词法与语法分析工具设计与实现

小组分工

彭怀龙：语法编辑器代码编写，状态转换图绘制

范宇恒：语法编辑器代码编写，报告撰写

杨 淞：词法分析器代码与报告，PPT 制作

1. 背景介绍

随着现代编程语言的快速发展，编译器前端技术作为程序语言实现的核心环节，其设计与实现质量直接影响着整个编译系统的性能和可用性。本报告基于Python 3.12.6实现了一个Rust风格语言的词法和语法分析工具，旨在深入理解编译器前端的工作原理，并探索如何高效地构建语言处理工具链。

本次课程项目的开发背景源于编译原理课程实践需求，参考了Rust语言的语法特性，包括变量声明、控制结构、函数定义等核心语法元素。Rust作为一门现代系统编程语言，其严谨的语法设计和丰富的语言特性为编译技术研究提供了良好的素材。我们选择Python 3.12.6作为实现语言，主要考虑到其强大的字符串处理能力和清晰的面向对象特性，能够有效支持词法分析和语法分析模块的开发。

在技术层面，本项目采用了经典的递归下降分析法来实现语法分析，这种方法直观易懂，特别适合教学场景下的编译器实现。词法分析器采用确定有限自动机的概念手工构建，能够准确识别各类词法单元，包括关键字、标识符、字面量、运算符等。整个系统严格遵循编译器的分层设计原则，词法分析和语法分析模块解耦良好，便于后续的功能扩展和维护。

本报告将详细阐述该词法和语法分析工具的设计思路、实现细节以及测试结果。

1. 词法分析器

词法分析是编译过程的第一个阶段，它的主要任务是将输入的源程序字符流转换为词法单元（Token）序列。

2.1 原理分析

词法分析器的核心原理是基于有限状态自动机（FSA），通过读取输入字符流，根据预定义的规则将连续的字符组合成词法单元。具体来说，词法分析器会逐个读取输入字符，根据当前字符和后续字符的组合，判断其属于哪种词法单元类型，如关键字、标识符、数值等。

2.2 关键数据结构

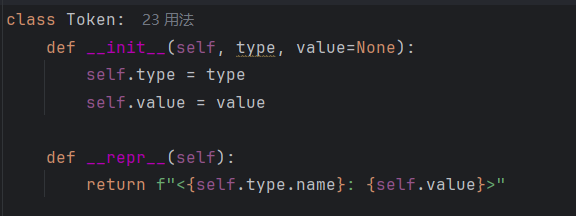
定义Token类，定义type （词法单元类型）与value（词法单元值）。

定义TokenType类，定义基础token类型

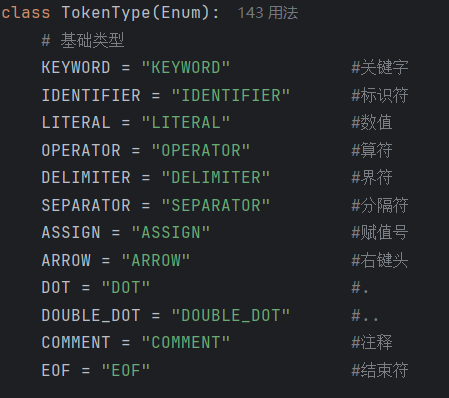
定义Lexter类，包含词法分析器核心模块，负责读取输入字符流并将其转换为词法单元序列。

2.3 详细设计

Token类：表示一个词法单元，包含词法单元的类型和值。\_\_repr\_\_方法用于方便调试和输出。



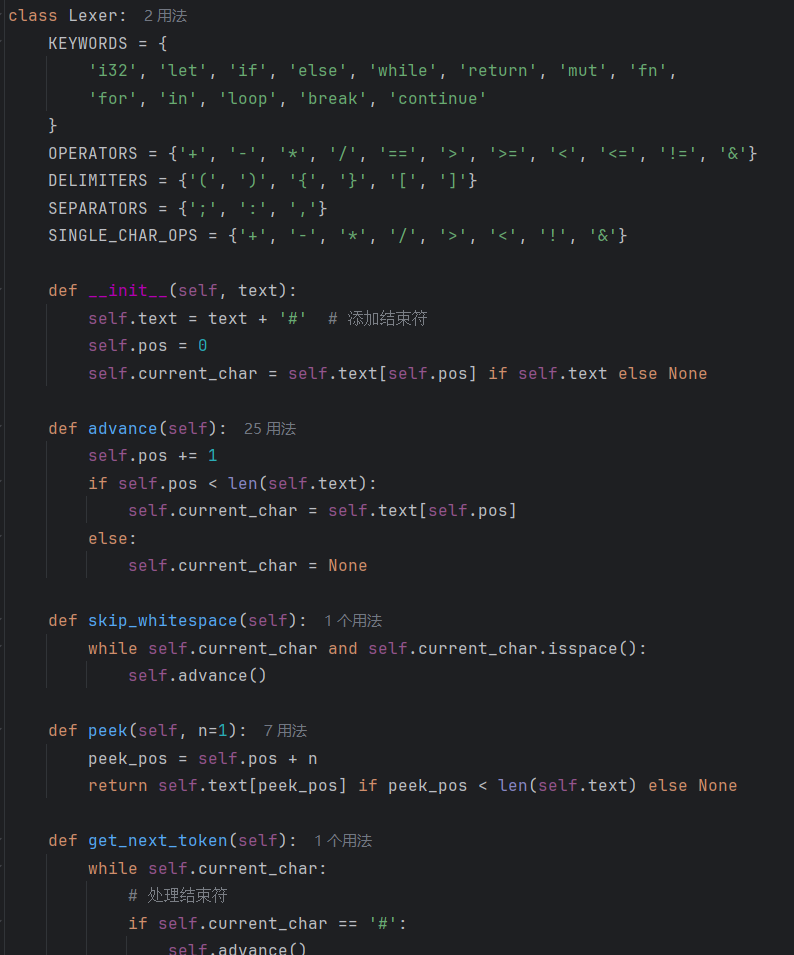
TokenType类：该枚举类定义了所有可能的词法单元类型，方便后续代码的使用和维护。



Lexer 类是词法分析器的核心，包含以下主要方法：

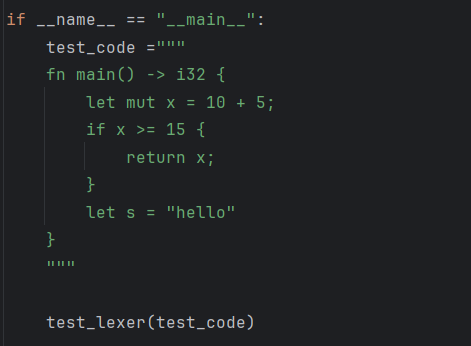
1. \_\_init\_\_：初始化输入文本和当前字符位置。
2. advance：将当前字符位置向后移动一位。
3. skip\_whitespace：跳过输入文本中的空白字符。
4. peek：查看下一个字符。
5. get\_next\_token：获取下一个词法单元。
6. read\_identifier：读取标识符或关键字。
7. read\_number：读取数值字面量。
8. read\_operator：读取算符。
9. read\_separator：读取分隔符。
10. read\_delimiter：读取界符。
11. read\_arrow：读取右箭头。
12. read\_dot：读取点号。
13. read\_double\_dot：读取双点号。
14. read\_line\_comment：读取行注释。
15. read\_block\_comment：读取块注释。

部分代码如下图所示：

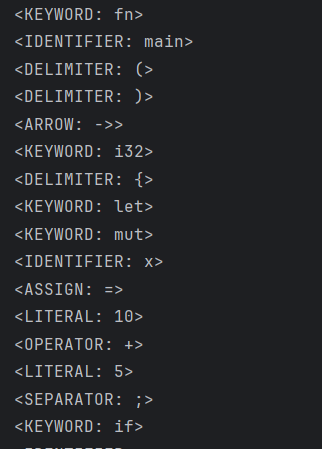


2.4 测试用例

编写测试程序，使用多段代码测试，均验证通过。



测试代码



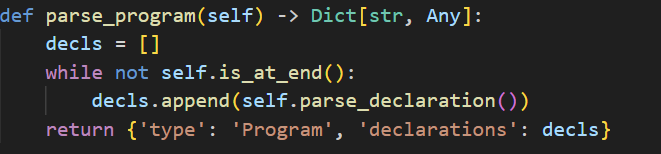
部分运行结果

1. 语法分析器

3.1 基础程序、语句、返回语句（1.1,1.2,1.3）

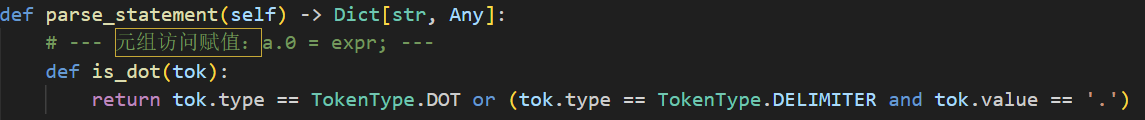
3.1.1 基础程序

parse() 方法作为解析过程的统一入口，程序由多个声明组成，先初始化空声明列表，使用 while 循环持续解析直到 token 流结束，对每个声明调用 parse\_declaration 最后生成包含类型标识和声明列表的字典结构，返回 AST 节点



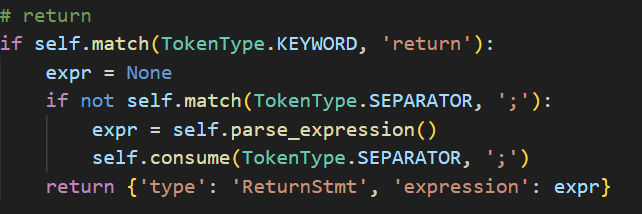
3.1.2 语句

检查是否为 return 关键字，检查是否为分号（空语句），都不匹配则抛出语法错误

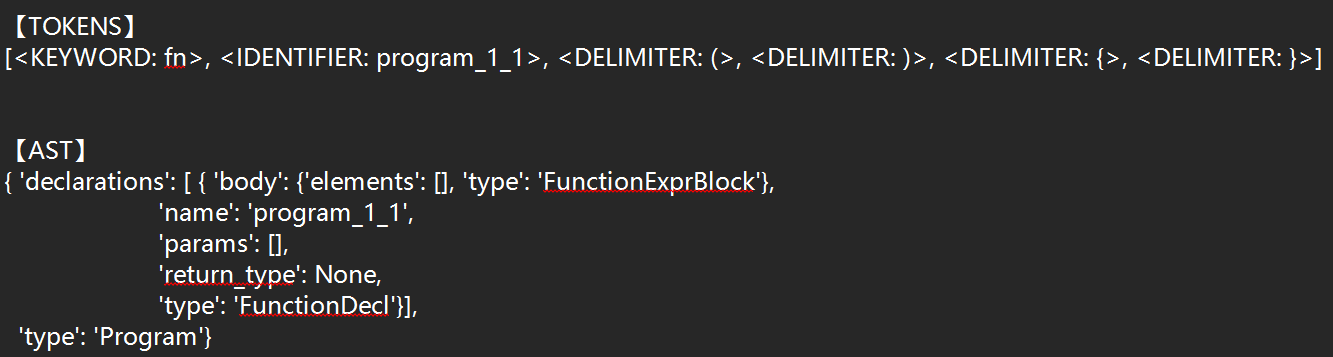


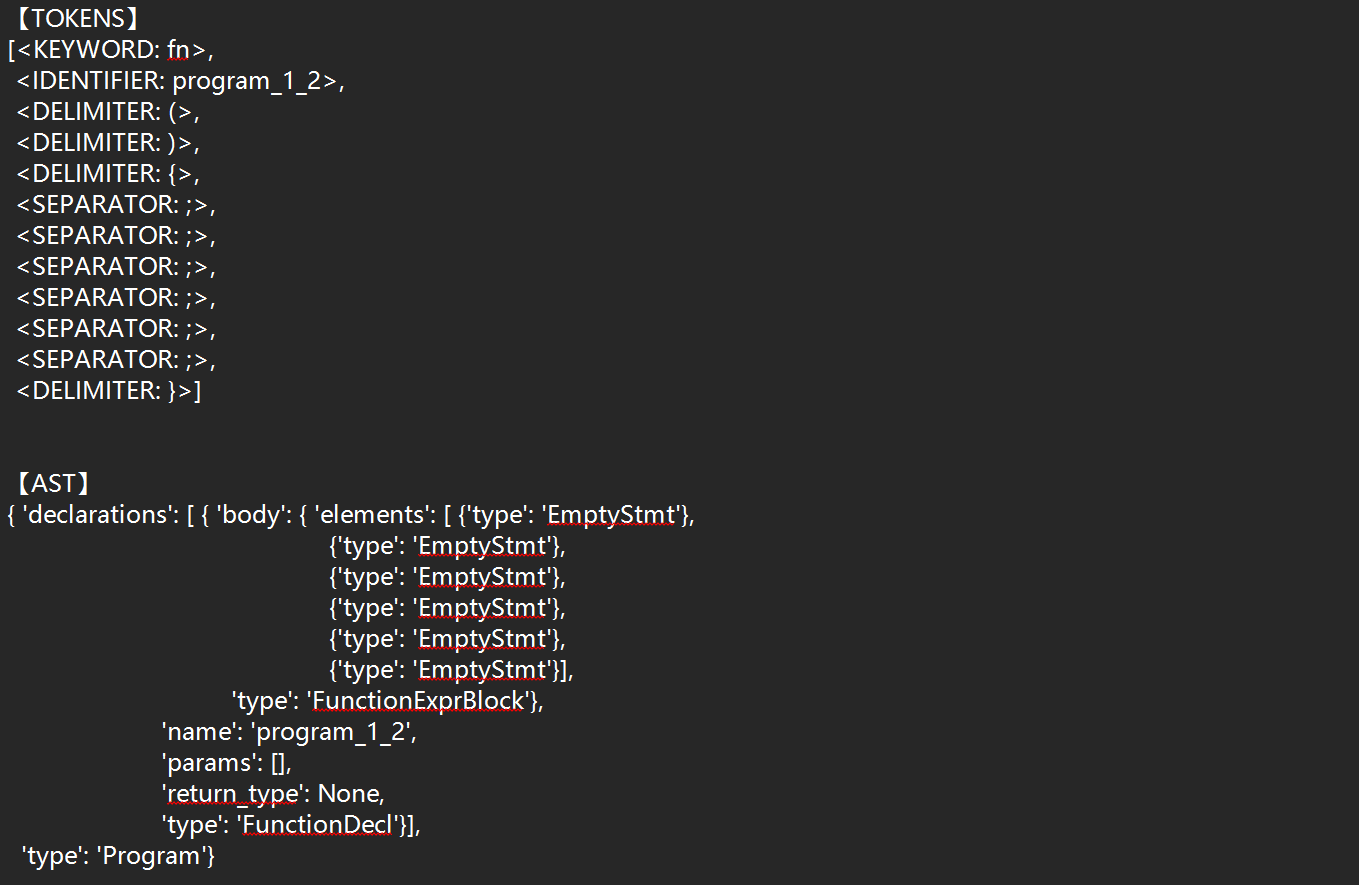
3.1.3 返回语句

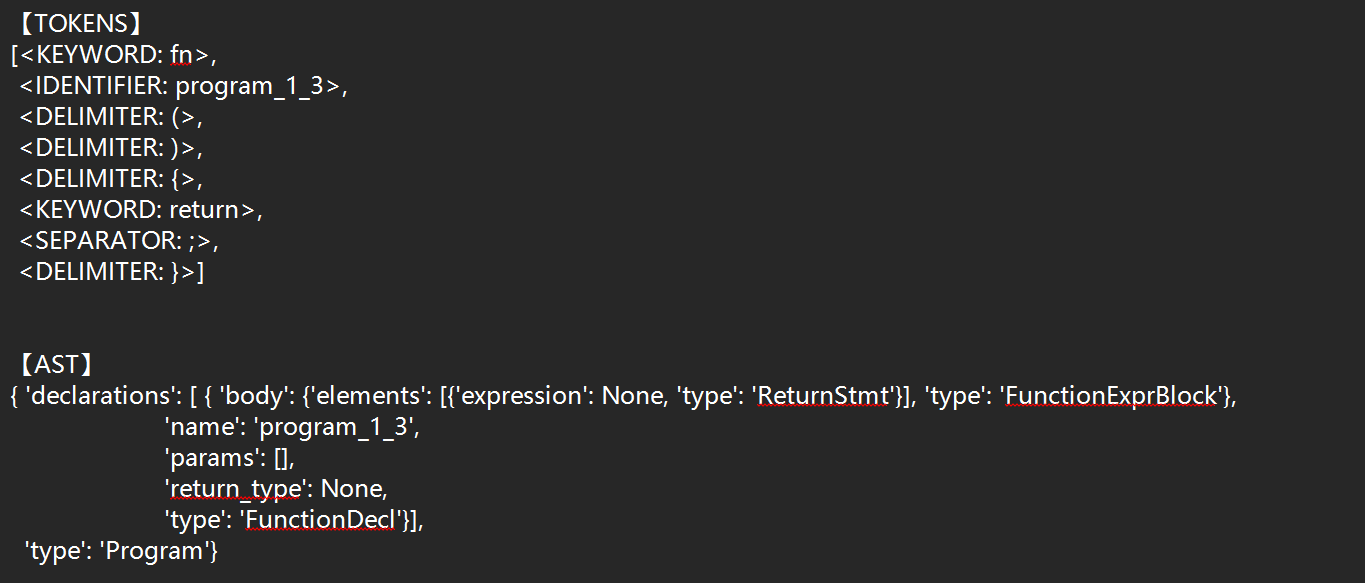
使用 peek() 前瞻检查表达式存在性，将字面量统一标识为 Literal 类型，同时必须显式识别到并消除分号保证语法正确性



3.1.4 运行结果







3.2 函数输入（1.4）

3.2.1 核心方法组成

parse\_declaration(): 函数声明入口

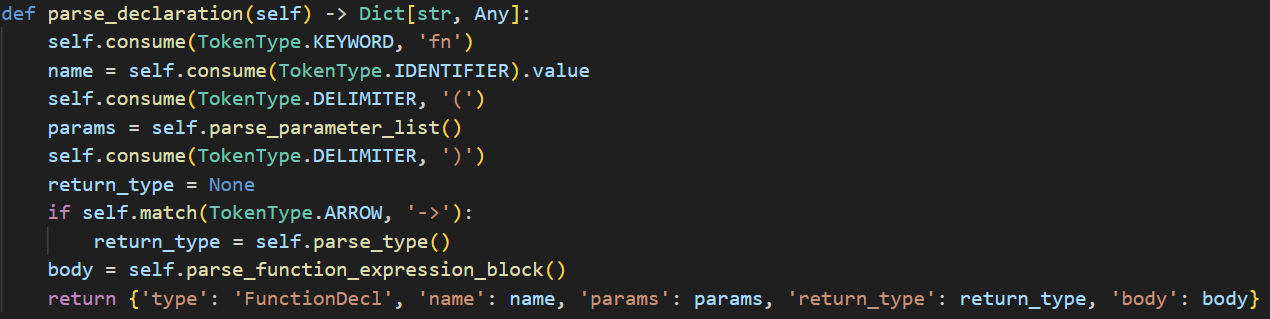
parse\_parameter\_list(): 参数列表解析

parse\_parameter(): 单个参数解析

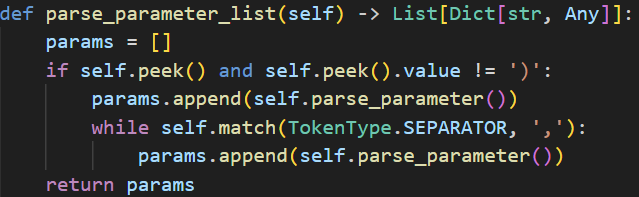
parse\_type(): 类型注解解析

3.2.2 关键方法实现

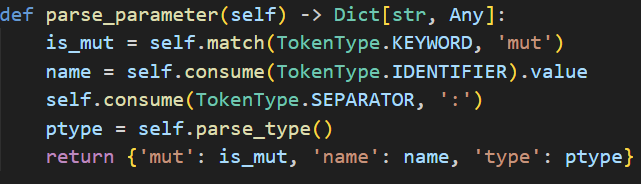
parse\_declaration() 是函数入口，先检查 fn 关键字，获取函数名标识符，再验证左括号，调用 parse\_parameter\_list()，然后验证右括号，处理可选返回类型，解析函数体并返回函数声明 AST 节点。



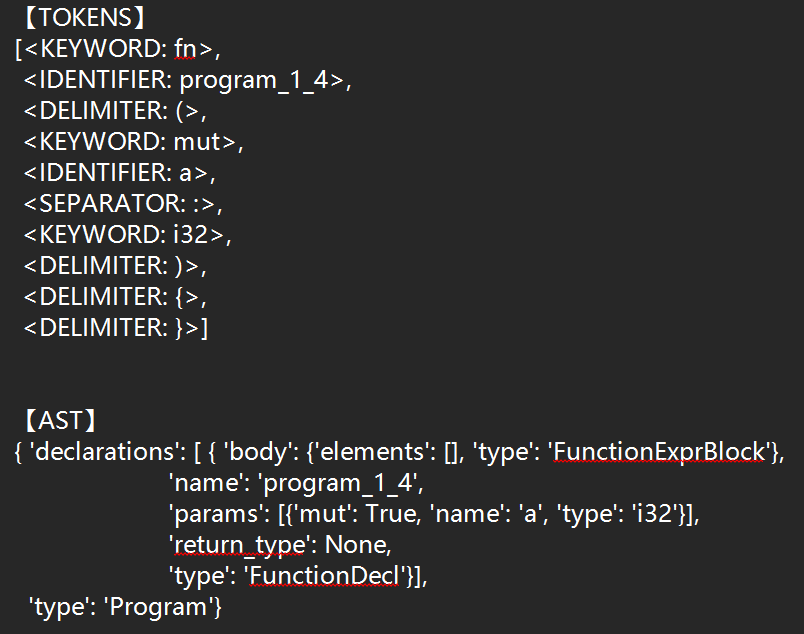
parse\_parameter\_list() 先检查非空参数，解析第一个参数，调用 parse\_parameter(),再解析后续参数，返回函数列表。



在解析单个参数，即 parse\_parameter() 函数中，先检查 mut 关键字，获取参数名，消耗类型冒号，再解析类型，返回参数结构。



3.2.3 运行结果



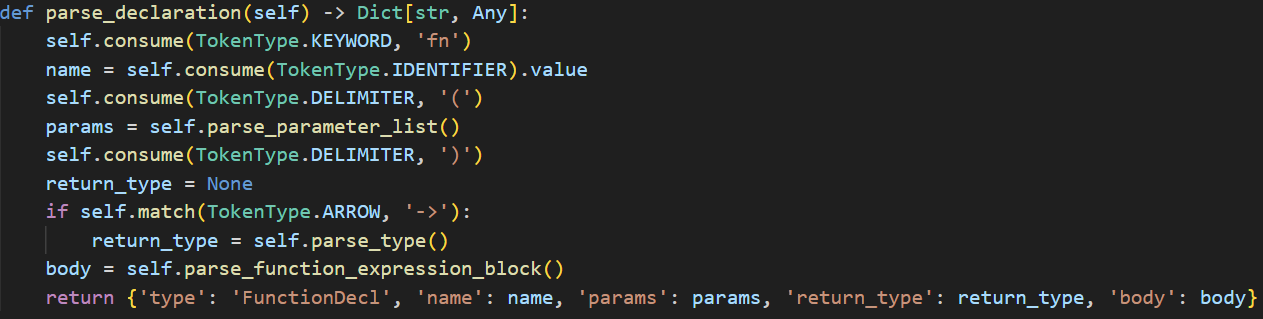
3.3 函数输出（1.5）

3.3.1 模块功能

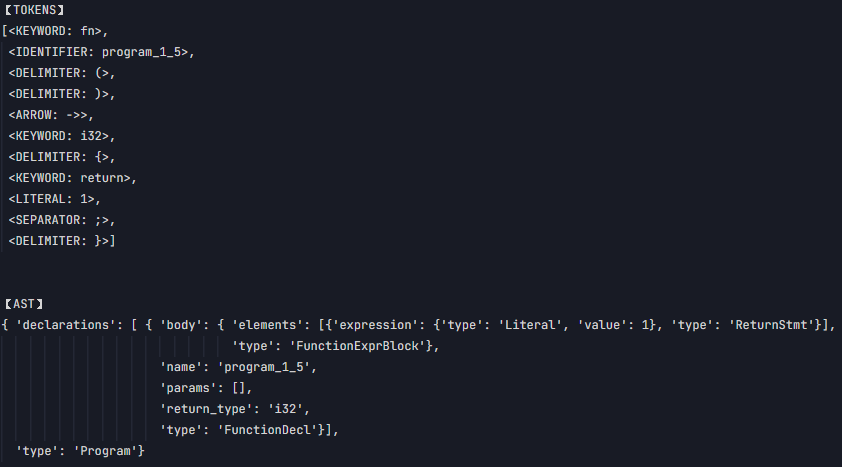
函数输出模块负责解析类 Rust 语法中的函数声明结构，包括函数名的识别，形参列表解析，返回值类型解析和函数体解析

3.3.2 实现方法

参数列表解析支持 mut 修饰符和类型注解，自动处理引用、数组、元祖等符合类型，具有遇到缺失参数类型时抛出带行列号的详细错误的错误恢复机制，支持同时包含语句块和表达式块的混合函数体



3.3.3 运行结果



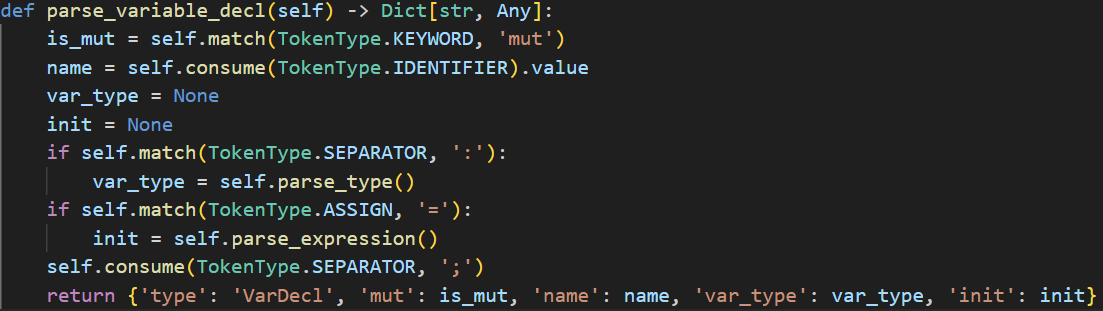
3.4 变量声明语句（2.1）

3.4.1 模块功能

处理以 let 关键字为起始的变量声明，支持可变和不可变变量，支持类型注解和初始化表达式

3.4.2 实现方法

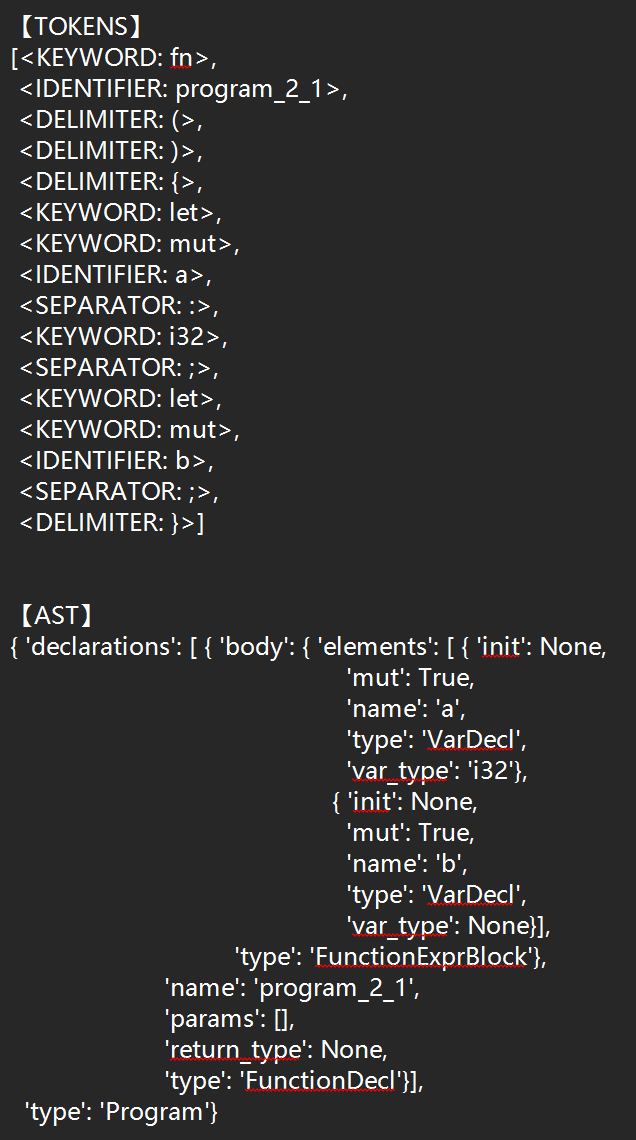
parse\_variable\_decl() 函数检查 mut 关键字，初始化表达式，对冒号（：）、等号（=）、分号（；）进行识别匹配，并处理类型注解



3.4.3 语法规则

**VariableDecl → 'let' ('mut')? IDENTIFIER (':' Type)? ('=' Expression)? ';'**

3.4.4 运行结果



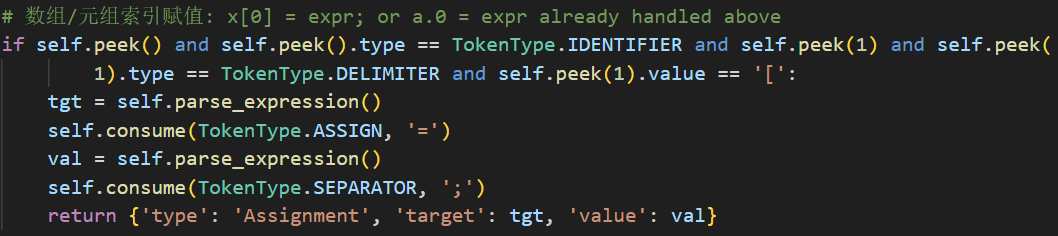
3.5 赋值语句（2.2）

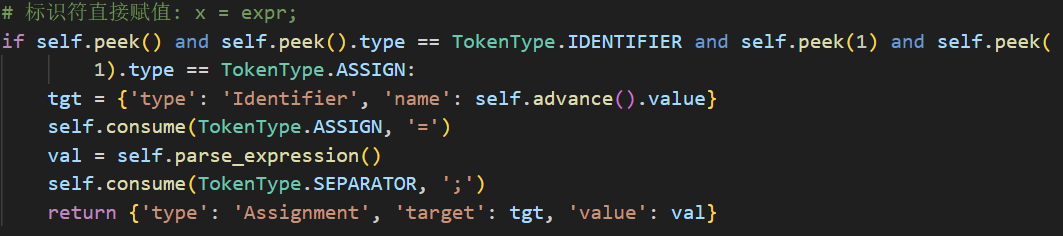
3.5.1 模块功能

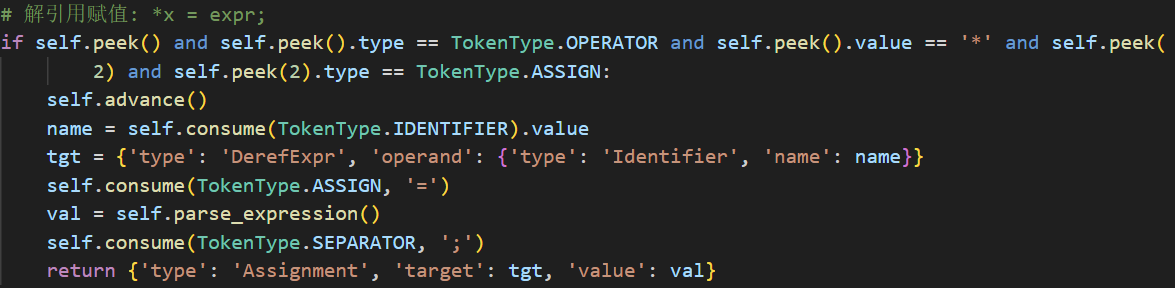
处理多种形式的赋值操作，包括普通变量、数组索引、元祖字段等，支持复合赋值运算符

3.5.2 实现方法

首先根据类型注解识别类型，并分别进行普通变量赋值操作、数组索引复制操作、元祖字段赋值操作





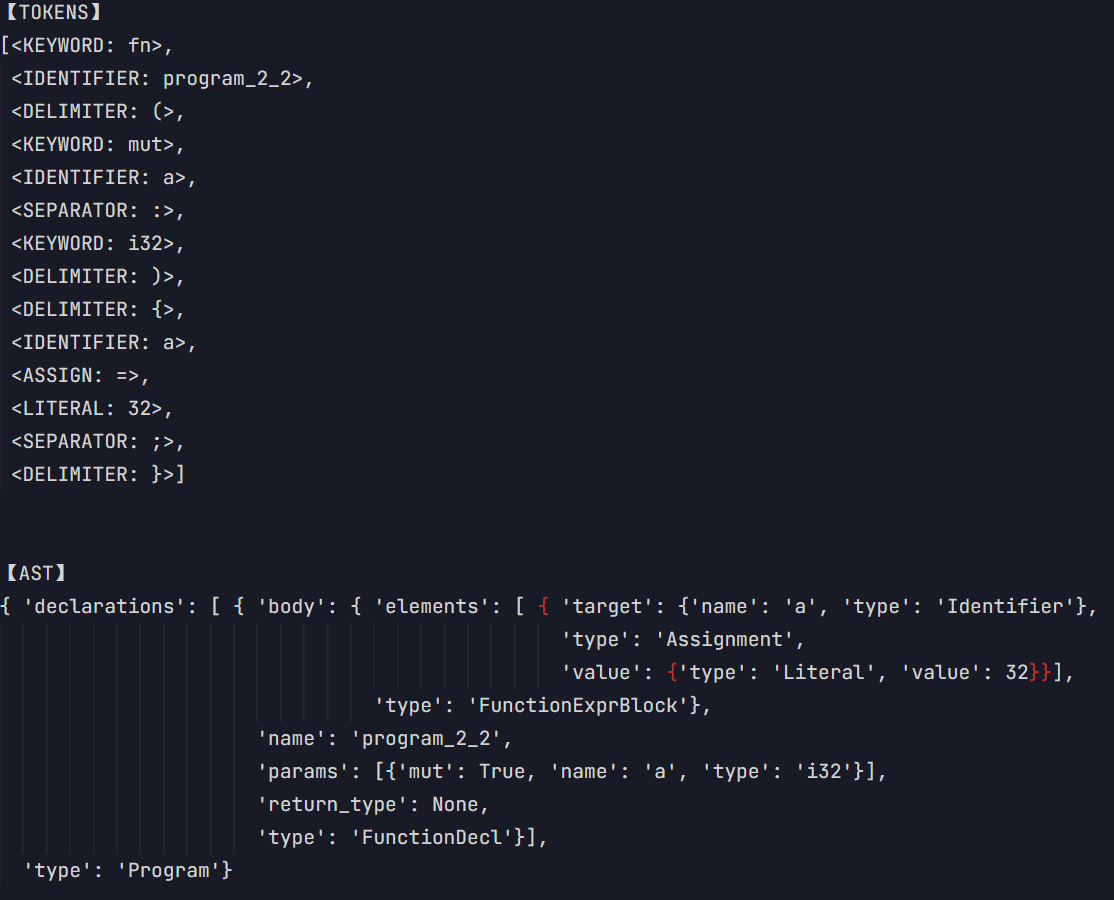


3.5.3 语法规则

**Assignment → Target '=' Expression ';'**

**Target → Identifier | ArrayIndex | TupleField**

3.5.4 运行结果



3.6 基本表达式（3.1）

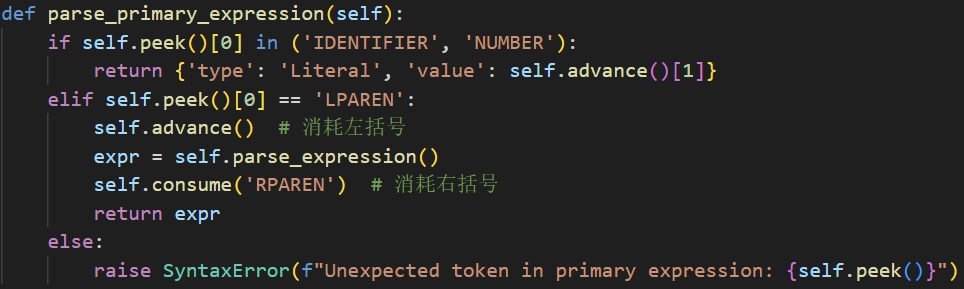
3.6.1 模块功能

·处理原子表达式元素，包括字面量、标识符、括号表达式

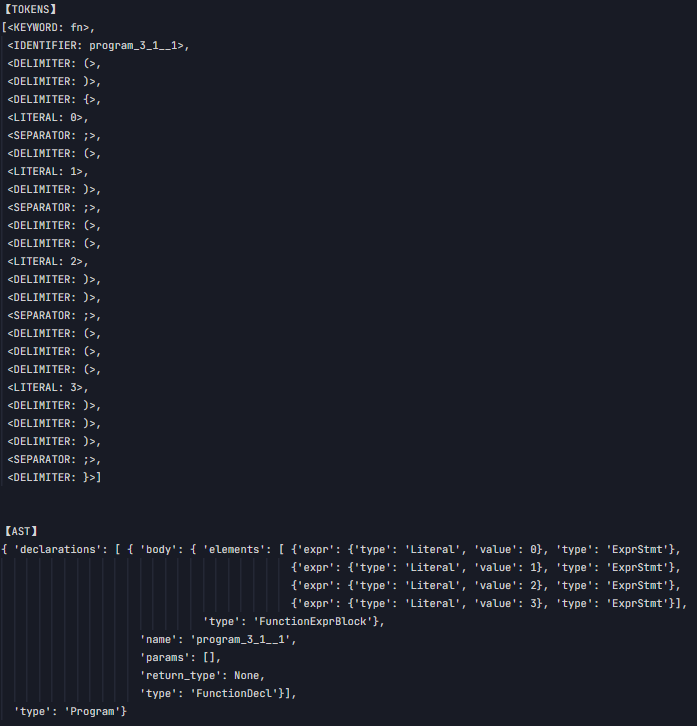
·为复杂表达式提供基础组件

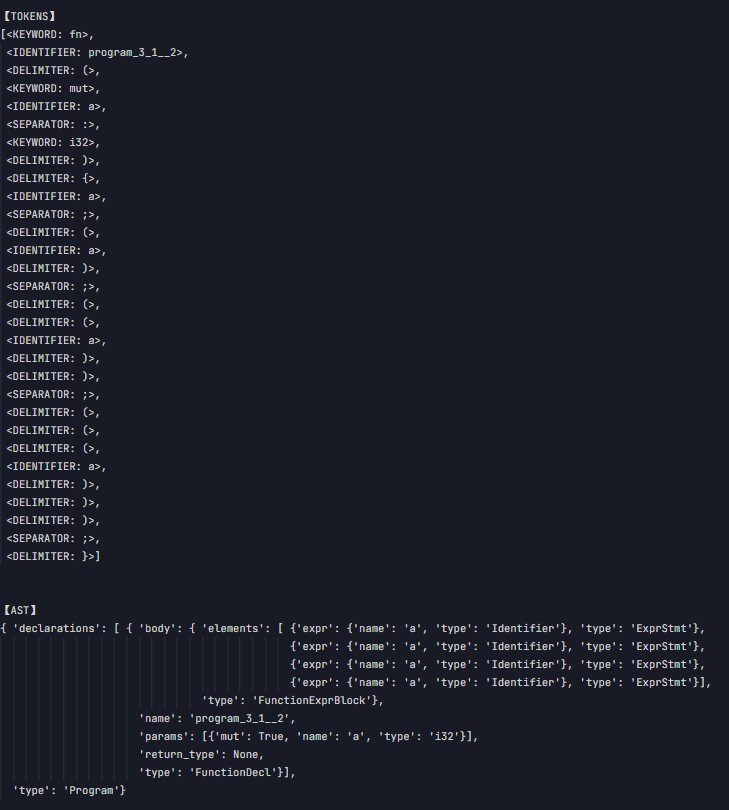
3.6.2 实现方法

在 parse\_primary\_expression() 函数中，首先根据类型注解识别基本表达式类型是标识符、字面量或者括号表达式，若是括号表达式则先消耗左括号，再将括号内的表达式作为新表达式，并消耗右括号。最后统一返回 AST 节点



3.6.3 运行结果





3.7 表达式的计算与比较（3.2）

3.7.1 分层解析结构

**parse\_expression() → parse\_comparison() → parse\_additive() → parse\_term() → parse\_factor()**

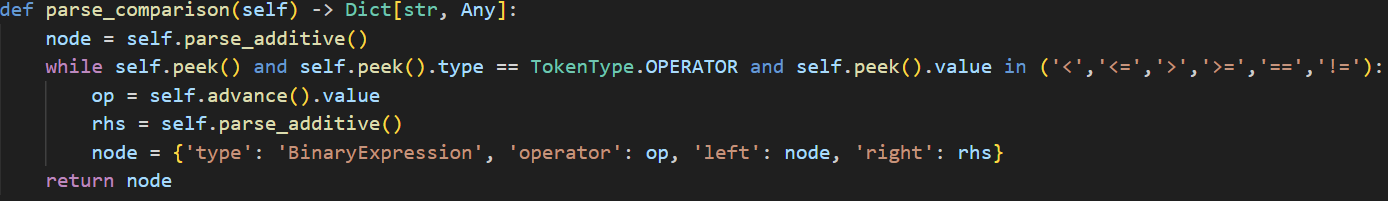
3.7.2 运算符优先级处理

|  |  |  |
| --- | --- | --- |
| 层级 | 运算符 | 处理方法 |
| 比较 | < <= > >= == != | parse\_comparison() |
| 加减 | + - | parse\_additive() |
| 乘除 | \* / | parse\_term() |
| 原子 | 基础元素 | parse\_factor() |

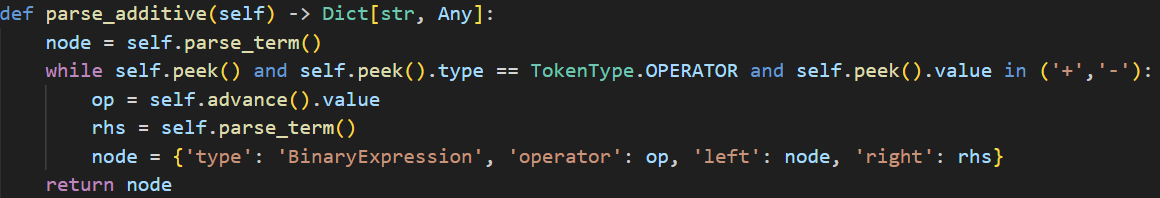
表3.1 运算符优先级表格

3.7.3 实现方法

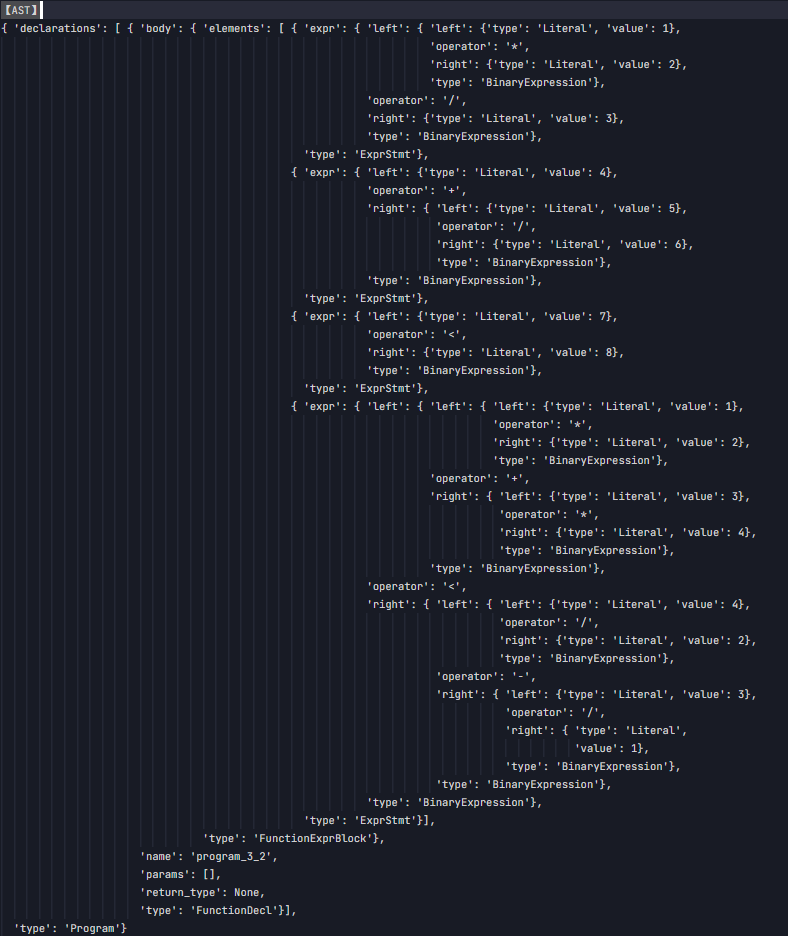
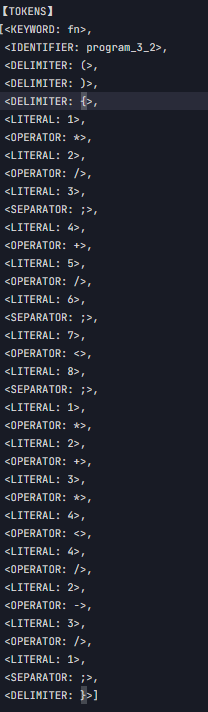
·表达式的比较



·表达式的计算



3.7.4 运行结果



3.8 函数调用（3.3）

3.8.1 模块功能

·处理函数调用表达式

·解析参数列表

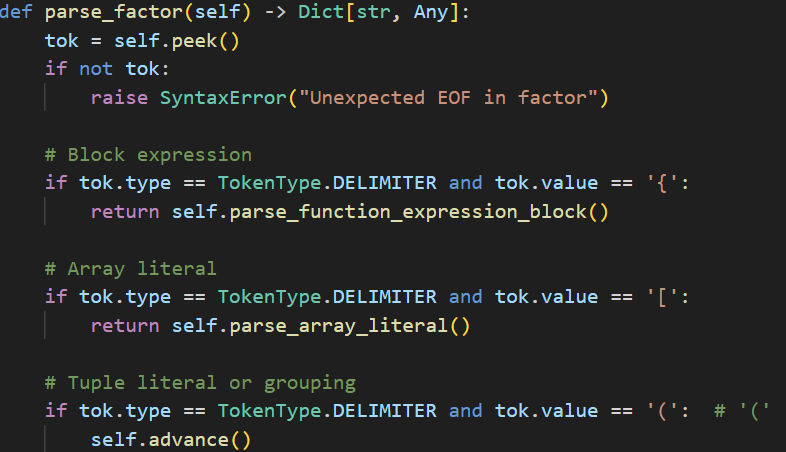
3.8.2 实现方法

**parse\_factor(检测到函数调用模式)**

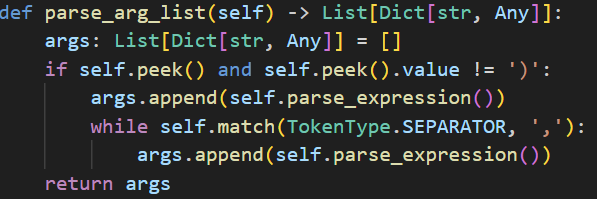
**→ parse\_arg\_list**

**→ parse\_expression (多次)**

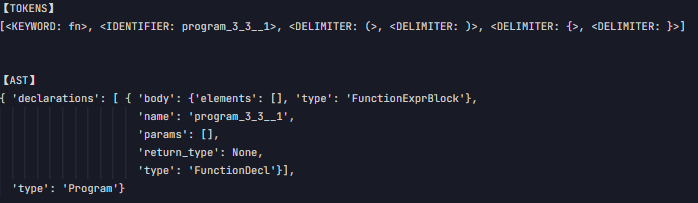
在 parse\_factor() 函数中，检查当前token是标识符且下一个token是(，前进到标识符token，获取函数名称，消耗左括号，解析参数列表，再消耗右括号，返回 AST 节点

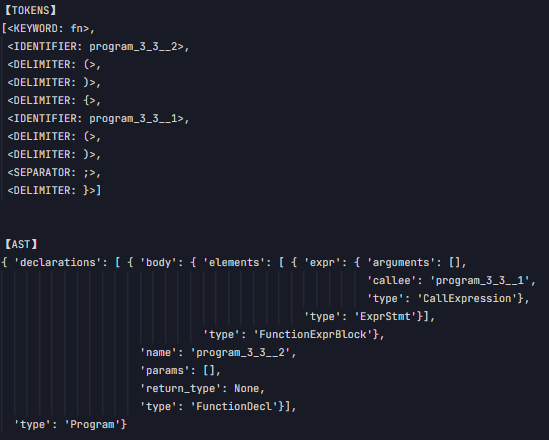


在 parse\_arg\_list() 函数中，首先初始化空参数列表，检查下一个 token 不是“)”，即非空，则调用 parse\_expression 解析第一个参数，消耗逗号（,），并解析下一个参数，依次循环直到结束，返回 args



3.8.3 运行结果



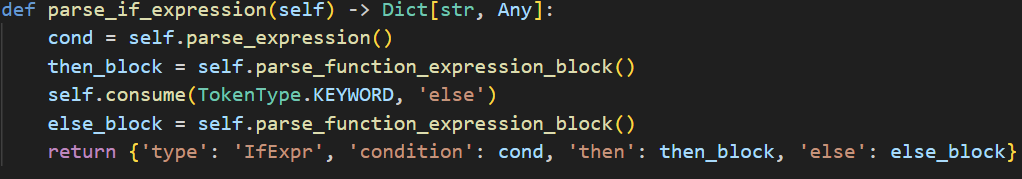


3.9 选择结构（4.1）

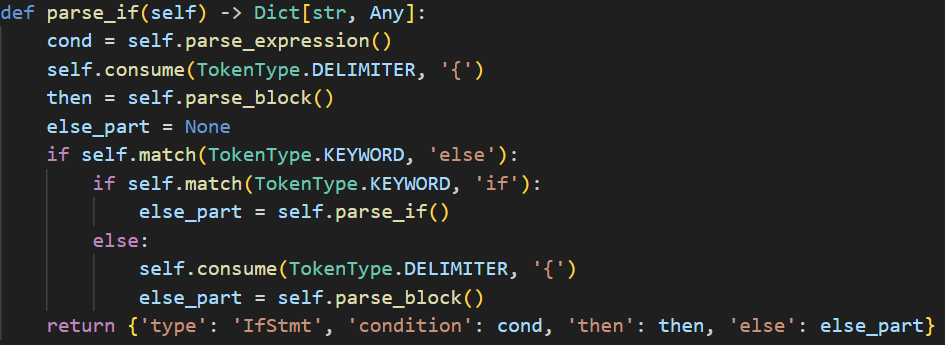
3.9.1 模块功能

处理 if 表达式和 if 语句

3.9.2 实现方法

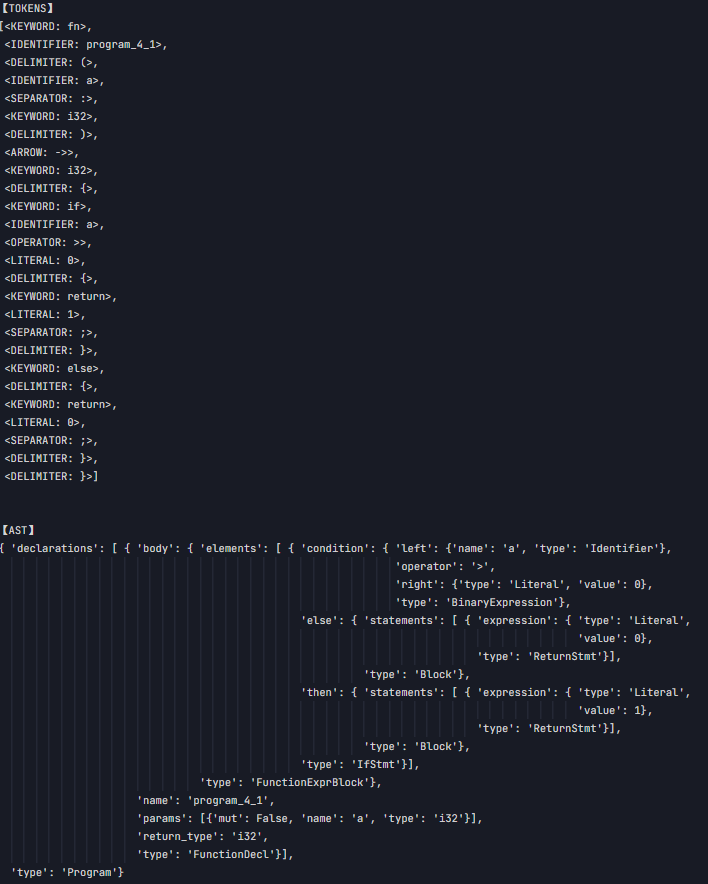


if 表达式



if 语句

3.9.3 运行结果



3.9.4 语法差异

|  |  |  |
| --- | --- | --- |
| 特性 | if 表达式 | if 语句 |
| 语法位置 | 表达式上下文 | 语句上下文 |
| 分支块 | 必须使用{} | 必须使用{} |

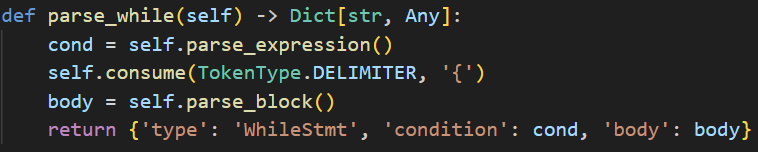
3.10 while循环结构（5.1）

3.10.1 模块功能

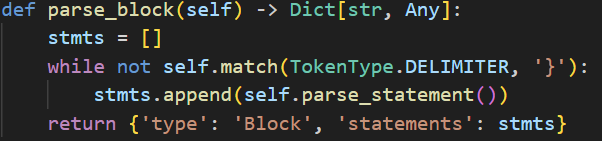
处理 Rust 风格的 while 循环语法，包括条件表达式和循环体的解析，生成对应的 AST 节点

3.10.2 实现方法

在 parse\_while() 函数中，首先消耗 while 关键字，解析条件表达式，再消耗“{”，调用 parse\_block() 函数解析循环体代码块，返回 AST 节点



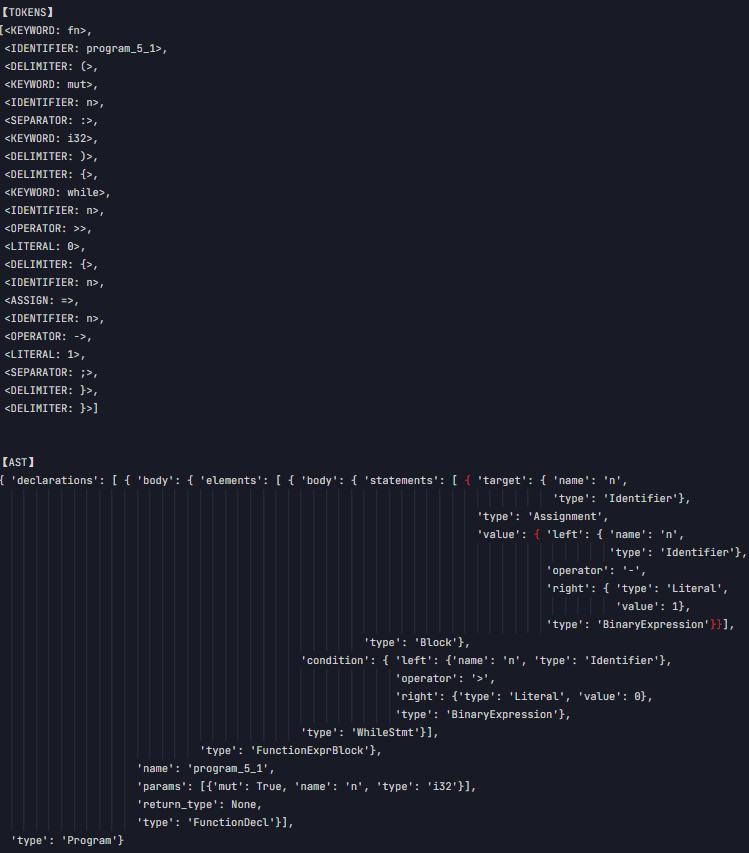
对于 parse\_block() 函数，先初始化语句列表，再循环解析语句直到遇到“}”，返回



3.10.3 语法规则

**WhileStatement → 'while' Expression '{' Statement\* '}'**

3.10.4 运行结果



1. 总结

本次实验我们对**所有的类 RUST 词法规则和语法规则**使用python 3.12.6 进行了实现，本次报告中则对作业所要求的基础词法、语法规则进行了详细地说明，并对各部分的核心代码与示例运行结果进行了截图展示。

在实现的过程中，我们采用了自底向上的模块化开发方式来构建代码。首先我们完成了词法分析器的基础框架，通过正则表达式定义了各种 Token 的匹配规则，包括数字、运算符、括号等基本元素，并特别处理了关键字与普通标识符的区分。初始版本仅支持简单Token识别，随后逐步加入了对注释（//和/\* \*/两种形式）和空白符的过滤功能，以及终止符 # 的特殊处理，使词法分析更加完善。

接着我们在开发语法分析器时，采用递归下降的方法，从最顶层的 Program 规则开始实现。最初只支持空函数声明（如fn main() {}），通过不断迭代扩展语法规则：先加入Block块解析，然后逐步支持Return语句和空语句（单独分号）。每次新增语法规则时，都需要在Lexer中补充对应的Token支持，并调整Parser的消费逻辑。例如，实现Return语句时，先在Lexer中确保能正确识别return关键字和数字字面量，然后在Parser中添加parse\_return()方法，处理表达式（当前仅支持字面量）和强制分号结尾的校验。

错误处理机制是随着功能增加逐步完善的，从最初简单的Token消费失败报错，到后来在peek()和consume()等方法中加入更精确的语法错误定位。测试用例的编写贯穿整个开发过程，从最简单的空函数开始，逐步测试各种边界情况（如缺少括号、分号等），这些测试帮助发现了多个消费逻辑中的边界条件问题。整个开发过程体现了编译器前端各组件协同工作的特点：Lexer的改动会影响Parser的Token流，而Parser规则的扩展又常常需要Lexer提供新的Token支持，两者通过精心设计的接口保持了解耦和灵活性。

经过本次实验，我们深入理解了编译器前端工作的基本原理和实现方法，对词法分析和语法分析的关键技术有了更直观的认识。在构建Lexer的过程中，我们学习到正则表达式在词法分析中的核心作用，掌握了如何通过模式匹配高效提取Token，并理解了关键字与标识符的优先级处理机制。Parser的开发让我们实践了递归下降分析法的具体应用，体会到语法规则的层级分解如何转化为代码的递归调用结构，特别是消费（consume）和匹配（match）操作对语法正确性的保障作用。在调试过程中，我们深刻认识到编译器设计中严谨性的重要性——即使是分号或括号的微小遗漏也会导致解析失败，这加深了我们对编程语言语法设计精密性的理解。此外，通过观察AST的生成过程，我们更清晰地认识到源代码如何被结构化表示，为后续可能的语义分析阶段奠定了基础。这次实验不仅巩固了编译原理的理论知识，更培养了实现复杂系统时所需的模块化思维和调试能力，对编程语言底层工作原理有了更系统的认知。

1. 相关文献
2. 码哥比特. 编译原理——用Python实现简易词法分析器[EB/OL]. [2025-03-22]. <https://blog.csdn.net/weixin_40960130/article/details/135056753.>
3. 悲伤的小朋友. 编译原理--词法分析器（python语言实现）[EB/OL]. [2022-04-08]. https://blog.csdn.net/m0\_54759841/article/details/123435506.
4. 编译程序中的词法分析器（python)[EB/OL]. [发布时间未知]. https://zhuanlan.zhihu.com/p/374080540.
5. minstrel（译者）. Rust参考手册中文版[EB/OL]. [发布时间未知]. https://rustwiki.org/zh-CN/reference/introduction.html.