**《编译原理课程设计》报告**

Pybicc编译器设计与实现

**分组序号：02**

**设计地点：微501**

**电子邮件：**

**分组成绩：**

**任课教师：李村合**

|  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- |
| 专业班级 | **计算2101** | **计算2101** | **计算2101** | **计算2104** | **计算2104** | **计算2104** | **计算2104** |
| 姓名 | **许祖耀** | **吕栋** | **颜辉** | **宁健** | **韩燕** | **王序之** | **张茹月** |
| 学号 | **2107010120** | **2107010114** | **2107010121** | **2103020311** | **2103030126** | **2103010414** | **2114010408** |
| 成绩比例(%) | 16% | 16% | 11% | 12.5% | 15.125% | 14.25% | 15.125% |
| 成绩 |  |  |  |  |  |  |  |

**2024 年 4 月17 日**

[1 课程设计目的 4](#_Toc164187662)

[2 课程设计内容 4](#_Toc164187663)

[2.1 功能要求 4](#_Toc164187664)

[2.2 性能要求 5](#_Toc164187665)

[2.3 安全要求 5](#_Toc164187666)

[2.4 可扩展性要求 5](#_Toc164187667)

[2.5 交互要求 5](#_Toc164187668)

[3 课程设计原理 5](#_Toc164187669)

[3.1 程序的总体架构 6](#_Toc164187670)

[3.2 程序编译过程 6](#_Toc164187671)

[3.3 前端设计的整体流程 7](#_Toc164187672)

[4 系统需求分析 8](#_Toc164187673)

[4.1 Pybicc编译程序总体介绍 8](#_Toc164187674)

[4.2 功能需求 8](#_Toc164187675)

[4.3 用到的主要关键词 9](#_Toc164187676)

[4.4 Pybicc语言的文法 10](#_Toc164187677)

[4.5 解释汇编语言所支持的指令 11](#_Toc164187678)

[5 系统设计与实现 11](#_Toc164187679)

[5.1 Pybicc编译程序主要功能模块 11](#_Toc164187680)

[5.2 编译器 12](#_Toc164187681)

[5.3 解释器 20](#_Toc164187682)

[5.4 前端页面 24](#_Toc164187683)

[6 系统测试与运行结果分析 26](#_Toc164187684)

[6.1 测试程序1 26](#_Toc164187685)

[6.2 测试程序2 27](#_Toc164187686)

[6.3 测试程序3 27](#_Toc164187687)

[6.4 测试程序4 30](#_Toc164187688)

[6.5 测试程序5 31](#_Toc164187689)

[6.6 测试结果1 31](#_Toc164187690)

[6.7 测试结果2 33](#_Toc164187691)

[6.8 测试结果3 35](#_Toc164187692)

[6.9 测试结果4 37](#_Toc164187693)

[6.10 测试结果5 39](#_Toc164187694)

[7 心得体会 41](#_Toc164187695)

# 课程设计目的

（1）根据所掌握的编译原理课程的基本知识，使用Python语言编写出一个类C语言编译器+汇编代码解释器。

（2）了解编译器的基本结构，分析编译器的设计原理。

（3）了解解释器的基本结构，分析解释器的设计原理。

（4）增强阅读和编写程序的能力。

（5）深入理解词法分析器的运作，掌握词法分析技巧，并能用所选编程语言实现一个简易词法分析器。使用该分析器对特定程序段进行词法分析。

（6）深化对语法分析器的工作机制的认识，特别是递归下降法的应用，并用所选编程语言编写一个基础的语法分析器。用这个分析器对给定代码段进行语法解析。

（7）理解并掌握语义分析及语法制导翻译流程，在LR语法分析基础上扩展以进行语义分析，产生源代码的四元组表示，并了解语法与语义分析之间的交互。

（8）掌握中间代码优化的关键原则和技术，包括局部、循环及全局优化，对四元组代码进行基本块划分及局部优化以节省存储并提高效率。

（9）实现编译器通过编辑、编译、链接步骤将源代码转换成可执行程序的功能。

（10）实现解释器通过分析和运行步骤直接执行源代码的功能。

# 课程设计内容

设计一个使用 Python 开发的 Pybicc 编译器图形界面，设计需注重用户体验和视觉效果，支持 C语言的编译功能。

## 功能要求

(1) 实现C语言的基本语法结构，包括变量声明、赋值语句、条件语句、循环语句、函数定义和调用等。

(2) 实现C语言的运算符和表达式，包括算术运算、逻辑运算、比较运算、位运算等。

(3) 支持C语言的整数、字符串、浮点数字面值常量等。

(4) 支持C语言的IF-ELSE、FOR、WHILE、SWITCH-CASE语句等。

(5) 支持C语言的指针、数组、函数、结构体等高级数据结构。

(6) 支持C语言的语句和运算：

a) 数据类型：int，char，short，long，double

b) 语句：赋值（=），if，while，else，for，void，main，break，switch，return

c) 算术运算：+，－，\*，/，取模%，++，--，

d) 关系运算：==，>，<，>=，<=

e) 逻辑运算：&&，||，!

f) 支持函数的定义、调用

g) 支持复合语句，即 ｛｝ 包含的语句

h) 位运算：&，|，!，(左移<<，右移>>)

## 性能要求

(1) 需要确保编译过程高效，以便能够迅速处理大规模代码文件。

(2) 输出的目标代码应当执行效率高，且能够适应不同的运行环境。

## 安全要求

(1) 必须对用户提供的代码执行细致入微的审查，以防止代码注入或恶意行为。

(2) 需要对产出的目标代码实施严格的安全审计，确保不存在如缓冲区溢出等安全隐患。

## 可扩展性要求

(1) 需要兼容多个操作系统，如Windows和Linux。

(2) 应提供灵活的编译配置，包括优化级别、调试模式以及指定目标操作系统等选项。

## 交互要求

(1) 系统应提供一个图形用户界面(GUI)，通过该界面可直观展示代码执行的输出和结果。此外，界面应支持用户交互，如输入数据、调整执行参数等，从而提高用户体验和程序的灵活性。

# 课程设计原理

通过Python编写的类C语言编译器和解释器允许在任何安装了Python的环境中运行C语言代码。这一创新方法意味着，无论使用者的机器类型如何，只要能够运行Python，就能够编译和执行C语言程序。

编译过程从tokenize函数开始，该函数负责对源文件进行词法分析，将其拆分为一系列的tokens。接着，parse函数以自顶向下的方式解析这些tokens，创建函数和全局变量的抽象语法树（AST），并将这些信息存储在全局变量中。为了提高效率和简化代码，解析的最后阶段会标记并移除所有未被引用的代码片段。

在代码生成阶段，codegen函数计算局部变量在栈上的位置，并构建出.data、.bss和.text段，其中.text段包含了遍历AST得到的具体汇编指令，这些指令随后被输出到一个临时文件中。最后，在解释阶段，interpreter函数模拟执行汇编代码，将执行结果通过图形化界面呈现给用户，完成了从源代码到执行输出的整个流程。

## 程序的总体架构

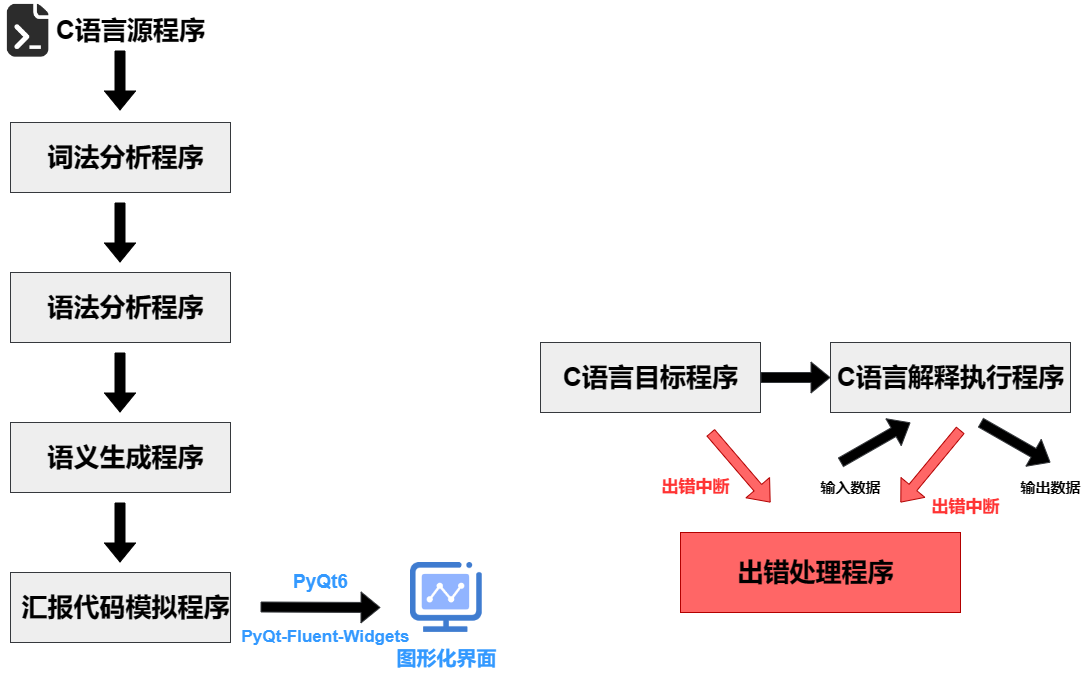


图1 程序的架构概述图

## 程序编译过程

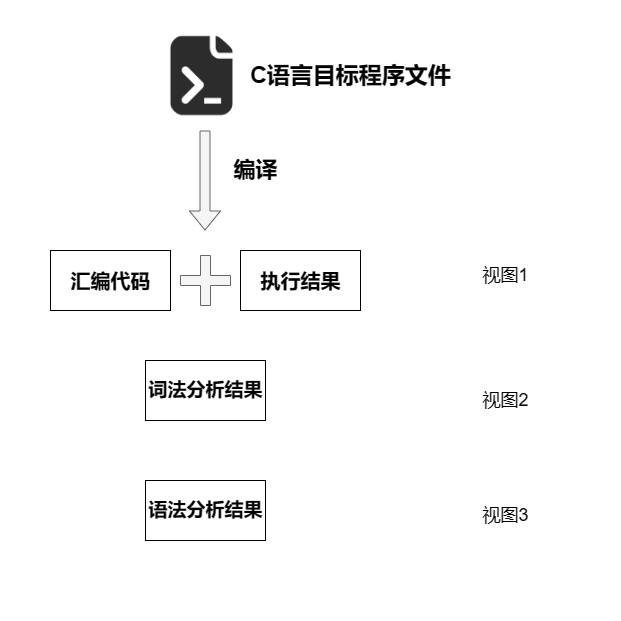


图2 程序的编译过程概述图

## 前端设计的整体流程

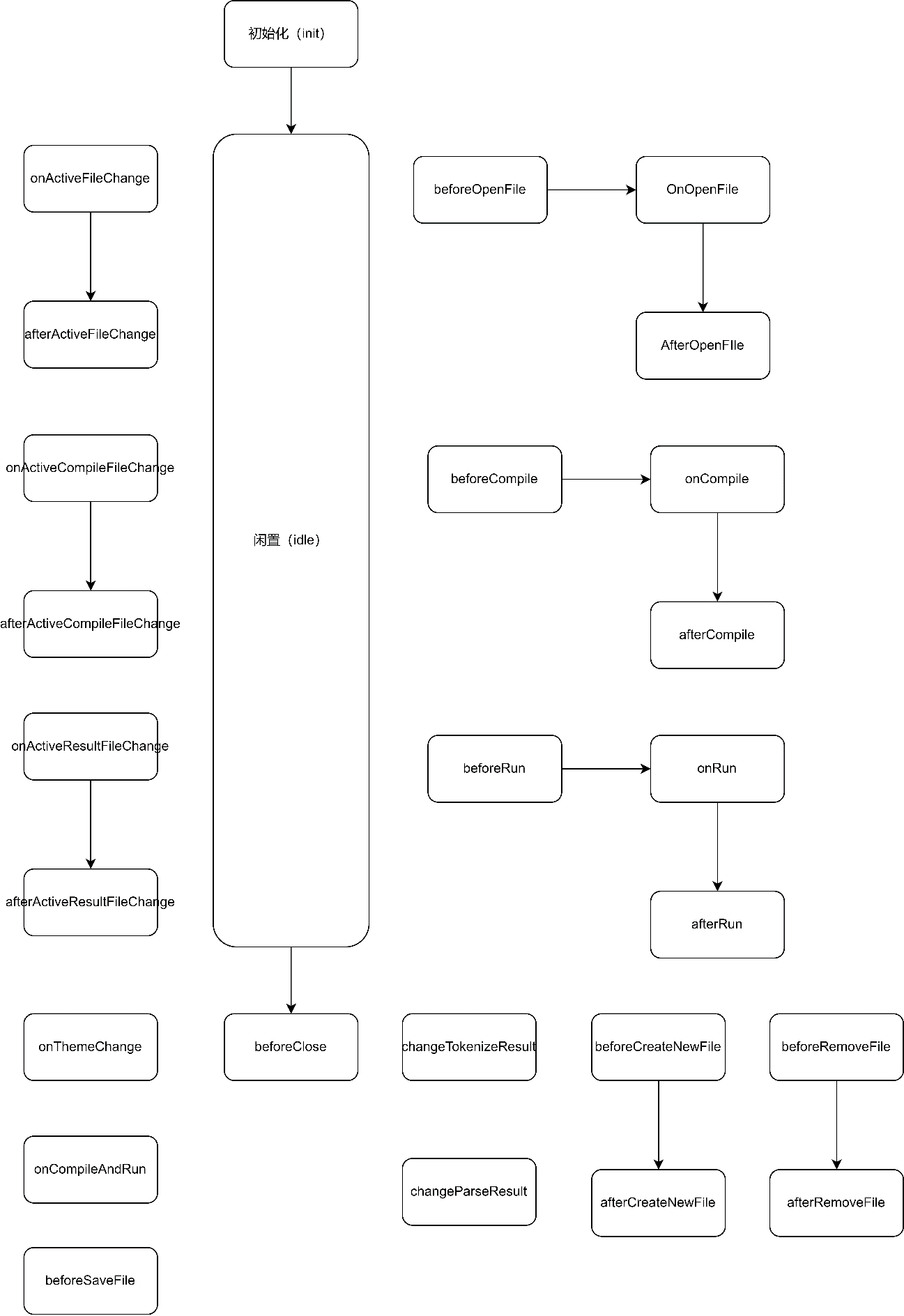


图3 前端设计流程图

# 系统需求分析

Pybicc语言编译将输入的C语言代码编译成汇编代码，并提供了一个汇编代码解释器来执行编译后的代码，最后输出执行结果。该编译器具有词法分析、语法分析和语义生成等功能，以及一个简单的汇编代码解释器，可以模拟CPU执行汇编指令，并输出执行结果。它的工作过程如下所示：

输入源代码 -> 词法分析 -> 语法分析 -> 语义生成 -> 汇编代码解释 -> 输出执行结果。

## Pybicc编译程序总体介绍

该编译器将输入的C语言代码编译成汇编代码，并提供了一个汇编代码解释器来执行编译后的代码，最后输出执行结果。首先，词法分析器将源代码转换为一个Token链表，其中每个Token包含了单词的类型和字符串值。 在语法分析阶段，词法分析生成的Token链表会被转换成一个抽象语法树（Abstract Syntax Tree，AST），用于表示程序的语法结构，其中每个节点代表一个语法结构，例如表达式、语句、函数声明等。在语义生成阶段，抽象语法树会被转换成相应的汇编代码，处理语法结构中的语义信息，例如变量的声明、赋值、运算等，并将其翻译成等效的汇编指令序列。最后，汇编代码解释器负责执行生成的汇编代码，并输出执行结果。

## 功能需求

1、实现了类C语言编译过程。

2、从左到右扫描输入符号串，进行LL(1)文法分析；采用递归子程序法实现语法分析，并用 Python 语言实现了词法分析器、语法分析器、语义生成、汇编代码解释器，能直接生成 intel 80x86汇编代码。

3、在声明中实现了对静态常量、变量和函数声明的支持；赋值语句的 = ，基本的数学运算 +、-、\*、/ 。

4、能够识别单行注释（//）和多行注释（/\* \*/）并跳过。

5、支持函数定义、函数调用、\*取地址，&解引用。

6、基本的逻辑运算符和算术运算符以及取地址解引用运算符，如"==", "!=", "<<", ">>"等。

7、实现一维数组，多维数组，且支持数组的下标访问。

8、支持char类型，string类型的字符。

9、支持字面值常量，包括特殊字符如"\n, \0, \b\a,\t"等。

10、支持结构体。

11、加入bool, long long, enum类型，并允许void类型的函数。

12、支持switch-case语句，并加入break和continue。

13、使用 { } 实现复合语句。

14、使用函数 read() 来同时读入一个或多个数据，函数 print() 用来输出字符串。

15、在循环分支语句中实现了if语句， if……else……语句， for(…,……,……)…语句，while……语句。

16、实现二进制，十六进制，浮点数之间的转换。

## 用到的主要关键词

关键字等用枚举类型囊括：

enum TokenKind

{

// 关键字

"return", "if", "else", "while", "for","int", "short", "long", "void", "char", "bool","enum", "sizeof", "struct", "typedef", "static","break", "continue", "switch", "case", "default","goto", "union", "float", "double"

// 操作符

算术运算符：+、-、\*、/

关系运算符：==、!=、<、>、<=、>=

逻辑运算符：&&、||、!

// 界符

分号（;）

大括号（{}）

小括号（()）

中括号（[]）

逗号（,）

冒号（:）

双引号（"）

单引号（'）

反斜杠（\）

注释符号（/\* \*/ 和 //）

// 复杂操作符

位运算符：&、|、^、~、<<、>>

赋值运算符：=、+=、-=、\*=、/=、%=,^=,|=,<<=,>>=

自增自减运算符：++、--

成员访问运算符：、->

地址和间接寻址运算符：&、\*

sizeof 运算符

// 输入输出函数

read, print

};

## 类C语言的文法

- program = (typedef | global-var | function )\*

- typedef = typdef basetype ident ";"

- global-var = global-var = basetype declaration type-suffix ";"

- function = basetype declarator "(" params? ")" ("{" stmt\* "}" | ";")

- params = param ("," param)\*

- param = basetype declarator type-suffix

- basetype = basetype = builtin-type | struct-decl | typedef "\*"\*

- builtin-type = "void" | "bool" | "char" | "short" | "int" | "long"

- struct-decl = "struct" ident? ("{" struct-member "}")?

- declaration = basetype ident ("[" num "]")\* ("=" expr) ";"| basetype ";"

- stmt = "return" expr ";"

| "if" "(" expr ")" stmt ("else" stmt)?

| "switch" "(" expr ")" stmt

| "case" num ":" stmt

| "default" ":" stmt

| "while" "(" expr ")" stmt

| "for" "(" (expr? ";" | declaration) expr? ";" expr? ")" stmt

| "{" stmt\* "}"

| "typedef" basetype ident ("[" num "]")\* ";"

| "break" ";"

| "continue" ";"

| "goto" ident ";"

| ident ":" stmt

| declaration

| expr ";"

- expr = assign ("," assign)\*

- assign = logor (assign-op assign)?

- assign-op = "=" | "+=" | "-=" | "\*=" | %= | "/=" | "<<=" | ">>="

- equality = relational ("==" relational | "!=" relational)\*

- relational = shift ("<" shift | "<=" shift | ">" shift | ">=" shift)\*

- shift = add ("<<" add | ">>" add)\*

- add = mul ("+" mul | "-" mul)\*

- mul = cast ("\*" cast | "/" cast | "%" cast)\*

- cast = "(" type-name ")" cast | unary

- unary = ("+" | "-" | "\*" | "&" | "!")? cast

| ("++" | "--") unary

| postfix

- postfix = primary ("[" expr "]" | "." ident | "->" ident | "++" | "--")\*

- primary = "(" expr ")"

| "sizeof" unary

| ident func-args?

| str

| num

## 解释汇编语言所支持的指令

1、push ( source | offset source )

2、pop destination

3、add destination, source

4、sub destination, source

5、imul destination, source

6、idiv operand

7、cqo

8、cmp operand1 operand2

9、sete destination

10、setne destination

11、setl destination

12、setle destination

13、mov destination, source

14、movzb destination, source

15、movsx destination, source

16、movss destination, source

17、movsd destination, source

18、lea destination, source

19、and destination, source

20、or destination, source

21、not destination

22、xor destination, source

23、shl destination, source

24、shr destination, source

25、sal destination, source

26、sar destination, source

27、jnz label

28、je label

29、jne label

30、call label

31、ret

# 系统设计与实现

## Pybicc编译程序主要功能模块

Pybicc项目由三大部分组成，分别是编译器、解释器、图形化界面。

## 编译器

这部分由tokenize.py、parse.py、codegen.py三个主要python文件构成。其中tokenize.py负责词法分析，解析C源代码，生成Token链表；parse.py使用LL(1)文法分析，根据语法分析生成的Token链表，建立抽象语法树（AST）；codegen.py根据语法分析生成的抽象语法树，进行语义分析，翻译为Intel80x86汇编语言。

### Tokenize.py

代码实现了一个**词法分析器**，其主要功能是将输入的程序代码字符串分解成一系列词法单元（Tokens），这些词法单元代表编程语言中的关键字、运算符、标识符、数字和字符串等基本元素。通过对输入文本进行系统的扫描和分类，词法分析器为后续的语法分析阶段提供了清晰、结构化的数据。

1. 实现方式
2. **Token 类和 TokenKind 枚举类：**

Token 类用于表示单个 Token，包括 Token 的种类、字符串值和指向下一个 Token 的指针。TokenKind 枚举类定义了 Token 的各种种类，如关键字、标识符、数字等。

1. **函数和全局变量：**

实现了一系列函数用于处理 Token，如消耗 Token、查看 Token、期望特定种类的 Token等。使用了全局变量 token 来跟踪当前处理的 Token。

1. **Tokenize 函数：**

tokenize 函数是该词法分析器的核心部分，它通过迭代遍历源代码字符串，并根据一系列规则将其分解为 Token。在遍历过程中，忽略空白字符和注释，识别字符串和字符字面值，处理关键字、标识符、单字符运算符和数字字面值。

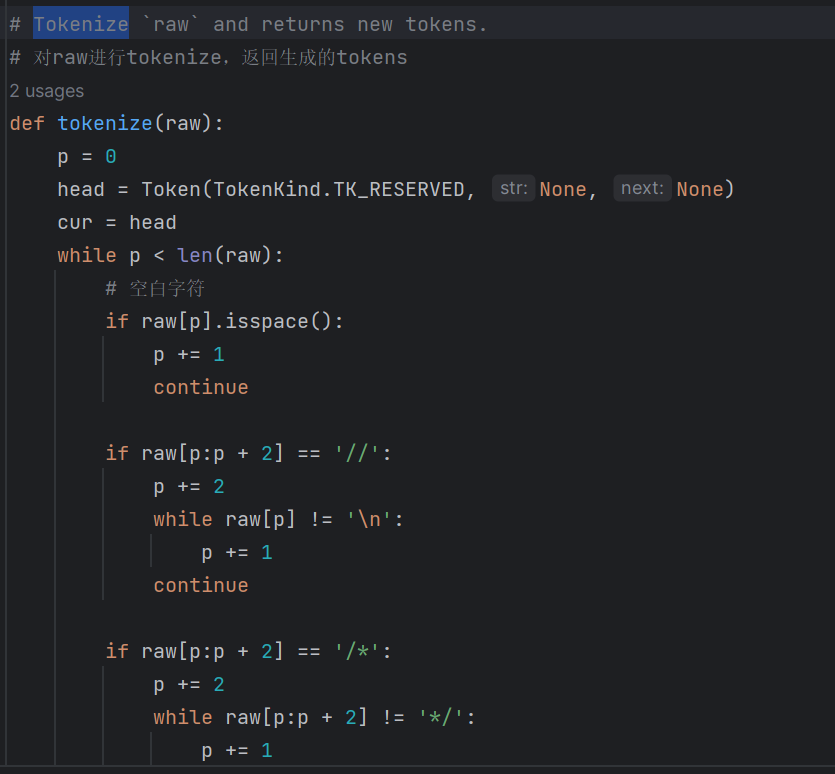


图4 tokenize函数

1. **字符串匹配和字符迭代：**

使用字符串操作函数和索引来对源代码字符串进行匹配和处理。通过迭代遍历源代码字符串，逐字符解析，并根据规则识别 Token 的种类和值。

1. 词法规则
2. **关键字和标识符：**

识别以字母开头，后跟字母、数字或下划线的字符串为标识符，如变量名、函数名等。预定义了一组关键字，如 if、else、while 等，这些关键字不能作为标识符使用。

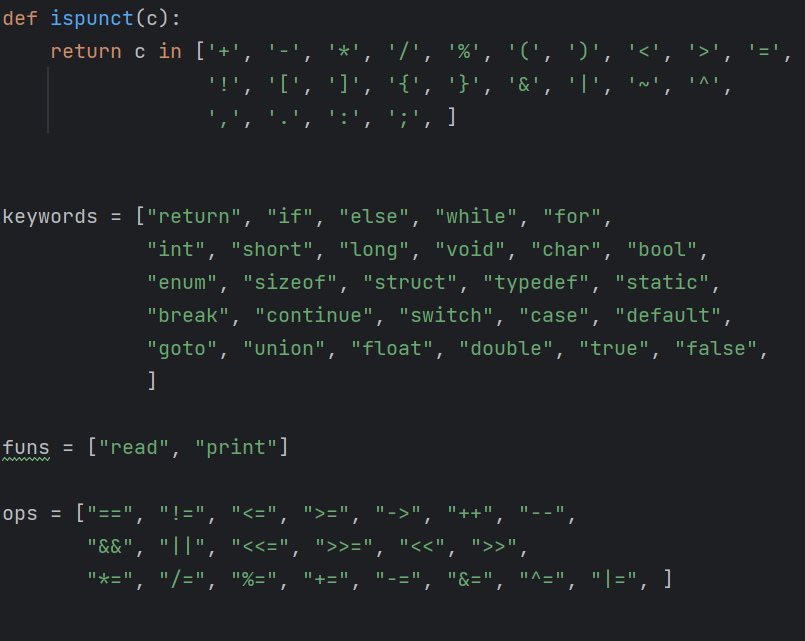


图5 关键字与运算符

1. **数字字面值：**

识别整数和浮点数字面值，包括十进制、十六进制、八进制和二进制表示。支持带有符号和无符号、长短类型修饰符的整数。

1. **字符串和字符字面值：**

识别双引号括起来的字符串字面值，支持转义字符；识别单引号括起来的字符字面值，支持转义字符。

1. **空白字符和注释：**

忽略空白字符和注释，包括空格、制表符和换行符。支持单行注释和多行注释，以 // 和 /\* ... \*/ 分别开头。

1. 测试结果

测试代码用例：

int main(){

int x = 1;

short y = 3;

long z = 5;

bool a = true;

return x + y + z + a;

}

测试结果输出：

TokenKind.TK\_RESERVED int

TokenKind.TK\_IDENT main

TokenKind.TK\_RESERVED (

TokenKind.TK\_RESERVED )

TokenKind.TK\_RESERVED {

TokenKind.TK\_RESERVED int

TokenKind.TK\_IDENT x

TokenKind.TK\_RESERVED =

TokenKind.TK\_NUM 1

TokenKind.TK\_RESERVED ;

TokenKind.TK\_RESERVED short

TokenKind.TK\_IDENT y

TokenKind.TK\_RESERVED =

TokenKind.TK\_NUM 3

TokenKind.TK\_RESERVED ;

TokenKind.TK\_RESERVED long

TokenKind.TK\_IDENT z

TokenKind.TK\_RESERVED =

TokenKind.TK\_NUM 5

TokenKind.TK\_RESERVED ;

TokenKind.TK\_RESERVED bool

TokenKind.TK\_IDENT a

TokenKind.TK\_RESERVED =

TokenKind.TK\_RESERVED true

TokenKind.TK\_RESERVED ;

TokenKind.TK\_RESERVED return

TokenKind.TK\_IDENT x

TokenKind.TK\_RESERVED +

TokenKind.TK\_IDENT y

TokenKind.TK\_RESERVED +

TokenKind.TK\_IDENT z

TokenKind.TK\_RESERVED +

TokenKind.TK\_IDENT a

TokenKind.TK\_RESERVED ;

TokenKind.TK\_RESERVED }

测试结果展示了对输入代码进行词法分析后得到的 Token 序列，每一行表示一个 Token，包括 Token 的种类和对应的字符串值。

### Parse.py

代码是一个编译器的核心部分，涉及到**语法分析**和代码生成。它定义了用于构建和管理抽象语法树（AST）的各种数据结构，并提供了处理源代码的功能，将其转换为编译器可以进一步处理的结构化格式。主要功能包括解析变量定义、函数定义、控制流语句（如if、while、for）、表达式和操作符等。

1. 语法分析实现方式

代码的核心是通过手写递归下降解析器实现语法分析，将源代码转换成语法树结构。这种方法通过一系列的函数来直接实现文法的各个产生式，每个函数代表一个语法规则。函数之间通过递归调用来解析嵌套的语法结构，使得代码结构直观且与文法规则紧密相关。

代码中使用了多个辅助函数，如 consume、expect 和 peek，用于处理和检查当前token。例如，consume 函数尝试匹配并消耗当前token，如果匹配则向前移动到下一个token；peek 函数查看当前token是否符合预期但不消耗它；而 expect 函数则要求当前token必须符合预期，不符合时会抛出错误。

1. 语法树设计

本代码中的语法树设计体现在 Node 类及其字段上，每个 Node 实例代表语法树中的一个节点，通过 kind 属性标识节点的类型（如加法、变量引用、函数调用等）。节点通过 lhs（左手子节点）、rhs（右手子节点）、cond（条件表达式）、then（then 分支）、els（else 分支）等属性与其他节点相连接，形成树状结构。



图6 构建的语法树

1. 测试结果

测试代码用例：

int main(){

int x = 1;

short y = 3;

long z = 5;

bool a = true;

return x + y + z + a;

}

测试结果：

===全局变量===

===变量结束===

=====函数名:main=====

===函数参数===

===局部变量===

变量名: a 大小: 1 offset: None

变量名: z 大小: 8 offset: None

变量名: y 大小: 2 offset: None

变量名: x 大小: 4 offset: None

===函数语句===

NodeKind.ND\_EXPR\_STMT

NodeKind.ND\_ASSIGN

NodeKind.ND\_VAR x

NodeKind.ND\_NUM 1

NodeKind.ND\_EXPR\_STMT

NodeKind.ND\_ASSIGN

NodeKind.ND\_VAR y

NodeKind.ND\_NUM 3

NodeKind.ND\_EXPR\_STMT

NodeKind.ND\_ASSIGN

NodeKind.ND\_VAR z

NodeKind.ND\_NUM 5

NodeKind.ND\_EXPR\_STMT

NodeKind.ND\_ASSIGN

NodeKind.ND\_VAR a

NodeKind.ND\_NUM 1

NodeKind.ND\_RETURN

NodeKind.ND\_ADD

NodeKind.ND\_ADD

NodeKind.ND\_ADD

NodeKind.ND\_VAR x

NodeKind.ND\_VAR y

NodeKind.ND\_VAR z

NodeKind.ND\_VAR a

=====函数结束======

测试结果展示了对给定函数的静态分析，包括了函数内部局部变量的声明和初始化、函数语句的执行顺序以及函数返回值的推导过程。

### Codegen.py

代码负责将抽象语法树（AST）转换为汇编语言。它实现了从高级编程语言到低级机器语言的关键转换步骤，使得编写的程序能够被计算机硬件执行。

主要功能包括：地址生成（gen\_addr）、加载和存储变量（load、store）、处理数据类型转换（truncate）、生成二进制运算和逻辑运算的汇编代码（gen\_binary），以及控制流语句如循环和条件分支的代码生成。此外，它还处理函数调用和返回，包括参数的传递和函数栈的管理。

1. 汇编语言生成

在文件顶部，首先引入了必要的模块，例如struct用于数据类型的转换，type和parse则提供了节点类型和类型检查的功能。之后定义了多个列表，用于处理函数参数在汇编代码中的寄存器分配，以及全局的序列变量用于生成唯一的标签。

函数gen\_addr生成地址加载的代码，用于处理变量和解引用等操作。gen\_lval则生成左值的汇编代码，用于赋值操作。浮点数的转换函数float\_to\_ieee754和double\_to\_ieee754将浮点数转换为IEEE 754格式的二进制字符串。

load和store函数用于生成加载和存储指令，这些指令会根据变量的类型（如布尔、整型等）来选择相应的操作。inc和dec函数分别生成增加和减少的汇编代码，这些函数处理自增和自减运算符。

gen\_binary函数处理二元运算符，如加、减、乘、除等，并根据操作的类型生成相应的汇编代码。gen是主要的代码生成函数，它根据传入的AST节点类型递归地生成整个程序的汇编代码。这个函数通过匹配节点类型，调用不同的代码生成函数。

在文件的底部，emit\_data和emit\_text分别生成数据段和代码段的汇编指令。数据段主要用于声明全局变量，而代码段包含了所有函数的定义和指令。

最后，codegen函数组合了上述功能，生成整个程序的汇编代码。整体而言，这段代码是一个典型的编译器代码生成模块，将高级语言的抽象表示转换为可以直接由机器执行的低级语言代码，涵盖了从基本的算术运算到复杂控制流和函数调用的处理。

1. 测试结果

测试用例：

int main(){

int x = 1;

short y = 3;

long z = 5;

bool a = true;

return x + y + z + a;

}

测试结果：

.intel\_syntax noprefix

.data

.text

.global main

main:

push rbp

mov rbp, rsp

sub rsp, 16

lea rax, [rbp-15]

push rax

push 1

pop edi

pop rax

mov [rax], edi

push edi

add rsp, 4

lea rax, [rbp-11]

push rax

push 3

pop di

pop rax

mov [rax], di

push di

add rsp, 2

lea rax, [rbp-9]

push rax

push 5

pop rdi

pop rax

mov [rax], rdi

push rdi

add rsp, 8

lea rax, [rbp-1]

push rax

push 1

cmp rdi, 0

setne dil

movzb rdi, dil

pop dil

pop rax

mov [rax], dil

push dil

add rsp, 1

lea rax, [rbp-15]

push rax

pop rax

movsxd rax, dword ptr [rax]

push rax

lea rax, [rbp-11]

push rax

pop rax

movsx rax, word ptr [rax]

push rax

pop rdi

pop rax

add rax, rdi

push rax

lea rax, [rbp-9]

push rax

pop rax

mov rax, [rax]

push rax

pop rdi

pop rax

add rax, rdi

push rax

lea rax, [rbp-1]

push rax

pop rax

movsx rax, byte ptr [rax]

push rax

pop rdi

pop rax

add rax, rdi

push rax

pop rax

jmp .L.return.main

.L.return.main:

mov rsp, rbp

pop rbp

ret

翻译成汇编代码，将源代码转换为计算机可执行的机器码指令，以便在计算机上执行。

## 解释器

解释器(interpreter.py) 为了在图形化界面（GUI）中得到运行结果，我们还额外编写了用于解释执行Intel80x86汇编语言的解释器，其依据汇编语言，逐行解释执行指令，模拟相关操作。

解释器以字节级别模拟实现了对存储器的访问存取，每个单元存储一个字节的数据，并实现了ALU的相应计算功能。

### 寄存器

寄存器（Register） 寄存器使用类进行模拟，所有寄存器（如rax,rbp,rsp）均为Register类的实例，访问存取通过成员函数提供的接口实现，所有实例共享shared\_storage（存储空间）变量。由此实现多个不同大小寄存器（rax,eax,ax,al）使用同一存储区域的特性。



图7 寄存器搭建

### 内存

内存（Memory） 内存使用Memory类进行模拟，访问存取通过成员函数提供的接口实现。模拟了一个内存管理器，可以在一个数组中插入和获取整型数据。

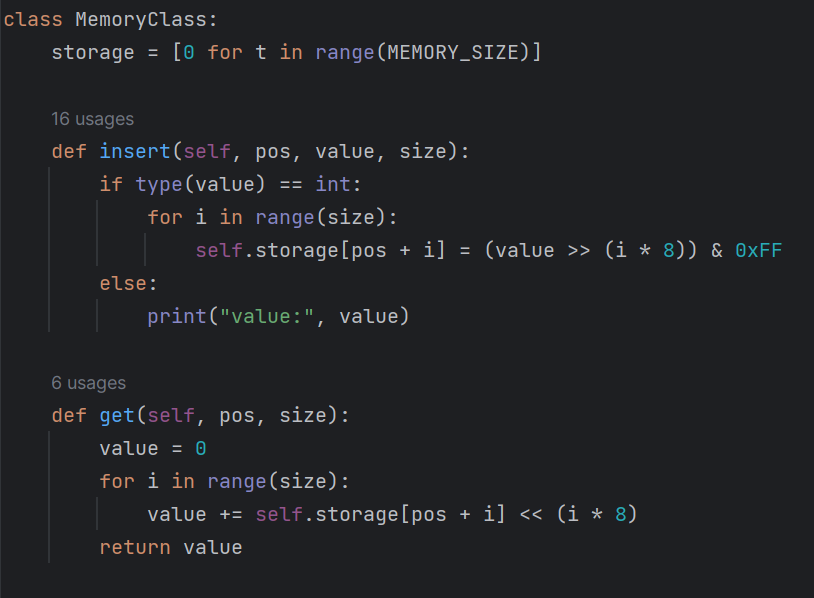


图8 内存管理器

### 寻址取数

在模拟指令运行时，解释器会将指令操作数送至addresing函数获取操作数的寻址方式，然后根据寻址方式 使用Memory.get(pos,size)从内存获取操作数的值，或使用getValueByAddressing获取操作数的值。

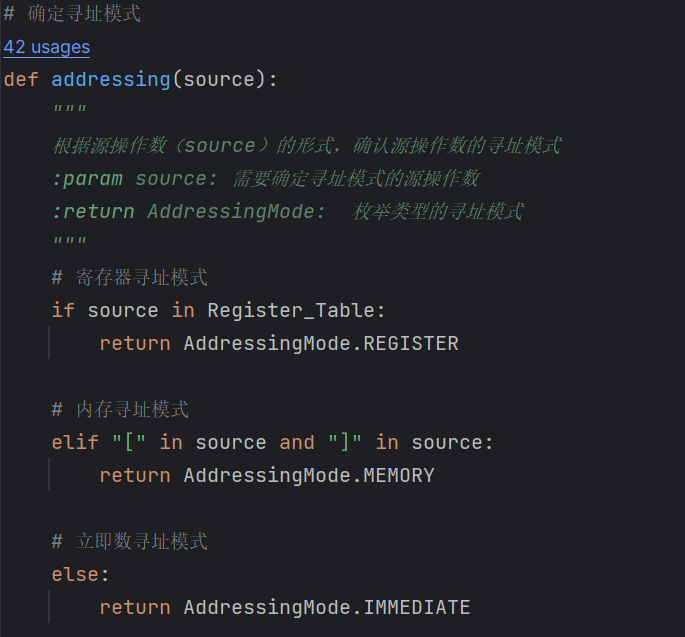


图9 addressing函数

### 内存寻址

在内存寻址时，通过getMomoryAddress(expr)获得内存地址，再根据表达式，转换为前缀表达式形式，计算出表达式的值，然后根据值计算出内存地址。

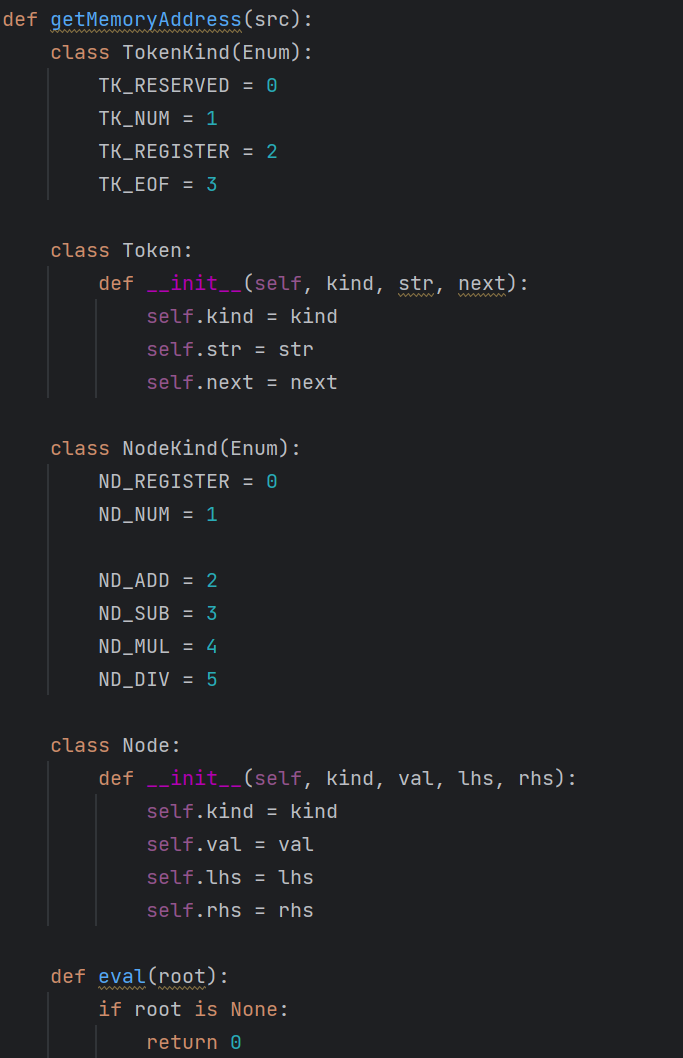


图10 getMemoryAddress函数

### 处理数据段

enterDataSegment函数用于处理数据段，将数据段中的数据存储至内存中。



图11 enterDataSegment函数

### 模拟执行汇编代码

run\_command()函数用于模拟执行汇编代码，根据指令的操作码，调用相应的函数模拟执行指令。

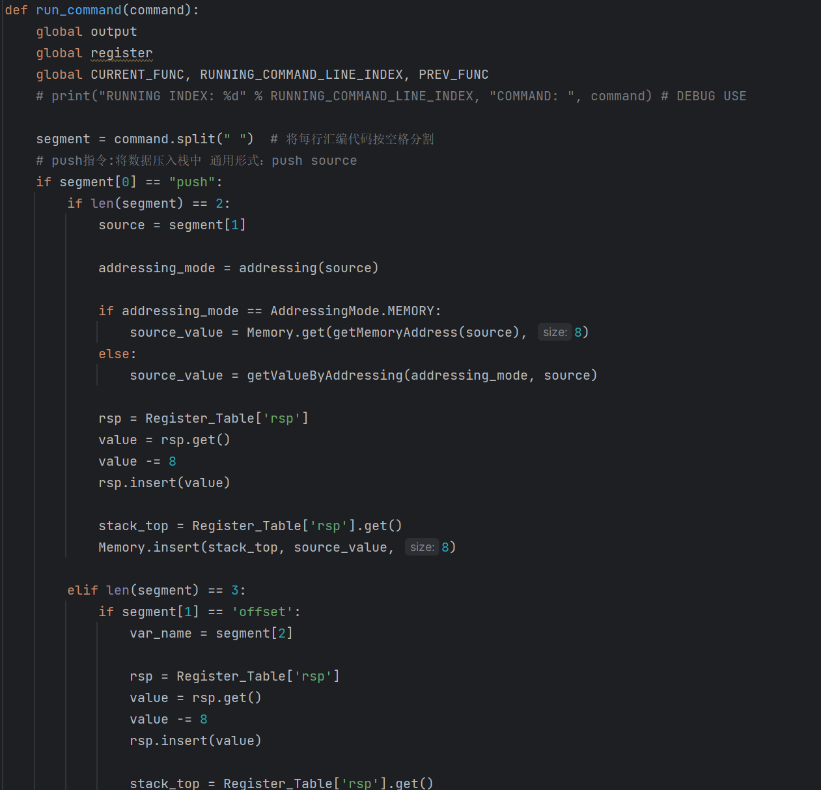


图12 run\_command()函数

## 前端页面

图形化界面(GUI) 图形化界面采用Fluent 2设计风格，使用PyQt6及PyQt-Fluent-Widgets实现，并提供了部分集成开发环境（IDE）的功能。

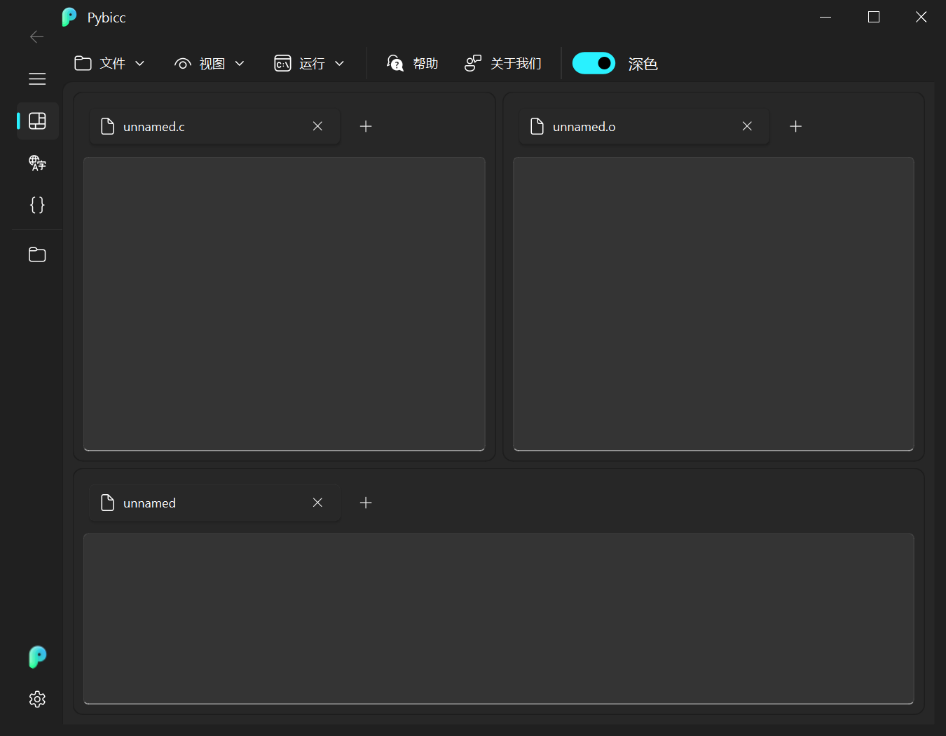


图13 深色版本编译器

图形化界面分为四大视图：总览视图、词法分析视图、语法分析视图、文件视图。

### 总览视图

显示总体功能，左上边框显示C源代码，右上边框显示编译后的汇编代码，下边框显示解释执行的结果。

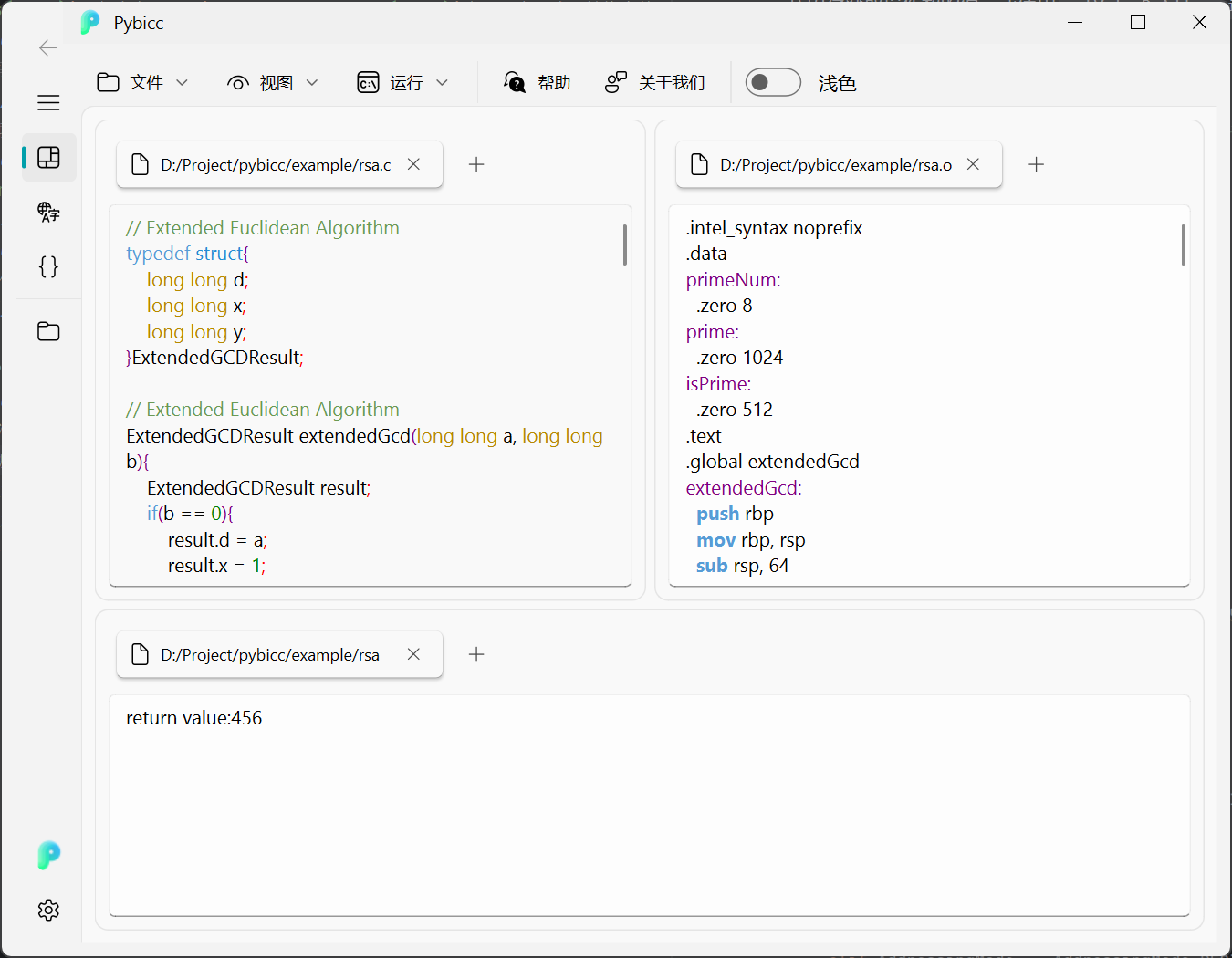


图14 总览视图

### 词法分析视图

显示词法分析的结果，左边框显示源代码，右边框显示词法分析的结果。

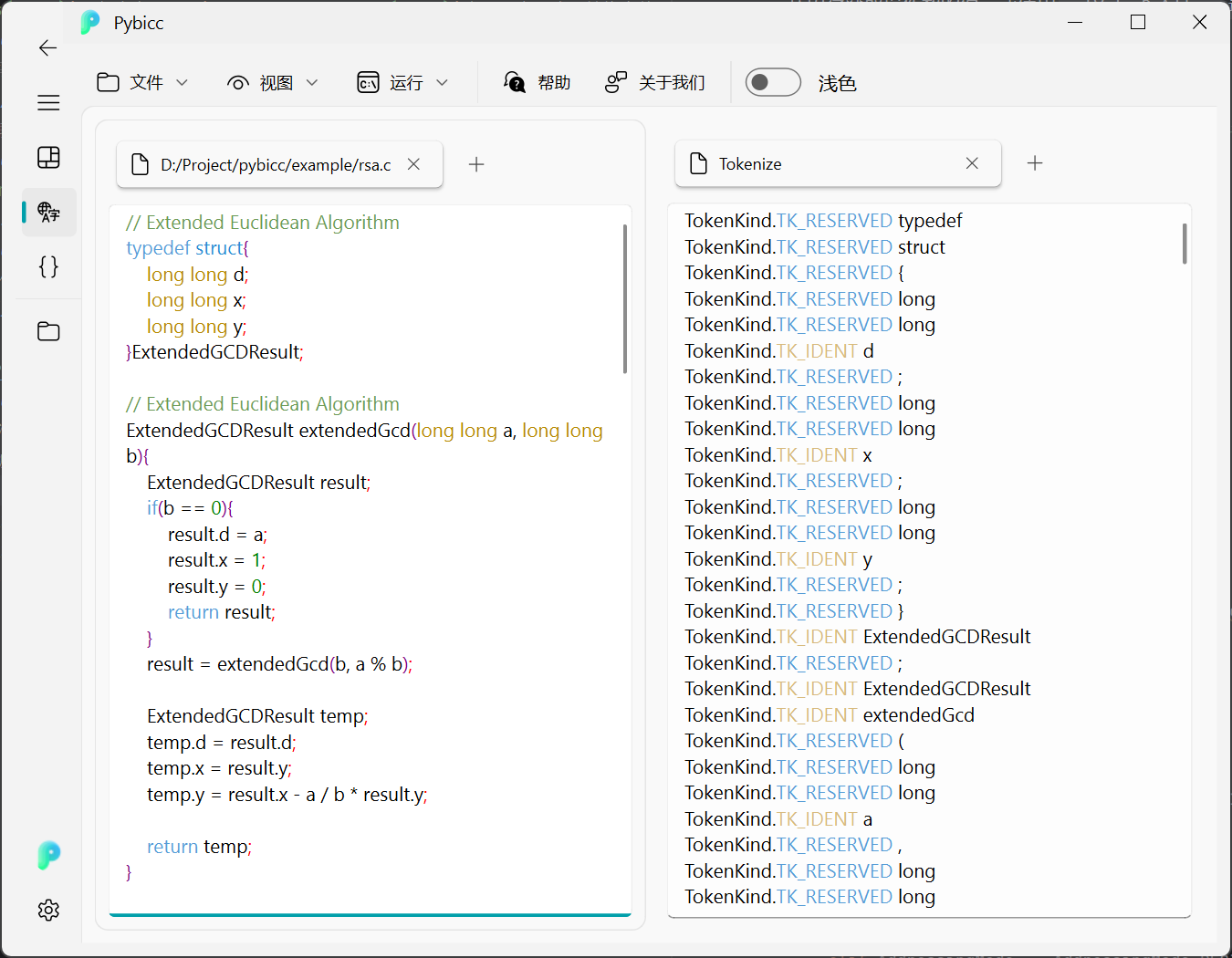


图15 词法分析视图

### 语法分析视图

显示语法分析的结果，左边框显示源代码，右边框显示语法分析的结果。

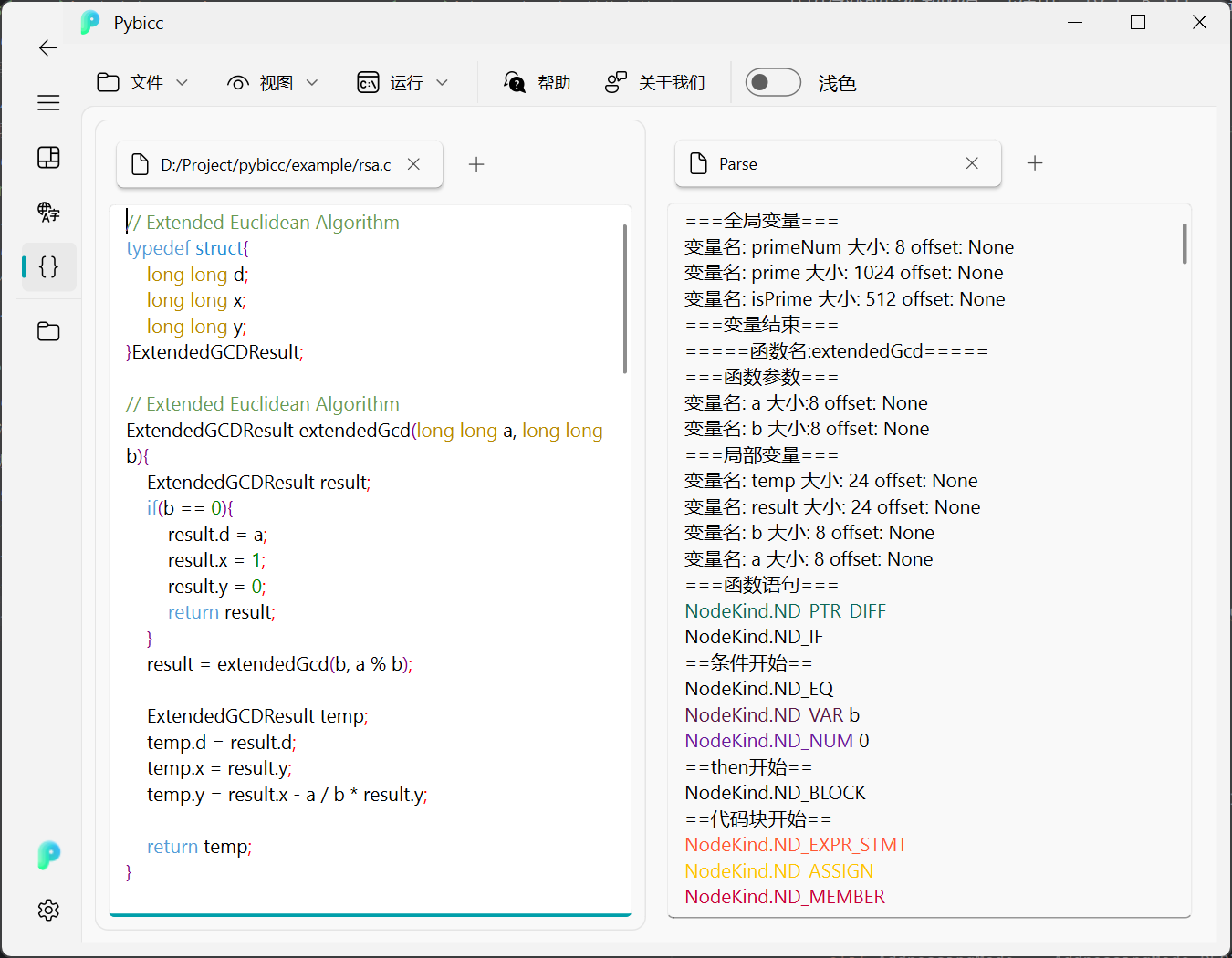


图16 语法分析视图

### 文件视图

显示文件的目录结构，通过双击文件可打开文件。

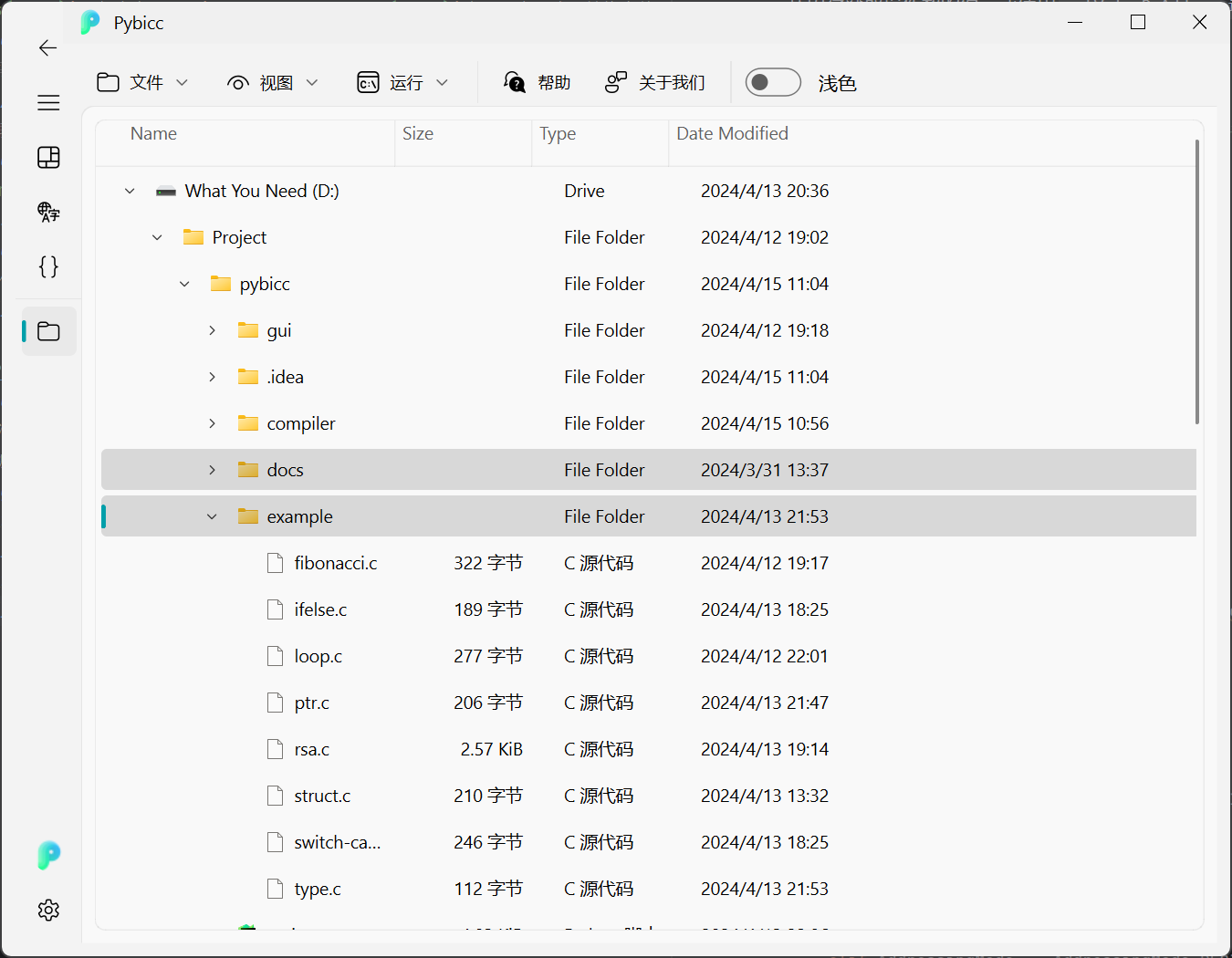


图17 文件视图

### 图形化界面

图形化界面提供了代码编辑器，汇编语言编辑器功能，同时支持深色模式，多文件打开与切换，文件保存等功能。

代码编辑器：支持代码高亮，代码折叠，代码提示，代码补全等功能。

汇编语言编辑器：支持汇编语言的编辑，高亮，折叠，提示等功能。

词法分析结果高亮，语法分析结果高亮。 图形化界面的开发引入了**生命周期**的概念，通过生命周期管理，实现了界面刷新，数据更新等功能。

# 系统测试与运行结果分析

## 测试程序1

//switch-case程序

int main() {

int num = 3;

//switch-case

switch (num) {

case 1:

write(123);

break;

case 2:

write(456);

break;

default:

write(789);

}

}

## 测试程序2

//struct结构体

typedef struct {

int x;

int y;

} Point;

void random\_point(Point \*p) {

p->x = 3;

p->y = 4;

return 0;

}

int main() {

Point p;

random\_point(&p);

return p.x + p.y;

}

## 测试程序3

//RSA算法

// Extended Euclidean Algorithm

typedef struct{

long long d;

long long x;

long long y;

}ExtendedGCDResult;

// Extended Euclidean Algorithm

ExtendedGCDResult extendedGcd(long long a, long long b){

ExtendedGCDResult result;

if(b == 0){

result.d = a;

result.x = 1;

result.y = 0;

return result;

}

result = extendedGcd(b, a % b);

ExtendedGCDResult temp;

temp.d = result.d;

temp.x = result.y;

temp.y = result.x - a / b \* result.y;

return temp;

}

// solve ax + by = gcd(a, b)

long long getGCD(long long a, long long b){

if(b == 0){

return a;

}

return getGCD(b, a % b);

}

// solve ax = 1 (mod m)

long long getModInverse(long long a, long long m){

if (getGCD(a, m) != 1){

return -1;

}

ExtendedGCDResult result = extendedGcd(a, m);

long long x = result.x;

return (x % m + m) % m;

}

int isPrime[128];

long long prime[128];

long long primeNum;

// Sieve of Eratosthenes

void sieve(long long n){

for(long long i = 0; i <= n; i++){

isPrime[i] = 1;

}

isPrime[0] = isPrime[1] = 0;

for(long long i = 2; i <= n; i++){

if(isPrime[i] == 1){

prime[primeNum++] = i;

for(long long j = i \* i; j <= n; j += i){

isPrime[j] = 0;

}

}

}

}

// Fast Power Encryption and Decryption

long long encrypt(long long m, long long e, long long n){

long long result = 1;

long long exp = e;

long long base = m;

while(exp > 0){

if(exp % 2 == 1){

result = result \* base % n;

}

base = base \* base % n;

exp /= 2;

}

return result;

}

long long decrypt(long long c, long long d, long long n){

long long result = 1;

long long exp = d;

long long base = c;

while(exp > 0){

if(exp % 2 == 1){

result = result \* base % n;

}

base = base \* base % n;

exp >>= 1;

}

return result;

}

int main(){

primeNum = 0;

sieve(32);

long long p = prime[primeNum - 1];

long long q = prime[primeNum - 2];

long long n = p \* q;

long long phi = (p - 1) \* (q - 1);

// e

long long e = 0;

for (e = phi; e >= 1; e--){

if(getGCD(e, phi) == 1){

break;

}

}

// d

long long d = getModInverse(e, phi);

// public key, private key

long long m = 456;

long long c = encrypt(m, e, n);

long long m2 = decrypt(c, d, n);

return m2;

}

## 测试程序4

//指针的使用

void change(int \*p){

\*p = 16;

return 0;

}

int main() {

int i = 3;

int \*p=&i;

\*p = 20;

write(\*p);

int \*\*z=&p;

change(\*z);

write(\*p);

return 0;

}

## 测试程序5

//斐波那契数列求和

int fibonacci(int n) {

if (n <= 0)

return 0;

else if (n == 1)

return 1;

else

return fibonacci(n - 1) + fibonacci(n - 2);

}

int main() {

int num\_terms;

read(&num\_terms);

for (int i = 0; i < num\_terms; ++i) {

write(fibonacci(i));

}

return 0;

}

## 测试结果1

1. 运行Pybicc，打开switch-case.c文件

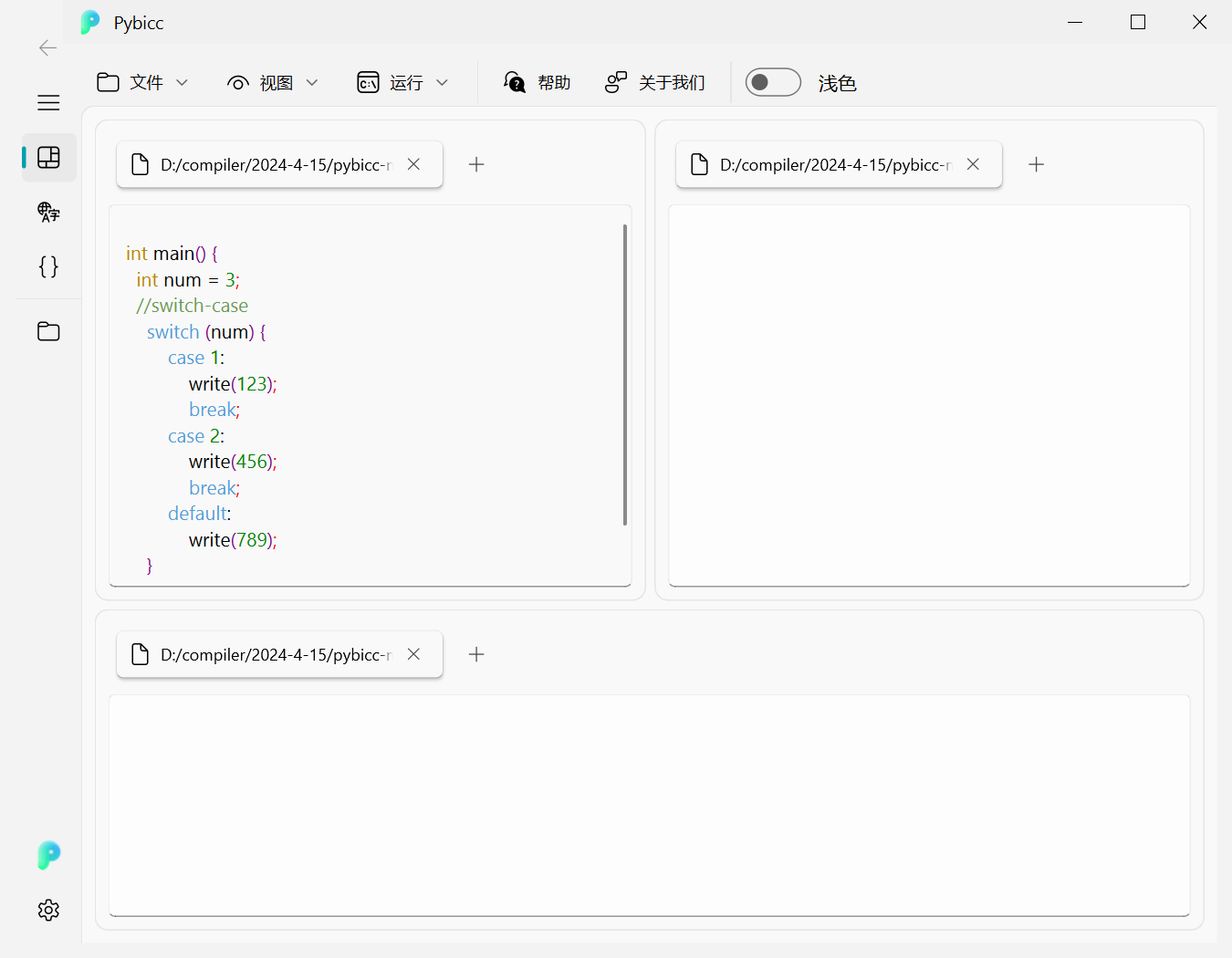


图18 打开文件

1. 选择菜单 运行->编译并运行

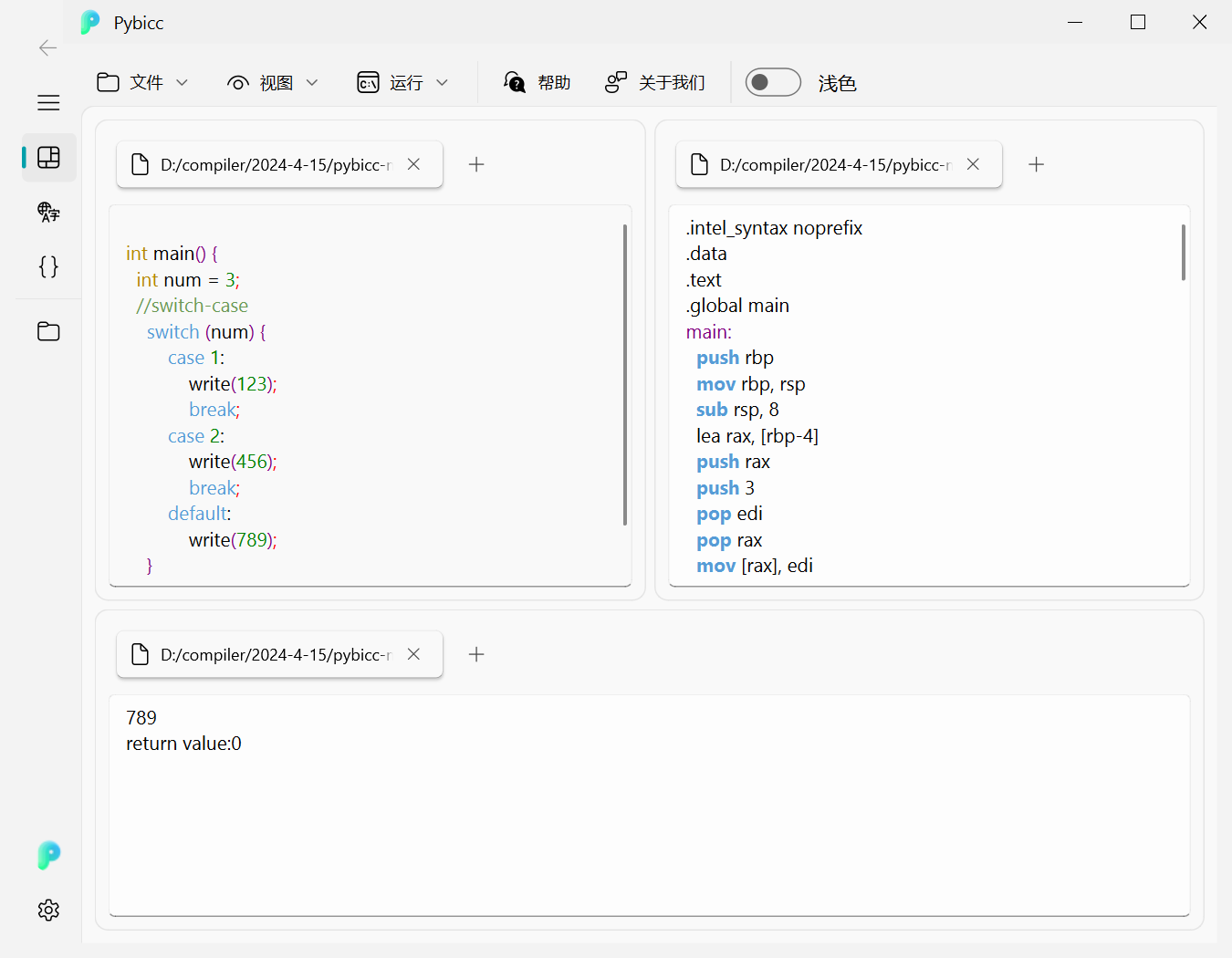


图19 编译运行

1. 在编译的同时，可以在导航栏点击词法分析与语法分析，查看中间步骤

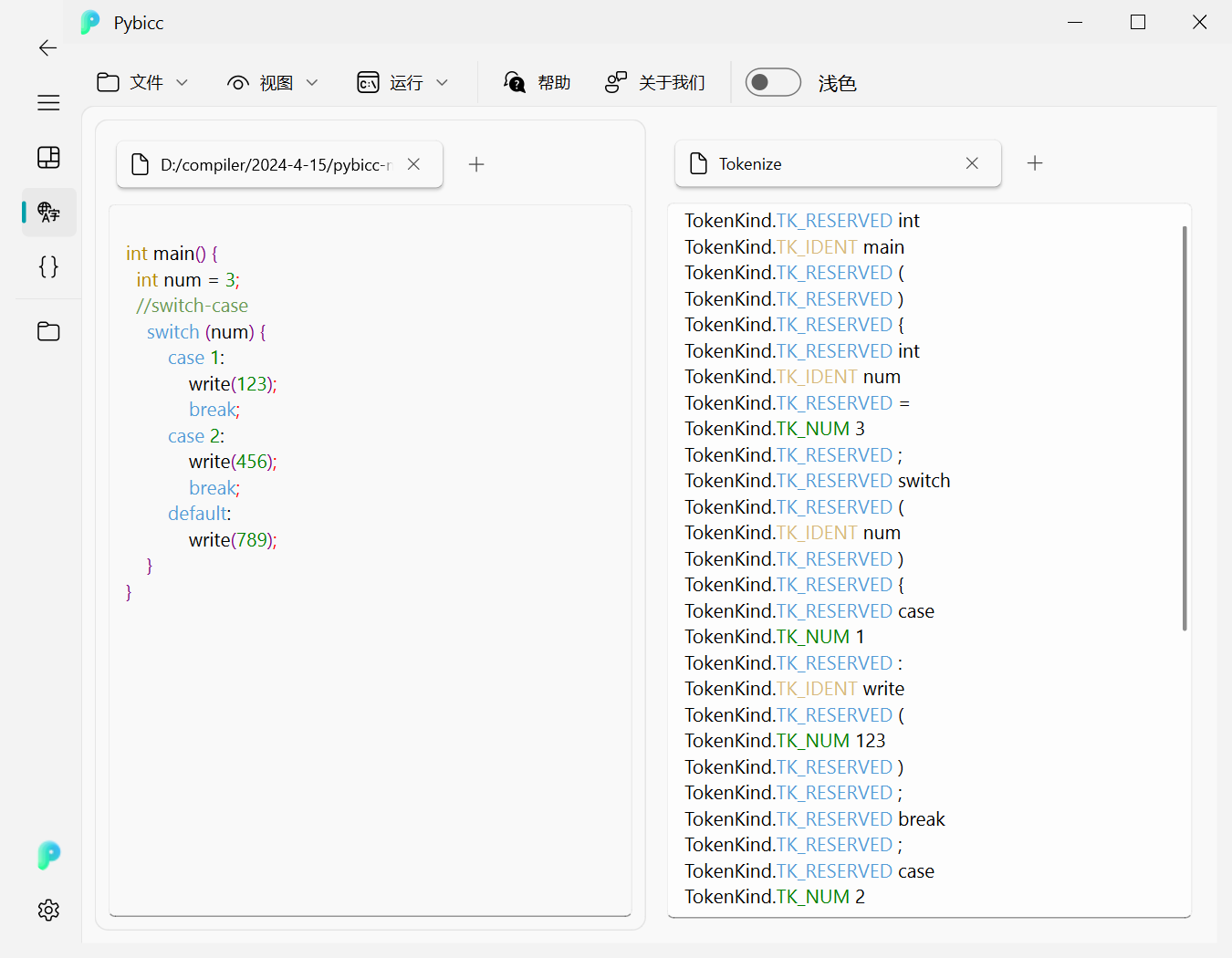


图20 查看词法分析

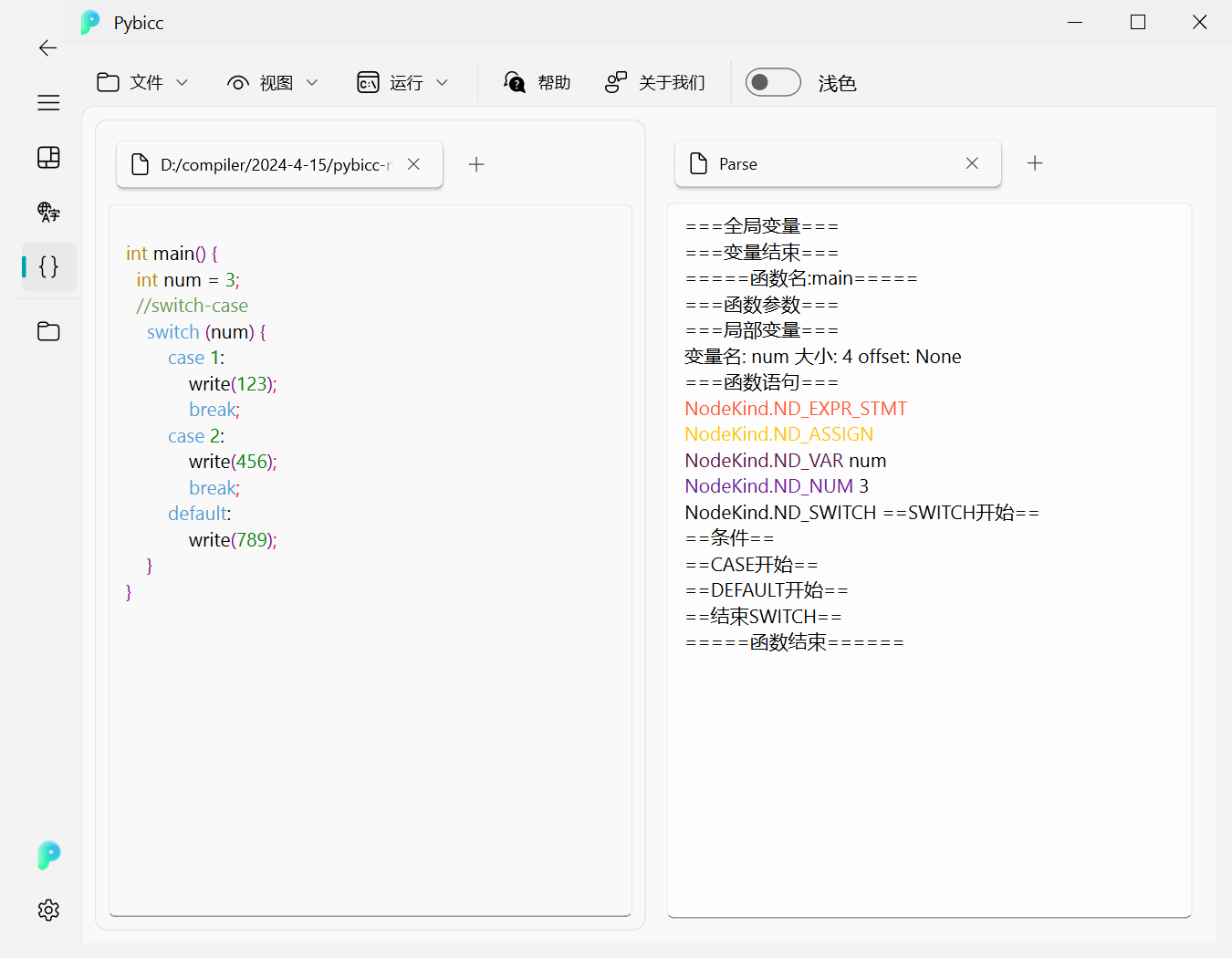


图21 查看语法分析

## 测试结果2

1. 运行Pybicc，打开struct.c文件

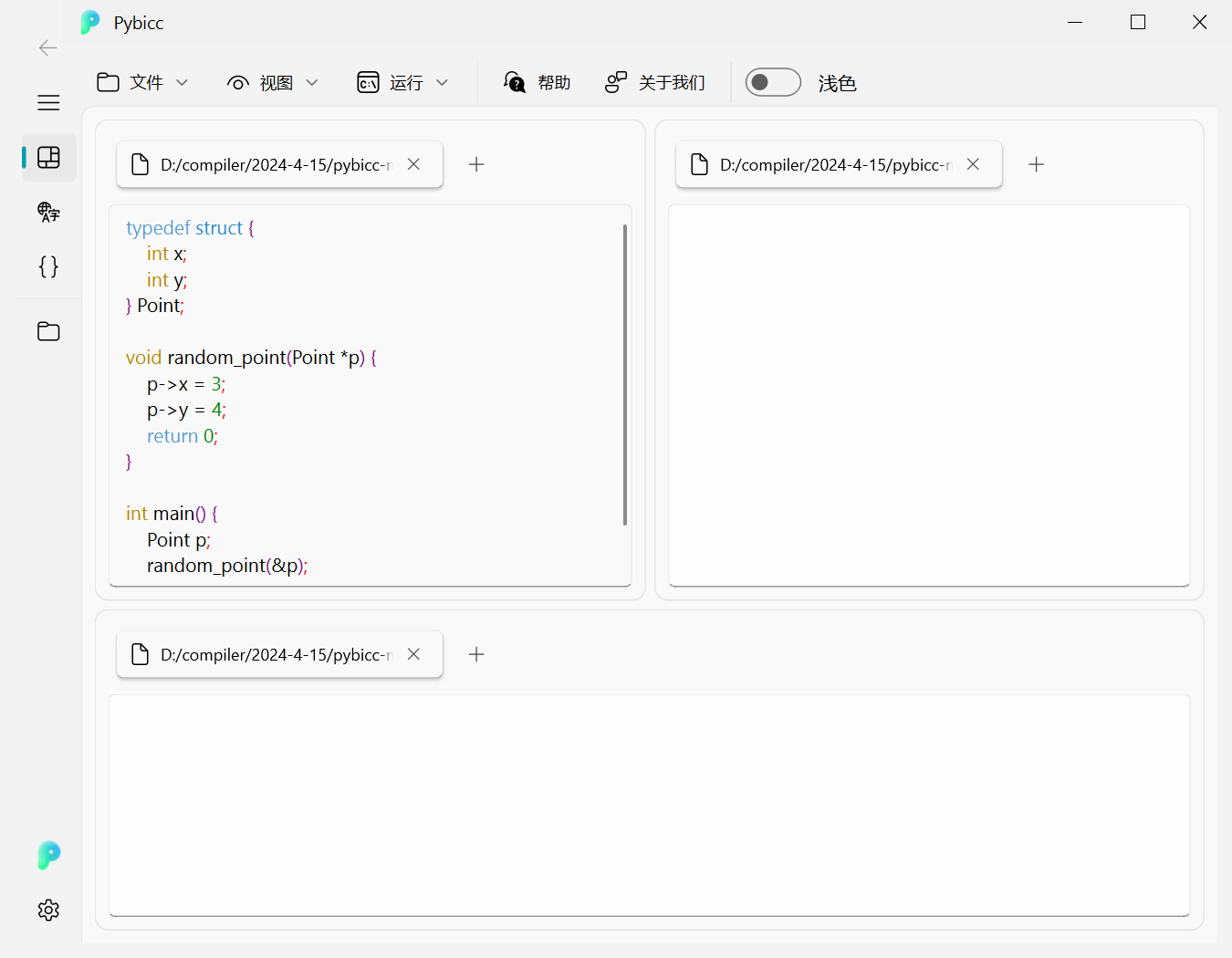


图 22 打开文件

1. 选择菜单 运行->编译并运行

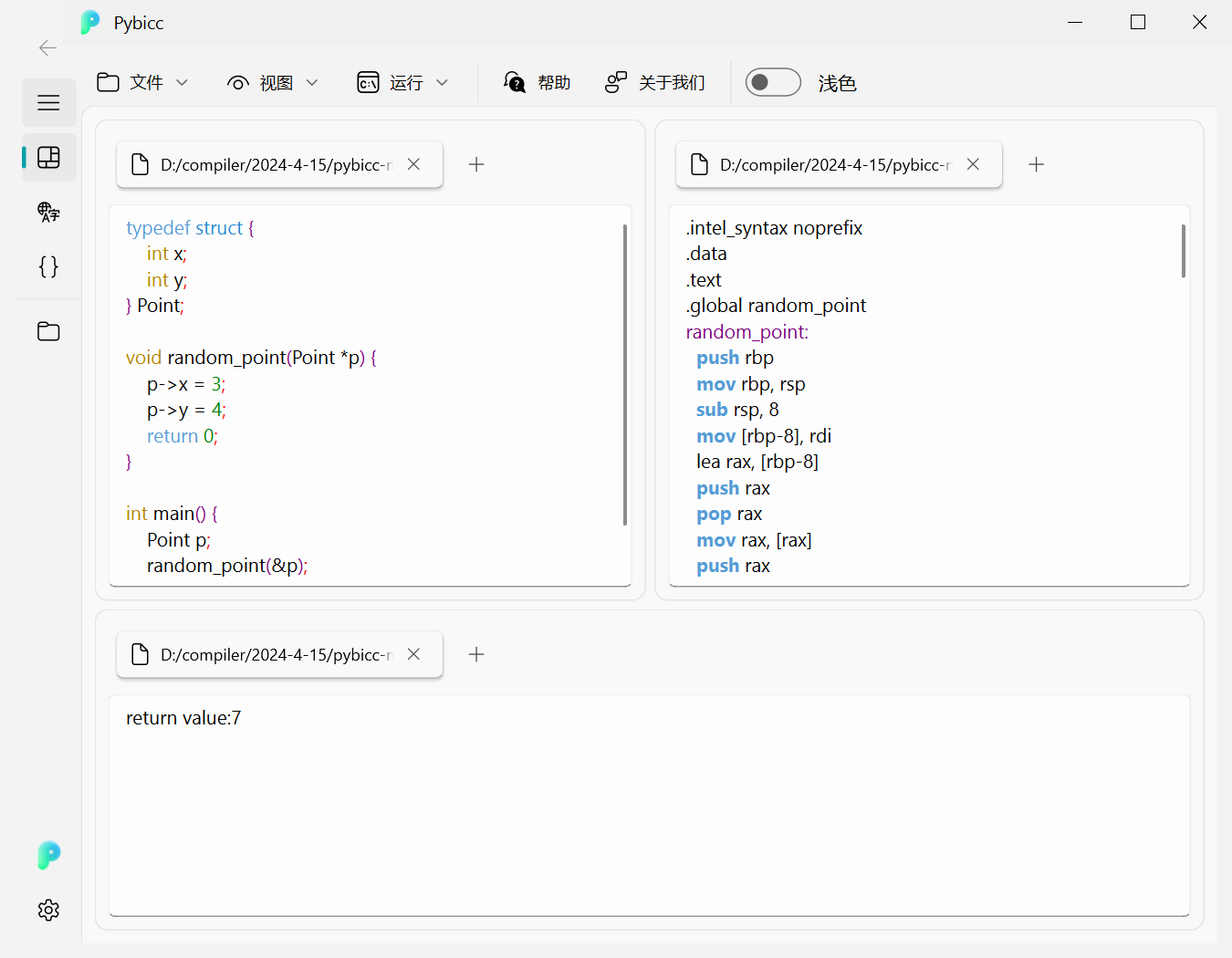


图23 编译运行

3、在编译的同时，可以在导航栏点击词法分析与语法分析，查看中间步骤

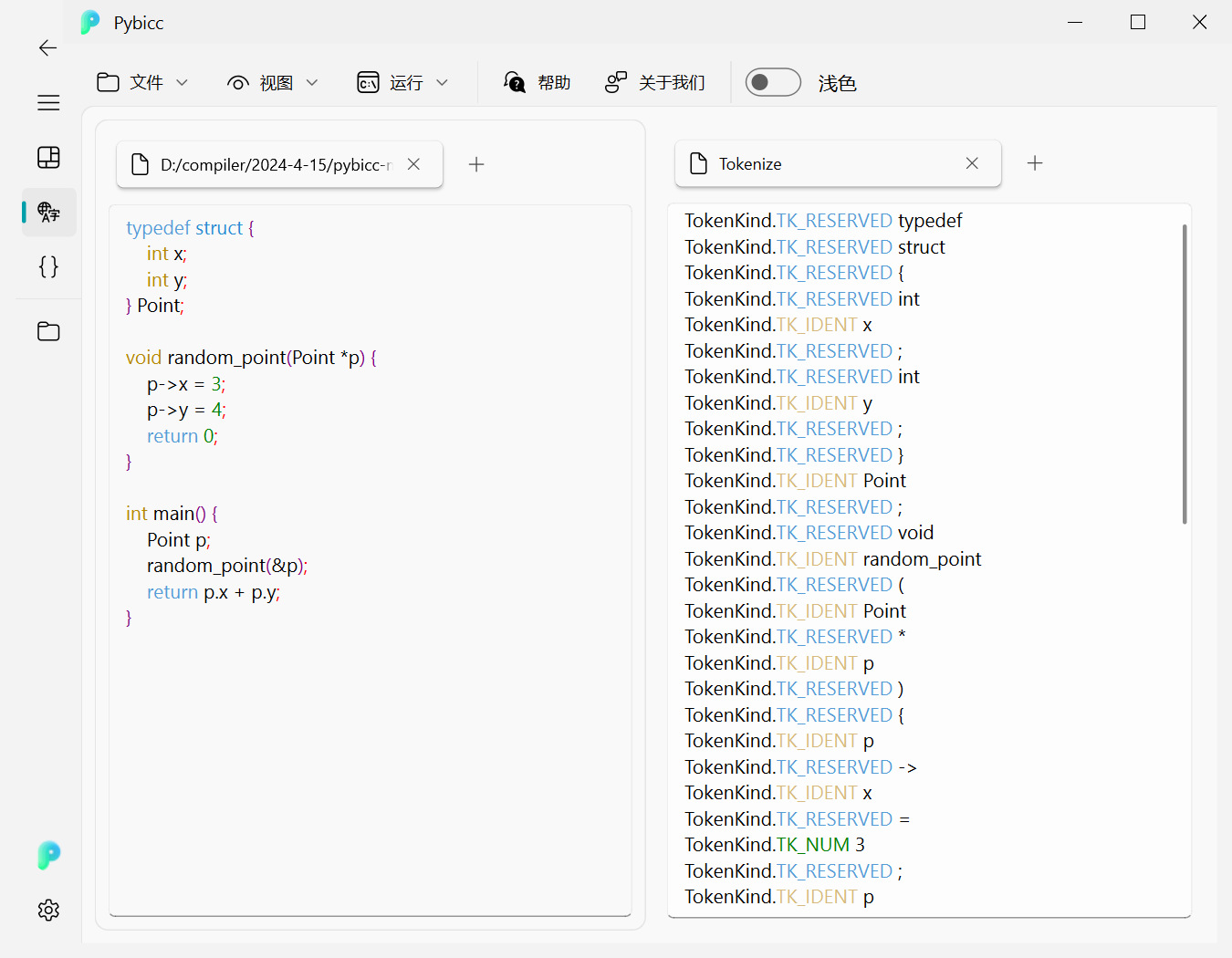


图24 查看词法分析

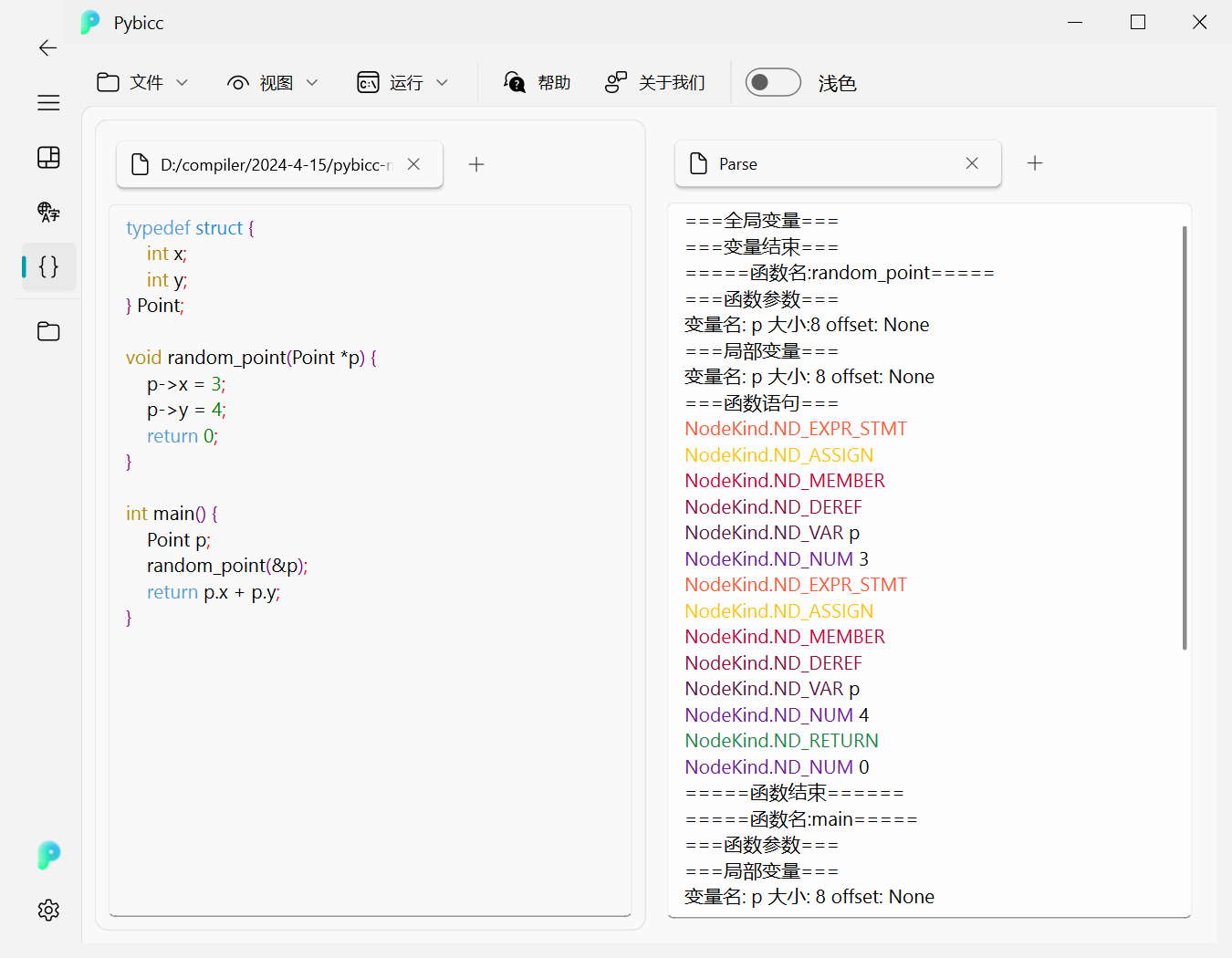


图25查看语法分析

## 测试结果3

1. 运行Pybicc，打开rsa.c文件

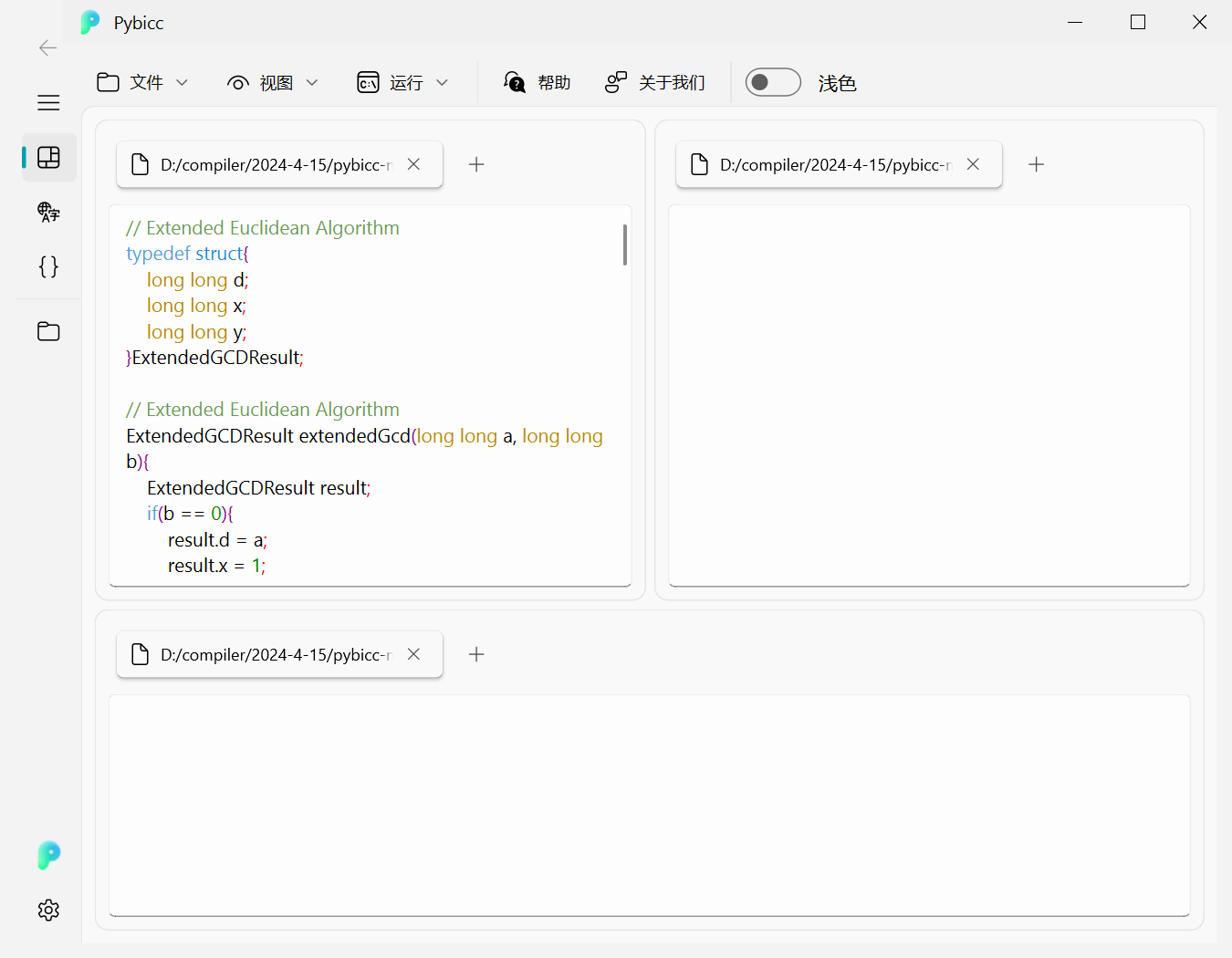


图26 打开文件

1. 选择菜单 运行->编译并运行

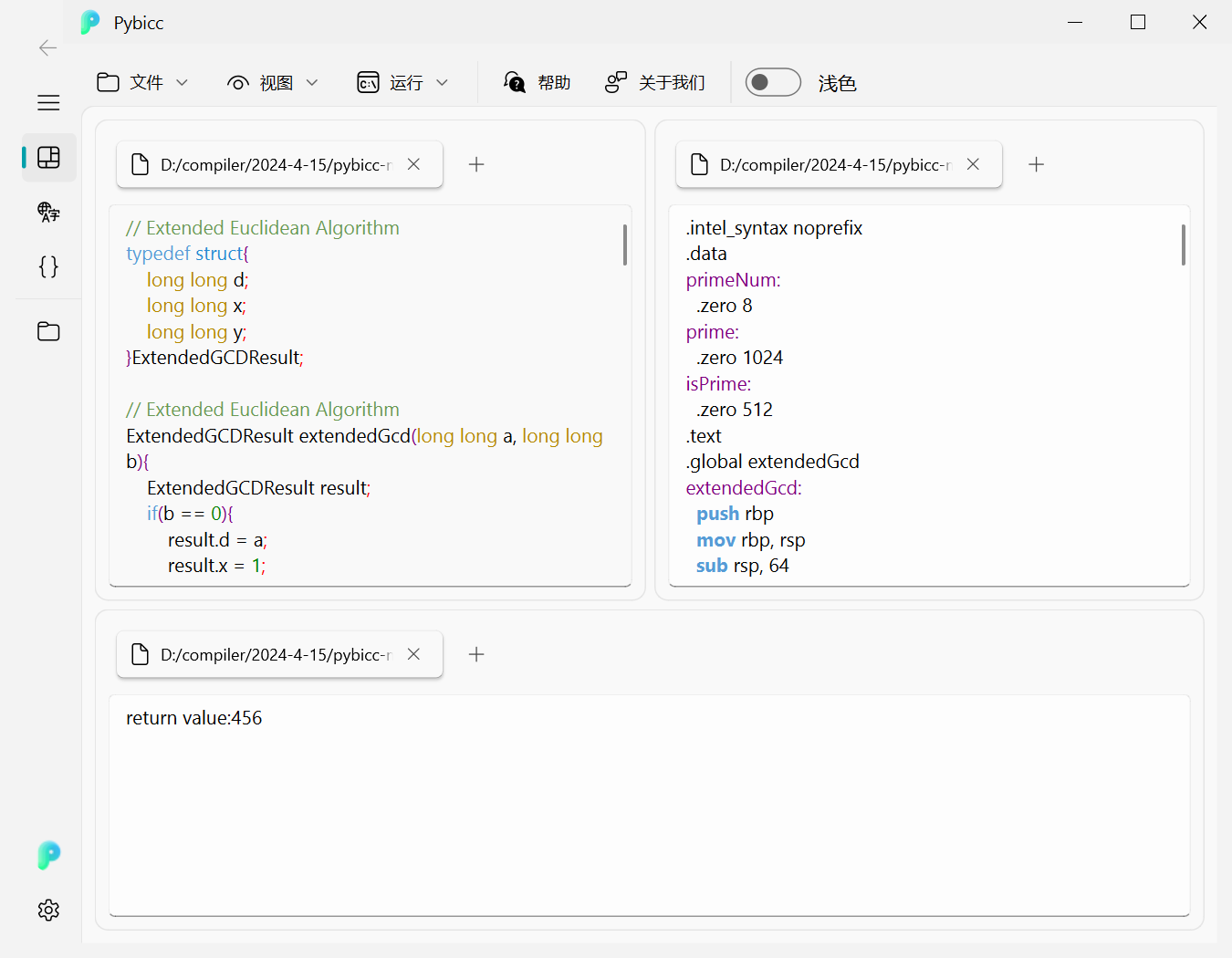


图27 编译运行

1. 在编译的同时，可以在导航栏点击词法分析与语法分析，查看中间步骤

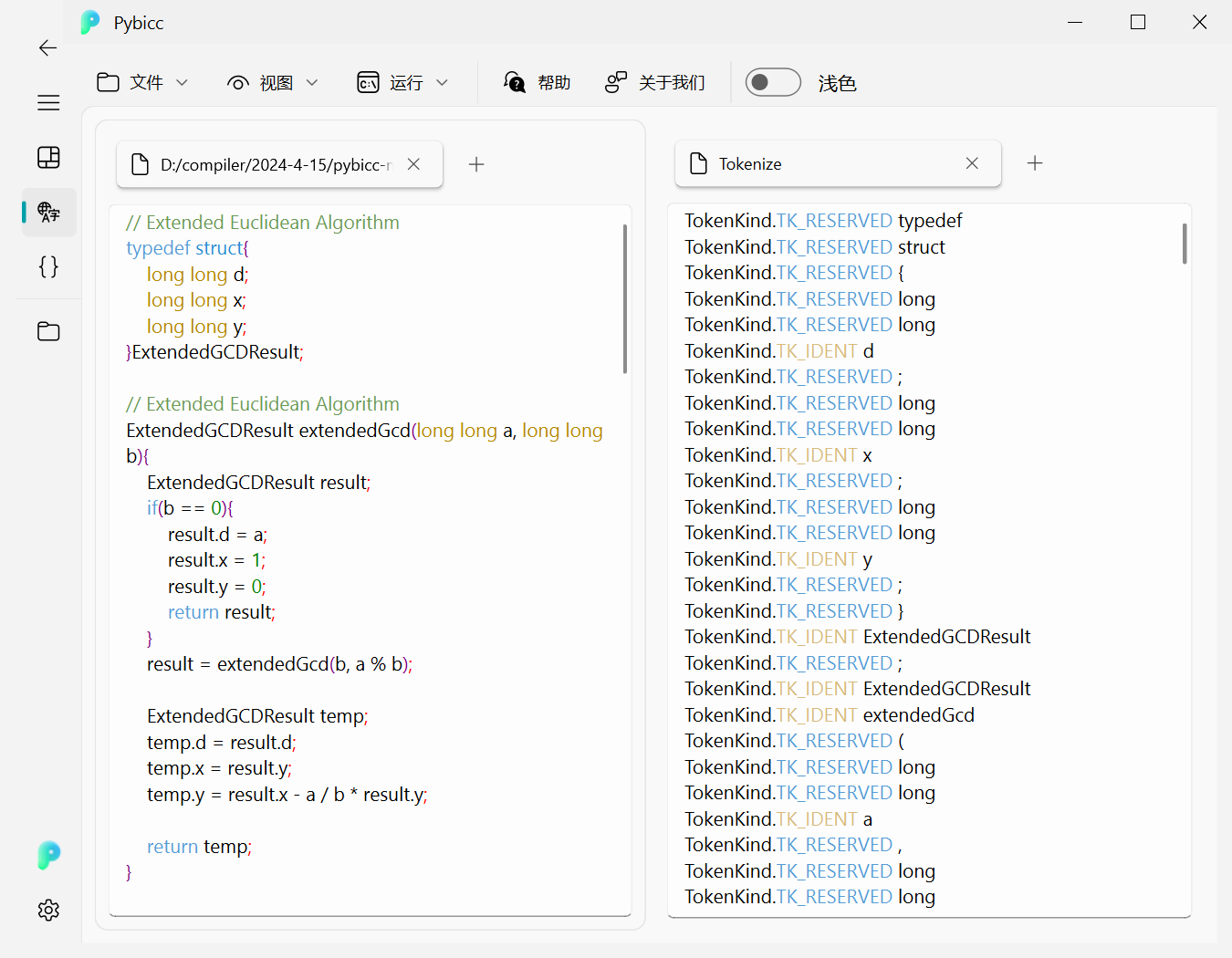


图28 查看词法分析

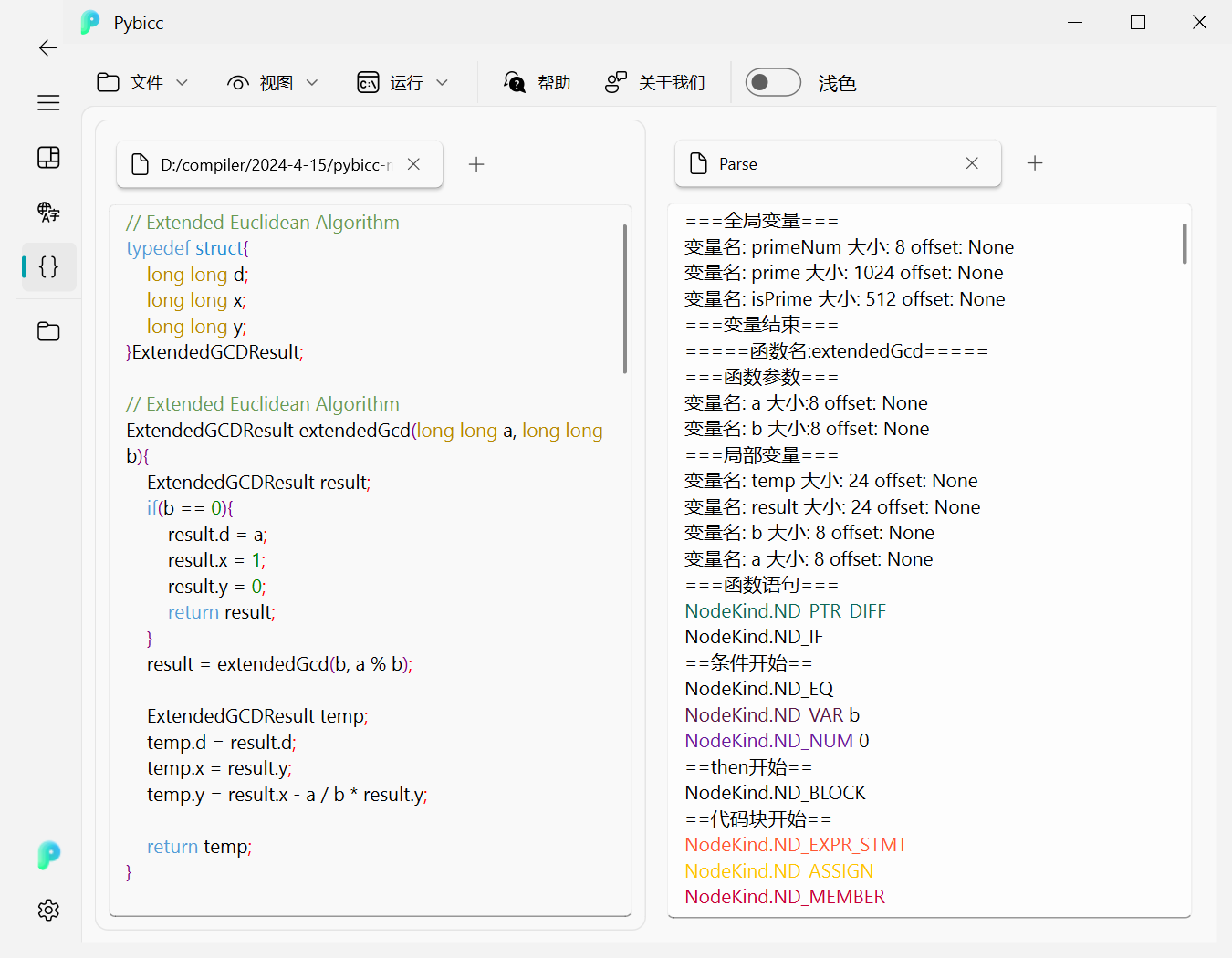


图29查看语法分析

## 测试结果4

1. 运行Pybicc，打开ptr.c文件

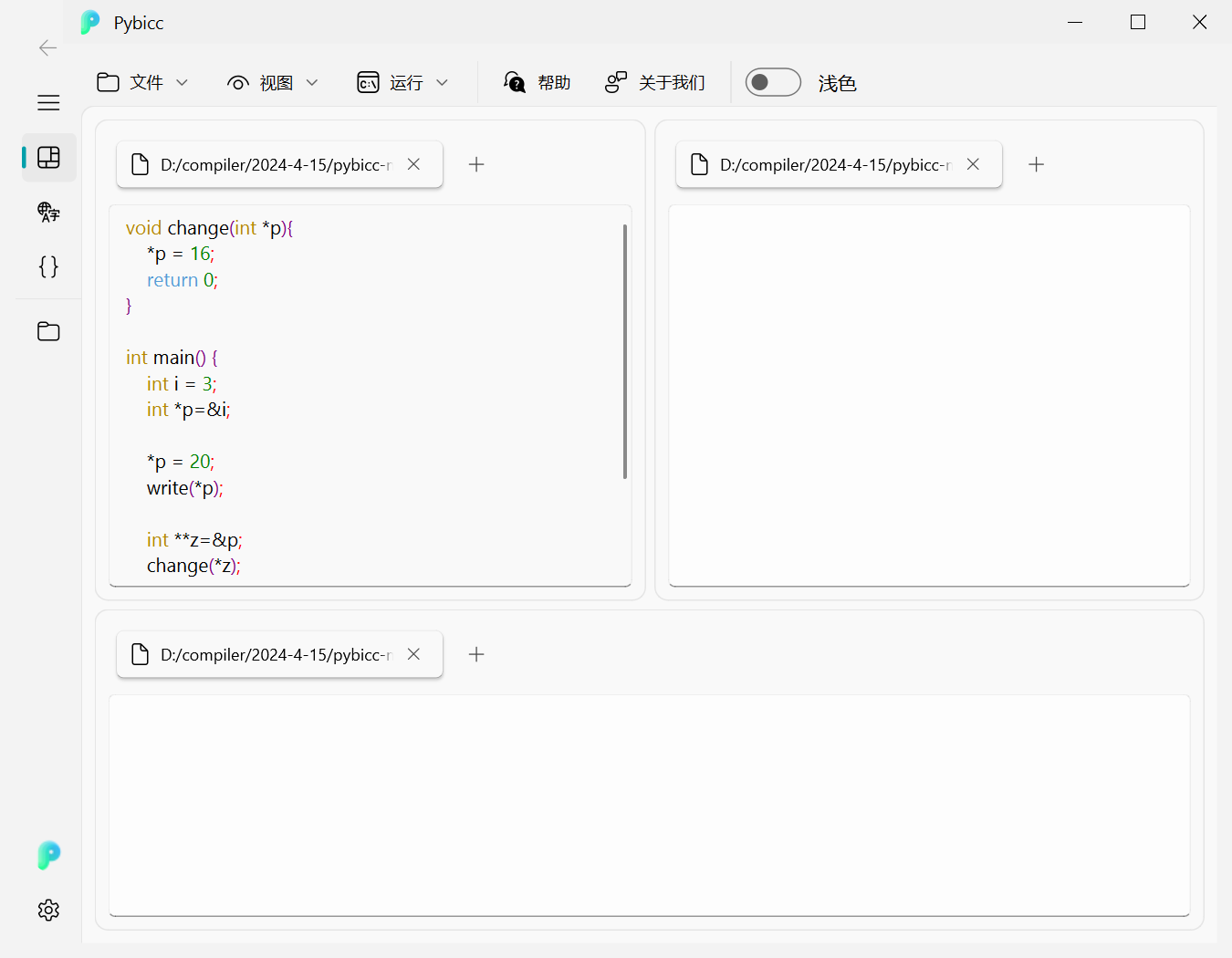


图30 打开文件

1. 选择菜单 运行->编译并运行

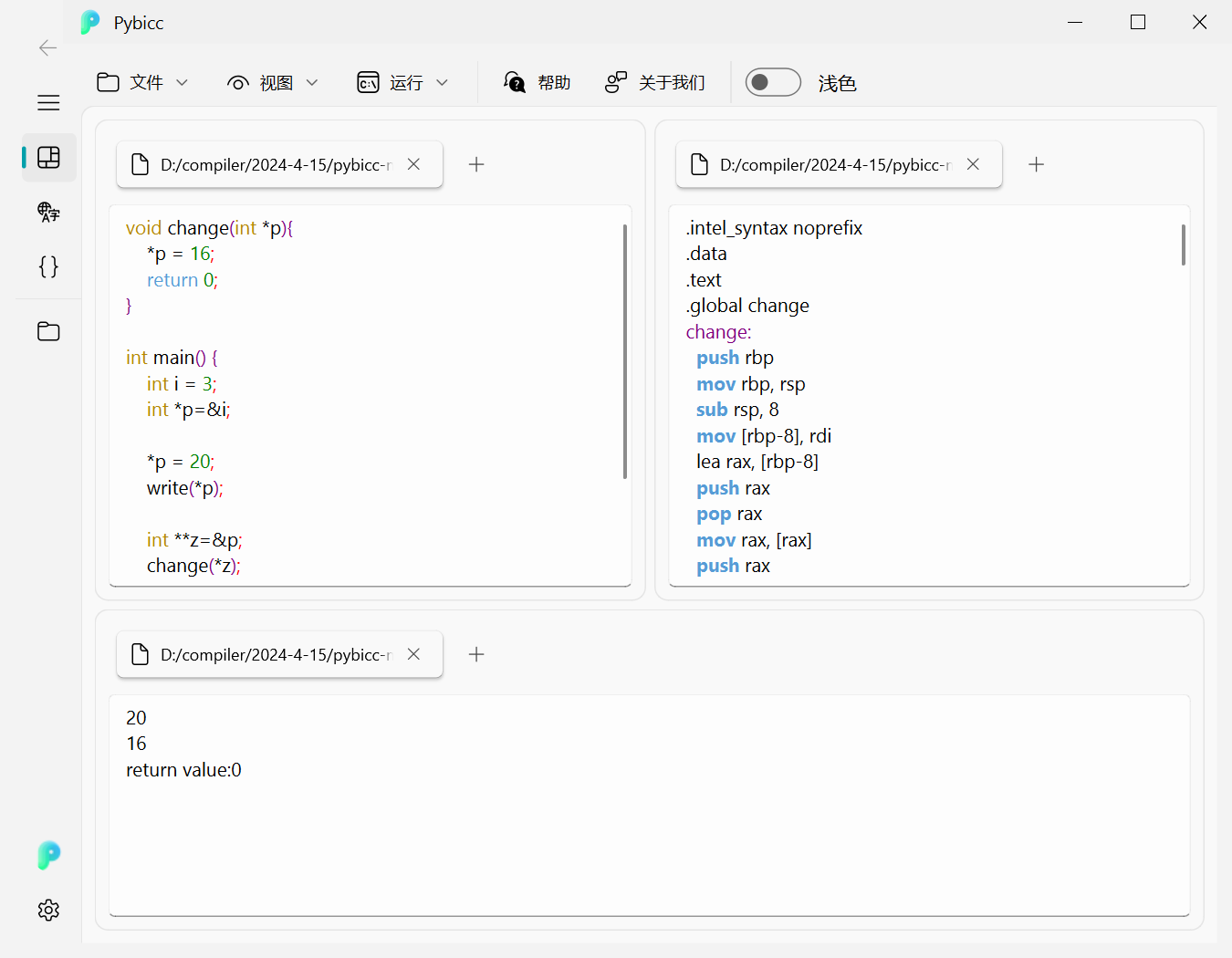


图31 编译运行

1. 在编译的同时，可以在导航栏点击词法分析与语法分析，查看中间步骤

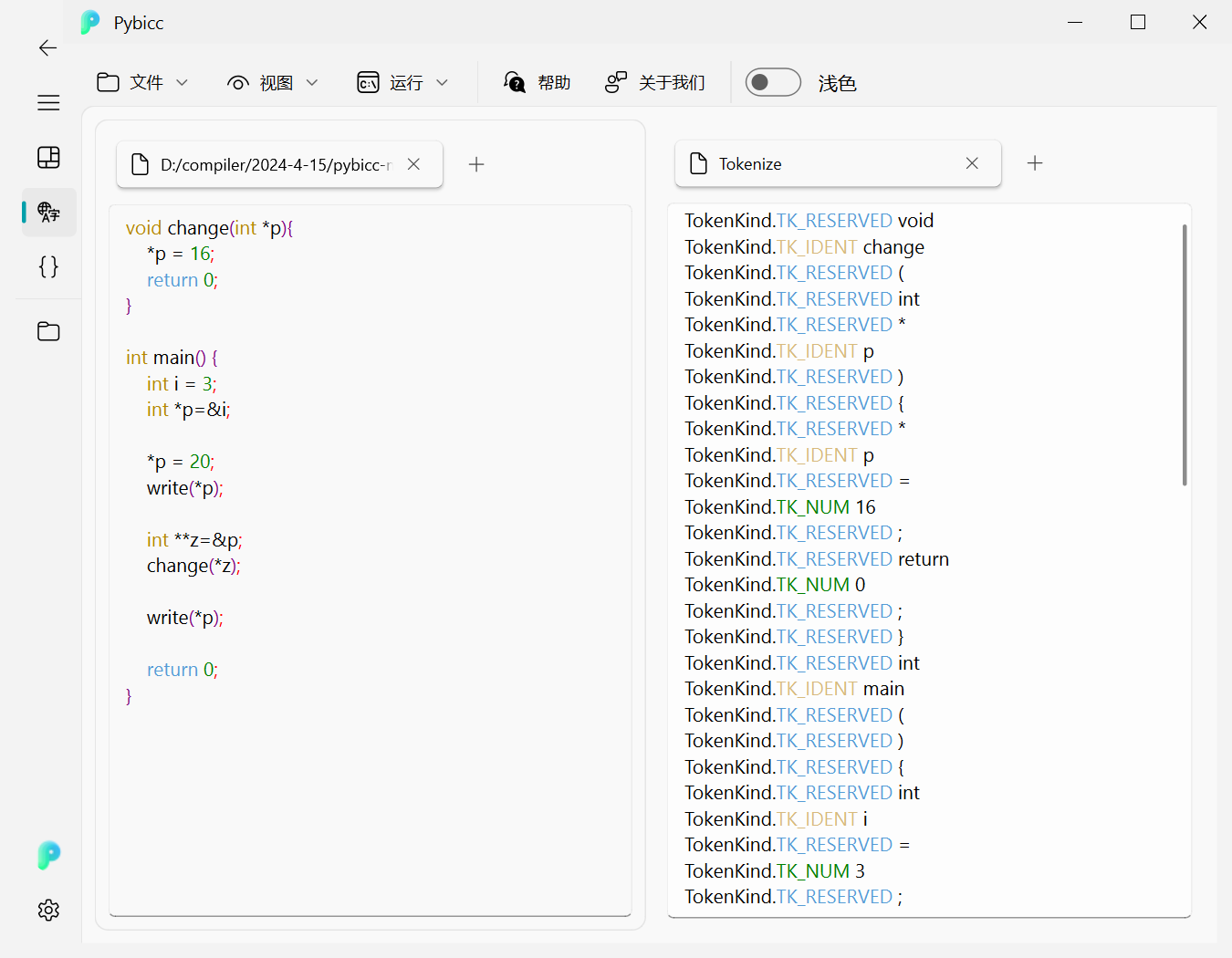


图32 查看词法分析

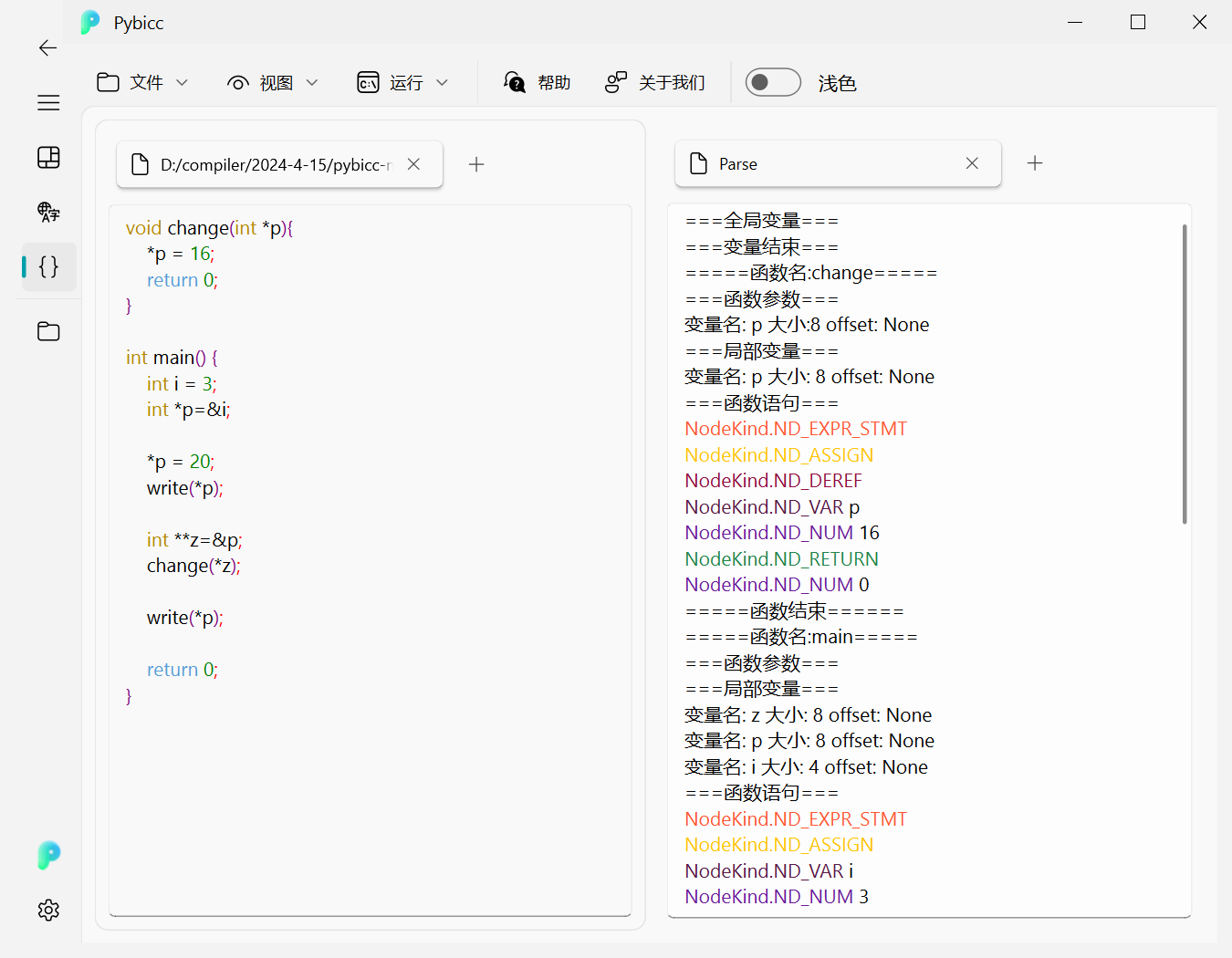


图33查看语法分析

## 测试结果5

1. 运行Pybicc，打开fibonacci.c文件

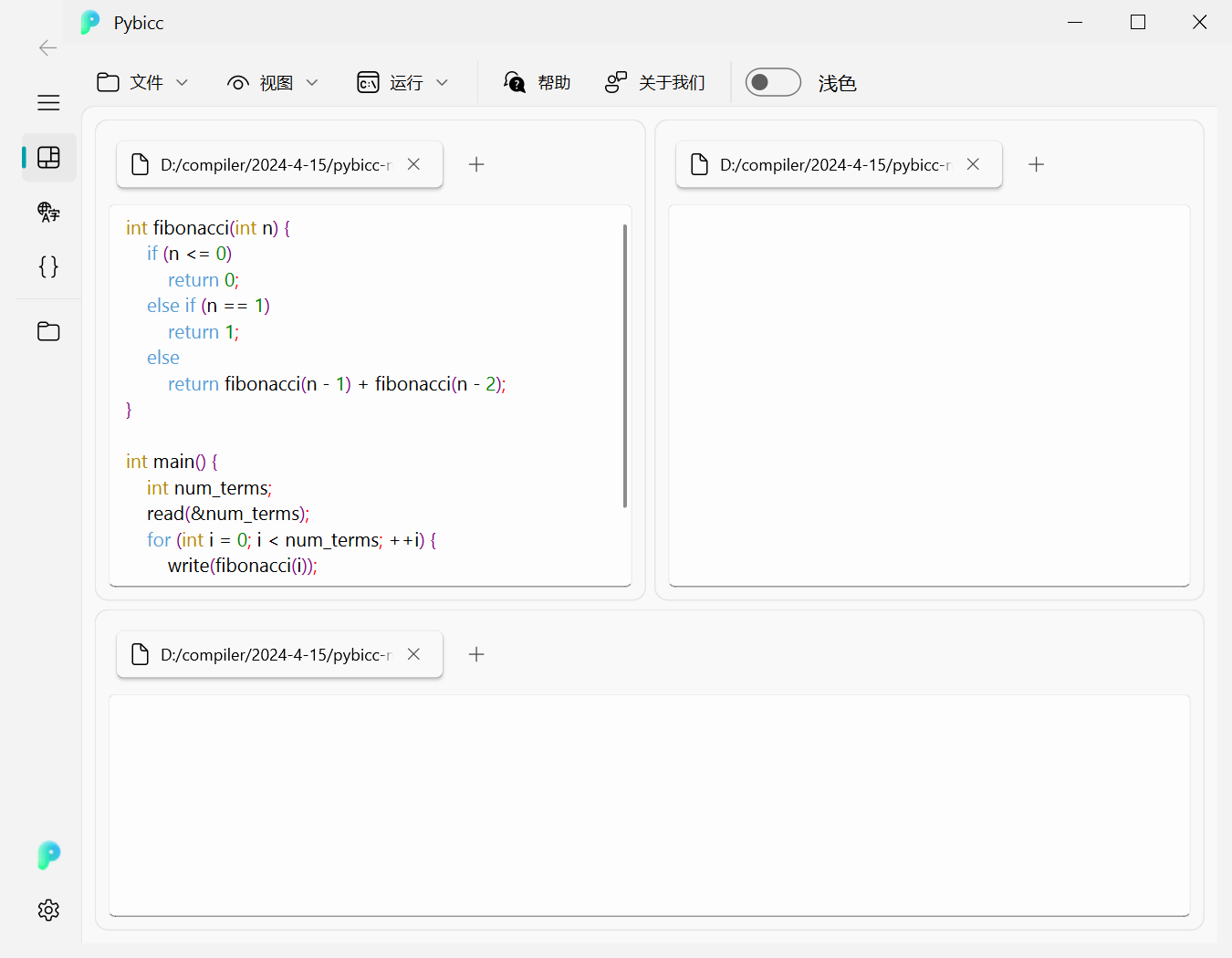


图34打开文件

1. 选择菜单 运行->编译并运行，先输入n=8

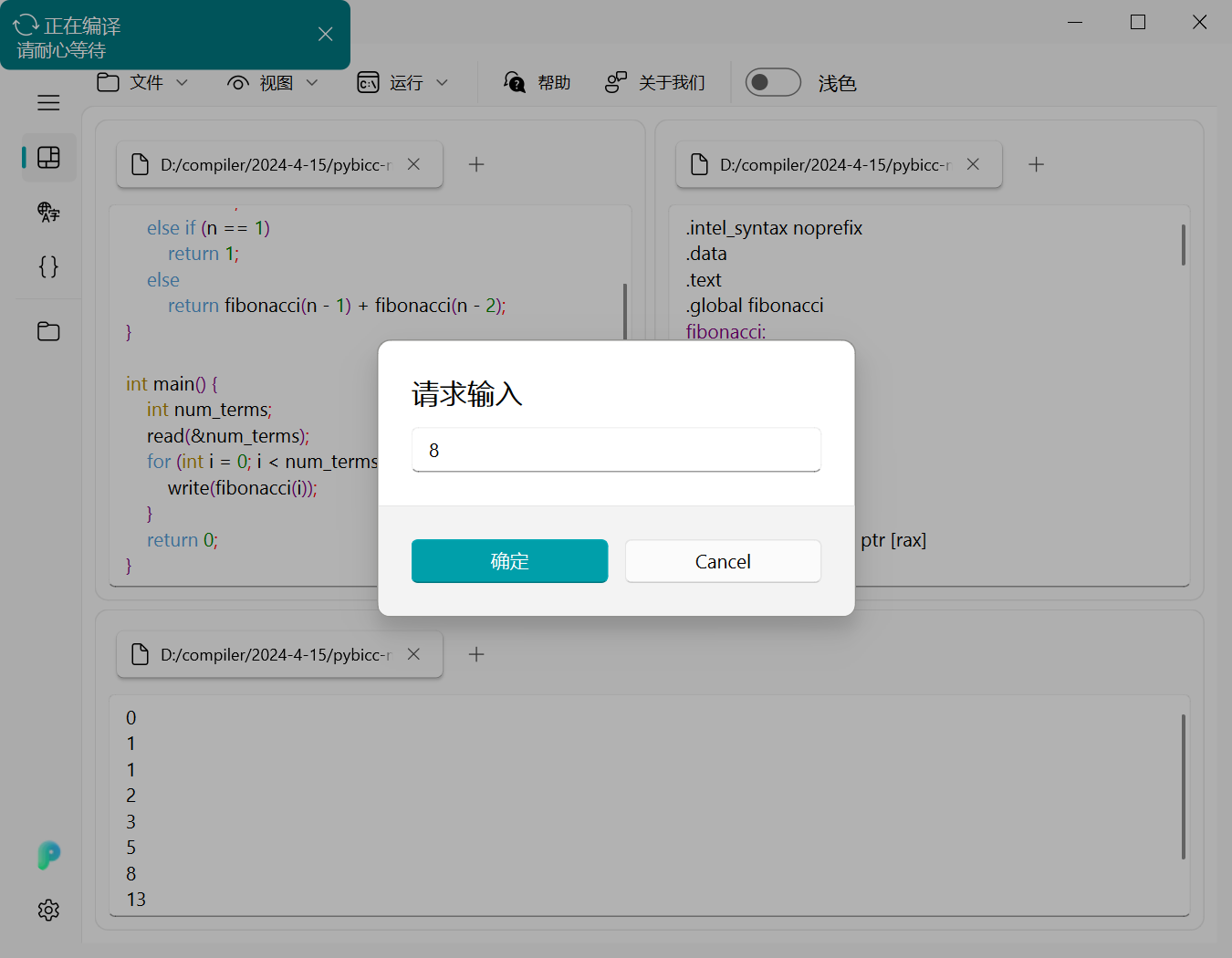


图35 输入n

1. 编译运行结果

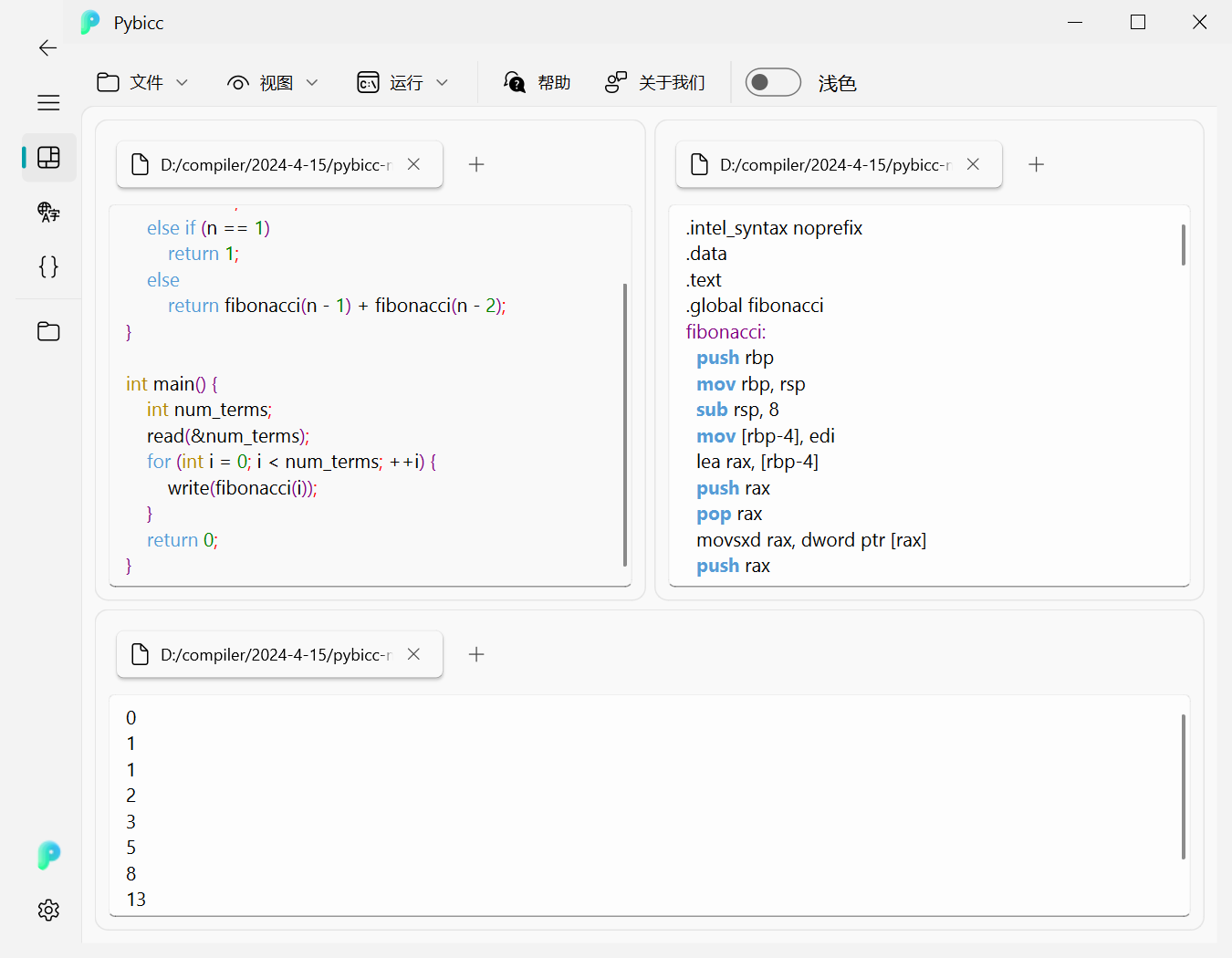


图36 编译运行

1. 在编译的同时，可以在导航栏点击词法分析与语法分析，查看中间步骤

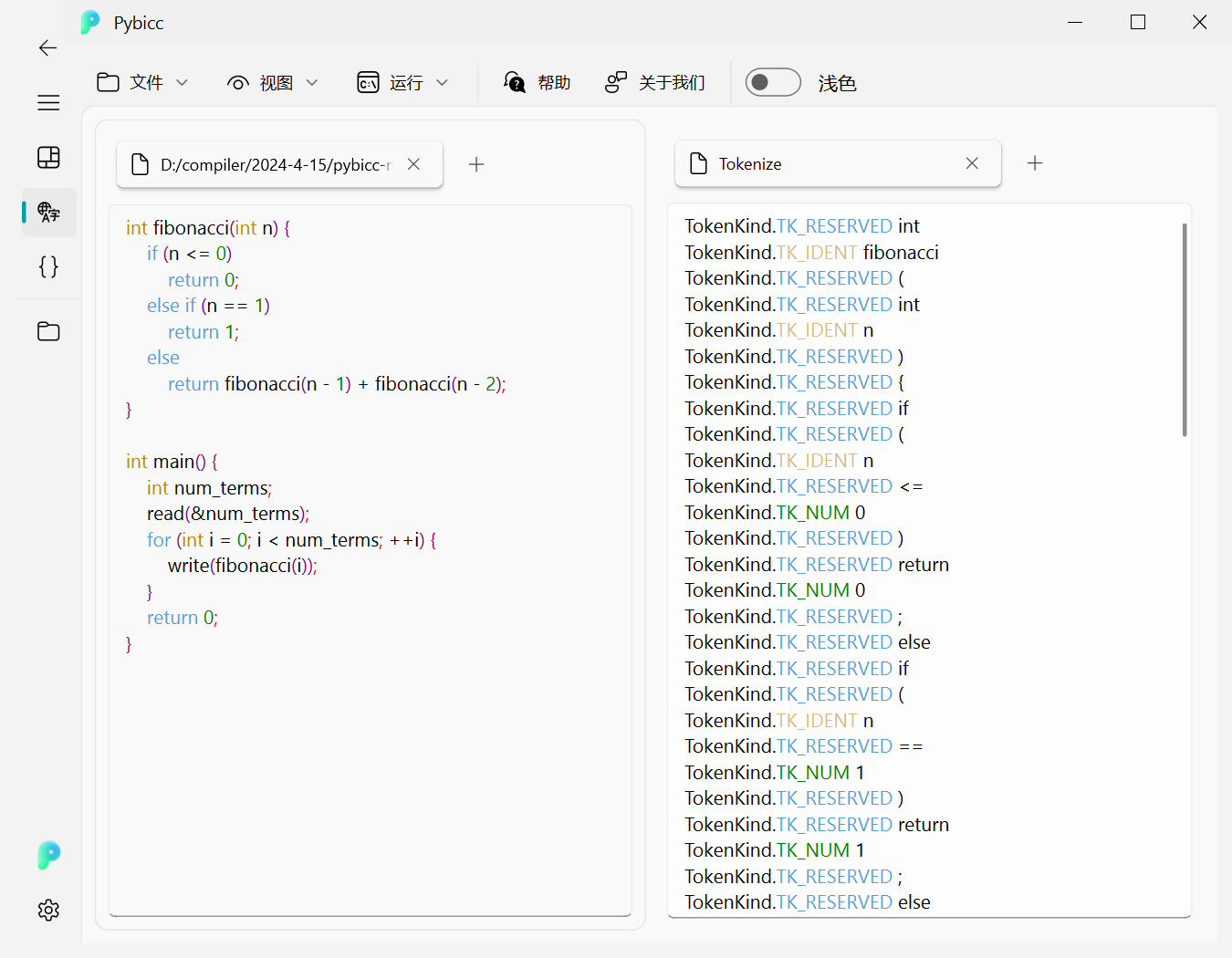


图37 查看词法分析

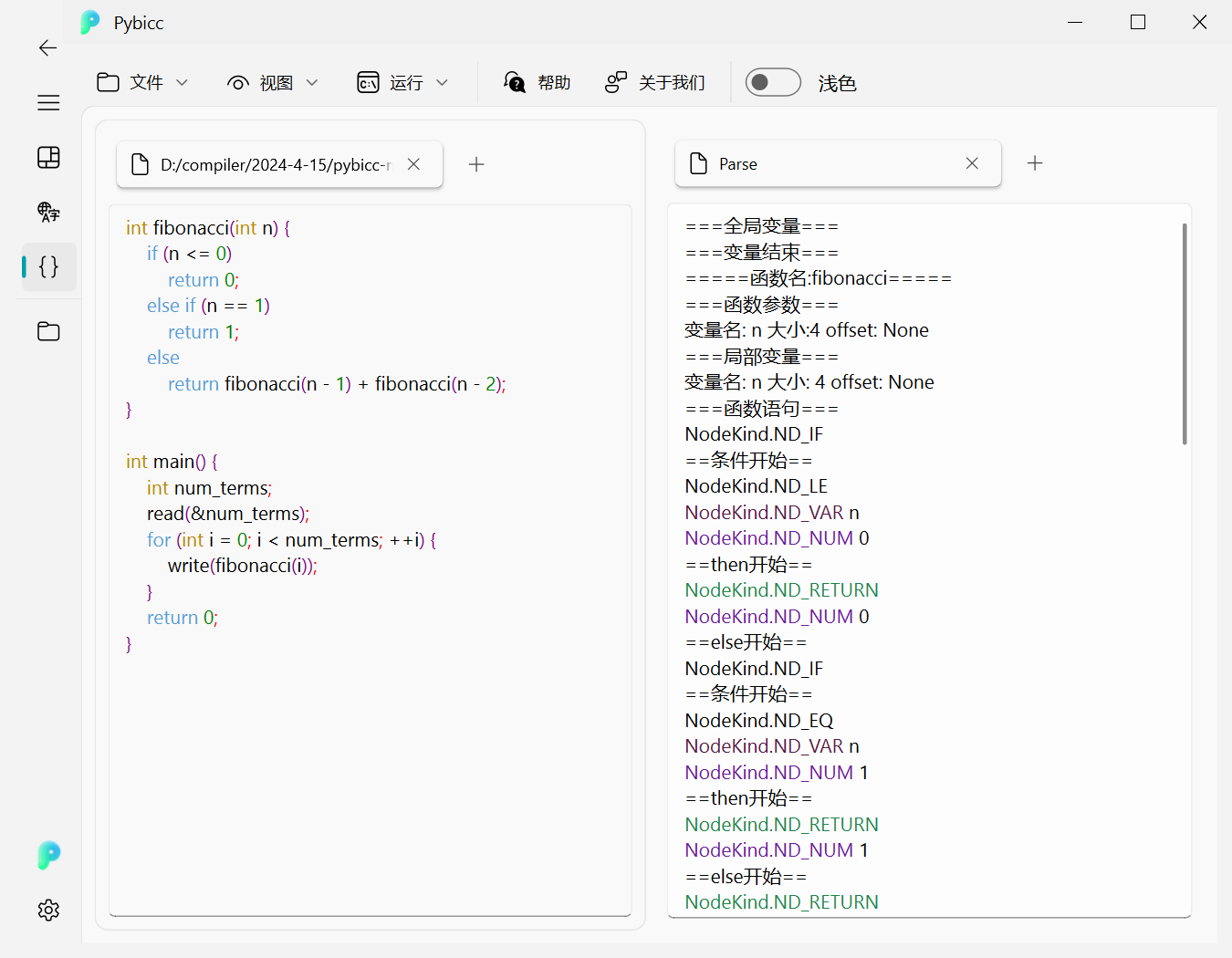


图38 查看语法分析

# 心得体会

随着本学期编译原理课程设计的接近尾声，我们可以回顾这段时间的学习经历，尽管课程时间紧张，每周仅两次课，但我们获益良多。

遵循老师的教学要求，我们在学习了文法知识后立即编写了所有相关的文法规则，并在掌握语句分析后完成了对应的分析部分。然而，我们最初面临的问题是不完全理解编译器的构建原理。经过深入查阅资料并进行小组讨论，我们逐渐明白了基本原理并确立了我们的目标功能和初步规划。

在之前使用Pycharm的过程中，我们深感其功能之强大，这也是我们羡慕的地方。虽然我们难以达到同等的完美和功能性，但我们尽力实现了一些常用功能，并根据时间安排进行了适当的调整。

项目的起步阶段是最困难的，一旦开始后，操作变得流畅许多。当然，项目进程中偶尔也会遇到难题，这些难题消耗了我们大量的时间去寻找解决方案，以便最终实现。

总结这个学期的课程设计，我们学到了许多知识。项目初期常常遇到问题，但借助发达的网络资源和图书馆的藏书，我们最终找到了解决方案。解决问题的过程不仅仅是解决单一问题，而是在这一过程中，我们从多个角度充实和提升了自己。

通过这次课程设计，我们将上学期学到的知识应用到了实际操作中。以往的学习往往仅停留在理论层面，尤其是那些抽象的概念，难以应用到实际中。本次课程设计很好地解决了这一问题。尽管将学习与实际应用相结合具有一定难度，但一旦克服，其好处显著。面对难题时，我们会查阅相关资料或者提前自学未涉及的内容。通过结合这些知识并运用相应的原理和方法，我们能够实现预定的结果。因此，在提升动手能力的同时，这次课程设计也巩固了我们之前学到的理论知识。

我们认为学习的一个高境界是能将理论知识应用于实践。学会并能够运用知识才是关键，我们的学习目的就是为了应用。将科学理论转化为实际技术，编译原理课程设计为我们提供了这样的机会。例如，尽管我们已学习文法和语句分析，但如何用程序实现这些理论，特别是自动化的实现方式，对我们来说仍然具有一定难度，但这正是将学到的知识应用于实践的绝佳机会。

尽管过程中遇到问题在所难免，但我们坚信通过不懈努力，一切问题都将迎刃而解。总之，通过这次编译原理课程设计，我不仅学到了很多，也收获颇丰。

**教 师 评 语 专 栏**

|  |  |  |  |
| --- | --- | --- | --- |
| **任**  **课**  **教**  **师**  **评**  **语** |  | | |
| **成**  **绩** |  | **任课教师：**  **年 月 日** |