SysY2022E 扩展语言编辑器

SysY2022E Language Support

系统设计说明书

成员信息： 李晓坤　信息安全　信安 211　U202141863

王若凡　信息安全　信安 211　U202141852

王宠爱　信息安全　信安 211　U202141853

成豪　信息安全　信安 211　U202141880

指导教师： 崔 晓 龙

目录

1. 引言 4
1.1. 编写目的 4
1.2. 范围 4
1.3. 术语定义 5
1.4. 引用标准 5
1.5. 参考资料 6
1.6. 版本更新信息 6
2. 系统设计概述 7
2.1. 系统设计概述 7
2.1.1. 用户界面层 7
2.1.2. 编辑器核心层 7
2.1.3. 分析与修复层 7
2.1.4. 重构引擎层 7
2.1.5. 后端服务层 7
2.1.6. 数据存储层 8
2.1.7. 插件系统 8
2.2. 模块划分和分布 8
2.2.1. 用户界面层（User Interface Layer） 8
2.2.2. 编辑器核心层（Editor Core Layer） 8
2.2.3. 分析与修复层（Analysis and Fix Layer） 9
2.2.4. 重构引擎层（Refactoring Engine Layer） 9
2.2.5. 后端服务层（Backend Service Layer） 9
2.2.6. 数据存储层（Data Storage Layer） 9
2.2.7. 插件系统（Plugin System） 9
2.3. 各层及其模块之间的关系和通信 9
2.3.1. 用户界面层与编辑器核心层 9
2.3.2. 编辑器核心层与分析与修复层 10
2.3.3. 编辑器核心层与重构引擎层 10
2.3.4. 各层与后端服务层 11
2.3.5. 各层与数据存储层 11
2.3.6. 用户界面层与插件系统 12

2.4. 系统采取的技术和实现方法 12
2.4.1. LSP 技术 12
2.4.2. ANTLR4 构建工具 13
2.4.3. TypeScript 语言 13
3. 详细设计概述 14
3.1. 用户界面 14
3.1.1. IDE 集成 14
3.1.2. 语法高亮 15
3.1.3. 悬浮提示 15
3.2. 编辑器核心 16
3.2.1. 语法检查与错误提示 16
3.2.2. 静态语义检查 17
3.3. 分析与修复 18
3.3.1. 修复建议与自动修复 18
3.3.2. 程序错误检查 19
3.4. 重构引擎 20
3.5. 优化层 21
3.5.1. 辅助编辑 21
3.5.2. 查找引用 21
3.5.3. 签名帮助 22
3.5.4. 查看引用 23
4. 详细设计 24
4.1. 语言规范 24
4.1.1. 词法规范 24
4.1.2. 语法规范 24
4.2. 用户界面 25
4.2.1. IDE 集成 25
4.2.2. 语法高亮 37
4.2.3. 悬浮提示 42
4.3. 编辑器核心 50
4.3.1. 语法检查与错误提示 50
4.3.2. 静态语义检查 57
4.4. 分析与修复 62

4.4.1. 修复建议与自动修复 62
4.4.2. 程序错误检查 68
4.5. 重构引擎 75
4.6. 优化层 82
4.6.1. 辅助编辑 82
4.6.2. 查找引用 89
4.6.3. 签名帮助 94
4.6.4. 查看定义 98
5. 程序提交清单 102

1. 引言

1.1. 编写目的

系统设计说明书是一份关键的项目文档，它详细描述了系统的整体设计和具体实现细节。编写这份文档的主要目的在于为项目的开发、实施、测试和维护提供一个清晰、全面的指南。通过该文档，项目团队成员能够更好地理解系统的架构、功能和设计考虑，从而确保项目的顺利进行。

该份文档主要介绍了系统的整体架构设计，包括系统的基本构成、模块划分以及各模块之间的关系。详细设计概述部分对系统的各个模块进行了更深入的描述，包括模块的功能、输入输出、性能要求等。而详细设计则进一步展开了每个模块的内部结构、算法、数据流程以及与其他模块的接口设计。这部分内容对于开发人员来说至关重要，因为它提供了实现系统所需的具体细节。程序提交清单列出了项目开发过程中需要提交的所有程序文件、配置文件、测试数据等。它不仅确保了项目交付的完整性，还为后续的项目管理和版本控制提供了便利。

开发人员依赖其详细设计来编写和调试代码，确保功能实现与设计要求一致；测试人员则据此编写测试用例，全面验证系统的功能和性能；项目管理人员通过它了解项目进展和设计思路，以优化项目规划、资源分配和风险管理；而维护人员在系统上线后，也可利用该文档快速定位和解决问题，确保系统的平稳运行。

1.2. 范围

这份系统设计说明书覆盖的范围包括系统的整体架构设计、详细的功能模块设计、数据库设计、界面设计、安全性与性能考量，以及程序提交的相关内容等。它全面阐述了从系统的高层架构到具体实现的各个细节，为项目的开发、测试、部署和维护提供了全面的指导。

在整个项目中，这份系统设计说明书充当着“项目蓝图”和“技术指南”的双重角色。作为“项目蓝图”，它帮助项目团队成员建立对系统的整体认识，明确各模块之间的关系和依赖，从而确保开发工作的协调性和一致性。作为“技术指南”，它为开发人员提供了具体的实现细节和编码规范，为测试人员提供了测试依据和验证标准，同时也为项目管理人员和维护人员提供了必要的信息和工具，以便他们更好地进行项目管理和系统维护。

通过这份系统设计说明书，项目团队能够解决在系统开发过程中可能遇到的多种问题，

如需求不明确、设计不合理、开发不规范等。它确保了项目的顺利进行，提高了开发效率，降低了维护成本，为项目的成功实施奠定了坚实的基础。

1.3. 术语定义

表 1 术语定义表

术语	定义
SysY2022E 语言	SysY2022 是 C 语言的一个子集，最初是为“全国大学生系统能力大赛-编译器大赛”设计的迷你编程语言。SysY2022E 语言是在其基础上拓展的语言。
LSP	微软指定的 Language Server Protocol，它标准化了语言工具和代码编辑器之间的通信。
ANTLR4	ANTLR4 是一款功能强大的解析器生成器，用于读取、处理、执行和翻译结构化文本或二进制文件。

1.4. 引用标准

1）IEEE 830-1998（已被 IEEE 29148 取代）：软件需求规格说明标准

提供了编写高质量软件需求规格说明书的指南，确保需求文档的完整性和一致性。

2）IEEE 1016-2009：软件设计描述标准

规范了软件系统设计文档的内容和格式，提供了系统设计的高级视图结构。

3）ISO/IEC/IEEE 29148-2018：系统与软件工程——生命周期过程——需求工程

结合了需求工程的最佳实践，适用于需求开发和管理，提供了编写需求和系统描述文档的详细指南。

4）IEEE 1233-1998（已被 IEEE 29148 取代）：系统需求规格说明标准

涉及系统级别的需求和设计，提供了编写系统需求说明文档的结构和指南。

5）ISO/IEC 25010:2011：系统与软件工程——系统和软件质量模型

提供了软件和系统质量的标准模型，用于评估系统的非功能性需求，如性能、安全性、可用性和可维护性。

6）ISO/IEC/IEEE 15288:2015：系统与软件工程——系统生命周期过程

定义了系统工程和软件工程中的生命周期过程，为系统开发和维护提供了标准化的指南。

1.5. 参考资料

[1] IEEE 830-1998. IEEE Recommended Practice for Software Requirements Specifications. Institute of Electrical and Electronics Engineers.

[2] IEEE 1016-2009. IEEE Standard for Information Technology—Systems Design—Software Design Descriptions. Institute of Electrical and Electronics Engineers.

[3] ISO/IEC/IEEE 29148-2018. Systems and software engineering — Life cycle processes — Requirements engineering. International Organization for Standardization / Institute of Electrical and Electronics Engineers.

[4] IEEE 1233-1998. IEEE Guide for Developing System Requirements Specifications. Institute of Electrical and Electronics Engineers.

[5] ISO/IEC 25010:2011. Systems and software engineering — Systems and software Quality Requirements and Evaluation (SQuaRE) — System and software quality models. International Organization for Standardization.

[6] ISO/IEC/IEEE 15288:2015. Systems and software engineering — System life cycle processes. International Organization for Standardization / Institute of Electrical and Electronics Engineers.

1.6. 版本更新信息

在项目开发过程中，使用 github 进行项目版本管理，共迭代了 5 个版本。在每一次版本迭代时，关于具体的修改时间、修改位置、修改内容均在 github 仓库中有详细记录。由于记录内容繁杂，不便于在报告中展示，因此在文档中省略关于版本更新信息的说明，如若想要查看相关版本更新信息，请查看本项目的 github 仓库。

表 2 版本更新表

修改编号	修改日期	修改后版本	修改位置	修改内容概述
U005	2024.06.08	v5.0	见仓库	见仓库

2. 系统设计概述

2.1. 系统设计概述

2.1.1. 用户界面层

- IDE 集成：提供与主流 IDE（如 Visual Studio Code, IntelliJ IDEA 等）的无缝集成，允许用户在熟悉的开发环境中使用 SysY2022 扩展语言。
- 语法高亮和悬浮提示：实现代码的语法高亮显示，以及当用户将鼠标悬停在代码元素上时，显示相关的提示信息。

2.1.2. 编辑器核心层

- 语法检查与错误提示：实时监控用户输入，进行语法分析，并在发现错误时立即提示用户。
- 静态语义检查：对代码进行静态语义分析，确保代码的语义正确性，并在发现问题时提供反馈。

2.1.3. 分析与修复层

- 修复建议：当检测到错误或潜在问题时，提供修复建议，帮助用户快速修正代码。
- 程序错误检查：深入分析代码逻辑，识别潜在的运行时错误，并给出警告。

2.1.4. 重构引擎层

- 代码重构：提供一系列重构操作，如重命名、提取方法、内联变量等，帮助用户改善代码结构，提高代码质量。

2.1.5. 后端服务层

- 集成编译器：与 SysY2022 语言的编译器集成，实现代码的编译功能，并提供编译错误反馈。

2.1.6.数据存储层

- 代码仓库：提供代码版本控制功能，支持用户管理代码的历史版本。
- 配置存储：存储用户的个性化配置，如主题、快捷键设置等。

2.1.7.插件系统

插件支持：开放插件接口，允许第三方开发者为编辑器开发扩展功能，如额外的代码分析工具、集成其他服务等。

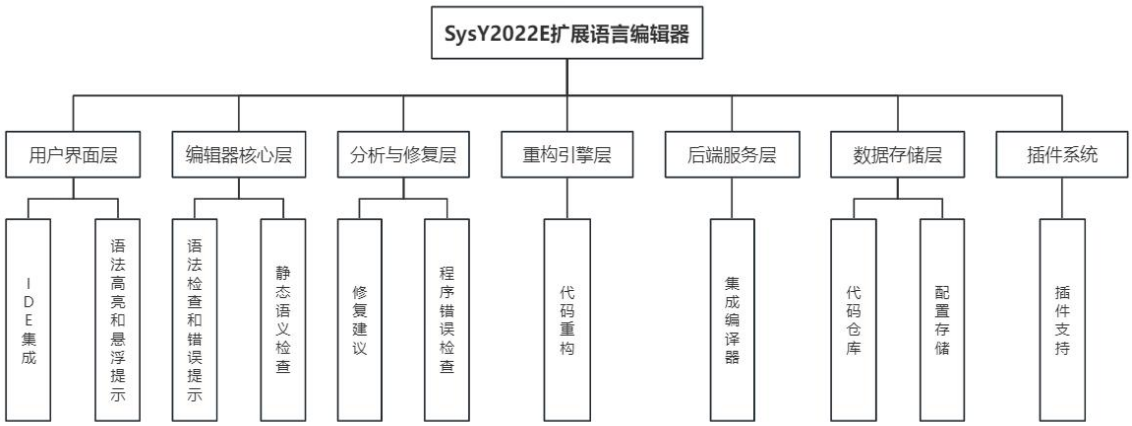

图 1 系统模块划分图

2.2. 模块划分和分布

2.2.1.用户界面层（User Interface Layer）

- IDE 集成：提供与主流 IDE（如 Visual Studio Code, IntelliJ IDEA 等）的无缝集成，允许用户在熟悉的开发环境中使用 SysY2022 扩展语言。
- 语法高亮和悬浮提示：实现代码的语法高亮显示，以及当用户将鼠标悬停在代码元素上时，显示相关的提示信息。

2.2.2.编辑器核心层（Editor Core Layer）

- 语法检查与错误提示：实时监控用户输入，进行语法分析，并在发现错误时立即提示用户。
- 静态语义检查：对代码进行静态语义分析，确保代码的语义正确性，并在发现问

题时提供反馈。

2.2.3.分析与修复层（Analysis and Fix Layer）

- 修复建议：当检测到错误或潜在问题时，提供修复建议，帮助用户快速修正代码。
- 程序错误检查：深入分析代码逻辑，识别潜在的运行时错误，并给出警告。

2.2.4.重构引擎层（Refactoring Engine Layer）

- 代码重构：提供一系列重构操作，如重命名、提取方法、内联变量等，帮助用户改善代码结构，提高代码质量。

2.2.5.后端服务层（Backend Service Layer）

- 集成编译器：与 SysY2022 语言的编译器集成，实现代码的编译功能，并提供编译错误反馈。

2.2.6.数据存储层（Data Storage Layer）

- 代码仓库：提供代码版本控制功能，支持用户管理代码的历史版本。
- 配置存储：存储用户的个性化配置，如主题、快捷键设置等。

2.2.7.插件系统（Plugin System）

插件支持：开放插件接口，允许第三方开发者为编辑器开发扩展功能，如额外的代码分析工具、集成其他服务等。

2.3. 各层及其模块之间的关系和通信

2.3.1.用户界面层与编辑器核心层

- 关系： 用户界面层接收用户的输入，并将这些输入传输到编辑器核心层进行处理。编辑器核心层分析用户的输入，执行语法检查和静态语义检查。当核心层发现错误或需要显示提示时，将这些信息传回给用户界面层，用户界面层再将这些信息展示给用户。
- 通信： 此通信通常是通过内部 API 调用实现的，可以是同步或异步的，这取决于具体

实现和用户操作的实时或非实时反馈需求。

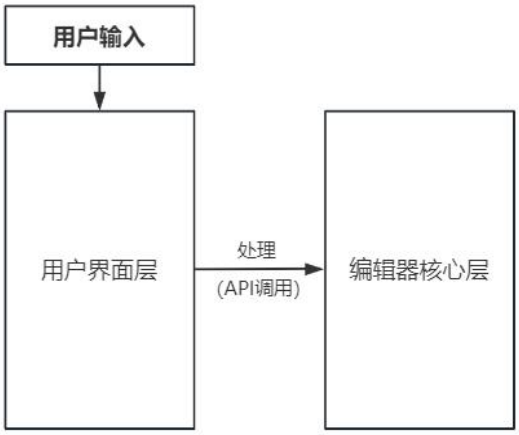

图 2 用户界面层与编辑器核心层关系图

2.3.2.编辑器核心层与分析与修复层

- 关系： 当编辑器核心层检测到潜在的代码问题时，会请求分析与修复层进行更深入的分析，并提出修复建议。分析与修复层处理这些请求，并返回一系列可能的解决方案。
- 通信： 通过内部调用进行，分析与修复层可能会作为一个独立的服务运行，使用消息队列或者 RPC（远程过程调用）机制与编辑器核心层交互。

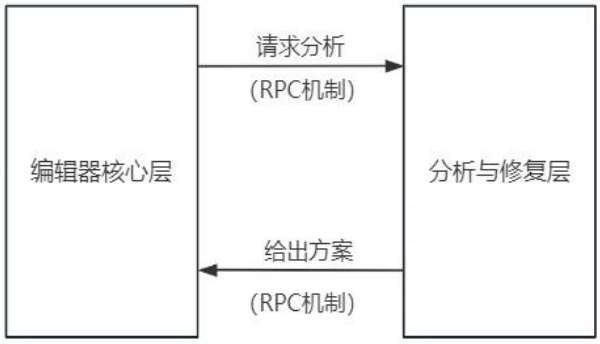

图 3 编辑器核心层与分析与修复层关系图

2.3.3.编辑器核心层与重构引擎层

- 关系： 当用户希望对代码进行结构性改变时，编辑器核心层会调用重构引擎层的功能。重构引擎层负责应用重构操作，并返回重构后的代码。
- 通信： 类似于分析与修复层的通信机制，重构引擎层可以通过内部 API 进行调用，也可能使用更复杂的通信协议，以支持重构操作的复杂性和计算需求。

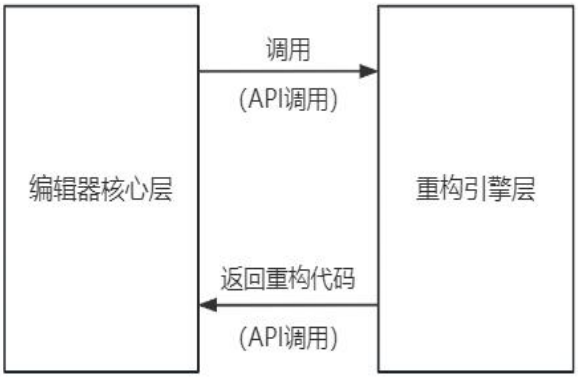

图 4 编辑器核心层与重构引擎层关系图

2.3.4.各层与后端服务层

- 关系： 后端服务层为其他层提供支持服务，如编译和调试。当用户请求编译或调试操作时，相应的层会与后端服务层通信，执行需要的服务。
- 通信： 服务层可能与其他层通过网络协议通信，特别是在分布式系统中。这可能包括 HTTP 请求或使用专用的编译器/调试器服务协议。

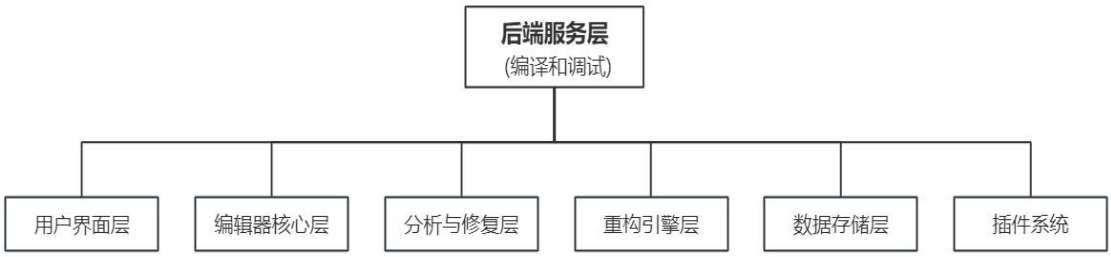

图 5 各层与后端服务层关系图

2.3.5.各层与数据存储层

- 关系： 数据存储层负责持久化用户的代码、配置和其他重要信息。当用户保存或加载代码时，相应层会与数据存储层通信。
- 通信： 数据存储层的通信通常使用数据库连接和访问协议，如 SQL，或者文件系统的 API 调用。

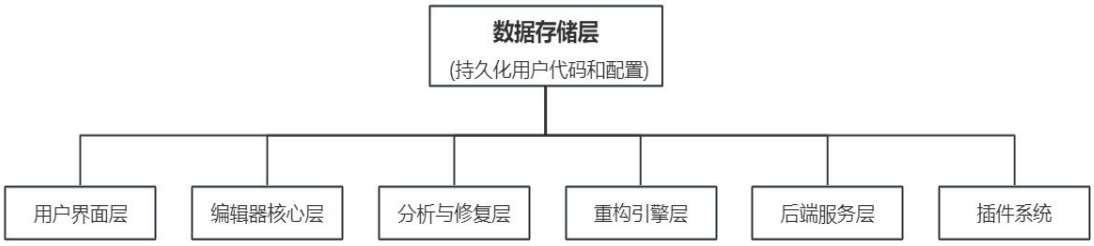

图 6 各层与数据存储层关系图

2.3.6.用户界面层与插件系统

- 关系： 插件系统允许第三方功能扩展与编辑器的核心功能集成。用户通过用户界面层管理和使用这些插件。
- 通信： 插件系统可能会通过定义良好的接口（例如，使用事件、钩子或者回调函数）与用户界面层和其他系统层交互。

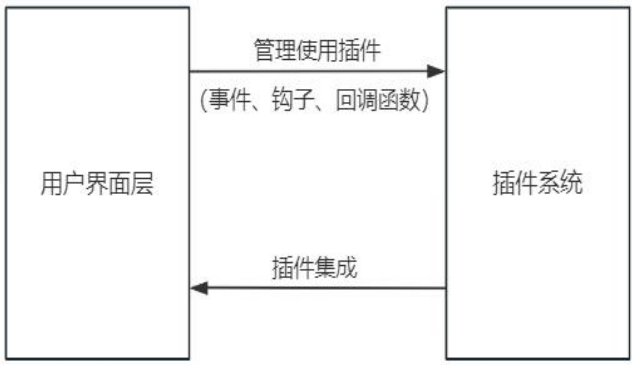

图 7 用户界面层与插件系统关系图

2.4. 系统采取的技术和实现方法

2.4.1.LSP 技术

Language Server Protocol(LSP)是由 Microsoft 开发的一种协议，旨在使得代码编辑器和语言服务器之间能够以标准化的方式进行通信。通过 LSP，开发者可以为不同的编程语言编写通用的语言服务器，从而使各种编辑器和 IDE 能够共享相同的语言智能功能（如代码补全、错误检查、语法高亮等）。其中，主要有 3 个核心概念：

1) 语言服务器：这是一个独立运行的进程，提供语言特定的功能，例如语法分析、语义分析、代码补全、重命名、格式化等。
2) 客户端：通常是一个编辑器或 IDE，它与语言服务器进行通信以获取语言服务功能。
3) 协议：LSP 定义了一组标准的请求和响应，用于在客户端和语言服务器之间传递信息。例如，客户端可以发送文档打开、文档更改、代码补全请求，语言服务器可以返回相应的结果。

LSP 的具体实现步骤如下：

1) 创建语言服务器：选择适合的编程语言和 LSP 库，例如 TypeScript 中的 vscode-languageserver。
2) 实现基础功能：实现 LSP 规范中的基础功能，如文档同步、代码补全、跳转到定义等。
3) 扩展功能：根据需要实现更多的语言特定功能，如语义分析、格式化、重命名等。
4) 测试和调试：对语言服务器进行全面的测试和调试，确保其在不同的编辑器中都能正常工作。
5) 发布和维护：将语言服务器发布为独立的软件包，并提供相应的文档和支持。

2.4.2.ANTLR4 构建工具

ANTLR4（Another Tool for Language Recognition）是一种强大的解析器生成器，广泛用于构建编译器、解释器和语言分析工具。它由 Terence Parr 开发，能够将语法定义转换为能够识别和处理相应语言的解析器代码。其中有 4 个核心概念：

1) 语法 Grammar：定义了语言的词法和语法结构。ANTLR4 的语法文件通常以 .g4 扩展名结尾，包含词法规则和语法规则。
2) 词法分析器 Lexer：负责将输入字符流分解为词法单元（tokens）。词法规则定义了语言中的基本构造，例如关键字、标识符、运算符等。
3) 语法分析器 Parser：负责根据语法规则将词法单元流转换为抽象语法树（AST）。语法规则定义了语言的结构，例如表达式、语句、函数等。
4) 抽象语法树 AST：是语法分析器生成的树状结构，表示源代码的语法结构。AST 是后续语义分析和代码生成的基础。

为了在我们的项目中使用 ANTLR 这一自动化构建工具，我们需要查阅 ANTLR4 的官方文档，并安装必须的依赖。通过学习官方文档，我们认为需要按照以下步骤进行实现：

1) 编写“g4”文件，该文件中包含 SysY2022E 语言的词法规则和语法规则，书写该文件时需要参考官方示例。
2) 利用 ANTLR4 官方项目中的 jar 包进行自动构建，通过设置相关参数控制不同的生成结果。
3) 利用自动化构建工具生成的文件，在项目中继承其中的类，添加属于我们项目的特性化需求。

2.4.3.TypeScript 语言

TypeScript 是一种由 Microsoft 开发和维护的开源编程语言，是 JavaScript 的超集。

它增加了静态类型和其他许多特性，使得开发大型和复杂应用程序变得更加容易。TypeScript 可以编译为纯 JavaScript,因此它可以运行在任何支持 JavaScript 的环境中。在我们的项目开发中，由于我们要开发一个与 IDE 高度集成的插件，因此我们选择 TypeScript 语言，其主要优势如下：

1) 类型安全：通过静态类型检查，可以在编译阶段捕获错误，从而减少运行时错误。
2) 开发工具支持：TypeScript 与现代编辑器和 IDE（如 Visual Studio Code）集成良好，提供强大的代码补全、重构和导航功能。
3) 兼容性：TypeScript 是 JavaScript 的超集，任何有效的 JavaScript 代码都是有效的 TypeScript 代码。此外，TypeScript 编译器可以将 TypeScript 代码编译为各种版本的 JavaScript，从而支持不同的运行环境。
4) 社区和生态系统：TypeScript 拥有庞大的社区和丰富的生态系统，许多流行的 JavaScript 库（如 React、Angular 和 Vue.js）都提供了 TypeScript 类型定义。

3. 详细设计概述

在进行详细设计时，按照总体设计的模块划分，进一步详细阐述其实现。在具体的实现过程中，受限于具体的操作环境和知识水平，部分模块的功能可能与设计有所偏差，但这并不会影响最终用户的体验，我们保证最终仍能够满足用户的需求。

3.1. 用户界面

根据总体设计方案，用户界面层主要有 IDE 集成功能、语法高亮和悬浮提示。下面，将对与此相关的模块进行描述。

3.1.1. IDE 集成

本项目开发的目标是得到一个可以与主流 IDE 集成的 SysY2022E 语言编辑器工具,具体而言，将开发与 vscode 集成的 SysY2022E 语言支持插件。项目开发的基础，为微软官方关于 vscode 插件开发的手脚架，该手脚架为 vscode 插件开发工作提供了基本框架。

关于 IDE 集成并没有显式的模块，因为项目本身即为 vscode 插件开发项目，IDE 集成自然而然与本项目深度绑定，无需显式地提供模块供其使用。

3.1.2.语法高亮

1) 模块用途

语法高亮模块的主要用途是提高代码的可读性和可维护性。通过对源代码中的不同元素（如关键字、变量、函数、注释等）使用不同的颜色或样式，开发者能够更容易地理解代码的结构和逻辑，迅速识别和定位代码中的各类元素和潜在错误。

2) 模块功能

- **词法分析：**解析源代码文本，将其分解为基本的语言标记（Token），如关键字、标识符、运算符、数字、字符串和注释。
- **高亮规则应用：**根据语言定义的语法规则，对不同类型的 Token 应用相应的高亮样式。这些规则由正则表达式或语言定义文件 JSON 指定。
- **实时高亮：**在开发者输入代码时，实时更新语法高亮显示，确保编辑器中的代码始终保持高亮状态。

3) 特别约定

- **高亮规则文件：**语法高亮规则存储在独立的配置文件中，便于更新和维护。配置文件使用标准格式 JSON，并提供详细的注释说明。
- **一致性：**高亮样式与用户界面的其他部分保持一致，避免使用过多的颜色或样式，确保整体界面的美观和一致性。
- **兼容性：**高亮模块兼容不同版本的 VS Code 编辑器和不同操作系统（如 Windows、macOS、Linux），确保在各种环境下都能正常工作。

3.1.3.悬浮提示

1) 模块用途

悬浮提示模块的主要用途是提供即时的代码上下文信息和文档提示，帮助开发者在编写代码时获得必要的参考信息，提高编码效率和代码质量。通过悬浮提示，开发者可以在光标悬停在代码元素上时，看到相关的注释、类型信息、函数签名等详细信息。

2) 模块功能

- **信息显示：**当开发者将鼠标悬停在特定的代码元素上时，显示相关的文档信息、注释、类型信息、函数签名、参数说明。
- **动态更新：**根据光标所在位置动态更新提示信息，确保提示内容始终与当前代码上下文相关。

- **语法解析：**利用语法解析和语义分析技术，准确识别代码元素并获取其相关信息。
- **文档链接：**提供文档链接，允许用户点击链接跳转到相关的文档或定义位置，获取更详细的信息。

3) 特别约定

- **信息来源：**悬浮提示信息来源于准确的文档和代码注释，确保提示内容的正确性和可靠性。可以通过 LSP 与语言服务器交互获取相关信息。
- **用户体验：**提示信息的显示美观且不干扰用户的正常编码操作，提示窗口应具有合适的大小和位置，避免遮挡代码视图。
- **一致性：**提示信息的样式和内容与编辑器界面的其他部分保持一致，确保整体界面的美观和一致性。

3.2. 编辑器核心

根据总体设计方案，该部分需要完成语法检查与错误提示和静态语义检查，下面，将对相关模块进行描述。

3.2.1. 语法检查与错误提示

1) 模块用途

语法检查与错误提示模块的主要用途是自动检测代码中的语法错误和潜在问题，并及时向开发者提供错误提示信息。通过此模块，开发者可以在编码过程中即时发现和修复语法错误，提高代码的正确性和质量，减少调试时间。

2) 模块功能

- **实时语法检查：**在开发者编写代码时，实时分析代码并检测语法错误、未闭合的括号、未声明的变量等常见问题。
- **错误标记：**在代码编辑器中，以红色波浪线标记检测到的语法错误，方便开发者快速定位问题。
- **错误提示信息：**提供详细的错误提示信息，包括错误类型、错误位置、错误描述等，帮助开发者理解错误原因并进行修复。
- **快速修复建议：**针对常见的语法错误，提供一键修复建议和自动修复选项，帮助开发者快速修正代码。

3) 特别约定

- **准确性：**语法检查尽量准确，避免误报和漏报，确保提示信息的正确性和可靠性。

可以通过引入权威的语言解析器和语法规则库来提高准确性。

- **性能优化：**由于语法检查需要频繁分析代码，模块实现高效的语法解析算法和检查机制，尽量减少对编辑器性能的影响，确保编辑器的流畅运行。
- **用户体验：**错误提示信息以直观、易理解的方式呈现，错误标记应不影响代码的可读性和美观性。提示窗口应具有合适的大小和位置，避免遮挡代码视图。
- **一致性：**错误提示信息的样式和内容与编辑器界面的其他部分保持一致，确保整体界面的美观和一致性。

3.2.2. 静态语义检查

1) 模块用途

静态语义检查模块的主要用途是通过在编译前分析源代码，检测代码中可能存在的语义错误和逻辑问题。静态语义检查有助于开发者在早期阶段发现和修正代码中的问题，提高代码的正确性、健壮性和可维护性，减少运行时错误。

2) 模块功能

- **类型检查：**验证变量和表达式的类型是否匹配，检测类型不一致、类型转换错误等问题。
- **变量声明和使用：**检查变量是否在使用前声明，检测未使用的变量和重复声明的变量。
- **作用域分析：**验证变量和函数的作用域，检测作用域内的冲突和不合法的引用。
- **控制流分析：**分析代码的控制流，检测未初始化变量的使用、不可达代码和潜在的无限循环等问题。
- **函数调用检查：**验证函数调用时的参数个数和类型是否正确，检测函数签名不匹配的问题。
- **常量检查：**检查常量的定义和使用，检测常量的重新赋值等问题。

3) 特别约定

- **准确性：**静态语义检查准确，避免误报和漏报，确保提示信息的正确性和可靠性。可以通过引入权威的语义分析器和语义规则库来提高准确性。
- **性能优化：**由于静态语义检查需要分析整个代码文件，模块实现高效的语义分析算法和检查机制，尽量减少对编辑器性能的影响，确保编辑器的流畅运行。
- **用户体验：**语义错误提示信息应以直观、易理解的方式呈现，错误标记应不影响代码的可读性和美观性。提示窗口应具有合适的大小和位置，避免遮挡代码视图。
- **一致性：**语义错误提示信息的样式和内容与编辑器界面的其他部分保持一致，确

保整体界面的美观和一致性。

3.3. 分析与修复

依据项目总体设计的内容，这一部分将实现修复建议和深层次的程序错误检查。下面，将对相关模块进行描述。

3.3.1.修复建议与自动修复

1) 模块用途

修复建议与自动修复模块的主要用途是为开发者提供代码问题的修复建议,并在可能的情况下自动修复这些问题。通过此模块,开发者可以更快速地解决代码中的错误和优化代码,提高编码效率和代码质量。

2) 模块功能

- **错误分析：**分析代码中的语法错误、语义错误和其他潜在问题，生成详细的错误报告。
- **修复建议：**根据分析结果，为每个检测到的问题提供详细的修复建议，包括具体的修改方案和代码示例。
- **自动修复：**在可能的情况下，提供一键自动修复功能，自动应用修复建议并修改代码。
- **用户确认：**在自动修复前，可以选择提示用户确认修复方案，确保自动修复符合用户预期。
- **修复历史记录：**记录所有自动修复操作的历史，允许用户查看、撤销和重做修复操作。

3) 特别约束

- **准确性：**修复建议尽量准确，确保修复方案的正确性和有效性。自动修复应在不影响代码功能的前提下进行。
- **性能优化：**修复建议与自动修复功能尽量高效，减少对编辑器性能的影响，确保编辑器的流畅运行。
- **用户体验：**修复建议和自动修复操作以直观、易理解的方式呈现，提示信息应不影响代码的可读性和美观性。修复提示窗口应具有合适的大小和位置，避免遮挡代码视图。

- **一致性：**修复建议和自动修复的样式和内容与编辑器界面的其他部分保持一致，确保整体界面的美观和一致性。
- **安全性：**自动修复功能确保修复操作的安全性，避免引入新的错误或漏洞。修复前应进行必要的验证和测试。

3.3.2. 程序错误检查

1） 模块用途

深层次程序错误检查模块的主要用途是检测代码中难以发现的深层次错误，例如无用变量、死循环、数组越界等。通过此模块，开发者可以发现和修复潜在的严重问题，提高代码的可靠性和安全性，减少运行时错误和崩溃的风险。

2） 模块功能

- **无用变量检测：**检查代码中未使用的变量，并提示开发者删除或重构相关代码。
- **死循环检测：**分析代码的控制流，检测潜在的死循环，并提供修复建议。
- **数组越界检查：**验证数组的访问索引是否超出其边界，防止数组越界错误。内存
- **泄漏检测：**检查代码中可能的内存泄漏问题，提示开发者释放未使用的内存。
- **空指针引用检测：**分析指针和引用的使用，检测可能的空指针引用问题，防止程序崩溃。
- **未初始化变量检查：**检查变量的初始化状态，防止使用未初始化的变量。
- **逻辑错误检测：**识别常见的逻辑错误，如条件表达式总为真或总为假，分支语句中缺少必要的分支等。
- **代码优化建议：**提供代码优化建议，如消除冗余代码、合并相似代码块等，提升代码的性能和可读性。

3） 特别约定

- **准确性：**深层次错误检查尽量准确，避免误报和漏报，确保提示信息的正确性和可靠性。可以通过引入高级静态分析技术和工具来提高检测的准确性。
- **性能优化：**由于深层次错误检查可能需要对代码进行深入分析，模块实现高效的分析算法和检查机制，尽量减少对编辑器性能的影响，确保编辑器的流畅运行。
- **用户体验：**错误提示信息以直观、易理解的方式呈现，错误标记应不影响代码的可读性和美观性。提示窗口应具有合适的大小和位置，避免遮挡代码视图。（4）一致性：错误提示信息的样式和内容应与编辑器界面的其他部分保持一致，确保整体界面的美观和一致性。
- **安全性：**检查功能确保安全性，避免引入新的错误或漏洞。检测前应进行必要的

验证和测试，防止误操作和数据丢失。

- **持续改进：**模块支持持续改进和更新，及时引入新的错误检查规则和优化建议，跟随编程语言和开发工具的发展，不断提升检查的准确性和有效性。

3.4. 重构引擎

在这一部分，按照系统总体设计的内容，实现一系列重构操作。下面，将对这些模块进行描述。

1) 模块用途

代码重构模块的主要用途是帮助开发者改善现有代码的结构和设计，以提高代码的可读性、可维护性和性能。通过此模块，开发者可以快速、安全地对代码进行重构，使其更符合设计原则和最佳实践。

2) 模块功能

- **提供重构建议：**根据代码的结构和设计原则，提供代码重构的建议，包括代码块合并、拆分、提取方法、重命名等。
- **自动化重构：**在用户确认重构建议后，自动应用重构操作，修改代码并保持其功能完整性。
- **安全检查：**在执行自动化重构前，进行必要的安全检查，确保重构操作不会引入新的错误或破坏现有功能。
- **支持多种重构操作：**支持常见的代码重构操作，如提取方法、内联方法、移动代码、重命名等，以及更复杂的重构操作，如重构继承关系、替换算法等。
- **实时反馈：**在进行重构操作时，实时反馈操作的进度和结果，以便用户随时了解重构的影响和变化。
- **撤销和重做：**支持撤销和重做重构操作，方便用户在需要时回退或重新执行操作。

3) 特别约定

- **准确性：**重构建议和自动化重构尽量准确，避免误导和破坏现有功能。提供清晰的提示和安全检查，确保重构操作的正确性。
- **性能优化：**重构操作尽量高效，减少对编辑器性能的影响，确保编辑器的流畅运行。
- **用户体验：**重构建议和操作以直观、易理解的方式呈现，提示信息不影响代码的可读性和美观性。重构操作的界面具有合适的大小和位置，避免遮挡代码视图。
- **一致性：**重构建议和操作的样式和内容与编辑器界面的其他部分保持一致，确保

整体界面的美观和一致性。

3.5. 优化层

该部分内容是在项目开发过程中增加的、可以提升用户体验的层次，包括辅助编辑、查找引用、签名帮助、查看定义。下面，将对这些辅助优化模块进行描述。

3.5.1.辅助编辑

1) 模块用途

辅助编辑模块的主要用途是提供一系列辅助功能，帮助开发者提高编码效率和代码质量。通过此模块，开发者可以快速完成常见的编辑操作，减少重复劳动，提高编码体验。

2) 模块功能

- **自动闭合**：根据语言语法，自动闭合代码中的括号、引号等符号，减少手动输入闭合符号的工作量。
- **自动缩进**：根据代码结构，自动调整代码的缩进，使代码层次清晰、易读。
- **自动环绕**：提供快捷键或菜单选项，可以快速将选中的代码块用指定的符号或模板包围起来，如 if 语句、函数等。
- **代码折叠**：支持代码折叠功能，可以折叠或展开代码块，方便浏览和编辑大型代码文件。

3) 特别约定

- **准确性**：辅助编辑功能应准确地识别代码结构和语法，确保自动闭合、缩进、环绕等操作的正确性。
- **性能优化**：操作应尽量高效，不影响编辑器的响应速度和性能。
- **用户体验**：提供直观、易用的界面和操作方式，确保操作的流畅性和便捷性。
- **一致性**：辅助编辑功能的样式和行为应与编辑器的其他部分保持一致，确保整体界面的美观和一致性。

3.5.2.查找引用

1) 模块用途

查找引用模块的主要用途是帮助开发者查找代码中的引用，了解变量、函数、类等在代

码中的使用情况，方便代码的理解和修改。通过此模块，开发者可以快速定位和浏览代码中的引用位置。

2) 模块功能

- **查找变量引用：**根据用户选择的变量，查找并显示所有引用该变量的位置。查找
- **函数引用：**根据用户选择的函数，查找并显示所有调用该函数的位置。
- **查找类引用：**根据用户选择的类，查找并显示所有实例化该类的位置。查找引用
- **路径：**显示引用路径，即代码中引用的层次关系，帮助用户理清代码的逻辑结构。
- **快速导航：**提供快速导航到引用位置的功能，方便用户查看和编辑引用位置的代码。

3) 特别约定

- **准确性：**查找引用功能应准确地识别代码中的引用，确保显示的引用位置是正确的。
- **性能优化：**查找引用操作应尽量高效，对大型代码库也能快速响应，保持编辑器的流畅性。
- **用户体验：**提供直观、易用的界面和操作方式，确保用户可以快速定位到引用位置。
- **一致性：**查找引用功能的样式和行为应与编辑器的其他部分保持一致，确保整体界面的美观和一致性。

3.5.3.签名帮助

1) 模块用途

签名帮助模块的主要用途是提供函数和方法的签名信息,帮助开发者正确地调用函数和方法，并提供参数信息和返回值类型，以便开发者更加准确地编写代码。

2) 模块功能

- **函数签名显示：**在编辑器中显示函数和方法的签名，包括函数名、参数列表和返回值类型。
- **参数提示：**根据函数签名提示参数信息，包括参数名和参数类型。
- **参数类型检查：**在输入参数时进行类型检查，提示可能的参数类型错误。
- **返回值提示：**根据函数签名提示返回值类型，帮助开发者正确处理函数的返回值。
- **函数重载支持：**支持函数重载，根据参数类型和数量显示不同的函数签名。
- **快速导航：**提供快速导航到函数定义的功能，方便查看函数的实现和文档。

3) 特别约定

- **准确性：**签名帮助功能应准确地显示函数和方法的签名信息，确保提示的参数和返回值类型是正确的。
- **性能优化：**签名帮助功能应尽量高效，对大型代码库也能快速响应，保持编辑器的流畅性。
- **用户体验：**提供直观、易用的界面和操作方式，确保用户可以快速查看和理解函数签名信息。
- **一致性：**签名帮助功能的样式和行为应与编辑器的其他部分保持一致，确保整体界面的美观和一致性。

3.5.4.查看引用

1) 模块用途

查看引用模块的主要用途是帮助开发者查看代码中的引用情况，包括变量、函数、类等在代码中的所有引用位置。通过此模块，开发者可以更好地理解代码的使用方式和调用关系。

2) 模块功能

- **显示引用位置：**在编辑器中显示变量、函数、类等在代码中的所有引用位置。
- **查看引用路径：**显示引用路径，即代码中引用的层次关系，帮助用户理清代码的逻辑结构。
- **快速导航：**提供快速导航到引用位置的功能，方便用户查看和编辑引用位置的代码。
- **跳转到定义：**支持跳转到变量、函数、类等的定义位置，方便用户查看其定义和实现。

3) 特别约定

- **准确性：**查看引用功能应准确地显示代码中的引用位置，确保显示的引用位置是正确的。
- **性能优化：**查看引用操作应尽量高效，对大型代码库也能快速响应，保持编辑器的流畅性。
- **用户体验：**提供直观、易用的界面和操作方式，确保用户可以快速查看和理解引用位置。
- **一致性：**查看引用功能的样式和行为应与编辑器的其他部分保持一致，确保整体界面的美观和一致性。

4. 详细设计

在进行详细设计之前，应先对本项目的目标语言 SysY2022E 的词法、语法进行规范，方便后续处理。因此，本部分首先对 SysY2022E 语言进行规范说明，然后以模块为单位，对每个模块进行详细设计。

4.1. 语言规范

4.1.1. 词法规范

SysY2022E 语言的词法规范定义了该语言的基本词法单元，包括标识符、关键字、常量、运算符和分隔符。标识符由字母、数字和下划线组成，且必须以字母或下划线开头；关键字是语言保留的标识符，如 if、else、while 等，不能作为用户自定义的标识符使用；常量包括整数和浮点数，支持十进制和十六进制表示；运算符包括算术运算符、关系运算符、逻辑运算符等；分隔符包括括号、逗号、分号等符号，用于分隔代码中的各个部分。所有词法单元之间可以用空格、制表符和换行符分隔。

Flag（这里导出文件后在添加实际代码内容）

表 3 SysY2022E 语言词法规范

4.1.2. 语法规范

SysY2022E 语言的语法规范定义了程序的结构和组成规则，包括语句、表达式、函数定义和控制结构等。一个合法的 SysY2022E 程序由若干个函数组成，每个函数包含一个返回类型、函数名、参数列表和函数体。函数体由语句块构成，语句块可以包含变量声明、赋值语句、控制结构（如 if-else 语句、while 循环）、函数调用和返回语句。表达式包括算术表达式、逻辑表达式和比较表达式，可以嵌套使用。控制结构允许嵌套和组合，支持复杂的程序流程控制。所有代码必须遵循严格的语法规则，以保证程序的正确性和可读性。

Flag（导出后添加代码）

表 4 SysY2022E 语言语法规范

4.2. 用户界面

4.2.1. IDE 集成

IDE 集成模块作为项目的隐含模块，是整个项目开发的基础，该模块所涉及到的数据、算法均与其他各模块存在关联，在编码时应时刻注意这一点。

1) 模块定义

IDE 集成模块旨在将 SysY2022E 语言的开发功能无缝集成到主流集成开发环境 Visual Studio Code。该模块通过语法高亮、代码补全、语法检查、语义检查、签名帮助、代码重构等功能，提升开发者的编码效率和体验。

2) 模块关联

- 语法高亮模块：负责对代码中的关键字、标识符、运算符等进行高亮显示。
- 代码补全模块：提供基于上下文的代码补全建议，帮助开发者快速编写代码。
- 语法检查与错误提示模块：在开发过程中实时检查语法错误，并提供错误提示。
- 静态语义检查模块：检查代码中的语义错误，如类型不匹配、未定义的变量等。
- 签名帮助模块：在编写函数调用时，提供函数签名和参数信息提示。
- 代码重构模块：提供代码重构功能，如重命名、提取函数等。
- 悬浮提示模块：当鼠标悬停在标识符上时，显示相关信息（例如变量类型或函数描述）。

3) 数据说明

- 源码文件：开发者编写的 SysY2022E 语言代码文件，扩展名通常为.sy。
- 词法单元：从源码文件中解析出的基本词法单元，如标识符、关键字、运算符等。
- 语法树（AST）：由词法单元生成的抽象语法树，用于表示程序的语法结构。
- 符号表：在语义分析过程中生成的符号表，记录变量、函数等标识符的信息。
- 错误信息：包含语法错误和语义错误的信息，如错误位置、错误类型和提示信息。

4) 算法说明

- 词法分析：扫描源码文件，识别并生成词法单元。
  a) 输入：源码文件
  b) 输出：词法单元序列

c) 算法：正则表达式匹配、状态机

表 5 词法分析算法伪码描述

```
// 词法分析伪码
// 词法分析器主函数
function lexicalAnalysis(sourceCode) {
    tokens = []
    position = 0
    while (position < sourceCode.length) {
        matchFound = false
        for (tokenType, pattern in PATTERNS) {
            match = matchPattern(pattern, sourceCode, position)
            if (match) {
                token = createToken(tokenType, match)
                tokens.append(token)
                position += match.length
                matchFound = true
                break
            }
        }
        if (!matchFound) {
            throwError("Unknown token at position " + position)
        }
    }
    return tokens
}
// 匹配正则表达式模式
function matchPattern(pattern, text, startPos) {
    result = pattern.exec(text.slice(startPos))
    if (result && result.index === 0) {
        return result[0]
    }
    return null
```

```
}
// 创建词法单元
function createToken(type, value) {
    return {
        type: type,
        value: value
    }
}
// 示例使用
sourceCode = `
int main() {
    // 这是一个注释
    float x = 10.5;
    if (x > 5) {
        return 1;
    } else {
        return 0;
    }
}
tokens = lexicalAnalysis(sourceCode)
print(tokens)
```

- 语法分析：根据词法单元序列生成抽象语法树（AST）。
  a) 输入：词法单元序列
  b) 输出：抽象语法树
  c) 算法：递归下降解析、LR 解析

表 6 语法分析算法伪码描述

```
//语法分析伪码
// 语法分析器类
class Parser {
    List<Token> tokens;
    int currentTokenIndex;
```

```
    // 构造函数
    Parser(List<Token> tokens) {
        this.tokens = tokens;
        this.currentTokenIndex = 0;
    }
    // 获取当前词法单元
    Token currentToken() {
        return tokens[currentTokenIndex];
    }
    // 移动到下一个词法单元
    void advance() {
        currentTokenIndex++;
    }
    // 匹配并消耗当前词法单元
    void match(TokenType expectedType) {
        if (currentToken().type == expectedType) {
            advance();
        } else {
            throw new Error("Syntax Error: Unexpected token at line " +
currentToken().line + ", column " + currentToken().column);
        }
    }
    // 解析程序
    ASTNode parseProgram() {
        ASTNode node = new ASTNode(ASTNodeType.PROGRAM);
        node.children = parseStatements();
        return node;
    }
    // 解析语句序列
    List<ASTNode> parseStatements() {
        List<ASTNode> statements = [];
        while (currentTokenIndex < tokens.length) {
```

```
            statements.push(parseStatement());
        }
        return statements;
    }
// 执行语法分析示例
List<Token> tokens = lexicalAnalysis(sourceCode);
Parser parser = new Parser(tokens);
ASTNode ast = parser.parseProgram();
printAST(ast);
```

- 语义分析：使用抽象语法树进行语义检查，生成符号表并记录语义错误。
  a) 输入：抽象语法树
  b) 输出：符号表、语义错误信息
  c) 算法：符号表管理、类型检查、作用域解析

表 7 分析算法伪码描述

```
//语义分析伪码
// 语义分析器类
class SemanticAnalyzer {
    SymbolTable symbolTable;
    List<string> semanticErrors;
    // 构造函数
    SemanticAnalyzer() {
        this.symbolTable = new SymbolTable();
        this.semanticErrors = [];
    }
    // 语义分析入口函数
    void analyze(ASTNode ast) {
        analyzeNode(ast);
    }
    // 分析单个节点
    void analyzeNode(ASTNode node) {
        switch (node.type) {
```

```
            case ASTNodeType.PROGRAM:
                for (let child of node.children) {
                    analyzeNode(child);
                }
                break;
            case ASTNodeType.DECLARATION:
                analyzeDeclaration(node);
                break;
            case ASTNodeType.ASSIGNMENT:
                analyzeAssignment(node);
                break;
            case ASTNodeType.IF_STATEMENT:
                analyzeIfStatement(node);
                break;
            case ASTNodeType.WHILE_STATEMENT:
                analyzeWhileStatement(node);
                break;
            // 其他节点类型处理
            default:
                for (let child of node.children) {
                    analyzeNode(child);
                }
        }
    }
// 执行语义分析示例
ASTNode ast = parser.parseProgram();
SemanticAnalyzer semanticAnalyzer = new SemanticAnalyzer();
semanticAnalyzer.analyze(ast);
for (let error of semanticAnalyzer.semanticErrors) {
    console.log(error);
}
```

- 语法高亮：根据词法单元对代码进行高亮显示。

a) 输入：词法单元序列

b) 输出：高亮信息

c) 算法：匹配关键字、标识符类型等

表 8 高亮算法伪码描述

```
//语法高亮伪码
// 定义一个函数，用于根据词法单元序列生成高亮信息
function highlightCode(tokens) {
    let highlightedCode = []
    // 遍历词法单元序列
    for (let token of tokens) {
        let highlight = ''
        // 根据词法单元的类型设置高亮样式
        switch (token.type) {
            case 'KEYWORD':
                highlight = '<span style="color: blue;">' + token.value +
'</span>'
                break
            case 'IDENTIFIER':
                highlight = '<span style="color: green;">' + token.value +
'</span>'
                break
            case 'STRING':
                highlight = '<span style="color: red;">' + token.value +
'</span>'
                break
            case 'NUMBER':
                highlight = '<span style="color: darkorange;">' + token.value
+ '</span>'
                break
            case 'OPERATOR':
                highlight = '<span style="color: purple;">' + token.value +
'</span>'
```

```
                break
            case 'COMMENT':
                highlight = '<span style="color: gray;">' + token.value +
'</span>'
                break
            // 其他类型的高亮样式
            default:
                highlight = token.value // 默认不设置特殊样式
        }
        // 将高亮后的文本添加到结果中
        highlightedCode.push(highlight)
    }
    // 返回高亮后的代码字符串
    return highlightedCode.join('')
}
// 示例使用
let tokens = [
    { type: 'KEYWORD', value: 'if' },
    { type: 'IDENTIFIER', value: 'x' },
    { type: 'OPERATOR', value: '=' },
    { type: 'NUMBER', value: '10' },
    { type: 'STRING', value: '"Hello, World!"' },
    { type: 'COMMENT', value: ' This is a comment ' }
]
let highlighted = highlightCode(tokens)
console.log(highlighted)
```

- 代码补全：提供基于上下文的代码补全建议。
  a) 输入：当前编辑位置及上下文
  b) 输出：补全建议列表
  c) 算法：前缀匹配、符号表查找

表 9 补全算法伪码描述

```
//代码补齐伪码
// 定义一个函数，用于提供代码补全建议
function codeCompletion(currentPosition, context, symbolTable) {
    let suggestions = []
    // 获取当前编辑位置的前缀
    let prefix = getPrefix(currentPosition, context)
    // 遍历符号表，查找匹配的符号
    for (let symbol in symbolTable) {
        if (symbol.startsWith(prefix)) {
            suggestions.push(symbol)
        }
    }
    // 返回补全建议列表
    return suggestions
}
// 辅助函数，用于获取当前编辑位置的前缀
function getPrefix(position, context) {
    // 假设 context 是一个字符串，position 是一个索引
    let prefix = ''
    for (let i = position - 1; i >= 0 && context[i] !== ' '; i--) {
        prefix = context[i] + prefix
    }
    return prefix
}
// 示例使用
let symbolTable = {
    "print",
    "printf",
    "println",
    "input",
    "int",
    "float",
```

```
    "str",
    "list",
    "dict"
}
let context = "print the value of x"
let position = context.indexOf("x") + 1
let suggestions = codeCompletion(position, context, symbolTable)
console.log(suggestions)
```

- 签名帮助：在函数调用时提供参数和签名提示。
  a) 输入：当前编辑位置及上下文
  b) 输出：函数签名信息
  c) 算法：符号表查找、参数匹配

表 10 帮助算法伪码描述

```
//签名帮助伪码
// 定义一个函数，用于提供函数签名帮助
function signatureHelp(currentPosition, context, symbolTable) {
    // 分析当前编辑位置的上下文，找到函数名
    let functionName = extractFunctionName(currentPosition, context)
    // 在符号表中查找函数名对应的签名
    let functionSignature = symbolTable.lookup(functionName)
    // 如果找到函数签名，则返回该签名
    if (functionSignature) {
        return formatSignature(functionSignature)
    // 如果没有找到对应的签名，返回空或者错误信息
    return null
}
// 示例使用
let symbolTable = {
    // 假设符号表是一个对象，键为函数名称，值为函数签名
    "add": {
        name: "add",
```

```
        returnType: "int",
        params: [
            { name: "a", type: "int" },
            { name: "b", type: "int", defaultValue: "0" }
        ]
    },
}
let context = "let result = add(a, b"
let position = context.length - 1 // 假设我们当前在'b'的后面
let signatureInfo = signatureHelp(position, context, symbolTable)
console.log(signatureInfo)
```

- 代码重构：提供重命名、提取函数等重构操作。
  a) 输入：重构目标及上下文
  b) 输出：重构后的代码
  c) 算法：AST 变换、符号表更新

表 11 重构算法伪码描述

```
//代码重构伪码
// 定义一个函数，用于执行代码重构操作
function refactorCode(target, context, symbolTable) {
    // 根据重构目标选择相应的重构方法
    switch (target.type) {
        case 'RENAME':
            return renameSymbol(target, context, symbolTable)
        case 'EXTRACT_FUNCTION':
            return extractFunction(target, context, symbolTable)
        // 其他重构操作
        default:
            return null
    }
}
// 重命名符号的重构方法
```

```
function renameSymbol(target, context, symbolTable) {
    // 从上下文中提取旧的符号名称
    let oldName = target.oldName
    // 从目标中获取新的符号名称
    let newName = target.newName
    // 更新符号表中的符号名称
    symbolTable.update(oldName, newName)
    // 更新代码中的引用
    let updatedCode = updateReferences(oldName, newName, context)
    return updatedCode
}
// 提取函数的重构方法
function extractFunction(target, context, symbolTable) {
    // 从上下文中提取要提取的代码块
    let codeBlock = extractCodeBlock(target, context)
    // 创建新的函数定义
    let newFunction = createFunction(codeBlock, target.newName)
    // 更新符号表，添加新的函数
    symbolTable.add(newFunction)
    // 更新原始代码，移除提取的代码块并调用新函数
    let updatedCode = replaceCodeBlockWithFunctionCall(codeBlock, newFunction,
context)
    return updatedCode
}
// 示例使用
let symbolTable = new SymbolTable()
let context = "let x = 10; print(x);"
let target = { type: 'RENAME', oldName: 'x', newName: 'y' }
let refactoredCode = refactorCode(target, context, symbolTable)
console.log(refactoredCode)
```

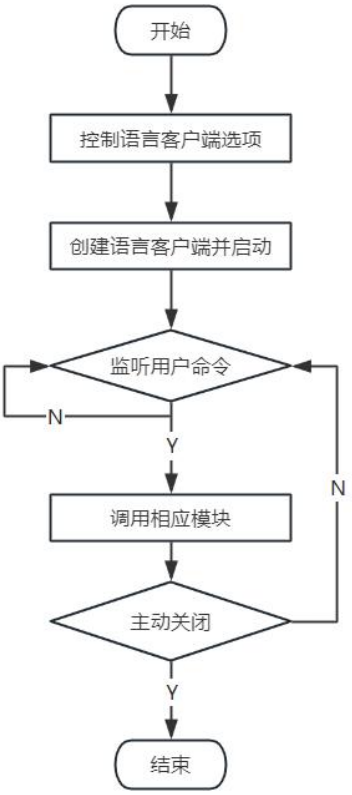

图 8 IDE 集成模块流程图（一）

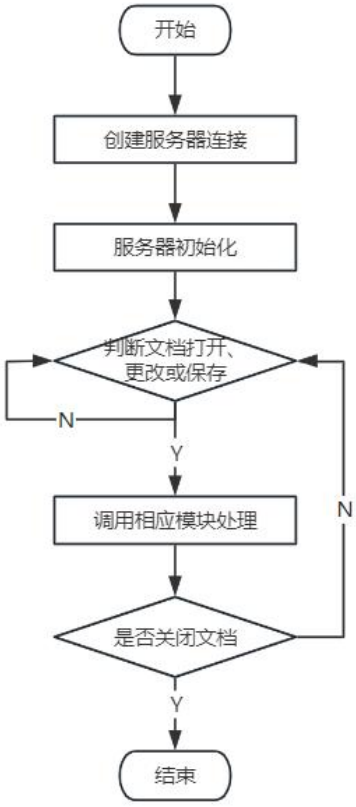

图 9 IDE 集成模块流程图（二）

4.2.2. 语法高亮

1) 模块定义

语法高亮模块旨在通过不同颜色和样式对代码中的关键字、标识符、运算符等进行高亮显示，从而提高代码的可读性和可维护性。该模块会根据 SysY2022E 语言的词法规范，将源码文件中的各个词法单元分类并进行相应的高亮显示。

2) 模块关联

- 词法分析模块：提供代码的词法单元，用于识别和分类不同类型的代码片段。
- IDE 集成模块：将语法高亮功能集成到 IDE 中，确保高亮效果在编辑器中可见。
- 配置管理模块：允许用户自定义高亮显示的样式和颜色。
- 语法检查模块：提供解析错误提示，与高亮模块协同工作，显示错误位置。

3) 数据说明

- 源码文件：用户编写的 SysY2022E 语言代码文件。
- 词法单元：从源码文件中提取的基本词法单元，如标识符、关键字、运算符、常量和分隔符。
- 高亮配置：定义不同类型词法单元的显示样式，包括颜色、字体样式等。

4) 说明

- 词法分析：通过词法分析模块对源码文件进行扫描，生成词法单元序列。
  a) 输入：源码文件
  b) 输出：词法单元序列
  c) 算法：正则表达式匹配、状态机
- 词法单元分类：根据 SysY2022E 语言的词法规范，对词法单元进行分类。
  a) 输入：词法单元序列
  b) 输出：分类后的词法单元
  c) 算法：匹配关键字表、识别标识符模式、运算符分类

表 12 词法单元分类算法伪码描述

```
//词法单元分类伪码
// 词法单元分类函数
function classifyTokens(tokens) {
    List<Token> classifiedTokens = [];


    for (Token token : tokens) {
        TokenType type = classifyToken(token);
        classifiedTokens.push(new Token(type, token.value, token.line,
token.column));
    }
    return classifiedTokens;
}
// 判断是否为数字字面量
function isNumberLiteral(string value) {
    return /^[0-9]+(\.[0-9]+)?$/ .test(value);
}
// 判断是否为字符串字面量
function isStringLiteral(string value) {
    return /^".*"$/.test(value);
}
// 判断是否为空白字符
function isWhitespace(string value) {
```

```
    return /^\s+$/.test(value);
}
// 判断是否为注释
function isComment(string value) {
    return /^\/\/.*$|^\/\*[\s\S]*?\*\/$/.test(value);
}
// 执行词法单元分类示例
List<Token> tokens = lexicalAnalysis(sourceCode);
List<Token> classifiedTokens = classifyTokens(tokens);
for (Token token : classifiedTokens) {
    console.log(`Token: ${token.type} | Value: "${token.value}" | Line:
${token.line} | Column: ${token.column}`);
}
```

- 应用高亮配置：根据高亮配置文件，为每个分类的词法单元应用相应的显示样式。
  a) 输入：分类后的词法单元、高亮配置
  b) 输出：高亮显示信息
  c) 算法：查找高亮配置、应用显示样式

表 13 应用高亮配置算法伪码描述

```
//应用高亮配置伪码
// 显示样式结构
struct HighlightStyle {
    string color;
    string fontWeight;
    string fontStyle;
    string textDecoration;
}
// 高亮显示函数
function applyHighlighting(tokens, highlightConfig) {
    List<string> highlightedTokens = [];
    for (Token token : tokens) {
        HighlightStyle style = highlightConfig[token.type];
```

```
        string highlightedToken = formatTokenWithStyle(token, style);
        highlightedTokens.push(highlightedToken);
    }
    return highlightedTokens;
}
// 格式化词法单元为带样式的字符串
function formatTokenWithStyle(Token token, HighlightStyle style) {
    return`<spanstyle="color:${style.color}; font-weight:${style.fontWeight};
font-style:${style.fontStyle};
text-decoration:${style.textDecoration};">${token.value}</span>`;
}
// 执行高亮示例
List<Token> tokens = classifyTokens(lexicalAnalysis(sourceCode));
List<string>highlightedTokens = applyHighlighting(tokens, highlightConfig);
for (string highlightedToken : highlightedTokens) {
    console.log(highlightedToken); // 或者将它们输出到 HTML 文档中
}
```

- 显示高亮效果：将高亮信息传递给 IDE，显示在编辑器中。
  a) 输入：高亮显示信息
  b) 输出：高亮显示的代码
  c) 算法：IDE 渲染引擎调用

表 14 显示高亮效果算法伪码描述

```
//显示高亮效果伪码
// 高亮显示函数
function applyHighlighting(tokens, highlightConfig) {
    List<HighlightRange> highlights = [];
    for (Token token : tokens) {
        HighlightStyle style = highlightConfig[token.type];
        HighlightRange range = createHighlightRange(token, style);
        highlights.push(range);
    }
```

```
    return highlights;
}
// 创建高亮范围
function createHighlightRange(Token token, HighlightStyle style) {
    return new HighlightRange(token.line, token.column, token.line,
token.column + token.value.length - 1, style.color);
}
// 传递高亮信息给 IDE
function sendHighlightingToIDE(highlights) {
    IDE.setHighlightRanges(highlights);
}
// 执行高亮示例
List<Token> tokens = classifyTokens(lexicalAnalysis(sourceCode));
List<HighlightRange>highlights = applyHighlighting(tokens, highlightConfig);
sendHighlightingToIDE(highlights);
```

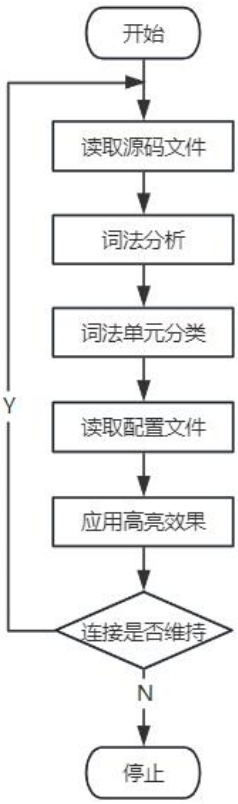

图 10 语法高亮模块流程图

4.2.3. 悬浮提示

1) 模块定义

悬浮提示模块用于在用户悬停在代码特定位置时显示详细信息，如变量类型、函数签名、文档注释等。该模块通过解析代码并提供上下文相关的信息，帮助开发者理解代码结构和功能。

2) 模块关联

- 词法分析模块：提供代码的词法单元，用于识别用户悬停位置的上下文信息。
- 语法分析模块：提供代码的语法树，用于确定悬停位置的语法结构。
- 语义分析模块：提供代码的语义信息，用于生成有意义的悬浮提示内容。
- IDE 集成模块：将悬浮提示功能集成到 IDE 中，确保提示信息在编辑器中显示。

3) 数据说明

- 源码文件：用户编写的 SysY2022E 语言代码文件。
- 词法单元：从源码文件中提取的基本词法单元，如标识符、关键字、运算符、常量和分隔符。
- 语法树：由语法分析模块生成的代码语法结构树。
- 语义信息：由语义分析模块生成的变量类型、函数签名等语义信息。
- 悬浮提示内容：根据用户悬停位置生成的提示信息。

4) 算法说明

- 获取悬停位置：监听用户在编辑器中的鼠标移动事件，确定悬停位置。
  a) 输入：鼠标位置
  b) 输出：源码文件中的字符位置
  c) 算法：事件监听

表 15 获取悬浮位置算法伪码描述

```
//获取悬停位置伪码
// 悬停位置结构
struct HoverPosition {
    int line;
    int column;
}
// 编辑器对象，假设它提供了事件监听接口
class Editor {
```

```
    // 监听鼠标移动事件
    function onMouseMove(callback) {
        this.editor.onDidChangeCursorPosition(callback);
    }
    // 获取鼠标位置对应的字符位置
    function getCharacterPosition(mousePosition) {
        return this.editor.getPosition(mousePosition);
    }
}
// 悬停处理器
class HoverHandler {
    Editor editor;
    // 初始化
    constructor(Editor editor) {
        this.editor = editor;
        this.editor.onMouseMove(this.handleMouseMove);
    }
    // 鼠标移动事件处理函数
    function handleMouseMove(mouseEvent) {
        // 处理悬停位置（例如，显示工具提示等）
        this.handleHoverPosition(position);
    }
}
// 示例：使用编辑器和悬停处理器
Editor editor = new Editor();
HoverHandler hoverHandler = new HoverHandler(editor);
```

- 词法单元查找：根据悬停位置，查找对应的词法单元。
    a) 输入：字符位置
    b) 输出：词法单元
    c) 算法：遍历词法单元列表

表 16 词法单元查找算法伪码描述

```
//词法单元查找伪码
// 查找词法单元函数
function findTokenAtPosition(tokens, HoverPosition hoverPosition) {
    for (Token token : tokens) {
        if (isPositionInToken(hoverPosition, token)) {
            return token;
        }
    }
    return null; // 未找到匹配的词法单元
}
// 编辑器对象，假设它提供了事件监听接口
class Editor {
    // 监听鼠标移动事件
    function onMouseMove(callback) {
        this.editor.onDidChangeCursorPosition(callback);
    }
    // 获取鼠标位置对应的字符位置
    function getCharacterPosition(mousePosition) {
        return this.editor.getPosition(mousePosition);
    }
}
// 悬停处理器
class HoverHandler {
    Editor editor;
    List<Token> tokens;
    // 初始化
    constructor(Editor editor, List<Token> tokens) {
        this.editor = editor;
        this.tokens = tokens;
        this.editor.onMouseMove(this.handleMouseMove);
    }
    // 处理找到的词法单元
```

```
    function handleTokenHover(Token token) {
        if (token != null) {
            console.log(`Hovering over token: ${token.value} of type: ${token.type}`);
            // 在此处执行需要的操作，例如显示工具提示
        } else {
            console.log("Hovering over an empty space.");
        }
    }
}
// 示例：使用编辑器和悬停处理器
Editor editor = new Editor();
List<Token> tokens = classifyTokens(lexicalAnalysis(sourceCode));
HoverHandler hoverHandler = new HoverHandler(editor, tokens);
```

- 语法结构解析：根据词法单元，获取其在语法树中的位置和上下文信息。
  a) 输入：词法单元
  b) 输出：语法树节点
  c) 算法：语法树遍历

表 17 词法结构解析算法伪码描述

```
//语法结构解析伪码
// 根据词法单元查找语法树节点
function findSyntaxNodeForToken(SyntaxNode root, Token targetToken) {
    if (isTokenInNode(root, targetToken)) {
        return root;
    }
    for (SyntaxNode child : root.children) {
        SyntaxNoderesult = findSyntaxNodeForToken(child, targetToken);
        if (result != null) {
            return result;
        }
    }
```

```
    return null;  // 未找到匹配的语法树节点
}
// 判断词法单元是否在语法树节点中
function isTokenInNode(SyntaxNode node, Token token) {
    return node.token != null &&
           node.token.startLine == token.startLine &&
           node.token.startColumn == token.startColumn &&
           node.token.endLine == token.endLine &&
           node.token.endColumn == token.endColumn;
}
// 获取语法树节点的上下文信息
function getSyntaxNodeContext(SyntaxNode node) {
    List<SyntaxNode> context = [];
    SyntaxNode current = node;
    while (current != null) {
        context.push(current);
        current = current.parent;
    }
   return context;  // 返回从当前节点到根节点的路径
}
// 词法单元查找处理器
class TokenHandler {
    SyntaxNode syntaxTreeRoot;
    // 初始化
    constructor(SyntaxNode syntaxTreeRoot) {
        this.syntaxTreeRoot = syntaxTreeRoot;
    }
    // 处理词法单元
    function handleToken(Token token) {
        SyntaxNode node = findSyntaxNodeForToken(this.syntaxTreeRoot, token);

        if (node != null) {
```

```
            List<SyntaxNode> context = getSyntaxNodeContext(node);
            this.printNodeContext(context);
        } else {
            console.log("Token not found in syntax tree.");
        }
    }
// 示例：使用语法树和词法单元查找处理器
SyntaxNode syntaxTreeRoot = parseSyntaxTree(sourceCode);
TokenHandler tokenHandler = new TokenHandler(syntaxTreeRoot);
```

- 生成提示内容：根据语法树节点，结合语义信息，生成悬浮提示内容。
  a) 输入：语法树节点、语义信息
  b) 输出：悬浮提示内容
  c) 算法：查找语义信息，格式化提示内容

表 18 生成提示内容算法伪码描述

```
//生成提示内容伪码
// 格式化悬浮提示内容
function formatHoverContent(SyntaxNodenode, SemanticInfo semanticInfo) {
    string content = "";
    content += `Node type: ${node.type}\n`;
    content += `Token value: ${node.token.value}\n`;
    content += `Description: ${semanticInfo.description}\n`;
    if (semanticInfo.references.size() > 0) {
        content += "References:\n";
        for (string reference : semanticInfo.references) {
            content += `- ${reference}\n`;
        }
    }
    return content;
}
// 悬停处理器
class HoverHandler {
```

```
    SyntaxNode syntaxTreeRoot;
    // 初始化
    constructor(SyntaxNode syntaxTreeRoot) {
        this.syntaxTreeRoot = syntaxTreeRoot;
    }
    // 处理词法单元
    function handleToken(Token token) {
        SyntaxNode node = findSyntaxNodeForToken(this.syntaxTreeRoot, token);
    // 显示悬浮提示内容
    function showHoverContent(string content) {
        displayHover(content);  // 示例
    }
}
// 示例：使用语法树和词法单元查找处理器
SyntaxNode syntaxTreeRoot = parseSyntaxTree(sourceCode);
HoverHandler hoverHandler = new HoverHandler(syntaxTreeRoot);
```

- 显示悬浮提示：将生成的提示内容显示在用户悬停的位置。
  a) 输入：悬浮提示内容
  b) 输出：悬浮提示框
  c) 算法：调用 IDE 渲染引擎

表 19 显示悬浮提示算法伪码描述

```
//显示悬浮提示伪码
// 悬停处理器
class HoverHandler {
    SyntaxNode syntaxTreeRoot;
    Editor editor;
    // 初始化
    constructor(SyntaxNode syntaxTreeRoot, Editor editor) {
        this.syntaxTreeRoot = syntaxTreeRoot;
        this.editor = editor;
    }
```

```
    // 处理词法单元
    function handleToken(Token token) {
        SyntaxNode node = findSyntaxNodeForToken(this.syntaxTreeRoot, token);
        if (node != null) {
            SemanticInfo semanticInfo = getSemanticInfo(node);
            string hoverContent = formatHoverContent(node, semanticInfo);
            this.showHoverContent(token, hoverContent);
        } else {
            console.log("Token not found in syntax tree.");
        }
    }
}
// 编辑器对象，假设它提供了显示悬浮提示内容的方法
class Editor {
    // 显示悬浮提示内容
    function displayHover(HoverPosition position, string content) {
     this.editor.showHover(position, content);
    }
// 示例：使用语法树和词法单元查找处理器
SyntaxNode syntaxTreeRoot = parseSyntaxTree(sourceCode);
Editor editor = new Editor();
HoverHandler hoverHandler=new HoverHandler(syntaxTreeRoot, editor);
```

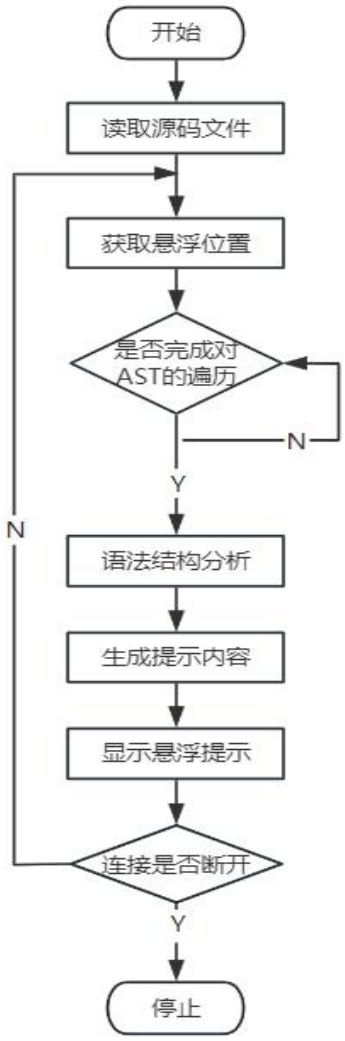

图 11 悬浮提示模块流程图

4.3. 编辑器核心

4.3.1. 语法检查与错误提示

1) 模块定义

语法检查与错误提示模块用于检查源代码中的语法错误，并在编辑器中提供即时的错误提示。该模块通过解析代码、识别语法错误并生成诊断信息，帮助开发者在编写代码时快速发现并修正语法错误。

2) 模块关联

- 词法分析模块：提供代码的词法单元，用于语法检查的基础。

- 语法分析模块：提供代码的语法树，用于语法结构的检查。
- IDE 集成模块：将语法检查与错误提示功能集成到 IDE 中，确保错误提示在编辑器中显示。
- 悬浮提示模块：在错误提示的同时提供详细的错误说明和修正建议。

3) 数据说明

- 源码文件：用户编写的 SysY2022E 语言代码文件。
- 词法单元：从源码文件中提取的基本词法单元，如标识符、关键字、运算符、常量和分隔符。
- 语法树：由语法分析模块生成的代码语法结构树。
- 诊断信息：包括错误位置、错误类型和错误消息的详细信息。
- 错误提示内容：在编辑器中显示的错误提示信息。

4) 算法说明

- 获取源码文件：监听用户在编辑器中保存或修改代码的事件，获取当前源码文件。
  a) 输入：编辑器事件
  b) 输出：源码文件
  c) 算法：事件监听

表 20 获取源码文件算法伪码描述

```
//获取源文件伪码
// 定义一个函数来初始化事件监听器
function initializeEditorEventListeners(editor) {
    // 监听保存事件
    editor.on('save', function(event) {
        // 获取当前源码文件
        currentSourceFile = getSourceCodeFile(event);
        // 打印当前源码文件路径或执行其他操作
        print("File saved: " + currentSourceFile.path);
    });
    // 监听修改事件
    editor.on('change', function(event) {
        // 获取当前源码文件
        currentSourceFile = getSourceCodeFile(event);
        // 打印当前源码文件路径或执行其他操作
```

```
        print("File changed: " + currentSourceFile.path);
    });
}
// 获取当前源码文件的方法实现
function getSourceCodeFile(event) {
    // 假设事件对象包含文件信息
    // 例如，事件对象的 file 属性包含文件路径和内容等信息
    return event.file;
}
// 示例：初始化事件监听器
var editor = getEditorInstance();
initializeEditorEventListeners(editor);
```

- 词法分析：对源码文件进行词法分析，生成词法单元。
  a) 输入：源码文件
  b) 输出：词法单元列表
  c) 算法：词法分析器
- 语法分析：对词法单元进行语法分析，生成语法树。
  a) 输入：词法单元列表
  b) 输出：语法树
  c) 算法：语法分析器
- 语法检查：遍历语法树，检查语法规则是否符合语言规范，生成诊断信息。
  a) 输入：语法树
  b) 输出：诊断信息列表
  c) 算法：语法检查器

表 21 语法检查算法伪码描述

```
//语法检查伪码
//诊断信息类
class Diagnostic:
    def __init__(self, message, node):
        self.message = message
        self.node = node
```

```
//语法检查器类
class SyntaxChecker:
    def __init__(self):
        self.diagnostics = []
    def check(self, node):
        if node.node_type == 'BIN_OP':
            self.check_bin_op(node)
        else node.node_type == 'INTEGER':
            self.check_integer(node)
    def check_bin_op(self, node):
        if len(node.children) != 2:
       self.diagnostics.append(Diagnostic("Binary operator node must have exactly 2 children", node))
        allowed_ops = ['+', '-', '*', '/']
        if node.value not in allowed_ops:
      self.diagnostics.append(Diagnostic(f"Invalid operator: {node.value}", node))
    def check_integer(self, node):
        if not isinstance(node.value, int):
       self.diagnostics.append(Diagnostic("Integer node must have an integer value", node))
    def get_diagnostics(self):
        return self.diagnostics
//示例使用
ast = ASTNode('BIN_OP', '+')
left_child = ASTNode('INTEGER', 3)
right_child = ASTNode('BIN_OP', '*')
right_child_left = ASTNode('INTEGER', 5)
right_child_right = ASTNode('INTEGER', 2)
right_child.children = [right_child_left, right_child_right]
ast.children = [left_child, right_child]
```

- 生成错误提示：根据诊断信息生成错误提示内容，并在编辑器中显示。

a) 输入：诊断信息列表

b) 输出：错误提示内容

c) 算法：格式化诊断信息

表 22 生成错误提示算法伪码描述

```
//生成错误提示伪码
//错误提示类
class ErrorHint:
    def __init__(self, message, line, column):
        self.message = message
        self.line = line
        self.column = column
//格式化诊断信息函数
def format_diagnostics(diagnostics):
    error_hints = []
    for diagnostic in diagnostics:
        // 假设每个节点包含行和列信息
        line = diagnostic.node.line if hasattr(diagnostic.node, 'line') else
None
        column = diagnostic.node.column if hasattr(diagnostic.node, 'column')
else None
        error_hint = ErrorHint(diagnostic.message, line, column)
        error_hints.append(error_hint)
    return error_hints
//显示错误提示函数
def display_error_hints(error_hints):
    for hint in error_hints:
        if hint.line is not None and hint.column is not None:
            print(f"Error at line {hint.line}, column {hint.column}:
{hint.message}")
        else:
            print(f"Error: {hint.message}")
//示例使用
```

```
checker = SyntaxChecker()
checker.check(ast)
diagnostics = checker.get_diagnostics()
error_hints = format_diagnostics(diagnostics)
display_error_hints(error_hints)
```

- 更新编辑器：将错误提示内容更新到编辑器界面中。
  a) 输入：错误提示内容
  b) 输出：编辑器界面更新
  c) 算法：调用 IDE 接口

表 23 更新编辑器算法伪码描述

```
//更新编辑器伪码
// 错误提示类
class ErrorHint:
    def __init__(self, message, line, column):
        self.message = message
        self.line = line
        self.column = column
// 格式化诊断信息函数
def format_diagnostics(diagnostics):
    error_hints = []
    for diagnostic in diagnostics:
        column = diagnostic.node.column if hasattr(diagnostic.node, 'column')
else None
        error_hint = ErrorHint(diagnostic.message, line, column)
        error_hints.append(error_hint)
    return error_hints
// 编辑器接口类（假设存在）
class EditorInterface:
    def __init__(self):
        // 假设编辑器有一个方法来清除现有错误
        self.clear_errors()
```

```
    def clear_errors(self):
        // 清除编辑器中现有的错误提示
        print("Clearing existing errors in the editor")
    def add_error(self, message, line, column):
        // 向编辑器添加新的错误提示
        if line is not None and column is not None:
            print(f"Adding error at line {line}, column {column}: {message}")
        else:
            print(f"Adding error: {message}")
    def update_errors(self, error_hints):
        self.clear_errors()
        for hint in error_hints:
            self.add_error(hint.message, hint.line, hint.column)
// 显示错误提示函数（更新至编辑器）
def display_error_hints_in_editor(editor_interface, error_hints):
    editor_interface.update_errors(error_hints)
// 示例使用
checker = SyntaxChecker()
checker.check(ast)
diagnostics = checker.get_diagnostics()
error_hints = format_diagnostics(diagnostics)
editor_interface = EditorInterface()
display_error_hints_in_editor(editor_interface, error_hints)
```

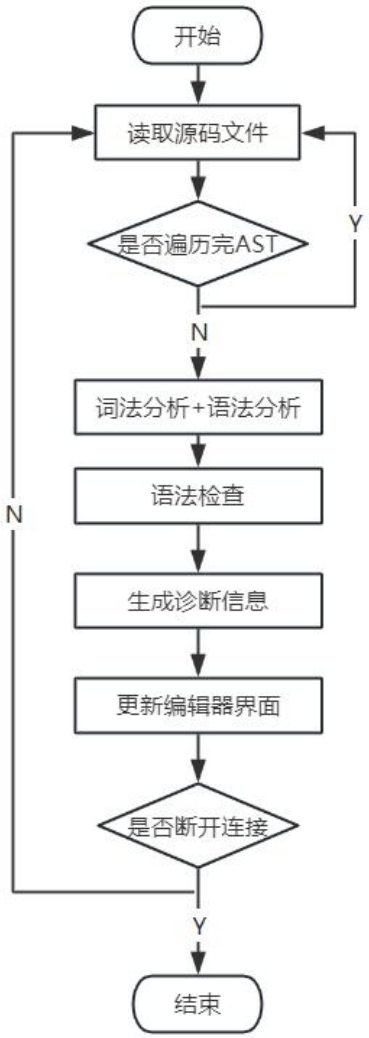

图 12 语法检查与错误提示模块流程图

4.3.2. 静态语义检查

1) 模块定义

静态语义检查模块用于在不执行程序的情况下对源代码进行语义分析，确保代码遵循语言的语义规则。该模块检查变量的定义和使用、类型匹配、作用域规则等，以发现潜在的语义错误。

2) 模块关联

- 词法分析模块：提供代码的词法单元，用于语义检查的基础。
- 语法分析模块：提供代码的语法树，用于语义结构的检查。
- 符号表管理模块：管理符号表，用于存储变量、函数等标识符的信息。
- IDE 集成模块：将静态语义检查功能集成到 IDE 中，确保语义错误提示在编辑器中显示。

- 语法检查与错误提示模块：结合语法检查结果，提供更全面的错误提示。

3) 数据说明

- 源码文件：用户编写的 SysY2022E 语言代码文件。
- 词法单元：从源码文件中提取的基本词法单元，如标识符、关键字、运算符、常量和分隔符。
- 语法树：由语法分析模块生成的代码语法结构树。
- 符号表：记录变量、函数等标识符的定义和属性。
- 诊断信息：包括错误位置、错误类型和错误消息的详细信息。
- 错误提示内容：在编辑器中显示的错误提示信息。

4) 算法说明

- 获取源码文件：监听用户在编辑器中保存或修改代码的事件，获取当前源码文件。
  a) 输入：编辑器事件
  b) 输出：源码文件
  c) 算法：事件监听
- 词法分析：对源码文件进行词法分析，生成词法单元。
  a) 输入：源码文件
  b) 输出：词法单元列表
  c) 算法：词法分析器
- 语法分析：对词法单元进行语法分析，生成语法树。
  a) 输入：词法单元列表
  b) 输出：语法树
  c) 算法：语法分析器
- 构建符号表：遍历语法树，收集变量和函数定义，构建符号表。
  a) 输入：语法树
  b) 输出：符号表
  c) 算法：符号表构建器

表 24 构建符号表算法伪码描述

```
//构建符号表伪码
// 符号表构建器类
class SymbolTableBuilder:
    def __init__(self):
        self.symbol_table = SymbolTable()
```

```
        self.current_scope = "global"
    def build(self, node):
        if node.node_type == 'FUNCTION_DEF':
            self.add_function(node)
        elif node.node_type == 'VARIABLE_DECL':
            self.add_variable(node)
        else:
            // 遍历子节点
            for child in node.children:
                self.build(child)
    def add_function(self, node):
        function_name = node.name
        self.symbol_table.add_entry(entry)
        // 更新作用域并遍历函数体
        previous_scope = self.current_scope
        self.current_scope = function_name
        for child in node.children:
            self.build(child)
        // 恢复作用域
        self.current_scope = previous_scope
// 示例使用
ast = ASTNode('PROGRAM')
function_node = ASTNode('FUNCTION_DEF', name='myFunction', line=1, column=1)
variable_node = ASTNode('VARIABLE_DECL', name='x', line=2, column=5)
function_node.children = [variable_node]
ast.children = [function_node]
builder = SymbolTableBuilder()
builder.build(ast)
symbol_table = builder.symbol_table
```

- 语义检查：遍历语法树，检查变量的定义和使用、类型匹配、作用域规则等，生成诊断信息。

  a) 输入：语法树、符号表

b) 输出：诊断信息列表

c) 算法：语义检查器

表 25 语义检查算法伪码描述

```
//语义检查伪码
// 语义检查器类
class SemanticChecker:
    def __init__(self, symbol_table):
        self.symbol_table = symbol_table
        self.diagnostics = []
        self.current_scope = "global"
    def check(self, node):
        if node.node_type == 'FUNCTION_DEF':
            self.check_function(node)
        elif node.node_type == 'VARIABLE_DECL':
            self.check_variable_declaration(node)
        elif node.node_type == 'VARIABLE_USE':
            self.check_variable_use(node)
        elif node.node_type == 'EXPRESSION':
            self.check_expression(node)
        else:
            // 遍历子节点
            for child in node.children:
                self.check(child)
    def check_function(self, node):
        function_name = node.name
        self.current_scope = function_name
        for child in node.children:
            self.check(child)
        self.current_scope = "global"
    def check_variable_use(self, node):
        variable_name = node.name
        entry = self.symbol_table.find_in_any_scope(variable_name)
```

```
        if not entry:
            self.diagnostics.append(Diagnostic(
                f"Variable '{variable_name}' is used but not defined.",
                node.line, node.column
semantic_checker.check(ast)
```

- 生成错误提示：根据诊断信息生成错误提示内容，并在编辑器中显示。
  a) 输入：诊断信息列表
  b) 输出：错误提示内容
  c) 算法：格式化诊断信息
- 更新编辑器：将错误提示内容更新到编辑器界面中。
  a) 输入：错误提示内容
  b) 输出：编辑器界面更新
  c) 算法：调用 IDE 接口

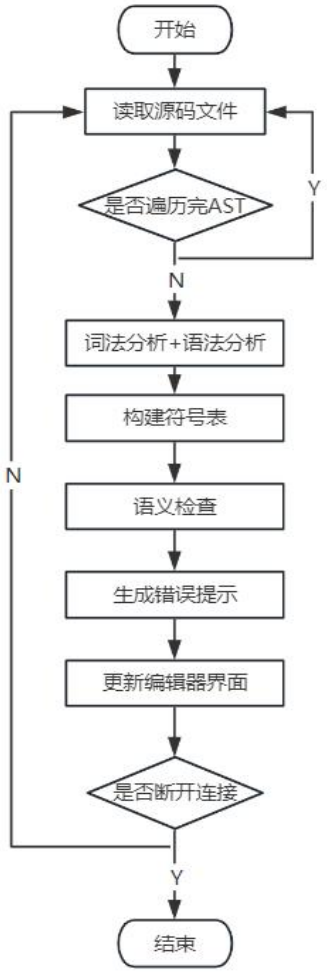

图 13 静态语义检查模块流程图

4.4. 分析与修复

4.4.1. 修复建议与自动修复

1) 模块定义

修复建议与自动修复模块用于分析代码中的语法和语义错误，并提供相应的修复建议。该模块还可以根据用户选择，自动应用修复建议，从而提高代码的正确性和开发效率。

2) 模块关联

- 词法分析模块：提供代码的词法单元，用于错误分析的基础。
- 语法分析模块：提供代码的语法树，用于语法结构的检查。
- 静态语义检查模块：提供代码的语义错误信息。
- IDE 集成模块：将修复建议和自动修复功能集成到 IDE 中，确保在编辑器中显示并应用修复建议。
- 用户交互模块：提供用户界面，用于显示修复建议并接受用户的修复操作。

3) 数据说明

- 源码文件：用户编写的 SysY2022E 语言代码文件。
- 词法单元：从源码文件中提取的基本词法单元，如标识符、关键字、运算符、常量和分隔符。
- 语法树：由语法分析模块生成的代码语法结构树。
- 诊断信息：包括错误位置、错误类型和错误消息的详细信息。
- 修复建议：针对每个诊断信息生成的修复方案。
- 用户选择：用户在编辑器中选择的修复建议。
- 修复操作：自动应用的修复操作。

4) 说明

- 获取诊断信息：从静态语义检查模块获取诊断信息。
  a) 输入：诊断信息列表
  b) 输出：诊断信息
  c) 算法：获取诊断信息

表 26 获取诊断信息算法伪码描述

```
//获取诊断信息伪码
// 诊断信息类
class Diagnostic:
```

```
    def __init__(self, message, line, column):
        self.message = message
        self.line = line
        self.column = column
// 获取诊断信息函数
def get_diagnostics(diagnostics_list):
    diagnostics_output = []

    for diagnostic in diagnostics_list:
        // 格式化诊断信息
        formatted_diagnostic = format_diagnostic(diagnostic)
        diagnostics_output.append(formatted_diagnostic)
    return diagnostics_output
// 格式化诊断信息函数
def format_diagnostic(diagnostic):
    return f"Line {diagnostic.line}, Column {diagnostic.column}:
{diagnostic.message}"
// 示例使用
diagnostics_list = [
    Diagnostic("Variable 'x' is not defined.", 10, 5),
    Diagnostic("Type mismatch in assignment.", 15, 8)
]
```

- 生成修复建议：根据诊断信息生成修复建议。
  a) 输入：诊断信息列表
  b) 输出：修复建议列表
  c) 算法：修复建议生成器

表 27 生成修复建议算法伪码描述

```
//生成修复建议伪码
// 诊断信息类
class Diagnostic:
    def __init__(self, message, line, column):
```

```
        self.message = message
        self.line = line
        self.column = column
// 修复建议类
class Suggestion:
    def __init__(self, message, line, column):
        self.message = message
        self.line = line
        self.column = column
// 修复建议生成器函数
def generate_suggestions(diagnostics_list):
    suggestions_list = []
    for diagnostic in diagnostics_list:
        // 基于诊断信息类型生成建议
        suggestion = create_suggestion_based_on_diagnostic(diagnostic)
        suggestions_list.append(suggestion)
    return suggestions_list
// 格式化修复建议函数
def format_suggestion(suggestion):
    return f"Line {suggestion.line}, Column {suggestion.column}:
{suggestion.message}"

// 示例使用
diagnostics_list = [
    Diagnostic("Variable 'x' is not defined.", 10, 5),
    Diagnostic("Type mismatch in assignment.", 15, 8),
    Diagnostic("Variable 'y' is already defined.", 20, 3)
]
```

- 显示修复建议：在 IDE 中显示修复建议，供用户选择。
  a) 输入：修复建议列表
  b) 输出：用户选择
  c) 算法：显示修复建议

表 28 显示修复建议算法伪码描述

```
//显示修复建议伪码
// 修复建议类
class Suggestion:
    def __init__(self, message, line, column):
        self.message = message
        self.line = line
        self.column = column
// 显示修复建议函数
def display_suggestions(suggestions_list):
    print("修复建议列表:")
    for index, suggestion in enumerate(suggestions_list, start=1):
        print(f"{index}. {format_suggestion(suggestion)}")
    // 获取用户选择
    user_choice = get_user_choice(len(suggestions_list))
    return suggestions_list[user_choice - 1]
// 格式化修复建议函数
def format_suggestion(suggestion):
   returnf"Line{suggestion.line},Column{suggestion.column}:
{suggestion.message}"
// 示例使用
suggestions_list = [
    Suggestion("Consider declaring the variable or checking if the name is
misspelled.", 10, 5),
    Suggestion("Ensure that both sides of the expression have the same type.",
15, 8),
    Suggestion("Try renaming the variable to avoid naming conflicts.", 20, 3)
]
```

- 应用修复操作：根据用户选择，自动应用修复操作。
  a) 输入：用户选择
  b) 输出：修复后的代码
  c) 算法：自动修复器

表 29 应用修复建议算法伪码描述

```
//应用修复操作伪码
// 修复建议类
class Suggestion:
    def __init__(self, message, line, column):
        self.message = message
        self.line = line
        self.column
// 代码文件类
class CodeFile:
    def __init__(self, lines):
        self.lines = lines
    def apply_suggestion(self, suggestion):
        if "declaring the variable" in suggestion.message:
            self.declare_variable(suggestion.line, suggestion.column)
        elif "ensure that both sides of the expression have the same type" in
suggestion.message:
            self.fix_type_mismatch(suggestion.line)
        elif "renaming the variable" in suggestion.message:
            self.rename_variable(suggestion.line, suggestion.column)
        else:
            print("无法应用这个修复建议。")
    def declare_variable(self, line, column):
        // 在指定位置前插入变量声明
        declaration = "var x;\n"
        self.lines.insert(line - 1, declaration)
    def fix_type_mismatch(self, line):
        // 修改指定行以确保类型匹配
        self.lines[line - 1] = self.lines[line - 1].replace("=", " = (int)")
    def rename_variable(self, line, column):
        // 重命名变量，假设新名字为"newVariable"
        self.lines[line - 1] = self.lines[line - 1].replace("y", "newVariable")
```

```
// 格式化修复建议函数
def format_suggestion(suggestion):
    return f"Line {suggestion.line}, Column {suggestion.column}: {suggestion.message}"
// 显示修复建议并获取用户选择函数
def display_suggestions(suggestions_list):
    print("修复建议列表:")
    for index, suggestion in enumerate(suggestions_list, start=1):
        print(f"{index}. {format_suggestion(suggestion)}")
// 示例使用
code_lines = [
    "int main() {",
    "    x = 10;",
    "    y = 20;",
    "    return 0;",
    "}"
]
```

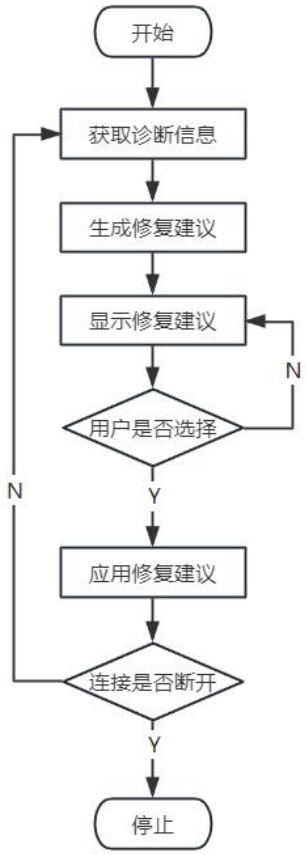

图 14 修复建议与自动修复模块流程图

4.4.2. 程序错误检查

1) 模块定义

程序错误检查模块用于检测代码中的常见错误，如无用变量、死循环、数组越界等，以提高代码的质量和可靠性。

2) 模块关联

- 静态语义检查模块：提供代码的语义错误信息。
- IDE 集成模块:将程序错误检查功能集成到 IDE 中,确保在编辑器中显示错误信息。
- 用户交互模块：提供用户界面，用于显示错误信息和接受用户的修复操作。

3) 数据说明

- 源码文件：用户编写的 SysY2022E 语言代码文件。
- 词法单元：从源码文件中提取的基本词法单元，如标识符、关键字、运算符、常量和分隔符。
- 语法树：由语法分析模块生成的代码语法结构树。
- 符号表：记录变量、函数等标识符的定义和属性。
- 诊断信息：包括错误位置、错误类型和错误消息的详细信息。

4) 算法说明

- 检查无用变量：遍历语法树和符号表，检查未使用的变量。
  a) 输入：语法树、符号表
  b) 输出：无用变量列表
  c) 算法：无用变量检查器

表 30 检查无用变量算法伪码描述

```
//检查无用变量伪码
# 节点类（表示语法树中的节点）
class Node:
    def __init__(self, type, children=None, value=None):
        self.type = type
        self.children = children if children is not None else []
        self.value = value
# 符号表类
```

```
class SymbolTable:
    def __init__(self):
        self.table = {}
    def add_variable(self, name):
        self.table[name] = True
    def mark_as_used(self, name):
        if name in self.table:
            del self.table[name]
    def get_unused_variables(self):
        return self.table.keys()
# 无用变量检查器函数
def check_unused_variables(ast, symbol_table):
    # 遍历语法树并标记变量的使用情况
   traverse_ast_and_mark_usage(ast, symbol_table)
    # 获取未使用的变量
    unused_variables = symbol_table.get_unused_variables()
    return unused_variables
# 示例使用
# 构建示例语法树
ast = Node('Program', [
    Node('VariableDeclaration', value='x'),
    Node('VariableDeclaration', value='y'),
    Node('Assignment', children=[
        Node('VariableUsage', value='x'),
        Node('Literal', value='10')
    ])
])
```

- 检查死循环：遍历语法树，检查可能导致死循环的代码结构。
  a) 输入：语法树
  b) 输出：死循环位置列表
  c) 算法：死循环检查器

表 31 检查死循环算法伪码描述

```
//检查死循环伪码
# 死循环检查器函数
def check_dead_loops(ast):
    dead_loops = []
    # 遍历语法树
    def traverse_ast(node):
        nonlocal dead_loops
        if node.type == 'LoopStatement':
            # 检查循环条件
            if is_loop_condition_always_true(node.children[0]):
                dead_loops.append(node.children[0])
        for child in node.children:
            traverse_ast(child)
    # 开始遍历语法树
    traverse_ast(ast)
    return dead_loops
# 示例使用
# 构建示例语法树
ast = Node('Program', [
    Node('LoopStatement', children=[
        Node('Expression', value='x < 10'),  # 假设这个条件总是为真
        Node('Block', children=[
            Node('Statement'),
        ])
    ])
])
# 检查死循环
dead_loops = check_dead_loops(ast)
# 输出死循环位置
print("死循环位置:", dead_loops)
```

- 检查数组越界：遍历语法树和符号表，检查数组访问是否越界。
  a) 输入：语法树、符号表

b) 输出：数组越界位置列表

c) 算法：数组越界检查器

表 32 检查数组越界算法伪码描述

```
//检查数组越界伪码
# 数组越界检查器函数
def check_array_out_of_bounds(ast, symbol_table):
    out_of_bounds_accesses = []
    # 遍历语法树
    def traverse_ast(node, symbol_table):
        nonlocal out_of_bounds_accesses
        if node.type == 'ArrayAccess':
            # 获取数组名和索引
            array_name = node.children[0].value
            index = node.children[1].value
            # 获取数组大小
            array_size = symbol_table.get_array_size(array_name)
            # 检查索引是否越界
            if index < 0 or index >= array_size:
                out_of_bounds_accesses.append((array_name, index))
        for child in node.children:
            traverse_ast(child, symbol_table)
    # 开始遍历语法树
    traverse_ast(ast, symbol_table)
    return out_of_bounds_accesses
# 示例使用
# 构建示例语法树
ast = Node('Program', [
    Node('ArrayDeclaration', value='myArray', children=[
        Node('Literal', value='10')  # 假设数组大小为 10
    ]),
    Node('ArrayAccess', children=[
        Node('VariableUsage', value='myArray'),
```

```
        Node('Literal', value='15')  # 假设这个索引会越界
    ])
])
symbol_table = SymbolTable()
symbol_table.add_array('myArray', 10)
# 检查数组越界
out_of_bounds_accesses = check_array_out_of_bounds(ast, symbol_table)
# 输出数组越界位置
print("数组越界位置:", out_of_bounds_accesses)
```

- 生成诊断信息：根据检查结果生成诊断信息。
  a) 输入：无用变量列表、死循环位置列表、数组越界位置列表
  b) 输出：诊断信息列表
  c) 算法：诊断信息生成器

表 33 生成诊断信息算法伪码描述

```
//生成诊断信息伪码
// 定义诊断信息生成器函数
function generateDiagnostics(unusedVars: List<Tuple<String, Int, Int>>,
deadLoops: List<Tuple<Int, Int>>, outOfBounds: List<Tuple<Int, Int>>):
List<Diagnostic> {
    var diagnostics = []
    // 处理无用变量
    for var in unusedVars {
        var name = var[0]
        var line = var[1]
        var column = var[2]
        var message = "Unused variable: " + name
        diagnostics.append(Diagnostic(message, line, column))
    }
    // 处理死循环
    for deadLoop in deadLoops {
        var line = deadLoop[0]
```

```
        var column = deadLoop[1]
        var message = "Potential infinite loop detected"
        diagnostics.append(Diagnostic(message, line, column))
    }
    // 处理数组越界
    for outOfBound in outOfBounds {
        var line = outOfBound[0]
        var column = outOfBound[1]
        var message = "Array index out of bounds"
        diagnostics.append(Diagnostic(message, line, column))
    }
    return diagnostics
}
// 示例：输入检查结果，生成诊断信息
var unusedVars = [Tuple("x", 10, 5), Tuple("y", 15, 8)] // 示例数据
var deadLoops = [Tuple(20, 3)] // 示例数据
var outOfBounds = [Tuple(30, 12)] // 示例数据
var diagnostics=generateDiagnostics(unusedVars,deadLoops, outOfBounds)
for diagnostic in diagnostics {
    print("Line " + diagnostic.line + ", Column " + diagnostic.column + ": " +
diagnostic.message)
}
```

- 显示错误信息：在 IDE 中显示错误信息。
  a) 输入：诊断信息列表
  b) 输出：显示错误信息
  c) 算法：显示错误信息

表 34 显示错误信息算法伪码描述

```
//显示错误信息伪码
// 定义诊断信息结构
class Diagnostic {
    message: String
```

```
    line: Int
    column: Int
    function Diagnostic(message: String, line: Int, column: Int) {
        this.message = message
        this.line = line
        this.column = column
    }
}
// 定义 IDE 显示错误信息的函数
function displayErrors(diagnostics: List<Diagnostic>) {
    for diagnostic in diagnostics {
        IDE.showError(diagnostic.message, diagnostic.line, diagnostic.column)
    }
}
class IDE {
    static function showError(message: String, line: Int, column: Int) {
        print("Error at Line " + line + ", Column " + column + ": " + message)
    }
}
// 示例：生成一些诊断信息并在 IDE 中显示
var diagnostics = [
    Diagnostic("Unused variable 'x'", 10, 5),
    Diagnostic("Potential infinite loop detected", 20, 3),
    Diagnostic("Array index out of bounds", 30, 12)
]
displayErrors(diagnostics)
```

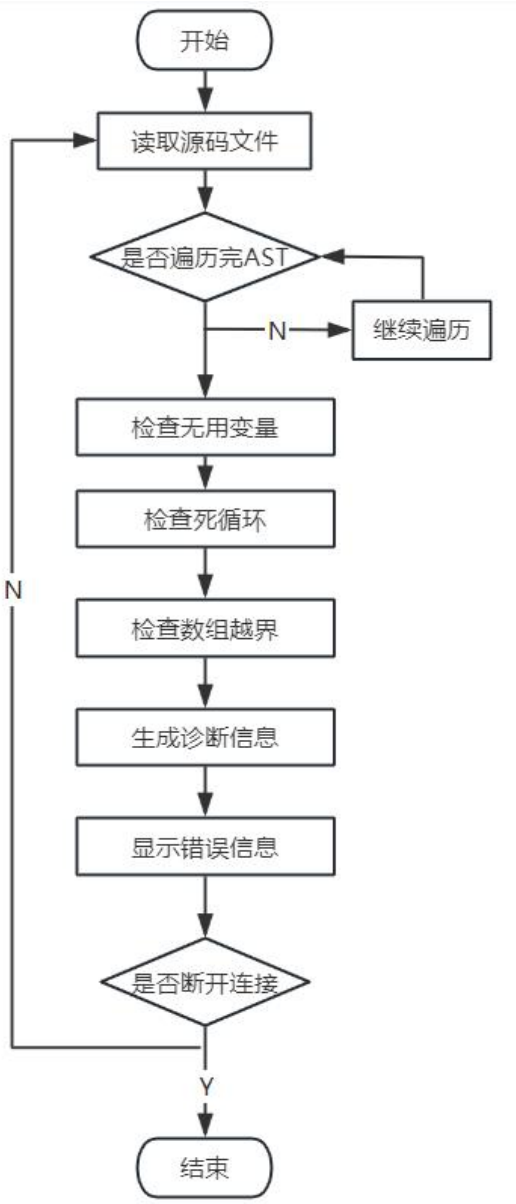

图 15 程序错误检查模块流程图

4.5. 重构引擎

1) 模块定义

重构引擎模块用于提供代码重构功能，包括变量/函数改名、抽取新函数、内联函数、提取局部变量等，以提升代码的可读性、可维护性和重用性。

2) 模块关联

- 语法分析模块：解析代码，生成语法树和符号表。
- 语义分析模块：提供代码的语义信息，确保重构后的代码仍然正确。
- IDE 集成模块：将重构功能集成到 IDE 中，提供用户接口和重构操作。

- 用户交互模块：提供用户界面，用于接受用户的重构操作指令并显示结果。

3) 数据说明

- 源码文件：用户编写的 SysY2022E 语言代码文件。
- 语法树：由语法分析模块生成的代码语法结构树。
- 符号表：记录变量、函数等标识符的定义和属性。
- 重构请求：用户发起的重构操作请求，包括重构类型和具体参数。
- 重构结果：重构操作的结果，包括重构后的代码和更新的语法树、符号表等。

4) 算法说明

- 变量/函数改名

a) 输入：语法树、符号表、重构请求（旧名称，新名称）

b) 输出：重构后的语法树

c) 算法：遍历语法树，找到所有使用旧名称的地方，并替换为新名称。

表 35 变量/函数改名算法伪码描述

```
//变量/函数改名伪码
// 定义语法树节点结构
class ASTNode {
    type: String
    value: String
    children: List<ASTNode>
    function ASTNode(type: String, value: String = "") {
        this.type = type
        this.value = value
        this.children = []
    }
// 定义符号表条目结构
class SymbolTableEntry {
    name: String
    type: String
    scope: String
    function SymbolTableEntry(name: String, type: String, scope: String) {
        this.name = name
        this.type = type
```

```
        this.scope = scope
    }
}
// 定义重构请求结构
class RefactorRequest {
    oldName: String
    newName: String
    function RefactorRequest(oldName: String, newName: String) {
        this.oldName = oldName
        this.newName = newName
    }
}
// 示例：重命名变量/函数
var root = ASTNode("PROGRAM")
var symbolTable = [
    SymbolTableEntry("x", "variable", "global"),
    SymbolTableEntry("myFunction", "function", "global")
]
var request = RefactorRequest("x", "y")
refactorAST(root, symbolTable, request)
```

- 抽取新函数
  a) 输入：语法树、符号表、重构请求（待抽取的代码段，函数名，参数）
  b) 输出：重构后的语法树
  c) 算法：将指定代码段替换为函数调用，并在合适位置插入新函数的定义。

表 36 抽取新函数算法伪码描述

```
//抽取新函数伪码
//遍历并重构语法树
 function    refactorAST(root:ASTNode,symbolTable:    List<SymbolTableEntry>,
request: RefactorRequest) {
    if root == null {
        return
```

```
    }
    // 替换代码段为函数调用
    replaceCodeSegment(root,request.codeSegment,        request.functionName,
request.parameters)
    insertNewFunction(root,     request.functionName,     request.parameters,
request.codeSegment)
    symbolTable.append(SymbolTableEntry(request.functionName,     "function",
"global"))
}
// 替换代码段为函数调用
function replaceCodeSegment(node: ASTNode, codeSegment: ASTNode, functionName:
String, parameters: List<String>) {
    if node == null {
        return
    }
    // 递归处理子节点
    for i in 0...node.children.length {
        replaceCodeSegment(node.children[i],codeSegment,        functionName,
parameters)
    }
}
    // 递归处理子节点
    for child in node.children {
        insertNewFunction(child,functionName, parameters, codeSegment)
    }
}
// 示例：抽取新函数
var root = ASTNode("PROGRAM")
var symbolTable = [
    SymbolTableEntry("x", "variable", "global"),
    SymbolTableEntry("myFunction", "function", "global")
]
```

```
var codeSegment = ASTNode("BLOCK")
```

- 内联函数
  a) 输入：语法树、符号表、重构请求（函数名）
  b) 输出：重构后的语法树
  c) 算法：遍历语法树，找到所有函数调用，并替换为函数体内容。

表 37 内联函数算法伪码描述

```
//内联函数伪码
// 遍历并重构语法树
function refactorAST(root:ASTNode, symbolTable: List<SymbolTableEntry>,
request: RefactorRequest) {
    if root == null {
        return
    }
 // 替换所有函数调用为函数体
    inlineFunctionCalls(root, request.functionName, targetFunction.body)
}
// 查找符号表中的目标函数
function findFunctionInSymbolTable(symbolTable: List<SymbolTableEntry>,
functionName: String): SymbolTableEntry {
    for entry in symbolTable {
        if entry.name == functionName and entry.type == "function" {
            return entry
        }
    }
    return null
}
// 替换所有函数调用为函数体
function inlineFunctionCalls(node: ASTNode, functionName: String, functionBody:
ASTNode) {
    if node == null {
        return
```

```
    }
// 示例：内联函数
var root = ASTNode("PROGRAM")
var functionBody = ASTNode("BLOCK")
var symbolTable = [
    SymbolTableEntry("x", "variable", "global"),
    SymbolTableEntry("myFunction", "function", "global", functionBody)
]
var request = RefactorRequest("myFunction")
refactorAST(root, symbolTable, request)
```

- 提取局部变量
  a) 输入：语法树、符号表、重构请求（表达式，变量名）
  b) 输出：重构后的语法树
  c) 算法：将指定表达式替换为变量，并在适当位置插入变量声明和赋值。

表 38 提取局部变量算法伪码描述

```
//提取局部变量伪码
// 定义语法树节点结构
class ASTNode {
    type: String
    value: String
    children: List<ASTNode>
    function ASTNode(type: String, value: String = "") {
        this.type = type
        this.value = value
        this.children = []
    }
    function addChild(child: ASTNode) {
        this.children.append(child)
    }
}
// 替换所有匹配的表达式为变量
```

```
function replaceExpressionWithVariable(node: ASTNode, expression: ASTNode,
variableName: String) {
    if node == null {
        return
    }
    // 检查当前节点是否匹配表达式
    if compareAST(node, expression) {
        node.type = "IDENTIFIER"
        node.value = variableName
        node.children = []
        return
    }
}
// 深拷贝 AST 节点
function deepCopyAST(node: ASTNode): ASTNode {
    if node == null {
        return null
    }
    var newNode = ASTNode(node.type, node.value)
    for child in node.children {
        newNode.addChild(deepCopyAST(child))
    }
    return newNode
}
var root = ASTNode("PROGRAM")
var expression = ASTNode("EXPRESSION")
var symbolTable = [
    SymbolTableEntry("x", "variable", "global"),
    SymbolTableEntry("myFunction", "function", "global")
]
var request = RefactorRequest(expression, "tempVar")
refactorAST(root, symbolTable, request)
```

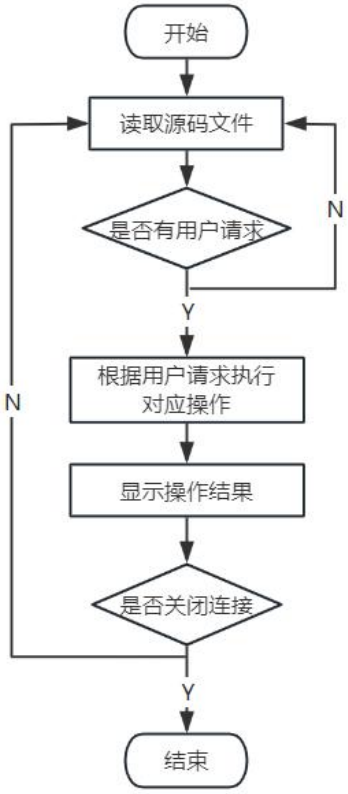

图 16 重构引擎模块流程图

4.6. 优化层

4.6.1. 辅助编辑

1) 模块定义

辅助编辑模块提供一系列编辑辅助功能，以提升编写代码的效率和便捷性。这些功能包括自动闭合、自动缩进、自动环绕、代码折叠等。

2) 模块关联

- 语法分析模块：解析代码结构，识别代码块和语句。
- IDE 集成模块：将辅助编辑功能集成到 IDE 中，提供用户接口和编辑操作。
- 用户交互模块：提供用户界面，用于接受用户的编辑操作指令并显示结果。

3) 数据说明

- 源码文件：用户编写的 SysY2022E 语言代码文件。
- 语法树：由语法分析模块生成的代码语法结构树。
- 用户操作请求：用户发起的编辑操作请求，包括自动闭合、自动缩进、自动环绕、

代码折叠等。

- 编辑结果：编辑操作的结果，包括编辑后的代码和更新的语法树等。

4) 算法说明

- 自动闭合

a) 输入：用户操作请求，当前光标位置

b) 输出：编辑后的代码

c) 算法：在光标位置检测需要闭合的符号（如括号、引号等），自动插入匹配的闭合符号。

表 39 自动闭合算法伪码描述

```
//自动闭合伪码
// 定义用户操作请求结构
class UserRequest {
    content: String
    cursorPosition: Integer
    function UserRequest(content: String, cursorPosition: Integer) {
        this.content = content
        this.cursorPosition = cursorPosition
    }
}
// 自动闭合符号
function autoCloseSymbol(request: UserRequest): EditResult {
    var content = request.content
    var cursorPosition = request.cursorPosition
    // 检查光标前的字符是否需要闭合
    if cursorPosition > 0 {
        var charBeforeCursor = content[cursorPosition - 1]
        if charBeforeCursor in symbolPairs {
            var closingSymbol = symbolPairs[charBeforeCursor]
            content = content.insert(cursorPosition, closingSymbol)
            return EditResult(content, cursorPosition)
        }
    }
```

```
    // 如果不需要闭合，返回原始内容
    return EditResult(content, cursorPosition)
}
// 插入字符串到指定位置
function String.insert(index: Integer, value: String): String {
    return this.substring(0, index) + value + this.substring(index)
}
// 示例：自动闭合
var request = UserRequest("function example() { return [", 24)
var result = autoCloseSymbol(request)
print("Edited content: " + result.content)
print("New cursor position: " + result.cursorPosition)
```

- 自动缩进
  a) 输入：用户操作请求，当前光标位置
  b) 输出：编辑后的代码
  c) 算法：根据代码的上下文和语法结构，自动调整当前行或代码块的缩进级别。

表 40 自动缩进算法伪码描述

```
//自动缩进伪码
// 自动缩进算法
function autoIndent(request: UserRequest): EditResult {
    var content = request.content
    var cursorPosition = request.cursorPosition
    var action = request.action
    var indent = "    " // 4 个空格
    if action == "enter" {
        var currentLineStart = findLineStart(content, cursorPosition)
       varcurrentLineIndent=getLineIndentation(content, currentLineStart)
        var    newContent    =    content.insert(cursorPosition,    "\n"    +
currentLineIndent)
// 查找当前行的起始位置
function findLineStart(content: String, cursorPosition: Integer): Integer {
```

```
    var position = cursorPosition - 1
    while position >= 0 and content[position] != '\n' {
        position -= 1
    }
    return position + 1
}
// 获取当前行的缩进
function getLineIndentation(content: String, lineStart: Integer): String {
    var indent = ""
    var position = lineStart
    while position < content.length and (content[position] == ' ' or
content[position] == '\t') {
        indent += content[position]
        position += 1
    }
    return indent
}
// 插入字符串到指定位置
function String.insert(index: Integer, value: String): String {
    return this.substring(0, index) + value + this.substring(index)
}
// 示例：自动缩进
var request = UserRequest("function example() {\n    return [\n", 32, "enter")
var result = autoIndent(request)
print("Edited content:\n" + result.content)
print("New cursor position: " + result.cursorPosition)
```

- 自动环绕
  a) 输入：用户操作请求，选中的代码段
  b) 输出：编辑后的代码
  c) 算法：在选中的代码段两侧自动插入指定的符号或结构（如括号、引号、代码块等）。

表 41 自动环绕算法伪码描述

```
//自动环绕伪码
// 自动环绕算法
function autoWrap(request: UserRequest): EditResult {
    var content = request.content
    var selectionStart = request.selectionStart
    var selectionEnd = request.selectionEnd
    var wrapType = request.wrapType
    var openingSymbol = ""
    var closingSymbol = ""
    // 根据 wrapType 确定环绕符号
    if wrapType == "parentheses" {
        openingSymbol = "("
        closingSymbol = ")"
    } else if wrapType == "quotes" {
        openingSymbol = "\""
        closingSymbol = "\""
    } else if wrapType == "singleQuotes" {
        openingSymbol = "'"
        closingSymbol = "'"
    } else if wrapType == "braces" {
        openingSymbol = "{"
        closingSymbol = "}"
    } else if wrapType == "brackets" {
        openingSymbol = "["
        closingSymbol = "]"
    } else {
        // 如果不支持的 wrapType，返回原始内容
        return EditResult(content, selectionStart, selectionEnd)
    }
    // 在选中的代码段两侧插入符号
    var newContent = content.insert(selectionStart, openingSymbol)
    newContent = newContent.insert(selectionEnd + 1, closingSymbol)
```

```
    var newSelectionStart = selectionStart
    var newSelectionEnd = selectionEnd + 2 // 因为插入了两个符号
    return EditResult(newContent, newSelectionStart, newSelectionEnd)
}
// 插入字符串到指定位置
function String.insert(index: Integer, value: String): String {
    return this.substring(0, index) + value + this.substring(index)
}
// 示例：自动环绕
var request = UserRequest("function example() {\n    return value;\n}", 21, 26,
"parentheses")
var result = autoWrap(request)
print("Edited content:\n" + result.content)
print("New selection start: " + result.selectionStart)
print("New selection end: " + result.selectionEnd)
```

- 代码折叠
  a) 输入：用户操作请求，语法树
  b) 输出：编辑后的代码视图
  c) 算法：根据代码的语法结构，自动折叠或展开指定的代码块，提升代码的可读性。

表 42 代码折叠算法伪码描述

```
//代码折叠伪码
// 代码折叠算法
function codeFolding(request: UserRequest, syntaxTree: SyntaxNode): EditResult
{
    var foldedLines = []
    var unfoldedLines = []
    // 递归函数，用于处理语法树节点
    function processNode(node: SyntaxNode) {
        if node.startLine >= request.startLine and node.endLine <=
request.endLine {
```

```
            if request.action == "fold" {
                for line in node.startLine+1 to node.endLine-1 {
                    foldedLines.append(line)
                }
            } else if request.action == "unfold" {
                for line in node.startLine+1 to node.endLine-1 {
                    unfoldedLines.append(line)
                }
            }
        }
        // 递归处理子节点
        for child in node.children {
            processNode(child)
        }
    }
    // 处理语法树根节点
    processNode(syntaxTree)
    return EditResult(foldedLines, unfoldedLines)
}
// 示例语法树
var syntaxTree = SyntaxNode("root", 0, 10, [
    SyntaxNode("function", 1, 4, []),
    SyntaxNode("class", 5, 9, [
        SyntaxNode("method", 6, 8, [])
    ])
])
// 示例：折叠代码块
var request = UserRequest("fold", 5, 9)
var result = codeFolding(request, syntaxTree)
print("Folded lines: " + result.foldedLines)
print("Unfolded lines: " + result.unfoldedLines)
```

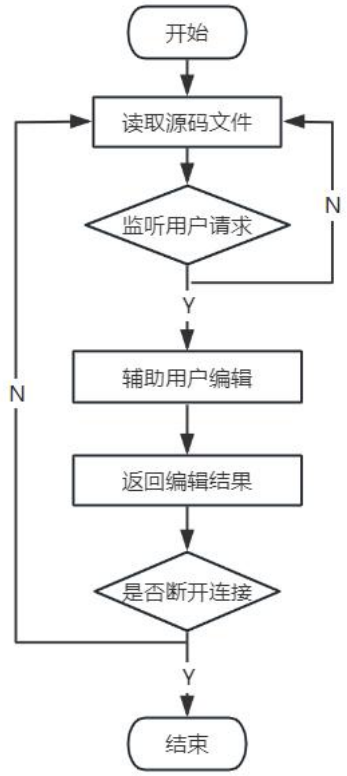

图 17 辅助编辑模块流程图

4.6.2. 查找引用

1) 模块定义

查找引用模块用于在代码中查找并列出某个标识符的所有引用位置。该模块提供用户在代码编辑过程中快速定位和导航到某个标识符的所有使用点。

2) 模块关联

- 语法分析模块：解析代码并生成语法树。
- 语义分析模块：对代码进行语义分析，生成标识符的引用信息。
- 用户交互模块：提供用户界面，用于接受用户的查找引用请求并显示结果。
- IDE 集成模块：将查找引用功能集成到 IDE 中。

3) 数据说明

- 源码文件：用户编写的 SysY2022E 语言代码文件。
- 语法树：由语法分析模块生成的代码语法结构树。
- 符号表：由语义分析模块生成的标识符和其引用信息的表。
- 用户操作请求：用户发起的查找引用请求，包括标识符及其所在位置。
- 引用列表：标识符的所有引用位置的列表。

4) 算法说明

- 标识符解析

a) 输入：用户操作请求，当前光标位置

b) 输出：标识符名称

c) 算法：根据光标位置在语法树中定位并提取标识符名称。

表 43 标识符解析算法伪码描述

```
//标识符解析伪码
// 标识符解析算法
function resolveIdentifier(request: UserRequest, syntaxTree: SyntaxNode): IdentifierResult {
    var cursorPosition = request.cursorPosition
    // 递归函数，用于在语法树节点中查找标识符
    function findIdentifier(node: SyntaxNode): String {
        // 检查光标是否在当前节点范围内
        if cursorPosition >= node.startPosition and cursorPosition <= node.endPosition {
            // 如果当前节点是标识符，返回其名称
            if node.type == "identifier" {
                return node.name
            }
            // 递归检查子节点
            for child in node.children {
                var result = findIdentifier(child)
                if result != null {
                    return result
                }
            }
        }
        return null
    }
    // 从根节点开始查找
    var identifierName = findIdentifier(syntaxTree)
```

```
    return IdentifierResult(identifierName)
}
// 示例语法树
var syntaxTree = SyntaxNode("root", 0, 50, "", [
    SyntaxNode("function", 0, 25, "exampleFunction", [
        SyntaxNode("parameters", 10, 20, "", [
            SyntaxNode("identifier", 11, 15, "param1", [])
        ])
    ]),
    SyntaxNode("variable", 26, 35, "variableName", [
        SyntaxNode("identifier", 27, 34, "variableValue", [])
    ])
])
// 示例：解析标识符
var request = UserRequest("function exampleFunction(param1) {}", 12)
var result = resolveIdentifier(request, syntaxTree)
// 输出结果
print("Identifier name: " + result.name)
```

- 引用查找
  a) 输入：标识符名称，符号表
  b) 输出：引用列表
  c) 算法：在符号表中查找与标识符名称匹配的所有引用位置。

表 44 引用查找算法伪码描述

```
//引用查找伪码
// 定义引用结果结构
class ReferenceResult {
    references: List<Integer> // 存储所有引用的位置
    function ReferenceResult(references: List<Integer>) {
        this.references = references
    }
}
```

```
// 引用查找算法
function findReferences(identifierName: String, symbolTable: List<SymbolTableEntry>): ReferenceResult {
    var references = []
    // 遍历符号表
    for entry in symbolTable {
        if entry.name == identifierName {
            // 添加所有匹配标识符的引用位置
            references.extend(entry.positions)
        }
    }
    return ReferenceResult(references)
}
// 示例符号表
var symbolTable = [
    SymbolTableEntry("exampleFunction", "function", [0, 50]),
    SymbolTableEntry("param1", "variable", [11, 30]),
    SymbolTableEntry("variableName", "variable", [26, 35]),
    SymbolTableEntry("variableValue", "variable", [27, 34, 45])
]
// 示例：查找引用
var identifierName = "variableValue"
var result = findReferences(identifierName, symbolTable)
// 输出结果
print("References for identifier '" + identifierName + "': " + result.references)
```

- 结果显示
  a) 输入：引用列表
  b) 输出：用户界面上的引用位置列表
  c) 算法：在用户界面上列出所有引用位置，并提供快速导航功能。

表 45 结果显示算法伪码描述

```
//结果显示伪码
```

```
// 定义引用结果结构
class ReferenceResult {
    references: List<Integer> // 存储所有引用的位置


    function ReferenceResult(references: List<Integer>) {
        this.references = references
    }
}
// 定义用户界面元素结构
class UIElement {
    type: String // 比如 "button", "list", etc.
    content: String
    position: Integer // 在界面中的位置，适用于可导航元素
    function UIElement(type: String, content: String, position: Integer) {
        this.type = type
        this.content = content
        this.position = position
    }
}
// 定义用户界面结构
class UserInterface {
    elements: List<UIElement>
    function UserInterface(elements: List<UIElement>) {
        this.elements = elements
    }
    // 在界面上添加元素
    function addElement(element: UIElement) {
        this.elements.append(element)
    }
    // 渲染界面
    function render() {
        for element in self.elements {
```

```
            print("Rendering " + element.type + " with content: " +
element.content)
        }
    }
}
// 示例引用结果
var referenceResult = ReferenceResult([11, 30, 27, 34, 45])
// 显示引用结果
var ui = displayReferences(referenceResult)
```

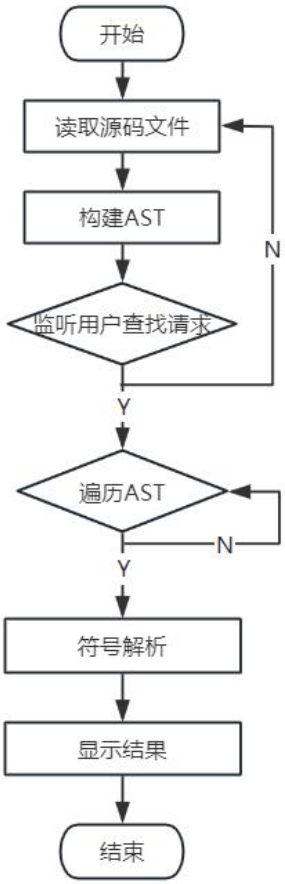

图 18 查找引用模块流程图

4.6.3. 签名帮助

1) 模块定义

签名帮助模块用于在用户输入函数调用时显示该函数的参数信息和用法提示。该模块帮助用户了解函数的参数要求，减少错误输入并提高代码编写效率。

2) 模块关联

- 语法分析模块：解析代码并生成语法树。
- 符号表模块：存储函数定义及其参数信息。
- 用户交互模块：提供用户界面，用于显示函数的签名帮助信息。
- IDE 集成模块：将签名帮助功能集成到 IDE 中。

3) 数据说明

- 源码文件：用户编写的 SysY2022E 语言代码文件。
- 语法树：由语法分析模块生成的代码语法结构树。
- 符号表：存储函数定义及其参数信息的表。
- 用户操作请求：用户在函数调用时的输入操作。
- 签名信息：包括函数名称、参数列表、参数类型、参数描述等信息。

4) 算法说明

- 签名解析
  a) 输入：用户操作请求，当前光标位置，语法树
  b) 输出：函数名称及其签名信息
  c) 算法：根据光标位置在语法树中定位当前函数调用，提取函数名称及其参数信息。

表 46 签名解析算法伪码描述

```
//签名解析伪码
// 签名解析算法
function resolveSignature(request: UserRequest, syntaxTree: SyntaxNode): FunctionSignature {
    var cursorPosition = request.cursorPosition
    // 递归函数，用于在语法树节点中查找函数调用
    function findFunctionCall(node: SyntaxNode): FunctionSignature {
        // 检查光标是否在当前节点范围内
        if cursorPosition >= node.startPosition and cursorPosition <= node.endPosition {
            // 如果当前节点是函数调用，提取其名称和参数信息
            if node.type == "function_call" {
                var functionName = node.name
                var parameters = []
```

```
                for child in node.children {
                    if child.type == "parameter" {
                        parameters.append(child.name)
                    }
                }
                return FunctionSignature(functionName, parameters)
            }
            // 递归检查子节点
            for child in node.children {
                var result = findFunctionCall(child)
                if result != null {
                    return result
                }
            }
        }
        return null
    }
// 示例：签名解析
var request = UserRequest("exampleFunction(arg1, arg2)", 55)
var result = resolveSignature(request, syntaxTree)
// 输出结果
if result != null {
    print("Function name: " + result.functionName)
    print("Parameters: " + result.parameters)
} else {
    print("No function call found at the cursor position.")
}
```

- 签名获取
  a) 输入：函数名称，符号表
  b) 输出：签名信息
  c) 算法：在符号表中查找与函数名称匹配的签名信息。

表 47 签名获取算法伪码描述

```
//签名获取伪码
//定义签名结果结构
class FunctionSignature {
    functionName: String
    parameters: List<String>
    function FunctionSignature(functionName: String, parameters: List<String>)
{
        this.functionName = functionName
        this.parameters = parameters
    }
}
// 示例符号表
var symbolTable = [
    SymbolTableEntry("exampleFunction", "function", ["param1", "param2"]),
    SymbolTableEntry("anotherFunction", "function", ["arg1"]),
    SymbolTableEntry("someVariable", "variable", [])
]
// 示例：签名获取
var functionName = "exampleFunction"
var signature = getSignature(functionName, symbolTable)
// 输出结果
if signature != null {
    print("Function name: " + signature.functionName)
    print("Parameters: " + signature.parameters)
} else {
    print("Function '" + functionName + "' not found.")
}
```

- 结果显示
  a) 输入：签名信息
  b) 输出：用户界面上的签名帮助提示
  c) 算法：在用户界面上显示函数签名及其参数信息。

图 19 签名帮助模块流程图

4.6.4. 查看定义

1) 模块定义

查看定义模块用于查找和显示代码中某个符号（如变量、函数、类等）的定义位置。该模块帮助用户快速导航到符号的定义位置，从而理解代码的实现细节。

2) 模块关联

- 语法分析模块：解析代码并生成语法树。
- 符号表模块：存储代码中的符号信息及其定义位置。
- 用户交互模块：提供用户界面，用于显示符号的定义位置。
- IDE 集成模块：将查看定义功能集成到 IDE 中。

3) 数据说明

- 源码文件：用户编写的 SysY2022E 语言代码文件。
- 语法树：由语法分析模块生成的代码语法结构树。
- 符号表：存储代码中符号信息及其定义位置的表。
- 用户操作请求：用户选择符号并请求查看其定义位置。
- 定义信息：包括符号名称、定义位置（文件名、行号、列号）等信息。

4) 算法说明

- 定义解析

  a) 输入：用户选择的符号，语法树

b) 输出：符号名称及其定义位置

c) 算法：根据用户选择的位置在语法树中定位符号定义，并提取符号名称及其定义位置。

表 48 定义解析算法伪码描述

```
//定义解析伪码
// 定义解析算法
function resolveDefinition(selection: UserSelection, syntaxTree: SyntaxNode): SymbolDefinition {
    var selectionStart = selection.selectionStart
    var selectionEnd = selection.selectionEnd
    // 递归函数，用于在语法树节点中查找符号定义
    function findSymbolDefinition(node: SyntaxNode): SymbolDefinition {
        // 检查选择范围是否在当前节点范围内
        if selectionStart >= node.startPosition and selectionEnd <= node.endPosition {
            // 如果当前节点是符号定义，提取其名称和定义位置
            if node.type == "function_definition" or node.type == "variable_declaration" {
                return SymbolDefinition(node.name, node.startPosition)
            }
            // 递归检查子节点
            for child in node.children {
                var result = findSymbolDefinition(child)
                if result != null {
                    return result
                }
            }
        }
        return null
    }
    // 从根节点开始查找
    var symbolDefinition = findSymbolDefinition(syntaxTree)
```

```
    return symbolDefinition
}
// 示例语法树
var syntaxTree = SyntaxNode("root", 0, 100, "", [
    SyntaxNode("function_definition", 0, 50, "exampleFunction", []),
    SyntaxNode("variable_declaration", 51, 55, "exampleVar", []),
    SyntaxNode("function_call", 56, 99, "exampleFunction", [])
])
// 示例：定义解析
var selection = UserSelection("exampleFunction", 10, 25)
var result = resolveDefinition(selection, syntaxTree)
// 输出结果
if result != null {
    print("Symbol name: " + result.symbolName)
    print("Definition position: " + result.definitionPosition)
} else {
    print("No symbol definition found in the selection range.")
}
```

- 定义查找
  a) 输入：符号名称，符号表
  b) 输出：定义信息
  c) 算法：在符号表中查找与符号名称匹配的定义位置。

表 49 定义查找算法伪码描述

```
//定义查找伪码
// 定义符号定义结果结构
class SymbolDefinition {
    symbolName: String
    definitionPosition: Integer
    function SymbolDefinition(symbolName: String, definitionPosition: Integer)
{
        this.symbolName = symbolName
```

```
        this.definitionPosition = definitionPosition
    }
}
// 示例符号表
var symbolTable = [
    SymbolTableEntry("exampleFunction", "function", 10),
    SymbolTableEntry("exampleVar", "variable", 20),
    SymbolTableEntry("anotherFunction", "function", 30)
]
// 示例：定义查找
var symbolName = "exampleFunction"
var definition = findDefinition(symbolName, symbolTable)
// 输出结果
if definition != null {
    print("Symbol name: " + definition.symbolName)
    print("Definition position: " + definition.definitionPosition)
} else {
    print("Symbol '" + symbolName + "' not found.")
}
```

- 结果显示
  a) 输入：定义信息
  b) 输出：用户界面上的定义位置
  c) 算法：在用户界面上显示符号的定义位置，并允许用户导航到相应的代码位置。

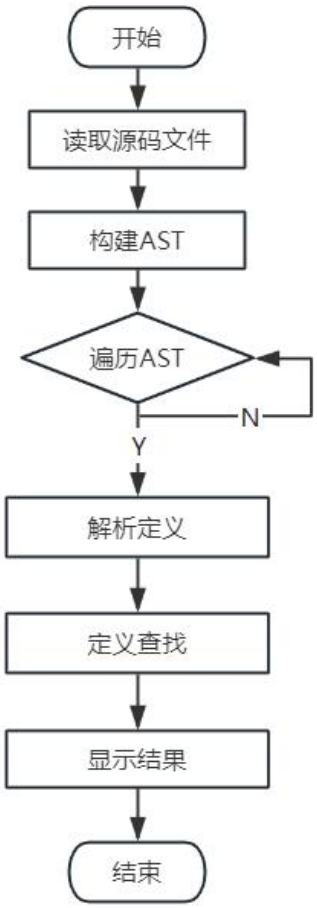

图 20 查看定义模块流程图

5. 程序提交清单

模块	文件名	文件类别	用途
IDE 集成	server.ts	核心代码文件	LSP 服务器端，向用户提供各种功能。
IDE 集成	client.ts	核心代码文件	LSP 客户端，向服务器端申请各种请求。
语法高亮	sy.tmLanguage.json	配置文件	提供用于语法高亮显示的不同类别标识。
语法高亮	SysY2022E-color-theme.json	主题文件	提供用于语法高亮显示的不同色彩主题。
悬浮提示	Symbol.ts	核心代码文件	定义符号表的数据结构

悬浮提示	SymbolTableGenerator.ts	核心代码文件	遍历 AST，基于源代码生成符号表
语法检查	syntaxChecker.ts	核心代码文件	遍历 AST，检查各个节点的语法错误，并记录。
错误提示	errorListener.ts	核心代码文件	将记录的错误显示给用户。
静态语义检查	syntaxChecker.ts	核心代码文件	遍历 AST，检查各个节点的静态语义错误，并记录。
修复建议	errorListener.ts	核心代码文件	显示错误提示的同时，向用户提供修复建议。
自动修复	codeAction.ts	核心代码文件	根据用户选择，进行对应的自动修复功能。
重构引擎	codeAction.ts	核心代码文件	提供重命名、抽取新函数、抽取变量、内联函数的功能。
辅助编辑	language-configuration.json	配置文件	辅助编辑模块提供一系列编辑辅助功能，以提升编写代码的效率和便捷性。这些功能包括自动闭合、自动缩进、自动环绕、代码折叠等。
查找引用	referenceRecord.ts	核心代码文件	定义用于引用查找的记录的数据结构。
查找引用	referenceHelper.ts	核心代码文件	提供引用查找功能，根据用户请求，查找记录。
签名帮助	signatureRecord.ts	核心代码文件	定义签名帮助的数据结构。
签名帮助	signature.ts	核心代码文件	提供查找签名帮助信息的功能。
查看定义	SymbolTableGenerator.ts	核心代码文件	遍历 AST，生成符号表，记录定义信息。
程序错误	runAnalysis.ts	核心代码文件	程序错误检查模块用于检

检查			测代码中的常见错误，如无用变量、死循环、数组越界等，以提高代码的质量和可靠性。
客户端配置	client/package.json	配置文件	提供客户端的配置。
服务端配置	server/package.json	配置文件	提供服务器端的配置。
项目配置	package.json	配置文件	提供整个项目的完整配置项。
语言定义	SysY2022E.g4	配置文件	定义 SysY2022E 语言的词法、语法等信息。
词法定义	SysY2022ELexer.ts	核心代码文件	提供 SysY2022E 语言的词法规则。
语法定义	SysY2022EParser.ts	核心代码文件	提供 SysY2022E 语言的语法规则。
visitor	SysY2022EVisitor.ts	核心代码文件	提供以访问者身份遍历 AST 的规则。
listener	SysY2022EListener.ts	核心代码文件	提供以监听者身份遍历 AST 的规则。
项目许可	license.md	配置文件	项目的 license
项目介绍	README.md	配置文件	简要介绍项目，提供项目的配置、运行等细节。
项目产品	SysY2022E-Language-Support-1.0.0.vsix	拓展文件	项目产品，可以直接安装到拓展市场中，进行使用。

程序提交清单以模块为单位分别进行描述。