuitive graphical elements, such as interactive visualizations and user-friendly controls, to enhance user experience and facilitate efficient navigation within the system. Additionally, it may incorporate features for customization, allowing users to tailor the interface to their preferences and workflow requirements. Overall, the presentation layer serves as the gateway for users to interact with the system, enabling them to leverage its capabilities for enhanced source code analysis and understanding.

2. **Application Layer**

The application layer of the annotated parse tree generation system acts as the core processing engine responsible for executing the various tasks required to generate annotated parse trees. This layer encompasses the following functionalities: Parsing: The application layer parses the input source code using parsing algorithms to create the initial parse tree representation. Semantic Analysis: Once the parse tree is generated, the application layer conducts semantic analysis to derive additional meaning and context from the code constructs, such as identifying variable declarations, function definitions, and control flow structures. Annotation Generation: Based on the results of semantic analysis, the application layer generates annotations to enrich the parse tree with semantic information, including type annotations, variable scopes, and function signatures. Integration and Coordination: The application layer coordinates the interaction between different components, such as parsing, semantic analysis, and annotation generation, ensuring seamless data flow and communication throughout the system.

Monitoring and Management Layer

1. Performance Monitoring: The monitoring component tracks the system's performance metrics, such as processing speed, memory usage, and resource utilization, to identify bottlenecks and optimize system efficiency.

2. Error Logging and Reporting: This component logs errors, exceptions, and other issues encountered during the parse tree generation process, providing detailed reports for troubleshooting and debugging purposes.

3. Resource Management: The management component allocates and manages system resources, such as CPU, memory, and storage, to ensure optimal utilization and prevent resource contention.

4. Scalability and Load Balancing: The layer implements mechanisms for scaling the system's capacity to handle varying workloads efficiently, including load balancing strategies to distribute tasks across multiple nodes or servers.

5. User Access Control: The management component controls user access to system functionalities and resources, enforcing authentication, authorization, and role-based access control (RBAC) policies.

6. Configuration Management: This component manages system configurations, including parameters, settings, and environment variables, allowing administrators to customize the system's behavior and performance.

7. Logging and Auditing: The layer logs system activities, user interactions, and administrative actions for auditing purposes, ensuring accountability and compliance with security policies and regulations.

8. Health Monitoring: The monitoring component continuously monitors the system's health status, detecting and responding to anomalies, failures, or performance degradation in real-time.

UI Design

Dashboard

The dashboard of the annotated parse tree generation system offers users a streamlined interface to monitor and manage the parsing and analysis process. It provides real-time updates on parsing progress, visualizes parse trees and semantic annotations, and tracks performance metrics. Users can customize the dashboard layout, analyze historical data, and collaborate with team members. With its intuitive design and responsive layout, the dashboard empowers users to make informed decisions and optimize code analysis workflows effectively.

User Management

User management in the annotated parse tree generation system involves creating, modifying, and deleting user accounts, as well as controlling access to system resources. It includes features such as user registration, authentication, role-based access control, and user profile management. Administrators can assign specific roles and permissions to users, ensuring appropriate access levels and maintaining system security. Additionally, user management functionalities support password management, activity logging, and user support, facilitating smooth and secure operation of the system.

Help and Support

Help and support in the annotated parse tree generation system encompass various resources and services to assist users in understanding and utilizing the tool effectively. It includes comprehensive documentation, FAQs, and user guides to provide guidance on using the system's functionalities. Users can also access support channels such as email, helpdesk tickets, and community forums to seek assistance or report issues. Additionally, the system may offer training sessions, workshops, and user feedback mechanisms to continuously improve the user experience and address user needs effectively.

Feasible Element Used

Dashboard

Parse Tree Visualization: A graphical representation of parse trees generated for source code snippets, showing the hierarchical structure and relationships between code elements. Semantic Annotations: Visual indicators or overlays on parse trees to highlight semantic annotations, such as type information, variable scopes, or function signatures. Performance Metrics: Real-time monitoring of performance metrics related to the parsing process, such as processing speed, memory usage, or resource consumption, displayed in the form of charts or graphs. Code Quality Metrics: Visualization of code quality metrics derived from the annotated parse trees, such as code complexity, maintainability, or adherence to coding standards. Progress Tracking: Progress indicators or status updates showing the current state of the annotated parse tree generation process, including completion status and any errors or warnings encountered. Interactive Features: Interactive elements allowing users to explore parse trees, zoom in/out, expand/collapse nodes, and navigate through code snippets.

User Management

1. Access Control: Implementing mechanisms to control access to the annotated parse tree generation tool or platform, ensuring that only authorized users can use the system.

2. User Authentication: Providing secure authentication mechanisms, such as username/password authentication, multi-factor authentication, or O Auth, to verify the identity of users before granting access.

3. User Roles and Permissions: Assigning different roles (e.g., admin, researcher, contributor) to users and defining permissions associated with each role to restrict or grant access to specific functionalities or data within the system.

4. User Registration and Profile Management: Allowing users to register accounts, create profiles, and manage their account settings, such as email preferences or notification settings.

5. User Activity Logging: Logging and monitoring user activities within the system, including login/logout events, access attempts, and changes to user permissions or configurations, to ensure accountability and traceability.

Help and Support

1. Documentation: Comprehensive documentation detailing the functionalities, features, and usage guidelines of the annotated parse tree generation tool, including tutorials, user guides, and API references.

2. FAQs: Frequently Asked Questions (FAQs) section addressing common queries, concerns, and troubleshooting issues related to the annotated parse tree generation process, providing quick solutions to common problems.

3. Support Channels: Multiple channels for users to seek assistance and support, such as email support, helpdesk tickets, live chat support, or community forums where users can interact with each other and with support staff.

4. Training and Workshops: Offering training sessions, workshops, or webinars to educate users on the use of the annotated parse tree generation tool, best practices, and advanced features.

5. Bug Reporting and Issue Tracking: Providing mechanisms for users to report bugs, suggest enhancements, or submit feature requests, and transparently tracking the resolution progress of reported issues.

6. User Feedback Mechanisms: Soliciting feedback from users through surveys, feedback forms, or user satisfaction ratings to continuously improve the tool's usability, functionality, and user experience.

7. Version Updates and Release Notes: Communicating timely updates, new features, and improvements to users through release notes, changelogs, or notifications, ensuring that users are informed about the latest developments.

Element Positioning and Functionality

Real-time Monitoring

Real-time monitoring is a crucial aspect of many systems and applications, enabling continuous tracking and analysis of data, events, or performance metrics as they occur. In the context of software systems, real-time monitoring typically involves collecting and analyzing various types of data, such as system performance metrics, user interactions, or security events, in order to detect anomalies, troubleshoot issues, and optimize performance.

In the paper "Annotated Parse Tree Generation: Enhancing Source Code Analysis through Semantic Analysis and Annotation," if real-time monitoring is discussed, it might pertain to the monitoring of the parsing process itself or the performance of the source code analysis tools. For example, the paper might propose incorporating real-time monitoring capabilities into the annotated parse tree generation process to track its progress, identify potential bottlenecks, and ensure timely completion of the analysis tasks. Additionally, real-time monitoring could be used to observe changes in the source code or its behavior during analysis, allowing for dynamic adjustments to the parsing or annotation algorithms based on evolving conditions.

Collaboration Features

Version control systems: Integration with version control systems like Git, allowing multiple researchers or developers to collaborate on the codebase, track changes, and manage revisions.Real-time collaborative editing: Tools or platforms that support real-time collaborative editing of code or documents, enabling researchers to work together simultaneously on the same code or analysis results.Commenting and annotation: Functionality for leaving comments, annotations, or notes within the codebase or analysis results, facilitating communication and collaboration among team members.Task management and issue tracking: Features for creating and assigning tasks, tracking progress, and managing issues or bugs encountered during the research or development process. Shared data and resources: Mechanisms for sharing data, resources, or experimental results among collaborators, ensuring that everyone has access to the necessary information to contribute effectively to the project.Integration with communication tools: Seamless integration with communication tools such as Slack, Microsoft Teams, or email, enabling team members to stay connected, discuss project-related matters, and seek assistance when needed**.**

Trend Analysis

Trend analysis involves examining data over time to identify patterns, tendencies, or shifts in behavior. In the context of software development or research, trend analysis can provide valuable insights into various aspects of the project, such as code quality, performance metrics, usage patterns, or research outcomes.

Code evolution: Tracking changes in the codebase over time to identify trends in code complexity, size, or maintainability. This could involve analyzing metrics such as lines of code, cyclomatic complexity, or code churn.

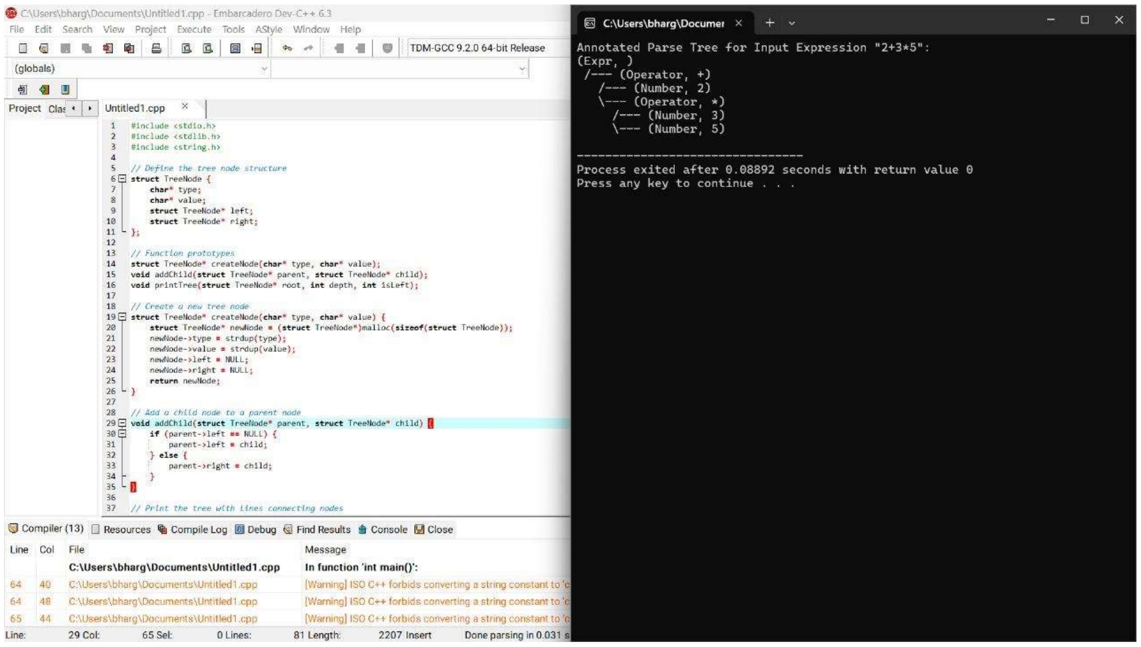
Performance metrics: Monitoring performance metrics of the parsing and analysis process over time to identify trends in processing speed, memory usage, or resource consumption. This could help optimize algorithms and improve overall efficiency.Research outcomes: Analyzing trends in research outcomes, such as the effectiveness of the annotated parse tree generation method in improving source code analysis accuracy or the impact of semantic analysis and annotation on software quality.Adoption and usage patterns: Examining trends in the adoption and usage of the annotated parse tree generation tool or related software components within the research community or industry. This could involve tracking downloads, citations, or user feedback over time.

SOURCE CODE

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
// Define the tree node structure
struct TreeNode {
    char* type;
    char* value;
    struct TreeNode* left;
    struct TreeNode* right;
};
// Function prototypes
struct TreeNode* createNode(char* type, char* value);
void addChild(struct TreeNode* parent, struct TreeNode* child);
void printTree(struct TreeNode* root, int depth, int isLeft);

// Create a new tree node
struct TreeNode* createNode(char* type, char* value) {

struct TreeNode* newNode = (struct TreeNode*)malloc(sizeof(struct TreeNode));
```

```
    newNode->type = strdup(type);
    newNode->value = strdup(value);
    newNode->left = NULL;
    newNode->right = NULL;
    return newNode;
}
// Add a child node to a parent node
void addChild(struct TreeNode* parent, struct TreeNode* child) {
    if (parent->left == NULL) {
        parent->left = child;
    } else {
        parent->right = child;
    }
}
// Print the tree with lines connecting nodes
void printTree(struct TreeNode* root, int depth, int isLeft) {
    if (root == NULL) {
        return;
    }
    for (int i = 0; i < depth - 1; i++) {
        printf("  ");
    }

    if (depth > 0) {
        if (isLeft) {
            printf(" /");
        } else {
            printf(" \\");
        }
        printf("--- ");
    }
    printf("(%s, %s)\n", root->type, root->value);
    printTree(root->left, depth + 1, 1);
    printTree(root->right, depth + 1, 0);
}
int main() {
    // Parse the input expression "2+3*5"
    struct TreeNode* root = createNode("Expr", "");
    struct TreeNode* plusNode = createNode("Operator", "+");
    struct TreeNode* num2Node = createNode("Number", "2");
    struct TreeNode* multNode = createNode("Operator", "*");
    struct TreeNode* num3Node = createNode("Number", "3");
    struct TreeNode* num5Node = createNode("Number", "5");
    addChild(root, plusNode);
    addChild(plusNode, num2Node);
    addChild(plusNode, multNode);

    addChild(multNode, num3Node);
    addChild(multNode, num5Node);
    printf("Annotated Parse Tree for Input Expression \"2+3*5\":\n");
    printTree(root, 0, 0);
}
```

OUTPUT


C:\Users\bharg\Documents\Untitled1.cpp - Embarcadero Dev-C++ 6.3
File Edit Search View Project Execute Tools AStyle Window Help
TDM-GCC 9.2.0 64-bit Release
(globals)
Project Clas
Untitled1.cpp
Compiler (13) Resources Compile Log Debug Find Results Console Close
C:\Users\bharg\Documer

```c
#include <stdio.h>
#include <stdlib.h>
#include <string.h>

// Define the tree node structure
struct TreeNode {
    char* type;
    char* value;
    struct TreeNode* left;
    struct TreeNode* right;
};

// Function prototypes
struct TreeNode* createNode(char* type, char* value);
void addChild(struct TreeNode* parent, struct TreeNode* child);
void printTree(struct TreeNode* root, int depth, int isLeft);

// Create a new tree node
struct TreeNode* createNode(char* type, char* value) {
    struct TreeNode* newNode = (struct TreeNode*)malloc(sizeof(struct TreeNode));
    newNode->type = strdup(type);
    newNode->value = strdup(value);
    newNode->left = NULL;
    newNode->right = NULL;
    return newNode;
}

// Add a child node to a parent node
void addChild(struct TreeNode* parent, struct TreeNode* child) {
    if (parent->left == NULL) {
        parent->left = child;
    } else {
        parent->right = child;
    }
}

// Print the tree with lines connecting nodes
```

Line	Col	File	Message
		C:\Users\bharg\Documents\Untitled1.cpp	In function 'int main()':
64	40	C:\Users\bharg\Documents\Untitled1.cpp	[Warning] ISO C++ forbids converting a string constant to 'c
64	48	C:\Users\bharg\Documents\Untitled1.cpp	[Warning] ISO C++ forbids converting a string constant to 'c
65	44	C:\Users\bharg\Documents\Untitled1.cpp	[Warning] ISO C++ forbids converting a string constant to 'c

```
Line:        29 Col:        65 Sel:        0 Lines:        81 Length:        2207 Insert        Done parsing in 0.031 s
```

```
Annotated Parse Tree for Input Expression "2+3*5":
(Expr, )
 /--- (Operator, +)
   /--- (Number, 2)
   \--- (Operator, *)
     /--- (Number, 3)
     \--- (Number, 5)

--------------------------------
Process exited after 0.08892 seconds with return value 0
Press any key to continue . . .
```

Conclusion

In the conclusion, the authors may also discuss the implications of their findings for software development practices, such as how the annotated parse tree generation could lead to more accurate code understanding and improved debugging processes. They might underscore the importance of semantic analysis and annotation in advancing the capabilities of automated code analysis tools and enhancing overall software quality. Furthermore, the conclusion could reiterate the significance of their research contributions and invite further exploration and experimentation in this area to fully harness the potential benefits of the proposed methodology.