**编译原理期末报告**

**类Rust中间代码生成器设计与实现**

**徽标

描述已自动生成**

**小组成员 [Truman-min-show](https://github.com/Truman-min-show)**

**[limeszstone](https://github.com/limeszstone)**

**[TheTry16](https://github.com/TheTry16)**

**院 系 同济大学计算机系**

**指导老师 丁志军**

**完成日期 2025.6.4**

**目录**

[一、 实验概述 3](#_Toc199921969)

[1、目的与意义 3](#_Toc199921970)

[2. 主要任务 3](#_Toc199921971)

[3、需求分析 4](#_Toc199921972)

[二、使用说明 5](#_Toc199921973)

[1、环境配置 5](#_Toc199921974)

[2、整体设计 5](#_Toc199921975)

[3、实现功能 8](#_Toc199921976)

[4、拓展功能 11](#_Toc199921977)

[三、详细设计 15](#_Toc199921978)

[1、语义分析 15](#_Toc199921979)

[2、中间代码生成 33](#_Toc199921980)

[3、可视化设计 44](#_Toc199921981)

[四、小组分工 47](#_Toc199921982)

[五、总结与展望 48](#_Toc199921983)

[1、总结 48](#_Toc199921984)

[2、展望 48](#_Toc199921985)

[六、参考文献 50](#_Toc199921986)

# 实验概述

## 1、目的与意义

编译器是计算机科学领域的基础核心技术之一，深刻理解其工作原理对于理解编程语言的底层机制，以及未来进行系统软件、编程语言设计等领域的研究与工作都具有重要意义。本实验通过实践，将理论知识转化为实际动手能力。实验旨在设计并实现一个针对简化版类Rust语言的编译器。其核心目的在于深入理解和实践编译器设计的基本原理与关键技术，具体包括：

* **深入理解语义分析的关键作用与技术：**认识到语义分析在确保程序意义正确性方面的重要性，掌握如符号表管理、类型检查、作用域分析等核心技术，并能将其应用于实际的编译器组件开发。
* **初步掌握中间代码生成的基本原理：**了解中间代码作为编译器前端与后端桥梁的角色，学习一种常见的中间表示形式（如四元式），并实践从抽象语法树到中间代码的转换过程。

## 2、主要任务

本项目的主要任务是分阶段完成类Rust语言编译器的核心组件，在前一阶段词法分析器、语法分析器和AST构建的基础上，本阶段主要任务集中在编译器的后续核心步骤：

* **语义分析器 (Semantic Analyzer):**

1. 设计并实现符号表，用于存储和管理程序中的标识符信息，如类型、作用域、可变性等。
2. 实现一个语义分析器，采用访问者模式遍历抽象语法树 (AST)，根据类Rust语言的语义规则，进行包括但不限于以下检查：变量的声明与使用检查、类型检查、作用域规则检查。
3. 能够在AST节点上标注类型信息，供后续阶段使用。
4. 能够检测并报告详细的语义错误，指出错误位置和原因。

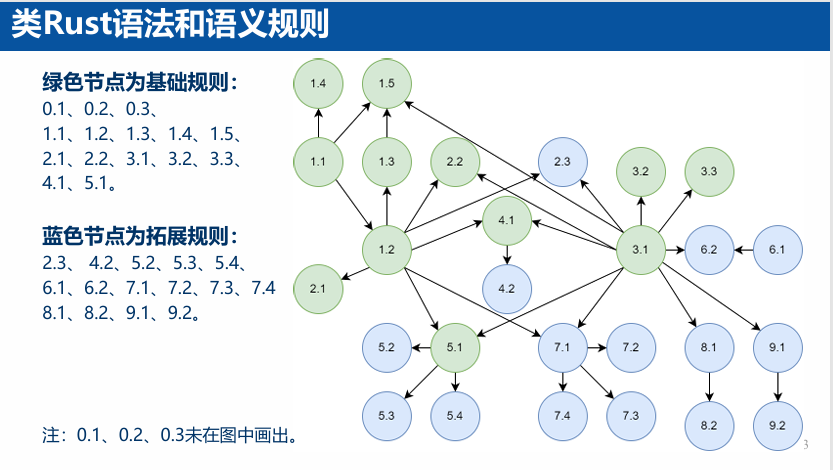
* **中间代码生成器 (IR Generator):**

1. 选择并定义一种中间表示形式，本项目选用四元式作为目标中间代码。
2. 设计四元式的操作符集合，覆盖算术运算、逻辑运算、赋值、跳转、函数调用、参数传递、返回等操作。
3. 实现一个中间代码生成器，同样采用访问者模式遍历（经过语义分析且无误的）AST，将AST节点的操作和结构转换为一个或多个四元式指令。
4. 管理临时变量的生成与使用，以及标签的生成。
5. 生成的四元式序列应能清晰地反映原始程序的控制流和数据流。

## 3、需求分析

本项目旨在实现的类Rust编译器前端需要满足以下基本需求：

* **输入：**接收包含类Rust源代码的文本文件。
* **语义分析：**在语法正确的AST基础上，进行上下文相关的语义检查：利用符号表管理标识符信息，执行类型检查，确保操作的合法性和类型兼容性，同时检查作用域规则和其他语言特定的静态语义约束。



* **中间代码生成：**在语义分析无误的前提下，将AST转换为四元式序列，生成的四元式应能表示源程序的计算逻辑和控制流程。
* **输出：**成功的语义分析应报告无语义错误，并输出生成的四元式序列；若存在错误，则输出明确的语义错误信息，包含错误类型、位置（行、列号）和相关描述。

# 二、使用说明

## 1、环境配置

以下命令需要在代码目录下的命令行界面中执行，需要先行安装python 3.11、pip等工具，建议使用虚拟环境进行环境配置。

1. 安装依赖

requirements.txt中包含了python运行该项目需要的环境依赖，安装依赖的命令为：

pip install -r requirements.txt

1. 运行

程序main函数在gui.py中，因此运行需要以gui.py为主文件，同时由于生成语法树需要依赖graphviz\_bin中的文件，因此需要将graphviz\_bin放在代码的同一目录下。运行程序的命令为：

python gui.py

1. 打包

打包程序需要用到pyinstaller，打包命令如下：

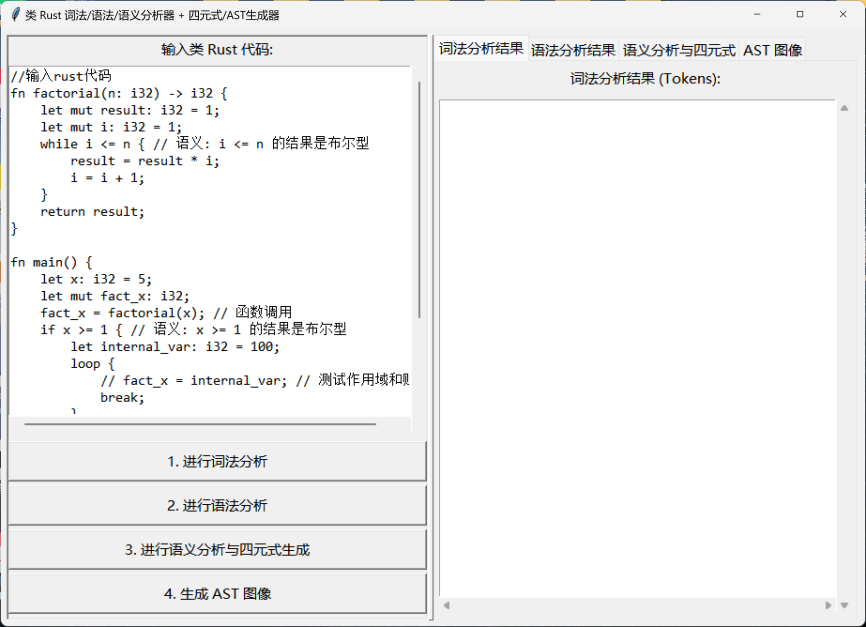
pyinstaller --onefile --noconsole --name=Rust-Compiler --icon=rust.ico gui.py

打包完成后的exe运行也需要graphviz\_bin的文件，因此运行打包后的Rust-Compiler.exe也需要将graphviz\_bin放在同一目录下运行。

## 2、整体设计

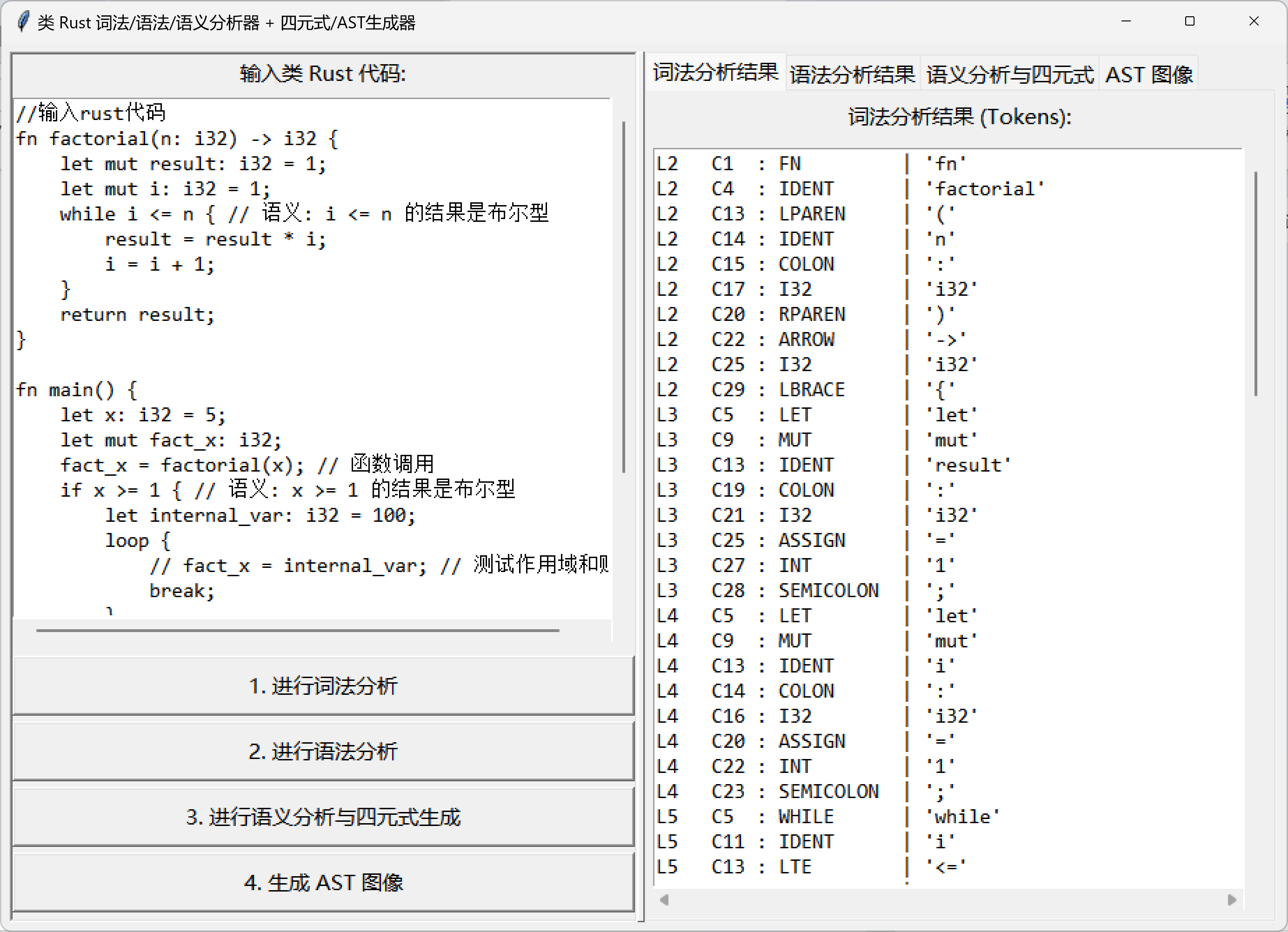
1. 初始界面

进入初始界面，左边是输入框，可以输入rust代码，初始展示的代码是写在gui.py内部的。我们的分析器可以通过在左侧编辑器输入代码，从而进行之后的分析过程。输入框下方有“进行词法分析”、“进行语法分析”、“进行语义分析与四元式生成”、“生成AST图像”四个按钮，点击按钮后，可以生成相应的词法分析、语法分析、语义分析四元式、AST等结果，具体结果查看可以通过点击右半界面上方对应的标签，进入相应的界面查看。



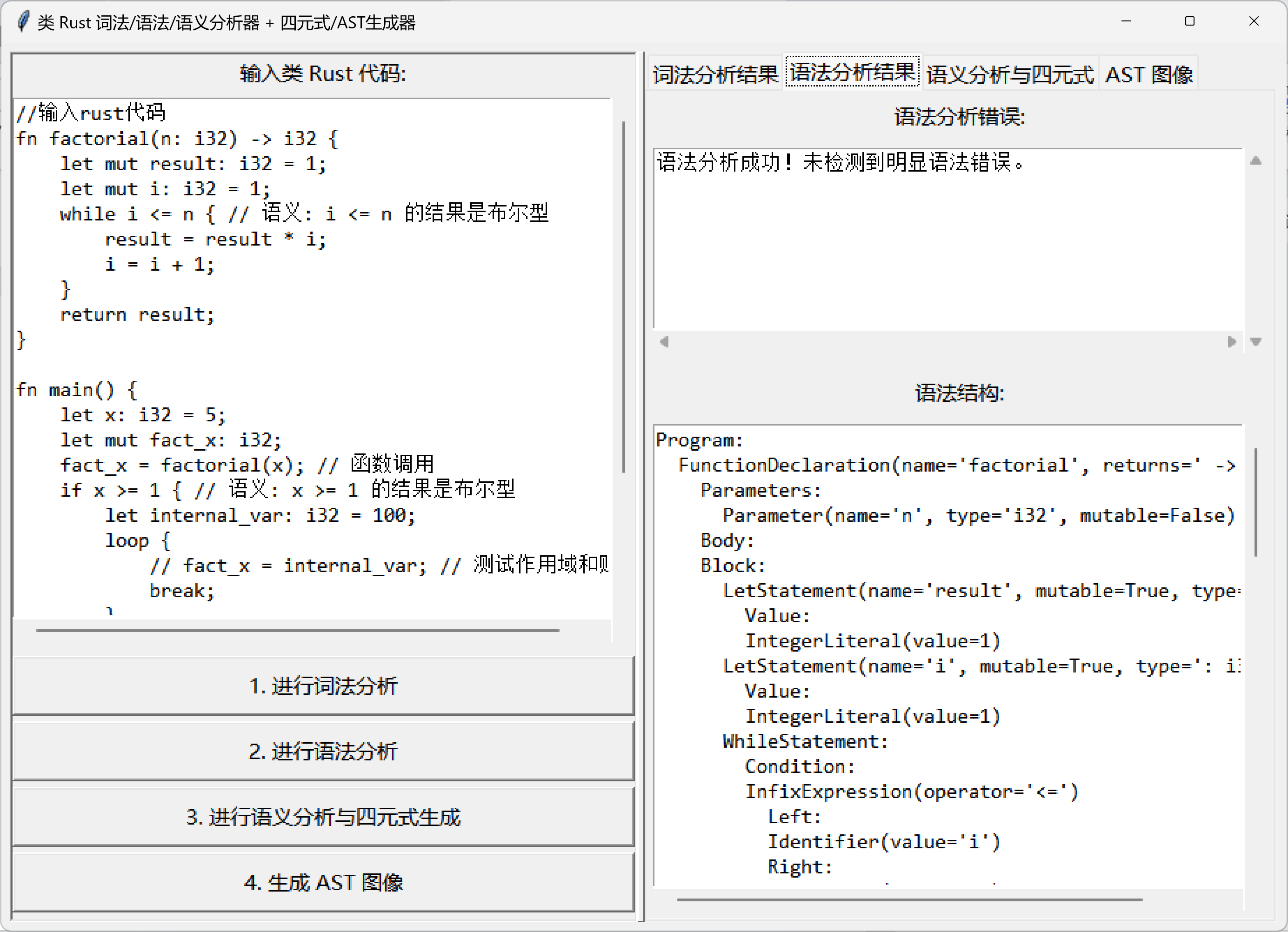
1. 词法分析结果

点击左下方的“进行词法分析”按钮，点击右侧“词法分析结果”，即可在右侧界面生成相关代码的词法分析结果。词法分析结果对代码的每一行每一个token都进行了分析展示。由于词法分析结果较多，可以通过滚轮进行上下滑动查看全部内容。



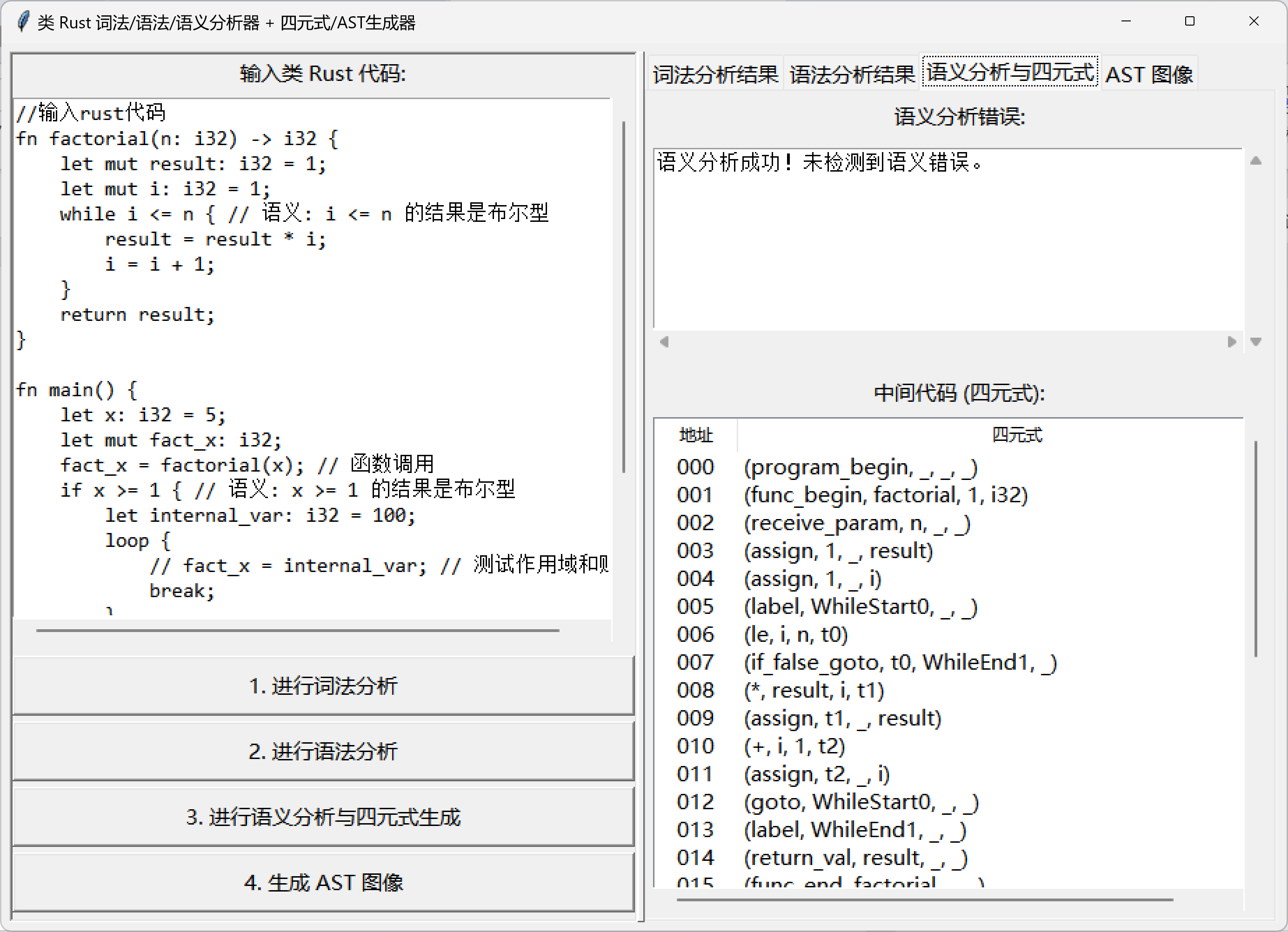
1. 语法分析结果

点击左下方的“进行语法分析”按钮，点击右侧“语法分析结果”，即可在右侧界面生成相关代码的语法分析结果。右侧界面上方输出语法分析是否有错误的结果，下方则是输出左边代码的语法结构。



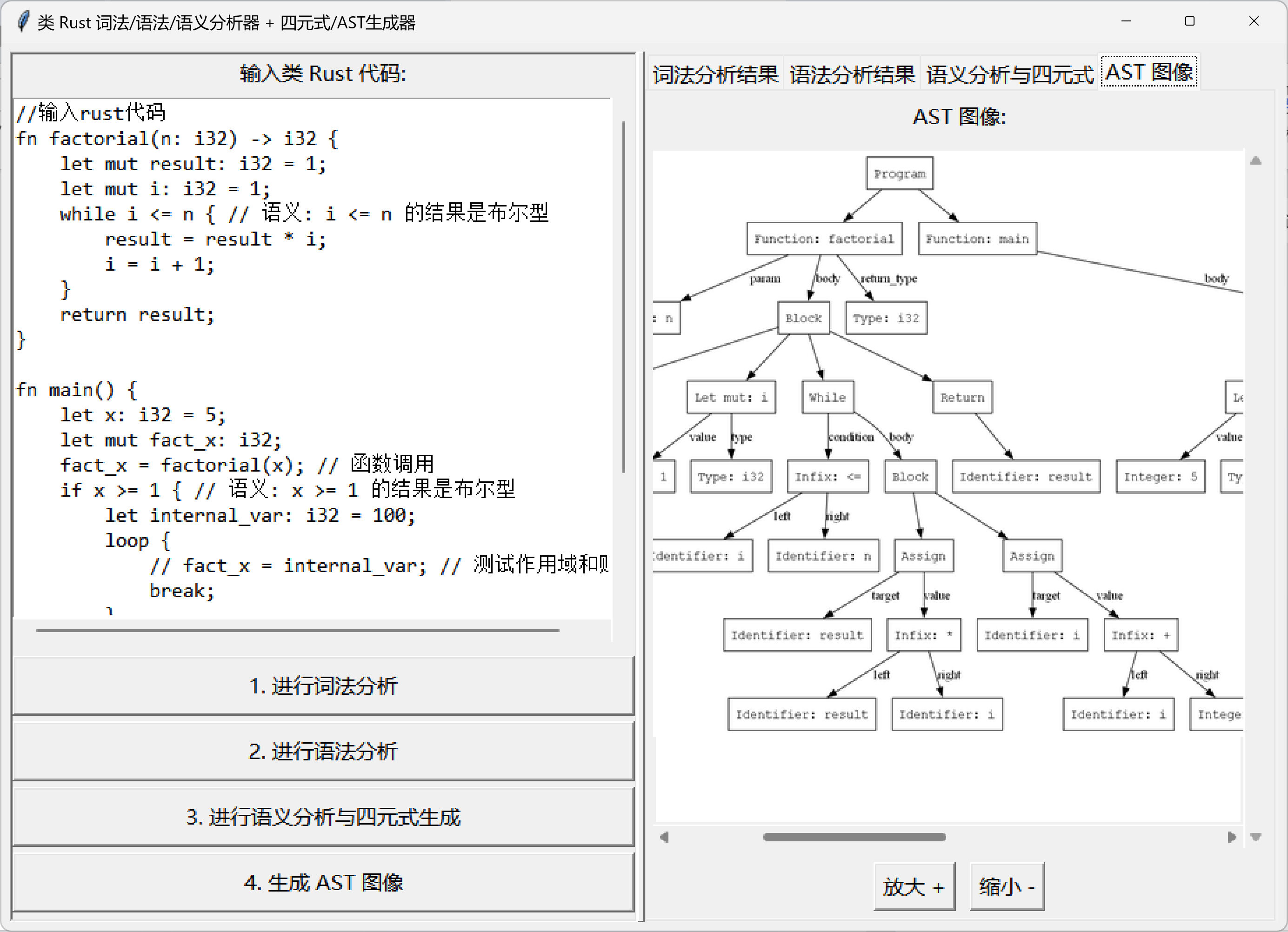
1. 语义分析结果

点击左下方的“进行语义分析与四元式生成”按钮，点击右侧“语义分析与四元式”，即可在右侧界面生成相关代码的语义分析结果。右侧界面上方输出语义分析是否有错误的结果，下方则是输出左边代码的语义分析生成的四元式，同时还可以看到不同中间代码的地址部分以及具体的四元式。



1. AST图像

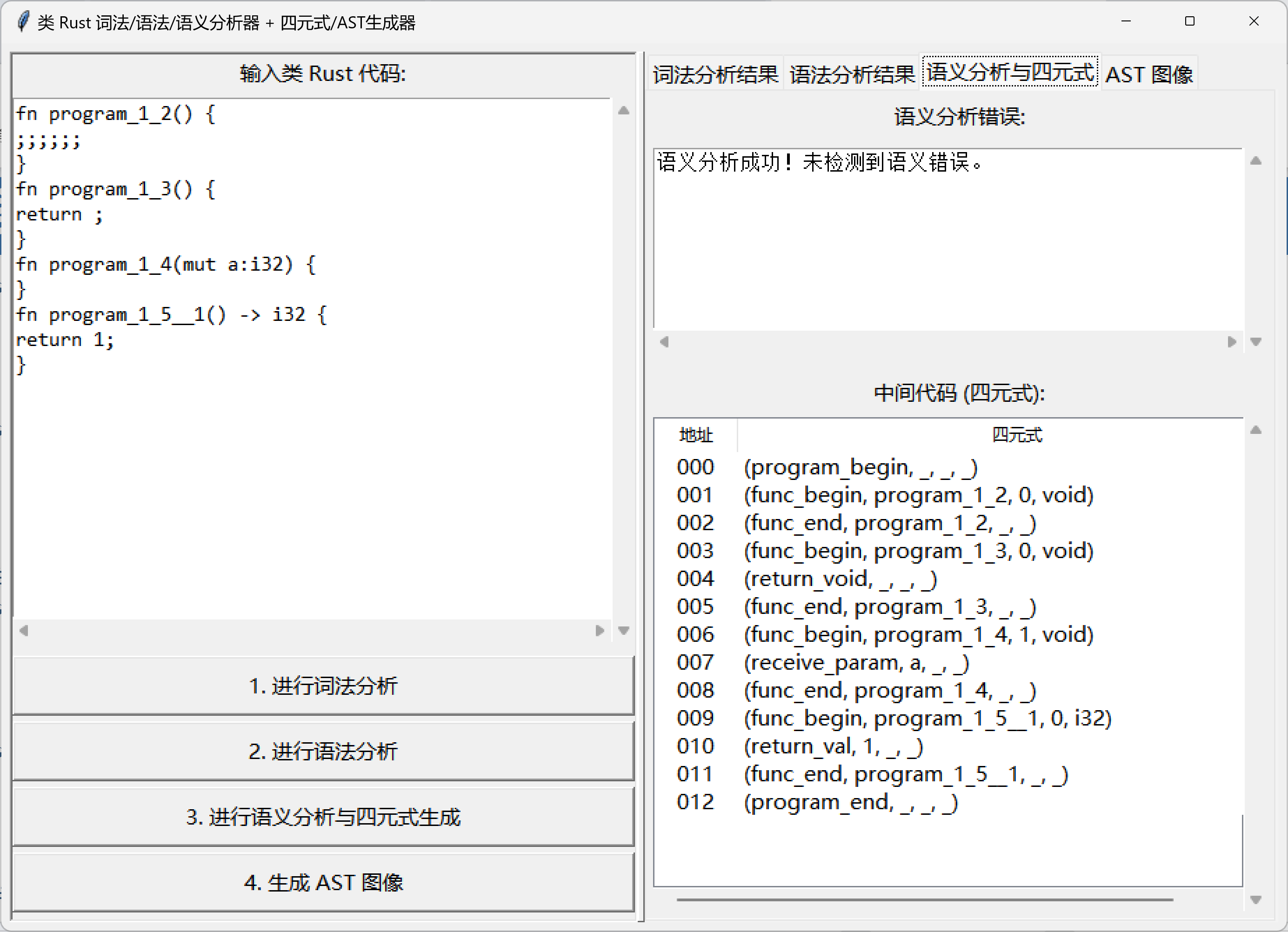
点击左下方的“生成AST图像”按钮，点击右侧“AST图像”，即可在右侧界面生成相关代码的语法树图像。语法树生成结果依赖于前面语法分析的语法结构，根据相关结构进行树的划分。同时，语法树图像还可以通过上下左右滚轮进行查看，同时在右侧界面下方提供放大、缩小按钮，可以通过其进行缩放，从而更好的查看图像。



## 3、实现功能

1. 基础程序和函数输出

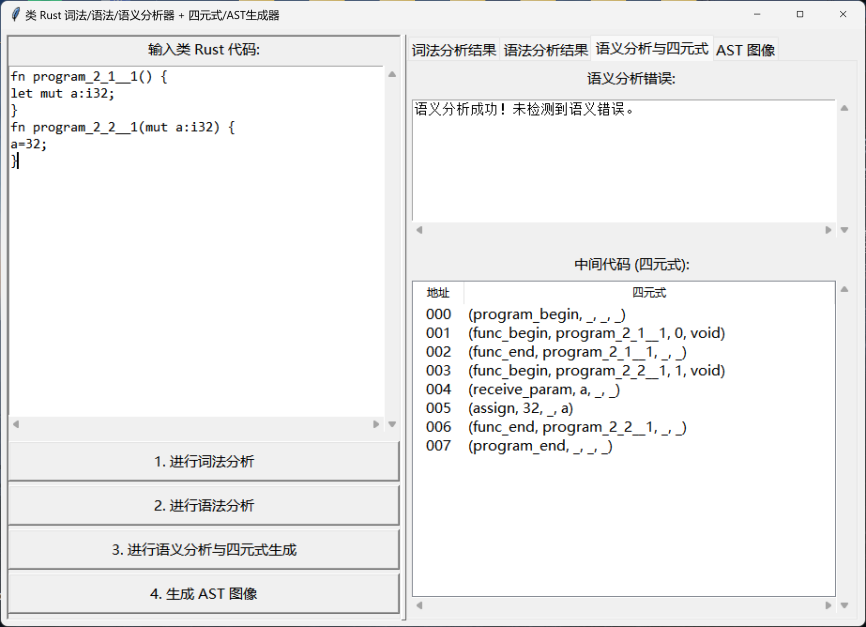
在这里，我们实现了1.1~1.5的基本功能，包括基础程序、语句、返回语句、函数输入、函数输出。具体测试代码见下图左侧的输入代码。其中词法分析、语法分析和AST图像的生成演示已在上次的期中报告中展示，本报告中所有的演示环节将主要展示语义分析和四元式生成。相关运行结果如下：





1. 变量声明和赋值语句

在这里，我们实现了2.1~2.3的功能，包括变量声明语句、赋值语句。具体测试代码见下图左侧的输入代码。相关运行结果如下：





1. 基本表达式和函数调用

在这里，我们实现了3.1~3.3的功能，包括基本表达式、表达式增加计算和比较、函数调用。具体测试代码见下图左侧的输入代码。相关运行结果如下：







1. 选择结构

在这里，我们实现了4.1的功能，即选择结构。具体测试代码见下图左侧的输入代码。相关运行结果如下：



1. 循环结构

在这里，我们实现了5.1的功能，即循环结构。具体测试代码见下图左侧的输入代码。相关运行结果如下：



1. 结果分析

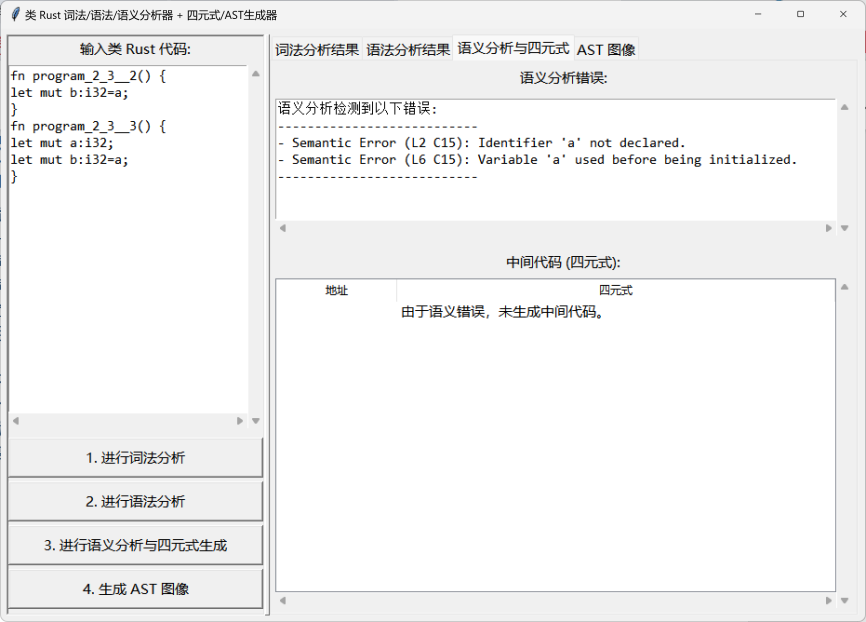
可以看到，程序均可以在上次作业中的语法分析的结果上进行语义分析并生成四元组。相应的词法分析结果、语法分析结果、语义分析结果、语法树图像无误，且可以精确给出程序报错的位置和类型，说明对测试程序的运行结果成功，相关功能实现成功。

## 4、拓展功能

除了上述基本功能之外，我们还实现了2.3、5.3、5.4和6.1的功能内容，包括变量声明赋值语句、loop循环结构、增加break和continue、声明不可变变量。具体测试代码见下图左侧的输入代码。相关运行结果如下：

2.3变量声明赋值语句：

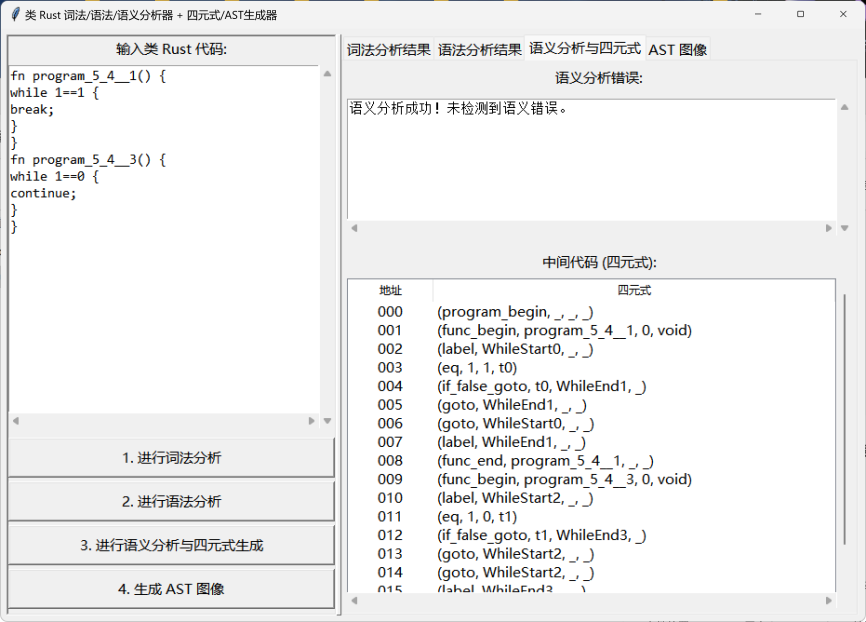


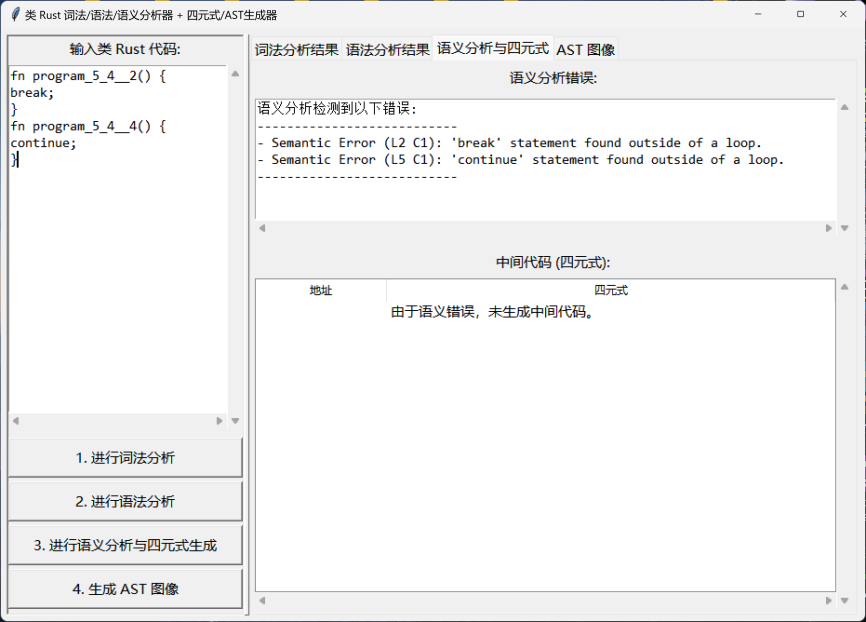


5.3 loop循环结构：



5.4 增加了break和continue语句：





6.1 声明不可声明变量：





# 三、详细设计

## 1、语义分析

1. 核心数据结构和思想

核心数据结构：

1. SymbolTable

符号表是语义分析阶段的核心数据结构，用于存储程序中所有已声明的标识符（变量、函数、参数等）的详细信息，并在不同作用域之间管理这些信息。

* 作用域栈 (self.scopes)：一个列表，每个元素代表一个作用域。列表的最后一个元素是当前作用域。每个作用域是一个字典，键为标识符名称（str），值为对应的Symbol对象。
* 当前作用域级别 (self.current\_scope\_level)：整数，表示当前所处的作用域嵌套深度，0 代表全局作用域。
* 语义错误列表 (self.semantic\_errors)：存储在语义分析过程中检测到的所有错误消息，以便统一报告。

class SymbolTable:

    def \_\_init\_\_(self):

        self.scopes = [{}]

        self.current\_scope\_level = -1

        self.semantic\_errors = []

1. Symbol

封装一个标识符的所有语义属性。

* name (str)：符号的名称。
* type (Type)：符号的类型（如i32,bool,fn(i32) -> i32）。
* kind (str)：符号的种类，如"variable","function","parameter"。
* is\_mutable (bool)：表示变量是否可变（mut关键字）。
* scope\_level (int)：符号被定义的所在作用域级别。
* attributes (dict)：额外属性，例如函数符号可以存储其参数信息。
* is\_initialized (bool)：表示变量是否已被初始化。这是检查“使用未赋值变量”的关键。

class Symbol:

def \_\_init\_\_(self, name, sym\_type: Type, kind: str, is\_mutable: bool = False,

                 scope\_level: int = 0, attributes=None, is\_initialized: bool = False):

        self.name = name

        self.type = sym\_type

        self.kind = kind

        self.is\_mutable = is\_mutable

        self.scope\_level = scope\_level

        self.attributes = attributes if attributes else {}

        self.is\_initialized = is\_initialized # ADDED

1. Type, FunctionType

定义了程序中所有值的类型，用于类型检查和推导。

class Type:

    def \_\_init\_\_(self, name):

        self.name = name

    def \_\_eq\_\_(self, other):

        if not isinstance(other, Type):

            return False

        return self.name == other.name

    def \_\_str\_\_(self):

        return self.name

    def \_\_repr\_\_(self):

        return f"Type('{self.name}')"

class FunctionType(Type):

    def \_\_init\_\_(self, param\_types, return\_type):

        super().\_\_init\_\_("function")

        self.param\_types = param\_types

        self.return\_type = return\_type

    def \_\_eq\_\_(self, other):

        if not isinstance(other, FunctionType):

            return False

        return self.param\_types == other.param\_types and self.return\_type == other.return\_type

    def \_\_str\_\_(self):

        params\_str = ", ".join(map(str, self.param\_types))

        return f"fn({params\_str}) -> {str(self.return\_type)}"

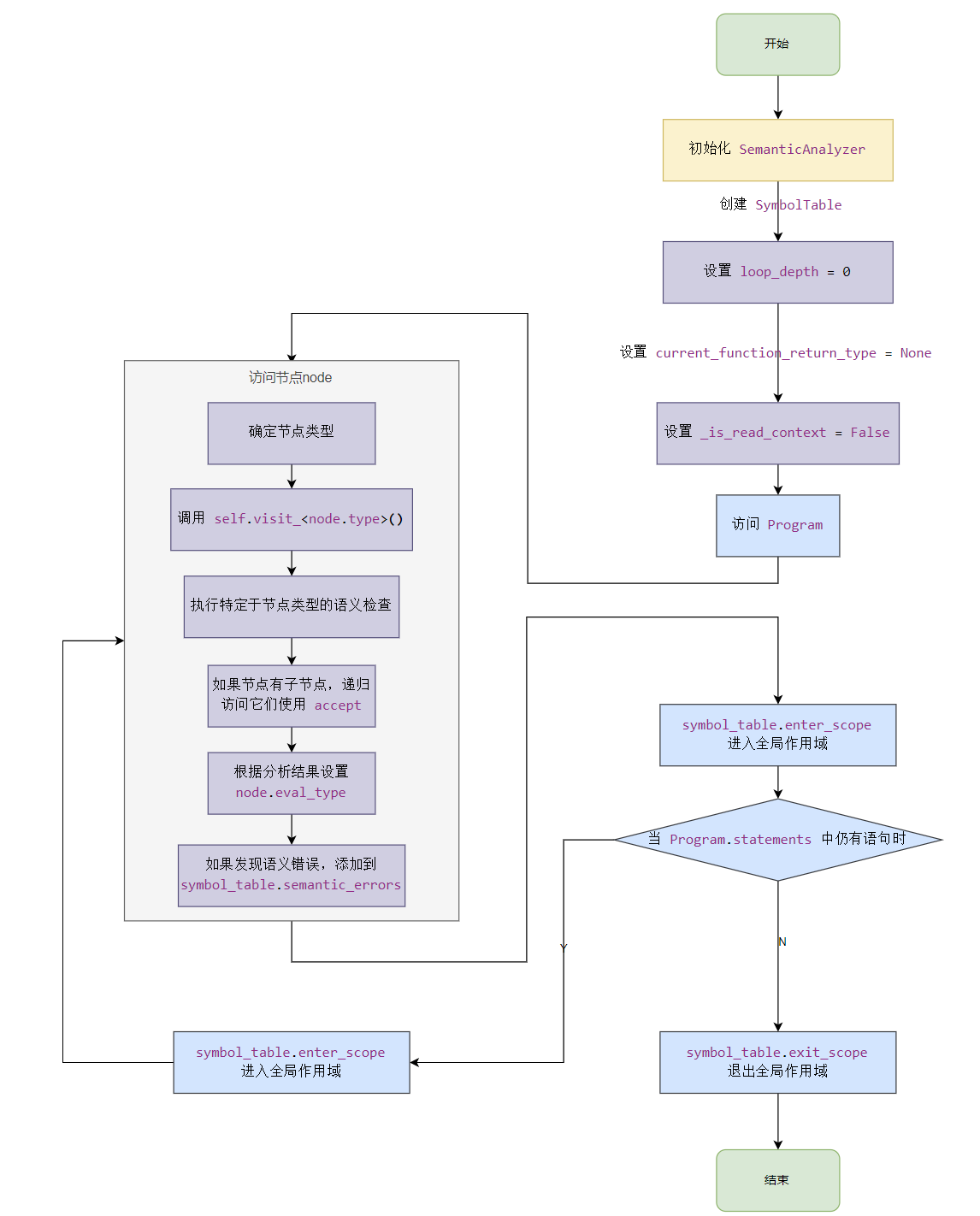
    def \_\_repr\_\_(self):

        return f"FunctionType(param\_types={self.param\_types!r}, return\_type={self.return\_type!r})"

核心思想：

语义分析是编译过程中的关键阶段，主要任务是检查程序的语义合法性，包括类型检查、变量声明与使用检查、作用域管理、控制流验证等。本阶段确保程序在逻辑上是正确的，为后续的中间代码生成提供可靠的依据。语义分析的核心在于对程序中各种符号（变量、函数等）及其属性进行有效的管理和查询，这主要通过符号表来实现。

1. 语义分析流程



1. 重点功能函数实现解析

**作用域与声明相关规则的处理**

1. visit\_program(node: Program)

def visit\_program(self, node: Program):

        self.symbol\_table.enter\_scope()

        for stmt in node.statements:

            stmt.accept(self)

        self.symbol\_table.exit\_scope()

        node.eval\_type = TYPE\_VOID

作为整个语义分析的入口，此方法首先调用self.symbol\_table.enter\_scope()来创建一个 全局作用域（Scope Level 0）。随后，它遍历Program节点下包含的所有Statement节点，并依次调用每个语句的accept(self)方法，从而启动对整个 AST 的深度优先遍历。在所有语句分析完成后，调用self.symbol\_table.exit\_scope()退出全局作用域。最终，将Program节点的eval\_type属性设置为TYPE\_VOID，表示整个程序作为一个执行单元，不返回特定值。

1. visit\_block\_statement(node: BlockStatement)

def visit\_block\_statement(self, node: BlockStatement):

        self.symbol\_table.enter\_scope()

        block\_eval\_type = TYPE\_VOID

        for i, stmt in enumerate(node.statements):

            stmt.accept(self)

            if i == len(node.statements) - 1 and isinstance(stmt, ExpressionStatement):

                if stmt.expression:  # Check if there's an actual expression

                    block\_eval\_type = self.\_get\_node\_type(stmt.expression)

        node.eval\_type = block\_eval\_type  # Could be TYPE\_VOID or type of last expression

        self.symbol\_table.exit\_scope()

每当遇到一个代码块，此方法会立即调用self.symbol\_table.enter\_scope()，为该代码块创建一个 新的局部作用域。这意味着在块内部声明的变量，在块外部将不可见。方法遍历node.statements列表，递归地对块内的每一个语句调用accept(self)。在Rust中，代码块可以作为表达式返回一个值。此方法实现了这一特性：它检查块的最后一个语句是否为ExpressionStatement且其内部包含实际表达式，如果是，那么该表达式的eval\_type将被提升为整个BlockStatement的eval\_type；否则，块的eval\_type默认为TYPE\_VOID。在块内所有语句分析完成后，调用self.symbol\_table.exit\_scope()退出当前局部作用域，同时销毁该作用域内定义的所有局部符号。

1. visit\_function\_declaration(node: FunctionDeclarationStatement)

def visit\_function\_declaration(self, node: FunctionDeclarationStatement):

        func\_name = node.name.value

        param\_type\_objects = []

        for p\_node in node.parameters:

            if p\_node.type\_info:

                p\_node.type\_info.accept(self)

                param\_type = self.\_get\_node\_type(p\_node.type\_info)

            else:  # Should be caught by parser if types are mandatory

                self.symbol\_table.add\_error(

                    f"Parameter '{p\_node.name.value}' in function '{func\_name}' is missing a type.", p\_node)

                param\_type = TYPE\_ERROR

            param\_type\_objects.append(param\_type)

        func\_return\_type\_obj = TYPE\_VOID

        if node.return\_type:

            node.return\_type.accept(self)

            func\_return\_type\_obj = self.\_get\_node\_type(node.return\_type)

        fn\_type\_signature = FunctionType(param\_type\_objects, func\_return\_type\_obj)

        existing\_symbol = self.symbol\_table.scopes[self.symbol\_table.current\_scope\_level].get(func\_name)

        if existing\_symbol:  # Rudimentary check for redefinition

            self.symbol\_table.add\_error(f"Identifier '{func\_name}' (function) redefined in the same scope.", node.name)

        self.symbol\_table.define(func\_name, fn\_type\_signature, "function", is\_mutable=False,is\_initialized=True, node\_for\_error=node.name)

        node.name.eval\_type = fn\_type\_signature

        # --- Process function body in new scope ---

        old\_return\_type = self.current\_function\_return\_type

        self.current\_function\_return\_type = func\_return\_type\_obj

        self.symbol\_table.enter\_scope()

        param\_names = set()

        for i, param\_node in enumerate(node.parameters):

            p\_name = param\_node.name.value

            p\_type = param\_type\_objects[i]

            param\_node.eval\_type = p\_type  # Type of ParameterNode itself

            param\_node.name.eval\_type = p\_type  # Type of Identifier within ParameterNode

            if p\_name in param\_names:

                self.symbol\_table.add\_error(f"Duplicate parameter name '{p\_name}' in function '{func\_name}'.",

                                            param\_node.name)

            else:

                param\_names.add(p\_name)

            if p\_type != TYPE\_ERROR:  # Only define if type is not erroneous

                self.symbol\_table.define(p\_name, p\_type, "parameter", param\_node.mutable,is\_initialized=True, node\_for\_error=param\_node.name)  # Params are initialized

        if node.body:

            node.body.accept(self)

        self.symbol\_table.exit\_scope()

        self.current\_function\_return\_type = old\_return\_type

        node.eval\_type = TYPE\_VOID  # Function declaration statement is void

* 函数签名构建：首先，该方法处理函数的参数 (node.parameters) 和返回类型 (node.return\_type)。它会递归访问每个ParameterNode和TypeNode来确定它们的具体Type对象。这些类型对象被收集起来，用于构建一个FunctionType实例，作为该函数的完整类型签名。
* 函数符号定义（外层作用域）：使用函数名 (node.name.value) 和刚刚构建的FunctionType签名，调用self.symbol\_table.define()在当前（函数声明所在）作用域中定义该函数符号。其kind被设置为"function"，is\_mutable为False，is\_initialized为True（函数本身在声明时即被认为是“初始化”可用的）。这里也会检查是否存在同名函数重定义，并报告错误。
* 进入函数体作用域：为了处理函数参数和函数体内部的局部变量，方法调用self.symbol\_table.enter\_scope()创建一个新的作用域，这个作用域是函数体内部的局部作用域。
* 参数符号定义（内层作用域）：遍历函数的所有参数。对每个参数，将其作为局部变量，在当前函数体作用域中调用self.symbol\_table.define()进行定义。参数的kind被设置为"parameter"，并且is\_initialized始终为True，因为参数在函数调用时就会被赋值。同时，检查函数参数列表中是否存在重复的参数名，若有则报告错误。
* 返回类型上下文：将当前函数的期望返回类型 (func\_return\_type\_obj) 存储到self.current\_function\_return\_type变量中。这个变量将在visit\_return\_statement中用于检查return语句的类型是否与函数声明的返回类型一致。
* 访问函数体：递归调用node.body.accept(self)来分析函数体内部的所有语句和表达式。
* 退出函数体作用域：恢复self.current\_function\_return\_type为进入函数前的状态，并调用self.symbol\_table.exit\_scope()退出函数体的局部作用域。

1. visit\_let\_statement(node: LetStatement)

def visit\_let\_statement(self, node: LetStatement):

        var\_name = node.name.value

        is\_mutable = node.mutable

        declared\_type\_obj = TYPE\_UNKNOWN

        if node.type\_info:

            node.type\_info.accept(self)  # Sets node.type\_info.eval\_type

            declared\_type\_obj = self.\_get\_node\_type(node.type\_info)

        value\_type\_from\_expr = TYPE\_VOID  # Type of the initializer expression

        r\_value\_is\_concrete\_and\_error\_free = False  # Can R-value be used for inference/initialization?

        if node.value:

            value\_type\_from\_expr = self.\_process\_expr\_in\_read\_context(node.value)

            if value\_type\_from\_expr not in [TYPE\_ERROR, TYPE\_VOID, TYPE\_UNKNOWN]:

                r\_value\_is\_concrete\_and\_error\_free = True

        final\_var\_type = TYPE\_ERROR  # The type to be stored for the variable

        is\_initialized\_by\_this\_let = False

        if declared\_type\_obj != TYPE\_UNKNOWN and declared\_type\_obj != TYPE\_ERROR:

            # Case 1: Explicit type annotation (e.g., let x: T)

            final\_var\_type = declared\_type\_obj

            if node.value:  # Has initializer (e.g., let x: T = val)

                if value\_type\_from\_expr != TYPE\_ERROR:  # Initializer expression is valid

                    if value\_type\_from\_expr == final\_var\_type or \

                            value\_type\_from\_expr == TYPE\_UNKNOWN:  # Compatible or inferable

                        is\_initialized\_by\_this\_let = True

                    else:  # Type mismatch

                        self.symbol\_table.add\_error(

                            f"Type mismatch for variable '{var\_name}'. Explicit type is '{final\_var\_type}', initializer is '{value\_type\_from\_expr}'.",

                            node

                        )

                # else: initializer had an error, so not initialized by this let.

            # else: no initializer (e.g. let x: T;), not initialized by this let.

        elif node.value:

            # Case 2: No explicit type, but has initializer (e.g., let x = val) - Type Inference

            if r\_value\_is\_concrete\_and\_error\_free:

                final\_var\_type = value\_type\_from\_expr

                is\_initialized\_by\_this\_let = True

            else:

                self.symbol\_table.add\_error(

                    f"Cannot infer type for variable '{var\_name}' from initializer of type '{value\_type\_from\_expr}'.",

                    node)

                # final\_var\_type remains TYPE\_ERROR

        else:

            # Case 3: No explicit type, no initializer (e.g., let x;) - Error

            self.symbol\_table.add\_error(

                f"Variable '{var\_name}' needs an explicit type or an initializer for type inference.", node)

            # final\_var\_type remains TYPE\_ERROR

        # Define the symbol to handle shadowing, even if its type is problematic or an error occurred.

        self.symbol\_table.define(var\_name, final\_var\_type, "variable", is\_mutable,

                                 is\_initialized=is\_initialized\_by\_this\_let, node\_for\_error=node.name)

        node.eval\_type = TYPE\_VOID  # Let statement itself is void

        node.name.eval\_type = final\_var\_type  # Store type on the Identifier node of the declaration

* 类型注解处理：如果let语句包含显式类型注解 (node.type\_info)，则先对其进行语义分析以确定declared\_type\_obj。
* 初始化表达式处理：如果let语句包含初始化表达式 (node.value)，这是关键点： 它会调用self.\_process\_expr\_in\_read\_context(node.value)来分析这个表达式。\_process\_expr\_in\_read\_context会将内部的\_is\_read\_context标志设置为True，确保在处理初始化表达式中可能出现的标识符时，会触发“使用前未初始化”的检查（如果适用）。
* 类型推导与匹配：

显式类型 + 初始化器 (let x: T = val;)：final\_var\_type被确定为declared\_type\_obj。然后，检查value\_type\_from\_expr是否与final\_var\_type匹配。若不匹配，则报告类型不一致错误。如果匹配，is\_initialized\_by\_this\_let设为True。

无显式类型 + 初始化器 (let x = val;)：final\_var\_type从value\_type\_from\_expr中推导得出。如果value\_type\_from\_expr是TYPE\_ERROR或TYPE\_UNKNOWN，则表示无法推导，报告错误。如果推导成功，is\_initialized\_by\_this\_let设为True。

显式类型 + 无初始化器 (let x: T;)：final\_var\_type为declared\_type\_obj。is\_initialized\_by\_this\_let设为False。

无显式类型 + 无初始化器 (let x;)：报告错误：“变量需要显式类型或初始化器进行类型推导。”final\_var\_type设为TYPE\_ERROR。

**变量使用与赋值相关规则的处理**

1. visit\_identifier(node: Identifier)

def visit\_identifier(self, node: Identifier):

        symbol = self.symbol\_table.resolve(node.value)

        if not symbol:

            self.symbol\_table.add\_error(f"Identifier '{node.value}' not declared.", node)

            node.eval\_type = TYPE\_ERROR

            return

        node.eval\_type = symbol.type  # Set the type of the AST identifier node

        if self.\_is\_read\_context and symbol.kind == "variable" and not symbol.is\_initialized:

            self.symbol\_table.add\_error(f"Variable '{node.value}' used before being initialized.", node)

* 声明检查：调用self.symbol\_table.resolve(node.value)尝试解析标识符。如果找不到对应的符号，则报告“标识符未声明”错误，并将node.eval\_type设为TYPE\_ERROR。
* 变量使用前未初始化的核心修正逻辑：如果标识符解析成功，并且当前语义分析器内部的self.\_is\_read\_context标志为True（表示该标识符正在作为右值被读取其内容），同时该符号是一个变量 (symbol.kind == "variable") 并且其is\_initialized属性为False，则报告“变量在使用前未初始化”错误。
* 将解析到的symbol.type赋值给node.eval\_type，以便类型信息能附着在 AST 节点上。

1. visit\_assignment\_statement(node: AssignmentStatement)

def visit\_assignment\_statement(self, node: AssignmentStatement):

        # --- 1. Process Target (L-value) ---

        # Temporarily set read\_context to False for the target itself

        original\_is\_read\_context = self.\_is\_read\_context

        self.\_is\_read\_context = False

        node.target.accept(self)  # This resolves the target (e.g., calls visit\_identifier)

        self.\_is\_read\_context = original\_is\_read\_context

        target\_ast\_node\_type = self.\_get\_node\_type(node.target)

        target\_symbol = None

        if isinstance(node.target, Identifier):

            target\_symbol = self.symbol\_table.resolve(node.target.value)

            if not target\_symbol:

                # Error should have been caught by node.target.accept() -> visit\_identifier

                node.eval\_type = TYPE\_ERROR

                return

            if target\_ast\_node\_type == TYPE\_ERROR:  # If identifier resolution or type itself is an error

                node.eval\_type = TYPE\_ERROR

                return

        else:

            # Add support for other L-values like array indexing, field access later

            self.symbol\_table.add\_error(

                f"Target of assignment must be a simple variable for now. Got {type(node.target).\_\_name\_\_}.",

                node.target)

            node.eval\_type = TYPE\_ERROR

            return

        if not target\_symbol.is\_mutable:

            self.symbol\_table.add\_error(

                f"Cannot assign to immutable variable '{target\_symbol.name}'. Variable must be declared 'mut'.",

                node.target)

            # Assignment is invalid due to immutability, but still check type of R-value.

        # --- 2. Process Value (R-value) ---

        value\_type\_from\_expr = self.\_process\_expr\_in\_read\_context(node.value)

        if value\_type\_from\_expr == TYPE\_ERROR:  # Error in evaluating the R-value

            node.eval\_type = TYPE\_ERROR  # Assignment statement is erroneous

            return

        # --- 3. Type Check and Update Initialization ---

        if target\_symbol.type != TYPE\_ERROR and \

                value\_type\_from\_expr != target\_symbol.type and \

                value\_type\_from\_expr != TYPE\_UNKNOWN:  # Allow assigning 'unknown' for now, might tighten later

            self.symbol\_table.add\_error(

                f"Type mismatch in assignment to '{target\_symbol.name}'. Expected '{target\_symbol.type}', got '{value\_type\_from\_expr}'.",

                node

            )

        else:  # Types are compatible (or target was error, or value was unknown)

            # Mark as initialized only if target is mutable and types are fine

            if target\_symbol.type != TYPE\_ERROR and target\_symbol.is\_mutable:

                target\_symbol.is\_initialized = True

        node.eval\_type = TYPE\_VOID  # Assignment statement itself is void

* 左值（L-value）处理：对赋值的目标 (node.target) 进行语义分析。在访问node.target时，self.\_is\_read\_context会被暂时设置为False。这是因为赋值的目标（例如一个变量名）本身并不是在“读取”其值，而是在“写入”其位置。如果node.target是一个Identifier，解析其对应的符号。如果目标符号未声明，则报告错误。检查目标变量的is\_mutable属性。如果变量是不可变的，则报告“无法对不可变变量赋值”错误。
* 右值（R-value）处理：对赋值表达式node.value调用self.\_process\_expr\_in\_read\_context(node.value)进行语义分析。这将获取右值的类型，并确保右值表达式中使用的任何变量都经过了初始化检查。比较左值（目标变量）的类型与右值表达式的类型。如果它们不兼容，则报告类型不匹配错误。
* 更新初始化状态：如果赋值操作在语义上是合法的（即目标变量已声明、可变，且类型匹配），则将目标变量在符号表中的is\_initialized属性设置为True。这表示该变量现在已被赋值，后续的读取操作将不再报告未初始化错误。

**表达式类型推导与检查**

1. visit\_integer\_literal(node: IntegerLiteral)和visit\_boolean\_literal(node: BooleanLiteral)

    def visit\_integer\_literal(self, node: IntegerLiteral):

        node.eval\_type = TYPE\_I32

   def visit\_boolean\_literal(self, node: BooleanLiteral):

        node.eval\_type = TYPE\_BOOL

这些是最简单的表达式，它们直接将自身的eval\_type属性设置为对应的基本类型TYPE\_I32或TYPE\_BOOL。

1. visit\_infix\_expression(node: InfixExpression)

def visit\_infix\_expression(self, node: InfixExpression):

        left\_type = self.\_process\_expr\_in\_read\_context(node.left)

        right\_type = self.\_process\_expr\_in\_read\_context(node.right)

        op = node.operator

        result\_type = TYPE\_ERROR

        if left\_type == TYPE\_ERROR or right\_type == TYPE\_ERROR:

            node.eval\_type = TYPE\_ERROR

            return

        if op in ['+', '-', '\*', '/']:

            if left\_type == TYPE\_I32 and right\_type == TYPE\_I32:

                result\_type = TYPE\_I32

            else:

                self.symbol\_table.add\_error(

                    f"Arithmetic operator '{op}' requires i32 operands, got '{left\_type}' and '{right\_type}'.", node)

        elif op in ['<', '<=', '>', '>=', '==', '!=']:

            if (left\_type == TYPE\_I32 and right\_type == TYPE\_I32) or \

                    (left\_type == TYPE\_BOOL and right\_type == TYPE\_BOOL and op in ['==', '!=']):

                result\_type = TYPE\_BOOL

            else:

                self.symbol\_table.add\_error(

                    f"Comparison operator '{op}' expects compatible types (i32-i32 or bool-bool for equality), got '{left\_type}' and '{right\_type}'.",

                    node)

        elif op in ['&&', '||']:  # Assuming these are added to parser

            if left\_type == TYPE\_BOOL and right\_type == TYPE\_BOOL:

                result\_type = TYPE\_BOOL

            else:

                self.symbol\_table.add\_error(

                    f"Logical operator '{op}' requires boolean operands, got '{left\_type}' and '{right\_type}'.", node)

        else:

            self.symbol\_table.add\_error(f"Unsupported infix operator '{op}'.", node)

        node.eval\_type = result\_type

* 操作分析：对左右操作数node.left和node.right均调用\_process\_expr\_in\_read\_context()，以确保它们作为右值被正确分析，并获取它们的类型。
* 错误传播：如果任何一个操作数的类型是TYPE\_ERROR，则整个中缀表达式的类型也将是TYPE\_ERROR。
* 类型检查逻辑（基于运算符）：

算术运算符 (+,-,\*,/)：严格要求左右操作数均为TYPE\_I32。如果满足，结果类型为TYPE\_I32。否则，报告类型不匹配错误。

比较运算符 (<,<=,>,>=,==,!=)：结果类型始终为TYPE\_BOOL。对于==,!=，允许TYPE\_I32与TYPE\_I32比较，或TYPE\_BOOL与TYPE\_BOOL比较。对于<,<=,>,>=，只允许TYPE\_I32与TYPE\_I32比较。若不满足上述条件，则报告类型不兼容错误。逻辑运算符 (&&,||)：严格要求左右操作数均为TYPE\_BOOL。如果满足，结果类型为TYPE\_BOOL。否则，报告类型不匹配错误。

1. visit\_prefix\_expression(node: PrefixExpression)

def visit\_prefix\_expression(self, node: PrefixExpression):

        right\_type = self.\_process\_expr\_in\_read\_context(node.right)

        op = node.operator

        result\_type = TYPE\_ERROR

        if right\_type == TYPE\_ERROR:

            node.eval\_type = TYPE\_ERROR

            return

        if op == '-':

            if right\_type == TYPE\_I32:

                result\_type = TYPE\_I32

            else:

                self.symbol\_table.add\_error(f"Unary minus operator '-' requires i32 operand, got '{right\_type}'.", node)

        elif op == '!':

            if right\_type == TYPE\_BOOL:

                result\_type = TYPE\_BOOL

            else:

                self.symbol\_table.add\_error(f"Logical NOT '!' requires boolean operand, got '{right\_type}'.", node)

        else:

            self.symbol\_table.add\_error(f"Unsupported prefix operator '{op}'.", node)

        node.eval\_type = result\_type

* 操作分析：对右操作数node.right调用\_process\_expr\_in\_read\_context()获取其类型。
* 错误传播：如果右操作数类型为TYPE\_ERROR，则前缀表达式结果也为TYPE\_ERROR。
* 类型检查逻辑：

一元减号 (-)：要求操作数为TYPE\_I32，结果为TYPE\_I32。否则，报告类型不匹配错误。

逻辑非 (!)：要求操作数为TYPE\_BOOL，结果为TYPE\_BOOL。否则，报告类型不匹配错误。

1. visit\_call\_expression(node: CallExpression)

def visit\_call\_expression(self, node: CallExpression):

        func\_signature\_type = self.\_process\_expr\_in\_read\_context(node.function)

        if not isinstance(func\_signature\_type, FunctionType):

            if func\_signature\_type != TYPE\_ERROR:

                func\_name\_str = node.function.value if isinstance(node.function, Identifier) else "Expression"

                self.symbol\_table.add\_error(f"'{func\_name\_str}' is not a function and cannot be called.", node.function)

            node.eval\_type = TYPE\_ERROR

            return

        expected\_param\_types = func\_signature\_type.param\_types

        num\_expected\_args = len(expected\_param\_types)

        num\_actual\_args = len(node.arguments)

        if num\_actual\_args != num\_expected\_args:

            func\_name\_str = node.function.value if isinstance(node.function, Identifier) else "Function"

            self.symbol\_table.add\_error(

                f"{func\_name\_str} expects {num\_expected\_args} arguments, but {num\_actual\_args} were provided.", node

            )

            node.eval\_type = TYPE\_ERROR  # Mismatch in arg count is a call error

            return  # Stop further processing for this call

        arg\_type\_error\_found = False

        for i in range(num\_actual\_args):

            arg\_expr = node.arguments[i]

            actual\_arg\_type = self.\_process\_expr\_in\_read\_context(arg\_expr)

            expected\_arg\_type = expected\_param\_types[i]

            if actual\_arg\_type != TYPE\_ERROR and expected\_arg\_type != TYPE\_ERROR:

                if actual\_arg\_type != expected\_arg\_type and actual\_arg\_type != TYPE\_UNKNOWN:

                    self.symbol\_table.add\_error(

                        f"Type mismatch for argument {i + 1} in call to '{node.function.value if isinstance(node.function, Identifier) else 'function'}'. Expected '{expected\_arg\_type}', got '{actual\_arg\_type}'.",

                        arg\_expr

                    )

                    arg\_type\_error\_found = True

            elif actual\_arg\_type == TYPE\_ERROR:  # Propagate error from argument expression

                arg\_type\_error\_found = True

        if arg\_type\_error\_found:

            node.eval\_type = TYPE\_ERROR

        else:

            node.eval\_type = func\_signature\_type.return\_type

* 函数名解析：首先对node.function（通常是一个Identifier节点，但理论上也可以是返回函数指针的表达式）调用\_process\_expr\_in\_read\_context()来获取其类型。
* 函数类型验证：检查获取到的类型是否是FunctionType的实例。如果不是（且不是TYPE\_ERROR），则报告错误：“不是函数，无法调用”。
* 参数数量检查：比较实参数量 (len(node.arguments)) 与函数签名中期望的形参数量 (len(func\_signature\_type.param\_types))。若不匹配，则报告错误。
* 参数类型检查：遍历每个实参 (node.arguments)，对每个实参表达式调用\_process\_expr\_in\_read\_context()来获取其类型。将实参的实际类型与函数签名中对应位置形参的期望类型进行比较。若不匹配（且都不是TYPE\_ERROR或TYPE\_UNKNOWN），则报告类型不匹配错误。如果所有检查都通过（没有类型错误或参数数量错误），则将CallExpression节点的eval\_type设置为函数签名的返回类型 (func\_signature\_type.return\_type)。否则，设置为TYPE\_ERROR。

**控制流语句的处理**

1. visit\_return\_statement(node: ReturnStatement)

def visit\_return\_statement(self, node: ReturnStatement):

        if self.current\_function\_return\_type is None:

            self.symbol\_table.add\_error("'return' statement found outside of a function.", node)

            node.eval\_type = TYPE\_VOID  # Or a "never" type

            return

        expected\_ret\_type = self.current\_function\_return\_type

        actual\_ret\_type = TYPE\_VOID

        if node.return\_value:

            actual\_ret\_type = self.\_process\_expr\_in\_read\_context(node.return\_value)

        if expected\_ret\_type != TYPE\_ERROR and actual\_ret\_type != TYPE\_ERROR:

            if expected\_ret\_type != actual\_ret\_type and actual\_ret\_type != TYPE\_UNKNOWN:

                self.symbol\_table.add\_error(

                    f"Return type mismatch. Function expects '{expected\_ret\_type}', but found '{actual\_ret\_type}'.",

                    node.return\_value if node.return\_value else node

                )

        node.eval\_type = TYPE\_VOID

* 上下文检查：首先检查self.current\_function\_return\_type是否为None。如果为None，意味着return语句出现在函数外部，则报告错误。
* 返回值处理：如果return语句带有返回值表达式 (node.return\_value)，则调用\_process\_expr\_in\_read\_context()对其进行语义分析，并获取其实际类型。
* 类型匹配： 比较返回值表达式的实际类型 (actual\_ret\_type) 与当前函数声明的期望返回类型 (expected\_ret\_type)。若两者不一致（且都不是TYPE\_ERROR或TYPE\_UNKNOWN），则报告“返回类型不匹配”错误。

1. visit\_if\_statement(node: IfStatement)

def visit\_if\_statement(self, node: IfStatement):

        cond\_type = self.\_process\_expr\_in\_read\_context(node.condition)

        if cond\_type != TYPE\_BOOL and cond\_type != TYPE\_ERROR and cond\_type != TYPE\_UNKNOWN:

            self.symbol\_table.add\_error(f"If condition must evaluate to a boolean, got '{cond\_type}'.", node.condition)

        node.consequence.accept(self)

        consequence\_type = self.\_get\_node\_type(node.consequence)

        alternative\_type = TYPE\_VOID

        if node.alternative:

            node.alternative.accept(self)

            alternative\_type = self.\_get\_node\_type(node.alternative)

        # Type of the if-(else) construct if used as an expression

        if node.alternative:  # if-else

            if consequence\_type != TYPE\_ERROR and alternative\_type != TYPE\_ERROR:

                if consequence\_type == alternative\_type:

                    node.eval\_type = consequence\_type

                elif consequence\_type == TYPE\_VOID or alternative\_type == TYPE\_VOID:

                    if consequence\_type == TYPE\_VOID and alternative\_type == TYPE\_VOID:

                        node.eval\_type = TYPE\_VOID

                    else:

                        self.symbol\_table.add\_error(

                            f"If-else branches have incompatible types for expression: 'then' is '{consequence\_type}', 'else' is '{alternative\_type}'.",

                            node

                        )

                        node.eval\_type = TYPE\_ERROR

                else:  # Mismatch, neither is void

                    self.symbol\_table.add\_error(

                        f"If-else branches have incompatible types: 'then' branch is '{consequence\_type}', 'else' branch is '{alternative\_type}'.",

                        node

                    )

                    node.eval\_type = TYPE\_ERROR

            else:  # One of the branches had an error

                node.eval\_type = TYPE\_ERROR

        else:  # if without else (evaluates to void or unit type)

            node.eval\_type = TYPE\_VOID

* 条件表达式：对node.condition调用\_process\_expr\_in\_read\_context()。检查其类型是否为TYPE\_BOOL。如果不是，则报告错误。
* 分支体分析：递归访问node.consequence（then 块）和node.alternative（else 块，如果存在）。如果存在else分支，语义分析器会获取consequence和alternative块各自的eval\_type。
* 类型处理：若两者类型相同，则整个if-else表达式的eval\_type即为该类型。若类型不一致，且至少有一个分支的类型不是TYPE\_VOID（即不是纯粹的语句），则报告“if-else 分支类型不兼容”错误。

1. visit\_while\_statement(node: WhileStatement)

def visit\_while\_statement(self, node: WhileStatement):

        self.loop\_depth += 1

        cond\_type = self.\_process\_expr\_in\_read\_context(node.condition)

        if cond\_type != TYPE\_BOOL and cond\_type != TYPE\_ERROR and cond\_type != TYPE\_UNKNOWN:

            self.symbol\_table.add\_error(f"While condition must be boolean, got '{cond\_type}'.", node.condition)

        node.body.accept(self)

        self.loop\_depth -= 1

        node.eval\_type = TYPE\_VOID

* 循环深度管理：在进入循环体之前，将self.loop\_depth计数器递增。这个计数器用于检查break和continue语句是否合法地出现在循环内部。
* 条件表达式：对node.condition调用\_process\_expr\_in\_read\_context()。检查其类型是否为TYPE\_BOOL。若不是，则报告错误。
* 循环体分析：递归访问node.body。
* 循环深度管理：在退出循环体之后，将self.loop\_depth计数器递减。

1. visit\_loop\_statement(node: LoopStatement)

def visit\_loop\_statement(self, node: LoopStatement):

        self.loop\_depth += 1

        node.body.accept(self)

        self.loop\_depth -= 1

        node.eval\_type = TYPE\_VOID

循环深度管理：与while循环类似，在进入loop体之前递增self.loop\_depth，之后递减。

循环体分析：递归访问node.body。

1. visit\_break\_statement(node: BreakStatement)

    def visit\_break\_statement(self, node: BreakStatement):

        if self.loop\_depth == 0:

            self.symbol\_table.add\_error("'break' statement found outside of a loop.", node)

        node.eval\_type = TYPE\_VOID  # or a "never" type conceptually

* 上下文检查：核心是检查self.loop\_depth。如果self.loop\_depth为 0，表示break语句出现在任何循环之外，则报告“'break' 语句在循环外部”错误。

1. visit\_continue\_statement(node: ContinueStatement)

    def visit\_continue\_statement(self, node: ContinueStatement):

        if self.loop\_depth == 0:

            self.symbol\_table.add\_error("'continue' statement found outside of a loop.", node)

        node.eval\_type = TYPE\_VOID

* 上下文检查：同样检查self.loop\_depth。如果为 0，则报告“'continue' 语句在循环外部”错误。

## 2、中间代码生成

1. 核心数据结构和思想

核心数据结构：

1. Quadruple

表示一条四元式指令，是生成的目标中间代码的基本单元。每个实例包含四个主要部分：op（操作符或字符串类型），arg1（第一个操作数），arg2（第二个操作数），以及result （存放结果的目标）。整个中间代码就是由一个 Quadruple 对象的列表构成。

class Quadruple:

    def \_\_init\_\_(self, op, arg1, arg2, result):

        self.op = op        # 操作符 (字符串，如 '+', '-', 'assign', 'goto', 'if\_false\_goto', 'call', 'label')

        self.arg1 = arg1    # 操作数1 (IRValue 实例, 或 None)

        self.arg2 = arg2    # 操作数2 (IRValue 实例, 或 None)

        self.result = result # 结果 (IRValue 实例, 或 None)

1. IRValue

用于封装四元式中的操作数和结果，它不仅仅存储一个值，还通过布尔标记来区分这个值的类型或含义，使得四元式中的各个字段具有更明确的语义，方便IR生成器内部逻辑判断以及后续阶段对IR的解析。

class IRValue:

    def \_\_init\_\_(self, value, is\_temp=False, is\_label=False, is\_const=False, is\_func\_name=False, var\_name=None):

        self.value = value # 实际值或名称 (如 t0, L1, 5, my\_var, my\_func)

        self.is\_temp = is\_temp

        self.is\_label = is\_label

        self.is\_const = is\_const

        self.is\_func\_name = is\_func\_name

        # var\_name 字段可以用来存储原始变量名，即使value是处理过的（例如加了作用域信息）

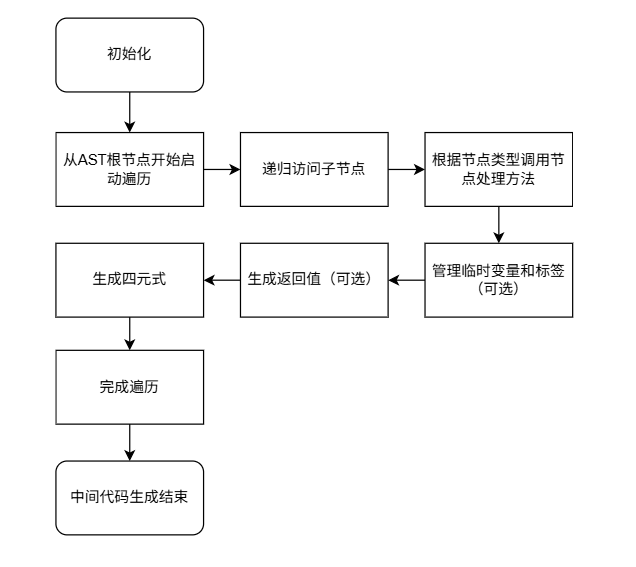
        # 但对于简单IR，value本身通常就是变量名或临时变量名

        self.original\_name = var\_name if var\_name else (str(value) if not (is\_temp or is\_label or is\_func\_name) else None)

核心思想：

中间代码生成的核心思想是采用访问者模式递归遍历经过语义分析的AST，针对每种AST节点类型，通过一套预定义的规则将其操作和结构系统性地转换为四元式指令，同时利用临时变量存储中间计算结果，并通过生成和使用标签来显式地表达程序的控制流程，最终形成能够清晰反映原始程序数据流和控制流的中间表示。

1. 中间代码生成流程



1. 重点功能实现函数解析
2. \_emit(self, op, arg1=None, arg2=None, result=None)

功能说明：创建并添加一个新的 Quadruple 对象到 self.quads 列表中。

实现过程：内部 wrap 函数负责将Python的简单类型或未包装的 IRValue 统一转换为 IRValue 对象或保持为 None。用给定的操作符 (op) 和处理过的操作数 (arg1, arg2, result) 创建 Quadruple 实例，并追加到 self.quads。

    def \_emit(self, op, arg1=None, arg2=None, result=None):

        # 确保操作数是 IRValue 类型或 None

        def wrap(val):

            if val is None or isinstance(val, IRValue):

                return val

            if isinstance(val, (int, float)): # bool 会被 isinstance(int) 捕获

                return IRValue(val, is\_const=True)

            if isinstance(val, bool): # 单独处理布尔值，通常转为 0 或 1

                 return IRValue(1 if val else 0, is\_const=True)

            if isinstance(val, str): # 假设字符串是变量名，除非它是特殊标记

                return IRValue(val) # 作为普通变量名

            raise TypeError(f"Cannot directly emit value of type {type(val)} as IR operand. Wrap in IRValue or handle specifically.")

        self.quads.append(Quadruple(op, wrap(arg1), wrap(arg2), wrap(result)))

1. visit\_program(self, node: Program)

功能说明：处理程序的根节点。

实现过程：发射一个 program\_begin 四元式作为程序的开始标记。遍历 node.statements（程序顶层的语句列表），对每个语句调用其 accept(self) 方法，递归地生成该语句的IR。

发射一个 program\_end 四元式作为程序的结束标记。

    def visit\_program(self, node: Program):

        self.\_emit("program\_begin")

        for stmt in node.statements:

            stmt.accept(self)

        self.\_emit("program\_end")

        return None

1. visit\_let\_statement(self, node: LetStatement)

功能说明：处理 let 变量声明语句。

实现过程：创建表示变量名的 IRValue。如果 node.value (初始化表达式) 存在，递归调用 node.value.accept(self) 生成初始化表达式的IR，其返回值应为一个 IRValue (代表存储表达式结果的临时变量或常量)，如果返回值有效，则发射一个 assign 四元式，形如 (assign, <value\_ir>, \_, <var\_ir\_value>)，将表达式的值赋给声明的变量。

    def visit\_let\_statement(self, node: LetStatement):

        var\_ir\_value = IRValue(node.name.value) # 变量名

        # 可以在这里发射 'declare' 四元式，如果后端需要明确的声明信息

        # self.\_emit('declare', var\_ir\_value, str(node.name.eval\_type)) # eval\_type 是语义分析时赋予的

        if node.value:

            value\_ir = node.value.accept(self)

            if value\_ir is not None: # 确保表达式有结果

                self.\_emit("assign", value\_ir, None, var\_ir\_value)

        return None

1. visit\_assignment\_statement(self, node: AssignmentStatement)

功能说明：处理赋值语句。

实现过程：递归调用 node.target.accept(self)，得到目标变量的 IRValue，递归调用 node.value.accept(self)，得到赋值表达式结果的 IRValue。如果两者都有效，发射 assign 四元式(assign, <value\_ir>, \_, <target\_ir\_value>)。

    def visit\_assignment\_statement(self, node: AssignmentStatement):

        # 假设 target 是 Identifier

        target\_ir\_value = node.target.accept(self) # 通常是 IRValue(name)

        value\_ir = node.value.accept(self)

        if target\_ir\_value is not None and value\_ir is not None:

            self.\_emit("assign", value\_ir, None, target\_ir\_value)

        return None

1. visit\_infix\_expression(self, node: InfixExpression)

功能说明：处理中缀表达式。

实现过程：递归调用 node.left.accept(self) 和 node.right.accept(self)，分别得到左右操作数的 IRValue，创建一个新的临时变量 result\_temp = self.\_new\_temp() 用于存储运算结果，使用 op\_map 将AST中的操作符映射到IR中更规范的操作符，发射相应的二元运算四元式。

    def visit\_infix\_expression(self, node: InfixExpression):

        left\_ir = node.left.accept(self)

        right\_ir = node.right.accept(self)

        result\_temp = self.\_new\_temp()

        # 映射操作符 (AST的 '==' '!=' 等到 IR 的 'eq' 'ne' 'lt' 等)

        op\_map = {

            '+': '+', '-': '-', '\*': '\*', '/': '/',

            '==': 'eq', '!=': 'ne',

            '<': 'lt', '<=': 'le',

            '>': 'gt', '>=': 'ge',

            # 如果有 '&&' '||', 它们通常需要转化为控制流，而不是简单的二元操作

        }

        ir\_op = op\_map.get(node.operator, node.operator) # 默认使用原操作符

        self.\_emit(ir\_op, left\_ir, right\_ir, result\_temp)

        return result\_temp

1. visit\_prefix\_expression(self, node: PrefixExpression)

功能说明：处理前缀表达式。

实现过程：递归调用 node.right.accept(self) 得到操作数的 IRValue，创建一个新的临时变量 result\_temp，使用 op\_map 映射操作符，发射一元运算四元式。

    def visit\_prefix\_expression(self, node: PrefixExpression):

        operand\_ir = node.right.accept(self)

        result\_temp = self.\_new\_temp()

        # 映射操作符

        op\_map = {'-': 'uminus', '!': 'not'} # '!' (逻辑非)

        ir\_op = op\_map.get(node.operator, f"unary\_{node.operator}") # 如 "unary\_&"

        self.\_emit(ir\_op, operand\_ir, None, result\_temp)

        return result\_temp

1. visit\_block\_statement(self, node: BlockStatement)

功能说明：处理块语句。

实现过程：遍历块内的所有语句 node.statements，对每个语句递归调用 accept(self)。块语句本身在简单IR中不产生值。

    def visit\_block\_statement(self, node: BlockStatement):

        for stmt in node.statements:

            stmt.accept(self)

        # 如果块作为表达式，最后一个表达式的值可能是块的值

        # 但作为语句，它不返回值。这需要语义分析的 `eval\_type` 来辅助判断。

        # 当前简单实现中，块语句本身不产生IR值。

        return None

1. visit\_if\_statement(self, node: IfStatement)

功能说明：处理 if-else 或 if 语句。

实现过程：递归调用 node.condition.accept(self) 得到条件求值结果的 IRValue，创建两个新标签else\_label (如果条件为假跳转到此) 和 end\_if\_label (if语句结束后跳转到此)。发射 if\_false\_goto 四元式：(if\_false\_goto, <cond\_ir>, <else\_label>, \_)。递归调用 node.consequence.accept(self) 生成 then 分支的IR，如果存在 node.alternative (else分支)，在 then 分支后发射一个无条件跳转 goto 到 end\_if\_label，以跳过 else 分支。发射 else\_label 标签：(label, <else\_label>, \_, \_)。如果存在 node.alternative，递归调用 node.alternative.accept(self) 生成 else 分支的IR。发射 end\_if\_label 标签：(label, <end\_if\_label>, \_, \_)。

    def visit\_if\_statement(self, node: IfStatement):

        cond\_ir = node.condition.accept(self)

        else\_label = self.\_new\_label("Else")

        end\_if\_label = self.\_new\_label("EndIf")

        # 如果条件为假 (0)，则跳转到 else\_label

        self.\_emit("if\_false\_goto", cond\_ir, else\_label, None)

        # Then 部分

        node.consequence.accept(self)

        if node.alternative: # 只有在有 else 分支时才需要跳过它

            self.\_emit("goto", end\_if\_label, None, None)

        self.\_emit("label", else\_label, None, None)

        if node.alternative:

            node.alternative.accept(self)

        self.\_emit("label", end\_if\_label, None, None)

        return None

1. visit\_while\_statement(self, node: WhileStatement)

功能说明：处理 while 循环语句。

实现过程：创建两个新标签：start\_loop\_label (循环开始，条件判断处) 和 end\_loop\_label (循环结束，跳出循环后)。将 end\_loop\_label压入 self.loop\_exit\_labels 栈 (供 break 使用)，将 start\_loop\_label压入 self.loop\_continue\_labels 栈 (供 continue 使用)。

发射 start\_loop\_label 标签。递归调用 node.condition.accept(self) 得到条件结果 cond\_ir。

发射 if\_false\_goto 四元式，如果条件为假，则跳转到 end\_loop\_label。递归调用 node.body.accept(self) 生成循环体的IR。发射 goto 四元式，无条件跳转回 start\_loop\_label 进行下一次条件判断。发射 end\_loop\_label 标签。从栈中弹出相应的标签。

    def visit\_while\_statement(self, node: WhileStatement):

        start\_loop\_label = self.\_new\_label("WhileStart")

        # body\_label = self.\_new\_label("WhileBody") # 可以不需要，直接在条件后

        end\_loop\_label = self.\_new\_label("WhileEnd")

        self.loop\_exit\_labels.append(end\_loop\_label)

        self.loop\_continue\_labels.append(start\_loop\_label) # continue 回到条件判断

        self.\_emit("label", start\_loop\_label, None, None)

        cond\_ir = node.condition.accept(self)

        self.\_emit("if\_false\_goto", cond\_ir, end\_loop\_label, None)

        # Loop body

        node.body.accept(self)

        self.\_emit("goto", start\_loop\_label, None, None) # 返回循环开始

        self.\_emit("label", end\_loop\_label, None, None)

        self.loop\_exit\_labels.pop()

        self.loop\_continue\_labels.pop()

        return None

1. visit\_loop\_statement(self, node: LoopStatement)

功能说明：处理无限 loop 语句。

实现过程：创建两个新标签：start\_label (循环体开始) 和 end\_label (循环结束，break 跳转目标)。将 end\_label 压入 self.loop\_exit\_labels，将 start\_label 压入 self.loop\_continue\_labels。发射 start\_label 标签。递归调用 node.body.accept(self) 生成循环体IR。发射 goto 四元式，无条件跳转回 start\_label。发射 end\_label 标签，从栈中弹出标签。

    def visit\_loop\_statement(self, node: LoopStatement):

        start\_label = self.\_new\_label("LoopStart")

        end\_label = self.\_new\_label("LoopEnd")

        self.loop\_exit\_labels.append(end\_label)

        self.loop\_continue\_labels.append(start\_label) # continue 回到循环体开始

        self.\_emit("label", start\_label, None, None)

        node.body.accept(self)

        self.\_emit("goto", start\_label, None, None) # 无条件跳回循环开始

        self.\_emit("label", end\_label, None, None) # break 跳到这里

        self.loop\_exit\_labels.pop()

        self.loop\_continue\_labels.pop()

        return None

1. visit\_break\_statement(self, node: BreakStatement)

功能说明：处理 break 语句。

实现过程：检查 self.loop\_exit\_labels 栈是否为空，若为空则表示 break 不在循环内。发射 goto 四元式，跳转到栈顶的出口标签 self.loop\_exit\_labels[-1]。

    def visit\_break\_statement(self, node: BreakStatement):

        if not self.loop\_exit\_labels:

            # 语义分析应已捕获此错误

            print("IR Gen Error: 'break' outside loop (should have been caught by semantic analysis).")

            return None

        # 如果是 'break <expr>;' (loop作为表达式)，这里会更复杂

        self.\_emit("goto", self.loop\_exit\_labels[-1], None, None)

        return None

1. visit\_continue\_statement(self, node: ContinueStatement)

功能说明：处理 continue 语句。

实现过程：检查 self.loop\_continue\_labels 栈是否为空，发射 goto 四元式，跳转到栈顶的继续点标签 self.loop\_continue\_labels[-1]。

    def visit\_continue\_statement(self, node: ContinueStatement):

        if not self.loop\_continue\_labels:

            print("IR Gen Error: 'continue' outside loop (should have been caught by semantic analysis).")

            return None

        self.\_emit("goto", self.loop\_continue\_labels[-1], None, None)

        return None

1. visit\_function\_declaration(self, node: FunctionDeclarationStatement)

功能说明：处理函数声明。

实现过程：创建表示函数名的 IRValue(node.name.value, is\_func\_name=True)，并存入 self.current\_function\_name。获取参数数量和返回类型信息，发射 func\_begin 四元式，标记函数开始，并可携带函数名、参数数量、返回类型等元信息。遍历函数参数 node.parameters，为每个参数发射 receive\_param 四元式，标记这是一个函数接收的参数。如果 node.body 存在，递归调用 node.body.accept(self) 生成函数体的IR。发射 func\_end 四元式，标记函数结束，清空 self.current\_function\_name。

    def visit\_function\_declaration(self, node: FunctionDeclarationStatement):

        func\_name\_ir = IRValue(node.name.value, is\_func\_name=True)

        self.current\_function\_name = func\_name\_ir

        # 获取参数数量和返回类型字符串（用于信息性输出）

        num\_params = len(node.parameters)

        return\_type\_str = "void"

        if node.return\_type and hasattr(node.return\_type, 'name'): # 从 TypeNode 获取

            return\_type\_str = node.return\_type.name

        elif hasattr(node.name, 'eval\_type') and isinstance(node.name.eval\_type, FunctionType): # 从语义分析结果获取

            return\_type\_str = str(node.name.eval\_type.return\_type)

        self.\_emit("func\_begin", func\_name\_ir, IRValue(num\_params, is\_const=True), IRValue(return\_type\_str)) # 传递类型字符串

        # 为参数生成 "receive\_param" 或类似指令（如果目标机器模型需要）

        for param\_node in node.parameters:

            param\_name\_ir = IRValue(param\_node.name.value)

            # param\_type\_str = str(param\_node.eval\_type) # eval\_type 是语义分析时赋予的

            self.\_emit("receive\_param", param\_name\_ir, None, None) # 标记参数

        if node.body:

            node.body.accept(self)

        # 确保非void函数总是有返回。void函数可以在末尾隐式返回。

        # 这一逻辑通常由语义分析保证，或在IR生成后进行检查/转换。

        # 如果函数声明了非void返回类型，但控制流可能不经过 return\_val，则是个问题。

        # 简单的处理是，如果函数末尾没有显式的 return，且函数是void，则可省略。

        # 如果非void，语义分析应该强制所有路径都有return。

        self.\_emit("func\_end", func\_name\_ir, None, None)

        self.current\_function\_name = None

        return None

1. visit\_return\_statement(self, node: ReturnStatement)

功能说明：处理 return 语句。

实现过程：如果 node.return\_value (返回值表达式) 存在：递归调用 node.return\_value.accept(self) 得到返回值表达式的 IRValue。发射 return\_val 四元式：(return\_val, <value\_ir>, \_, \_)。如果不存在返回值表达式，发射 return\_void四元式：(return\_void, \_, \_, \_)。

    def visit\_return\_statement(self, node: ReturnStatement):

        if node.return\_value:

            value\_ir = node.return\_value.accept(self)

            self.\_emit("return\_val", value\_ir, None, None)

        else:

            self.\_emit("return\_void", None, None, None) #  或简化为 "return"

        return None

1. visit\_call\_expression(self, node: CallExpression)

功能说明：处理函数调用表达式。

实现过程：遍历 node.arguments 列表，对每个参数表达式，递归调用 accept(self) 得到其 IRValue，然后发射 arg 四元式，表示将该参数值传递给即将调用的函数。递归调用 node.function.accept(self)，得到表示被调用函数的 IRValue。判断函数调用是否为 void 返回，如果函数有返回值，创建一个新的临时变量 result\_temp，发射 call 四元式：(call, <func\_target\_ir>, <num\_args\_ir>, <result\_temp>)；如果函数为 void 返回，发射 call\_void 四元式：(call\_void, <func\_target\_ir>, <num\_args\_ir>, \_)。

    def visit\_call\_expression(self, node: CallExpression):

        # 1. 计算所有参数表达式，并生成 "param\_push" 或 "arg" 指令

        # 通常参数从右到左压栈，或按顺序传递。这里我们按AST顺序生成 "arg" 指令。

        arg\_irs = []

        for arg\_expr in node.arguments:

            arg\_ir = arg\_expr.accept(self)

            arg\_irs.append(arg\_ir)

        for arg\_ir\_val in arg\_irs: # 按参数列表顺序发射

            self.\_emit("arg", arg\_ir\_val, None, None) # 'arg' 或 'param\_val'

        # 2. 获取函数名

        # node.function 可能是一个 Identifier，也可能是更复杂的表达式（如果支持函数指针/闭包）

        # 简单情况下，它是一个 Identifier

        func\_target\_ir = node.function.accept(self) # 应该返回 IRValue(func\_name, is\_func\_name=True)

        # 3. 生成调用指令

        # 需要知道函数是否返回void，以决定是否需要一个临时变量来接收结果

        # 这个信息应该来自语义分析阶段，存储在 node.function.eval\_type (一个 FunctionType 对象)

        result\_temp = None

        is\_void\_call = True # 默认

        if hasattr(node, 'eval\_type') and node.eval\_type is not None: # eval\_type 是整个调用表达式的类型

            if str(node.eval\_type) != "void" and str(node.eval\_type) != "TYPE\_VOID":

                is\_void\_call = False

        if not is\_void\_call:

            result\_temp = self.\_new\_temp()

            self.\_emit("call", func\_target\_ir, IRValue(len(node.arguments), is\_const=True), result\_temp)

        else:

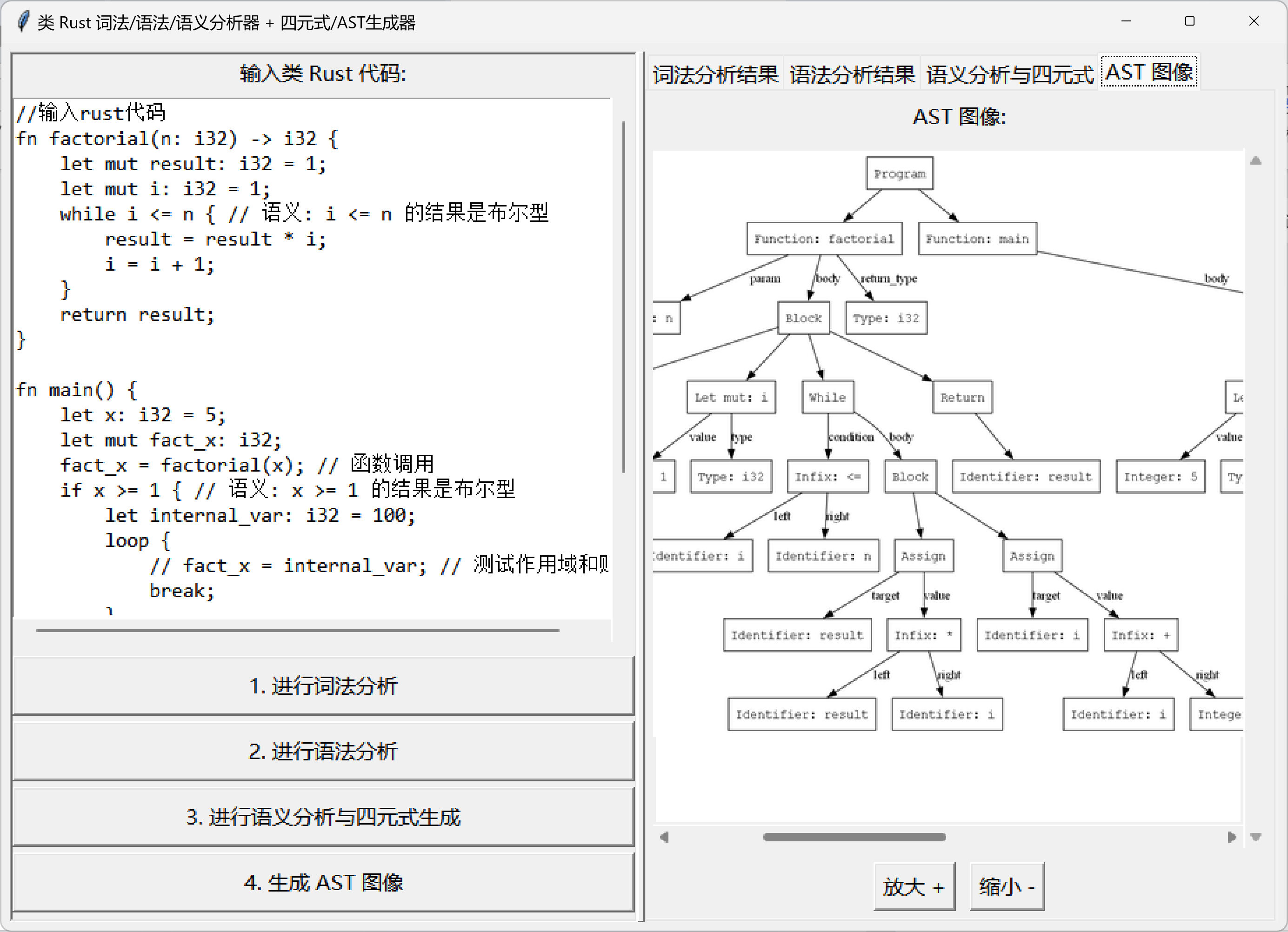
            self.\_emit("call\_void", func\_target\_ir, IRValue(len(node.arguments), is\_const=True), None)

        return result\_temp # 如果非void，返回存储结果的临时变量；否则返回None

## 3、可视化设计

在gui.py中，我们使用tkinter作为GUI的框架，对rust代码提供词法分析、语法分析以语法树的可视化展示。

1. 整体页面设计



界面使用 PanedWindow 分为左右两部分：

1. 左侧：代码输入区与操作按钮。

左侧面板的输入区域为一个可滚动的文本框 (ScrolledText)，用于输入 Rust 代码。同时还有操作按钮：

* "进行词法分析"：触发词法分析，显示 token 列表。
* "进行语法分析"：触发语法分析，显示错误信息和语法结构。
* "进行语义分析与四元式生成"：触发语义分析，显示错误信息并生成带有地址标号的四元式。该步和AST图像都需要先进行语法分析。
* "生成 AST 图像"：生成并显示 AST 的图像。

1. 右侧：结果展示区，包含词法分析、语法分析和 AST 图像三个标签页。

右侧面板为结果显示，包括：

* 词法分析结果：显示 token 列表，包括行号、列号、token 类型和字面值。
* 语法分析结果：
  + 错误信息：如果有语法错误，显示错误列表。
  + 语法结构：如果分析成功，显示 AST 的文本表示。
* 语义分析结果和四元式：
  + 错误信息：如果有语义错误，显示错误列表，并给出报错原因。
  + 四元式：如果语义分析成功，显示地址和四元式。
* AST 图像：显示 AST 的图像，支持缩放功能。

1. 语义分析和四元式生成功能实现函数解析

由于词法分析、语法分析和AST图像的功能实现已在期中报告中详细叙述过了，此处只进行语法分析和四元式生成可视化的分析。

四元式作为重要的中间代码形式，其展示设计力求清晰、准确、易于理解。

1. 专用标签页

在右侧的 ttk.Notebook 中，专门设置了一个名为“语义分析与四元式”的标签页 (semantic\_ir\_tab)，集中展示与此阶段相关的信息。

        semantic\_ir\_tab = tk.Frame(right\_notebook) # Renamed tab for clarity

        right\_notebook.add(semantic\_ir\_tab, text='语义分析与四元式')

1. 语义错误展示

标签页的上半部分，首先是一个文本标签“语义分析错误:”，其下方是一个可滚动的只读文本区域 (self.semantic\_error\_text)。

semantic\_error\_label = tk.Label(semantic\_ir\_tab, text="语义分析错误:", font=default\_font)

semantic\_error\_label.pack(pady=(5, 5), fill=tk.X, padx=5)

self.semantic\_error\_text = self.create\_scrollable\_text(semantic\_ir\_tab, height=7, font=text\_font)

* 内容：此区域用于逐条列出由 SemanticAnalyzer 检测到的所有语义错误。每个错误通常会包含错误信息、以及错误发生的源码位置（行号、列号，如果 SymbolTable.add\_error 支持）。

if semantic\_errors:

         semantic\_errors\_found = True

         self.semantic\_error\_text.insert(tk.END, "语义分析检测到以下错误:\n" + "---------------------------\n")

         for error in semantic\_errors:

             self.semantic\_error\_text.insert(tk.END, f"- {error}\n")

         self.semantic\_error\_text.insert(tk.END, "---------------------------\n")

         self.ir\_table.insert('', 'end', values=('', '由于语义错误，未生成中间代码。\n'))

     else:

         self.semantic\_error\_text.insert(tk.END, "语义分析成功！未检测到语义错误。\n")

* 清空机制：每次点击“进行语义分析与四元式生成”按钮时，此区域会先被清空，然后填充新的错误信息（或成功信息）。

1. 四元式表格展示

标签页的下半部分，是四元式展示的核心区域。一个文本标签“中间代码 (四元式):”位于该区域的顶部。

* 核心组件：使用 ttk.Treeview 组件 (self.ir\_table) 以表格形式展示生成的四元式序列。Treeview 提供了比纯文本更好的结构化展示能力。
* 列定义：
  + 地址 (addr)：第一列，显示四元式在序列中的索引或逻辑地址。通常从0或1开始，使用 f"{i:03d}" 格式化为固定三位数的数字，便于对齐和阅读。
  + 四元式 (quad)：第二列，显示四元式本身。其内容是 Quadruple对象的字符串表示（通过调用其 \_\_str\_\_ 方法得到），格式通常为 (op, arg1, arg2, result)。
* 表头：明确显示“地址”和“四元式”作为列标题。
* 对齐与宽度：“地址”列居中对齐，宽度较小；“四元式”列左对齐，占据主要宽度。
* 滚动条：配备了垂直和水平滚动条，以应对大量四元式或较长的四元式字符串。
* 清空机制：与语义错误区类似，每次生成新的四元式前，表格内容会被清空 (self.ir\_table.delete(row) in loop)。

  if not semantic\_errors\_found:

                try:

                    ir\_gen = IRGenerator()

                    ir\_gen.visit\_program(self.ast\_root\_for\_analysis) # 使用同一个AST

                    quadruples = ir\_gen.get\_quadruples()

                    if quadruples:

                        for i, quad in enumerate(quadruples):

                            self.ir\_table.insert('', 'end', values=(f"{i:03d}", str(quad)))

                    else:

                        self.ir\_table.insert('', 'end', values=('', '未生成中间代码。(可能是空程序或IR生成器未对所有情况处理)。\n'))

                except Exception as e\_ir:

                    self.ir\_table.insert('', 'end', values=('', f"生成中间代码时发生错误: {e\_ir}"))

                    import traceback

                    self.ir\_table.insert('', 'end', values=('', traceback.format\_exc()))

1. 数据来源与填充

四元式的生成和展示逻辑位于 perform\_semantic\_analysis\_and\_ir 方法中：

* 前提条件：检查 self.ast\_root\_for\_analysis 是否存在。如果不存在（即语法分析未成功），则提示用户先进行语法分析，并且不进行四元式生成。
* 语义分析：首先执行 SemanticAnalyzer 对AST进行分析。
* 条件生成：仅当语义分析未报告任何错误 (semantic\_errors\_found is False) 时，才继续进行四元式生成。
  + 实例化 IRGenerator。
  + 调用 ir\_gen.visit\_program(self.ast\_root\_for\_analysis) 来遍历AST并生成四元式。
  + 通过 ir\_gen.get\_quadruples() 获取包含 Quadruple 对象的列表。
* 表格填充：
  + 遍历获取到的四元式列表。
  + 对于每个 Quadruple 对象，使用其索引 i（格式化为三位数字）作为“地址”，使用 str(quad)（即调用 Quadruple.\_\_str\_\_()）作为“四元式”内容。
  + 调用 self.ir\_table.insert('', 'end', values=(...)) 将数据插入表格的新行。
* 空/错误状态处理：
  + 如果语义分析失败，表格会显示提示信息“由于语义错误，未生成中间代码。”。
  + 如果IR生成阶段出现意外，也会显示相应的错误信息。
  + 如果成功但未生成任何四元式（例如空程序），则显示“未生成中间代码。”。

# 四、小组分工

本次报告与PPT由小组成员共同完成，类Rust中间代码生成器设计实现分工如下：

[Truman-min-show](https://github.com/Truman-min-show)：主要负责语义分析设计编写，实现基础语义分析功能。

[limeszstone](https://github.com/limeszstone)：主要负责可视化设计，展示分析结果。

[TheTry16](https://github.com/TheTry16)：主要负责中间代码生成设计编写，实现拓展语义分析功能。

# 五、总结与展望

## 1、总结

本次大作业在期中项目的基础上，成功地将类Rust编译器项目推进到语义分析和中间代码生成阶段，进一步完善了编译器的前端功能。我们不仅巩固了词法和语法分析的理论与实践经验，更深入地探索了程序语义的合法性验证及其向低层表示的转换。

在语义分析方面，我们构建了一个基于访问者模式的SemanticAnalyzer，并引入了核心的SymbolTable数据结构，主要完成包括：

* **精确的符号管理与作用域解析：**通过构建带作用域栈的符号表，实现了对变量和函数等标识符的有效声明、重影及跨作用域查找，确保了所有引用的正确性和上下文一致性。
* **严格的类型系统与初始化检查：**全面实施了类型检查，覆盖了表达式运算、函数调用参数与返回类型、赋值操作以及控制流条件，确保了操作类型匹配并能推断结果类型。同时，引入了变量初始化状态跟踪，杜绝了使用未初始化变量的风险。
* **关键语义规则的全面验证：**严格执行了包括不可变变量赋值限制、循环控制语句合法性在内的多项核心语义规则，显著提升了代码的健壮性和逻辑正确性。

在此基础上，我们迈入了编译器的第二个主要阶段——中间代码生成。我们采用了四元式作为中间表示形式：

* **四元式设计：**定义了Quadruple类，包含操作符 (op)、两个操作数 (arg1,arg2) 和结果 (result) 字段，能够清晰地表达程序的基本操作。IRValue类则用于统一表示四元式中的操作数，包括变量、临时变量、常量、标签和函数名，便于后续处理。
* **生成器实现：**IRGenerator类作为Visitor，在语义分析后的 AST 上进行遍历。它为大部分 AST 节点实现了visit方法，将高级语言的结构逐步转换为一系列线性的四元式指令。

本次大作业不仅深化了我们对语义分析与中间代码生成原理的理解，更通过实践锻炼了我们处理复杂语言特性、管理数据结构和实现模块化编译组件的能力，为构建更完善的编译器奠定了坚实的基础。

## 2、展望

尽管本次作业成功实现了语义分析和四元式中间代码生成，并覆盖了大部分核心规则和指定拓展功能，但距离构建一个成熟类Rust编译器仍有进一步拓展和优化的空间。在未来的工作中，我们将着眼于以下几个方向，同时也将本次在语义分析和四元式生成阶段考虑到的高级语言特性和解决方案融入其中。

语义检查的深度与鲁棒性提升：

* **类型推导与统一：**当前的类型推导主要基于直接的初始化和赋值。未来可以引入更复杂的类型推导算法以支持更灵活的类型系统。
* **错误恢复策略：**尽管语义分析可以传播TYPE\_ERROR，但错误报告的友好性仍有提升空间。未来可以引入更精细的错误诊断，例如指出类型不匹配的具体原因，或者在发现第一个错误后尝试跳过部分代码继续分析，以收集更多的错误信息一次性报告给用户。

中间代码生成器的优化与拓展：

* **更丰富的IR指令：**随着语言特性的增加，四元式指令集也需扩展，例如支持数组和元组的访问、引用操作等。
* **临时变量管理：**改进临时变量的分配和重用策略，减少生成的四元式数量。
* **控制流图（CFG）的构建：**四元式序列是线性的，但程序执行是分支和循环的。构建CFG可以为后续的优化提供结构化的视图，可以增加数据结构来记录标签和跳转指令之间的关系，或者在四元式生成后进行后处理，构建基本的块和边。
* **函数调用约定：**明确函数参数的传递方式和返回值处理，使call指令更加具体化，可以在visit\_call\_expression和visit\_function\_declaration中，添加对参数和返回值的特定处理指令。

通过这些展望，我们期待将这个类Rust编译器项目打造成为一个功能更完善、性能更优越、诊断能力更强的编译系统，为深入理解编译器技术及未来更复杂的系统软件开发奠定坚实基础。

# 六、参考文献

1. 知乎专栏. "编译原理中是如何进行「词法分析」的" [EB/OL]. 知乎技术专栏, 2021. https://zhuanlan.zhihu.com/p/363589423
2. CSDN 博客. "编译原理：词法分析器设计与实现(C语言)" [EB/OL]. CSDN 技术博客, 2025. https://wenku.csdn.net/column/4amenkbjmk
3. CSDN 博客. "源代码解析艺术：词法分析器设计原理详解" [EB/OL]. CSDN 技术博客, 2020. https://blog.csdn.net/hpu2022/article/details/106530171
4. PingCode 文章. "如何用python写词法分析器" [EB/OL]. PingCode 百科文章, 2025. https://docs.pingcode.com/ask/1091084.html
5. 知乎专栏. "编译原理(6) 自顶向下的分析方法" [EB/OL]. 知乎技术专栏, 2023. https://zhuanlan.zhihu.com/p/654321098
6. 稀土掘金. "计算机编程语言原理与源码实例讲解：递归下降解析器的构建" [EB/OL]. 稀土掘金技术博客, 2023. https://juejin.cn/post/7245678901234567890
7. CSDN 博客. "手写一个Pratt Parser基本运算解析器" [EB/OL]. CSDN 技术博客, 2023. https://blog.csdn.net/qq\_45678901/article/details/136589823
8. 知乎专栏. "编程实践｜如何使用MoonBit编写Pratt解析器？" [EB/OL]. 知乎技术专栏, 2024. https://zhuanlan.zhihu.com/p/789012345
9. 咨信网．"2022年词法分析程序语法语义分析四元式生成实验报告" [EB/OL]．学术文档，2025．[https://www.zixin.com.cn/doc/9805382.html](https://www.zixin.com.cn/doc/9805382.html" \t "https://chat.deepseek.com/a/chat/s/_blank)
10. CSDN 博客．"词法分析程序+语法语义分析四元式生成+实验报告" [EB/OL]．CSDN 下载资源，2021．[https://download.csdn.net/download/wsbhm62/27688417](https://download.csdn.net/download/wsbhm62/27688417" \t "https://chat.deepseek.com/a/chat/s/_blank)
11. pandolia. 自己动手写编译器 [EB/OL]. (2019-01-01) [2025-05-08]. <https://pandolia.net/tinyc/index.html>
12. 掘金社区．"语义分析器四元式额生成过程" [EB/OL]．掘金，2021．  
    [https://juejin.cn/s/语义分析器四元式额生成过程](https://juejin.cn/s/%E8%AF%AD%E4%B9%89%E5%88%86%E6%9E%90%E5%99%A8%E5%9B%9B%E5%85%83%E5%BC%8F%E9%A2%9D%E7%94%9F%E6%88%90%E8%BF%87%E7%A8%8B" \t "https://chat.deepseek.com/a/chat/s/_blank)
13. Steve Klabnik, Carol Nichols, Rust Community. The Rust Programming Language [EB/OL]. (2023-05-20)[2025-05-08]. https://doc.rust-lang.org/book/
14. Overview of the compiler - Rust Compiler Development Guide [EB/OL]. (2023-05-20) [2025-05-08]. https://rustc-dev-guide.rust-lang.org/overview.html
15. Logos Handbook. Logos Handbook [EB/OL]. [2025-05-08]. https://logos.maciej.codes/