《编译原理》课程设计
实验报告书

班　　级：1622302

组　　长：陈梓鹏

组　　员：李　萌

组　　员：杨雨涵

组　　员：吴静柔

指导老师：杨志斌

课设地点：计算机学院楼 105

课设时间：2024 年 11 月至 2025 年 1 月

2025 年　1 月　8　日

目录

1. 小组成员及分工

1.1. 小组成员

162230217 陈梓鹏,162230203 李萌,162230205 杨雨涵,162230201 吴静柔.

1.2. 课程设计分工

姓名	分工
陈梓鹏	语法制导翻译、界面工具、错误单元处理、文档撰写
李　萌	解释器、目标代码生成、错误单元处理、文档撰写
杨雨涵	界面工具、语法分析、错误单元处理、文档撰写与排版
吴静柔	词法分析、错误单元处理、文档撰写

2. PL/0 语言的语法图描述

PL/0 语言的 BNF 描述（扩充的巴克斯范式表示法）：

<prog> → program <id>；<block>

<block> → [<condecl>][<vardecl>][<proc>]<body>

<condecl> → const <const>{,<const>};

<const> → <id>:=<integer>

<vardecl> → var <id>{,<id>};

<proc> → procedure <id>（[<id>{,<id>}]）;<block>{;<proc>}

<body> → begin <statement>{;<statement>}end

<statement> → <id> := <exp>

|if <lexp> then <statement>[else <statement>]

|while <lexp> do <statement>

|call <id>（[<exp>{,<exp>}]）

|<body>

|read (<id>{，<id>})

|write (<exp>{,<exp>})

<lexp> → <exp> <lop> <exp>|odd <exp>

<exp> → [+|-]<term>{<aop><term>}

<term> → <factor>{<mop><factor>}

<factor>→<id>|<integer>|(<exp>)

<lop> → =|<>|<|<=|>|>=

<aop> → +|-

<mop> → *|/

<id> → l{l|d}　（注：l 表示字母）

<integer> → d{d}

注释：

<prog>：程序 ；<block>：块、程序体 ；<condecl>：常量说明 ；<const>：常量；<vardecl>：变量说明 ；<proc>：分程序 ； <body>：复合语句 ；<statement>：语句；<exp>：表达式 ；<lexp>：条件 ；<term>：项 ； <factor>：因子 ；<aop>：加法运算符；<mop>：乘法运算符； <lop>：关系运算符。

故其语法图描述如下：

图 2.1 prog 语法图

图 2.2 block 语法图

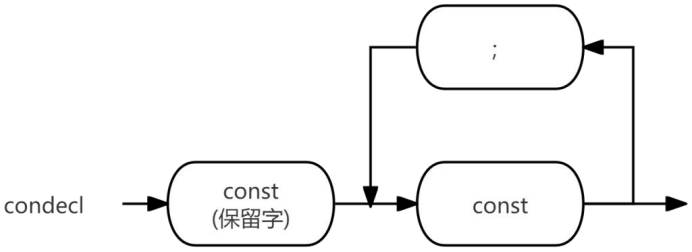


图 2.3 condecl 语法图

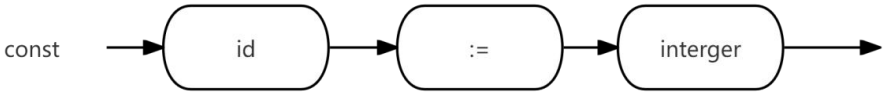


图 2.4 const 语法图

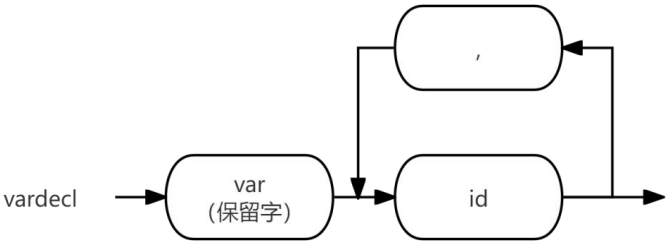


图 2.5 vardecl 语法图

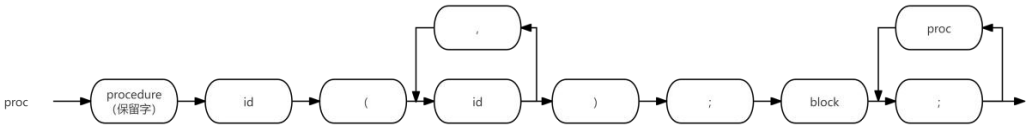


图 2.6 proc 语法图

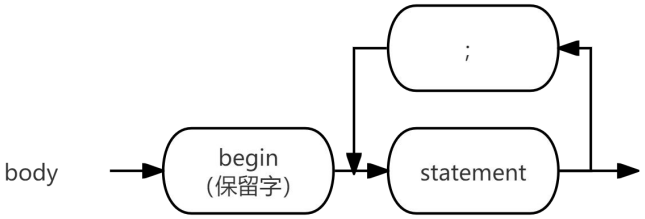


图 2.7 body 语法图

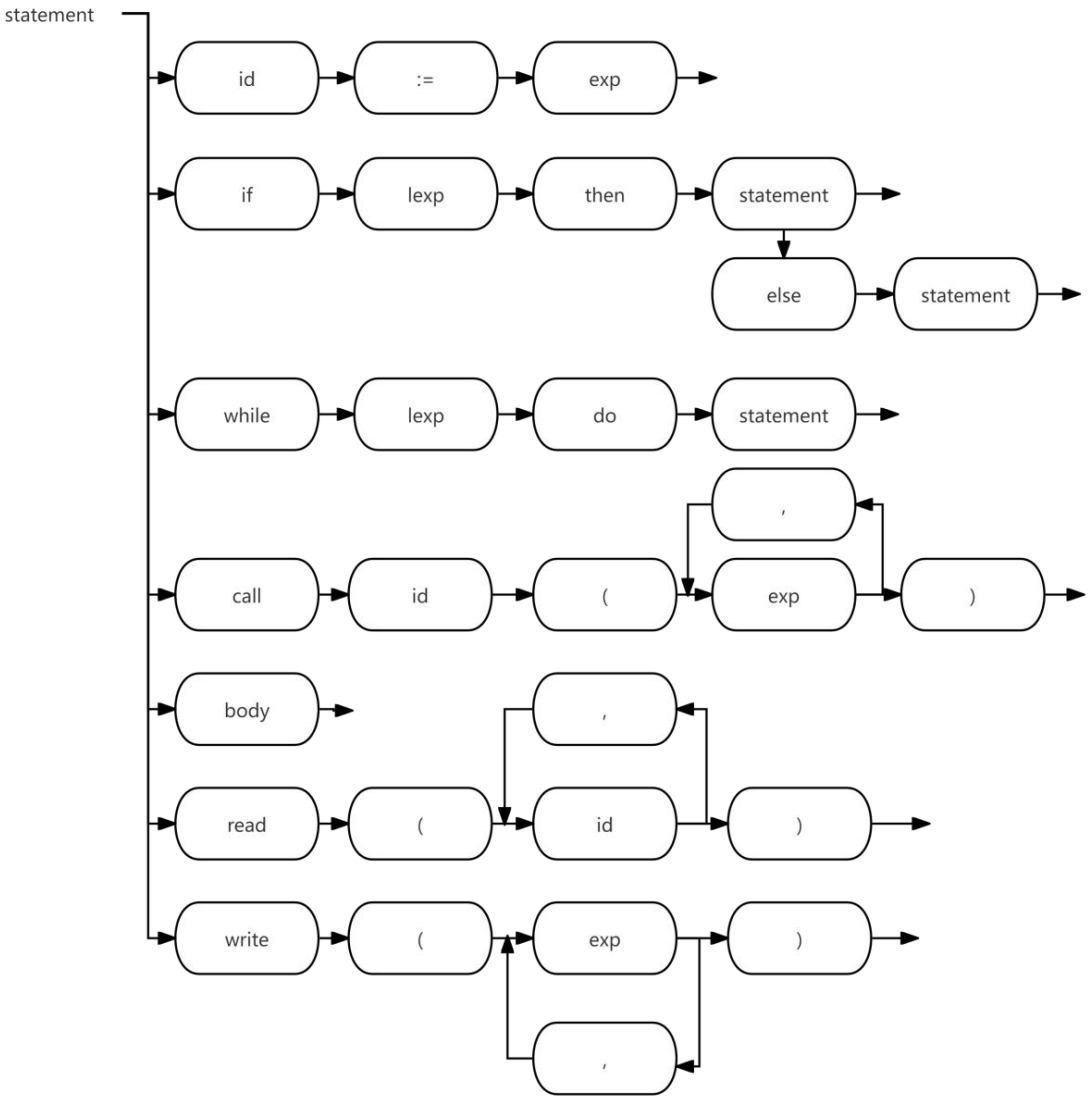


图 2.8 statement 语法图

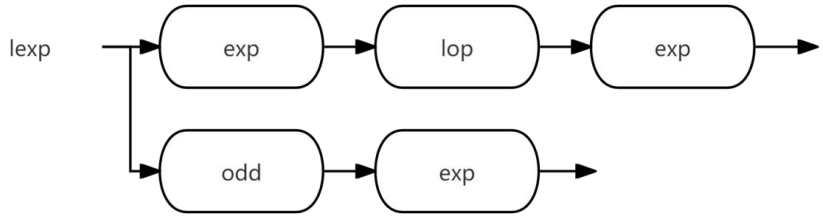


图 2.9 lexp 语法图

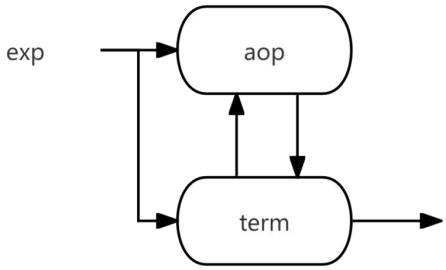


图 2.10 exp 语法图

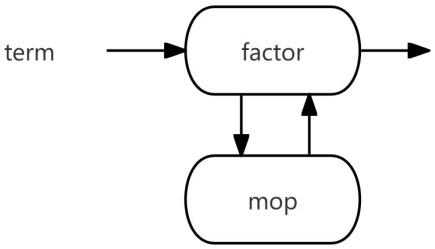


图 2.11 term 语法图

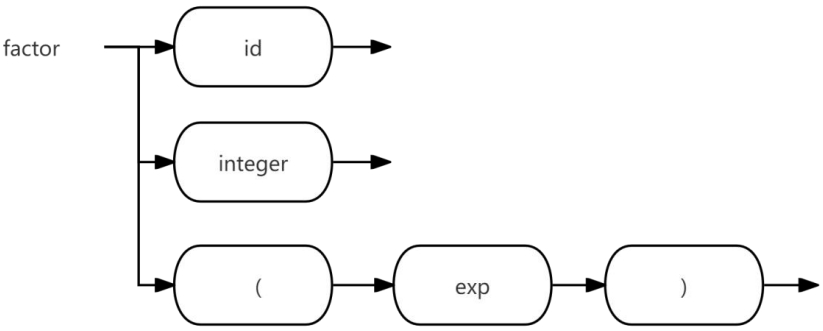


图 2.12 factor 语法图

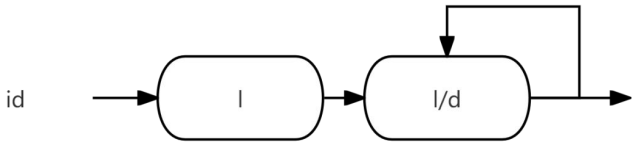


图 2.13 id 语法图

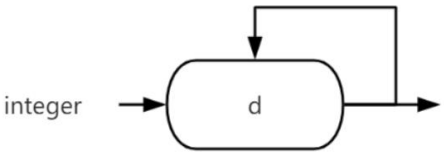


图 2.14 integer 语法图

3. 系统设计

3.1. 系统的总体架构

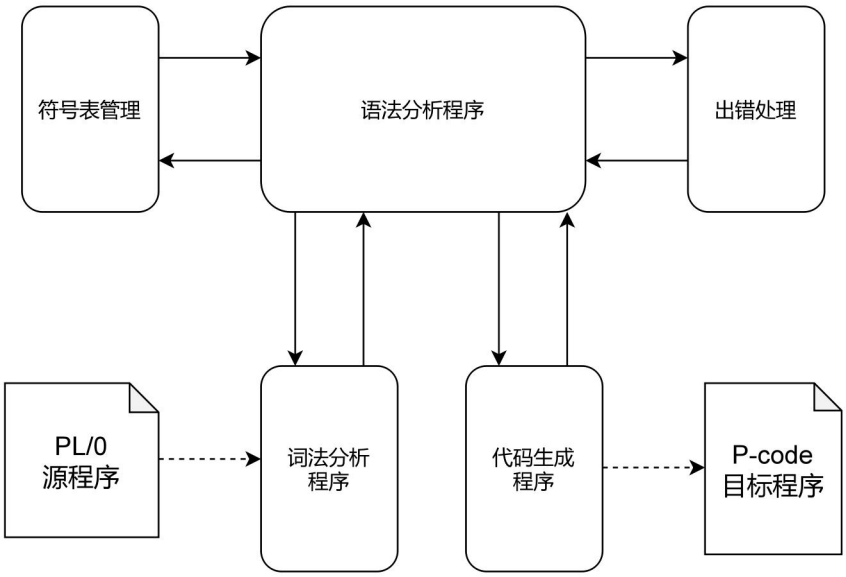


图 3.1 系统总体框架

编译器系统的总体结构如图 3.1 所示。箭头方向代表数据的流向。下面将分别介绍各模块的设计。

3.2. 主要功能模块的设计

3.2.1. 符号表

基本介绍：

符号表（Symbol Table）是编译器中一个重要的数据结构，用于存储程序中使用的各种符号（如变量、函数、常量等）及其相关信息。它在编译过程中起着至关重要的作用，特别是在语法分析、语义分析和代码生成及运行时分配内存。符号表初始化类图如 3.2 所示。

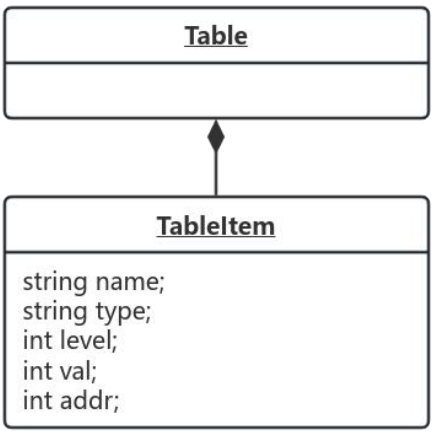


图 3.2 符号表初始化类图

常量 item 有三个 key，分别是:name（记录常量的符号名）、type（类型）、value（常量的值）；变量 item 有四个 key，分别是:name(记录变量的符号名)、type（类型）、level（当前层次）、addr（当前变量的地址）；过程 item 有四个 key，分别是：name（记录过程的符号名）、type（类型）、level（当前层次）、addr（程序地址）。

主要功能：

1) 静态语义检查：

上下文无关文法由于没有记忆性，无法检查出符号是否声明、符号是否重定义、形参与实参列表是否匹配的问题。此外，由于过程的嵌套定义产生了作用域的概念，相同名字的符号在不同的作用域也有不同的含义，内层作用域可以使用外层的符号，为了解决这些问题，需要借助符号表的帮助。

2) 运行时内存分配：

维护了一个全局 space，用来记录变量在子过程中的相对地址，以及整个子过程所占用的内存空间大小。在登入变量符号时将当前 offset 作为符号的属性同时登入，并令 space+=1;在过程的声明部分结束后，将当前 space 作为当前过程的所占空间的属性记入符号表，然后将 space 置 0，用于下一个子过程。当编译运行生成到内存分配相关的指令时，会依据符号表的 space 属性进行生成。

3) 目标代码生成

除了上述的内存分配任务需要符号表外，还有以下情况目标代码生成需要：

• 右值替换：使用常量类型的变量为右值，不会在运行时刻分配内存地址，而是使用符号表中记录的值直接替换。

• 跨子程序回填：由于不同子程序之间的信息不能共享，对于部分跨子过程的回填操作，需要借助符号表的帮助，例如到主程序体的跳转指令的回填。在生成需要被回填的跳转指令时，我会将当前跳转指令的地址与当前过程的符号的 addr 属性绑定，遇到当前过程的过程体的第一条目标代码语句时，便将该语句的地址回填至所在过程符号绑定的 addr 指令处。

3.2.2. 词法分析器

基本介绍：

词法分析器是一个子程序，接受从源代码文件的字符串，依据图 3.2 所示的状态转换图，从头至尾分析字符串中出现的所有同步符号，每分析出一个词法单元便提供给语法分析器使用，循环往复直到程序末尾。

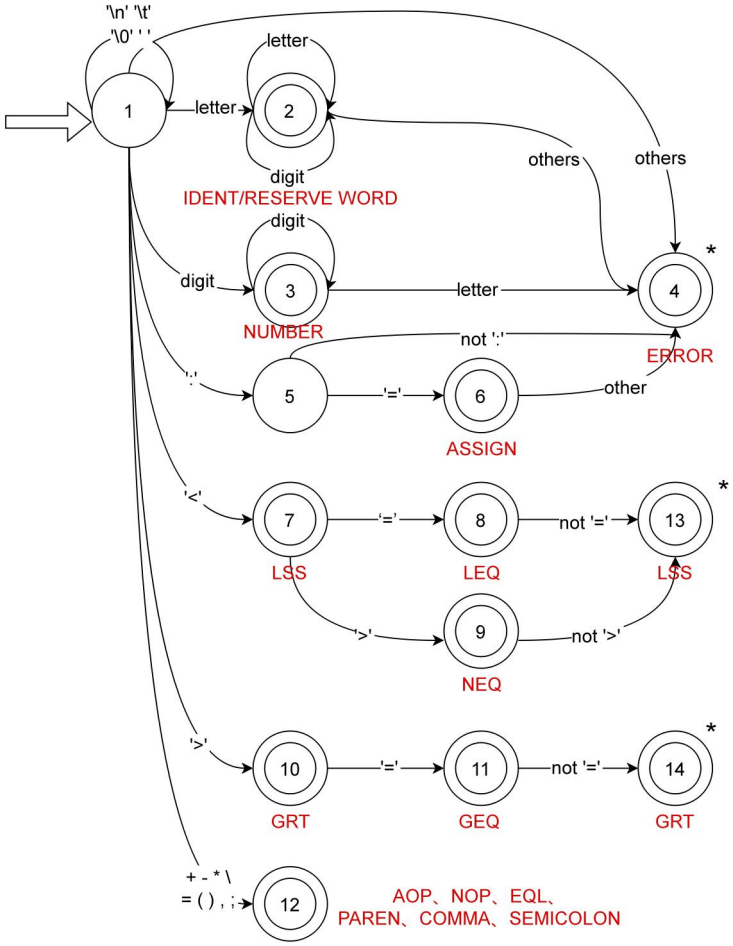


图 3.3 词法分析器状态转换图

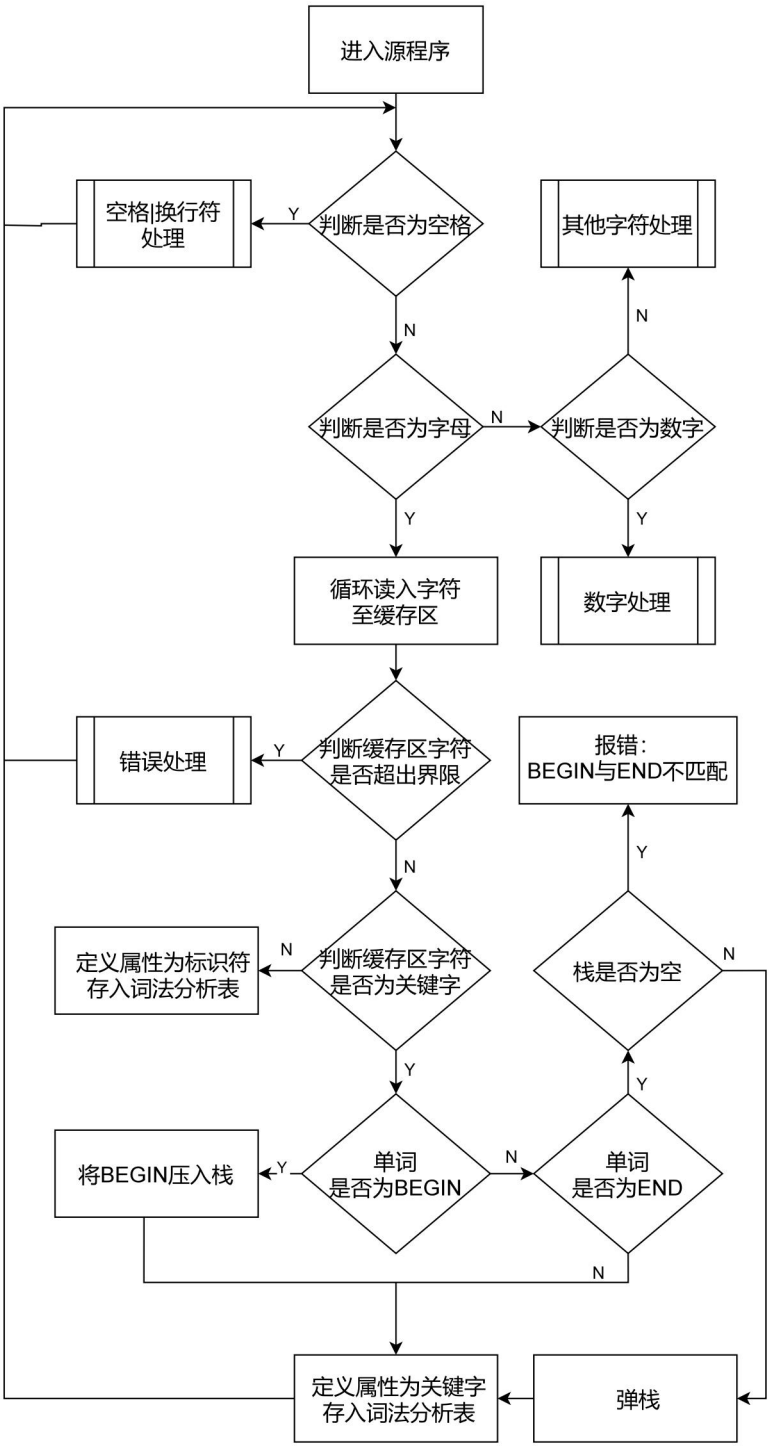
主要功能：

1) 实现同步符号的读取与识别：

为保证“当语法分析需要读单词时就调用词法分析程序”，当语法分析需要时，词法分析器读取程序字符串中一个词，依照图 3.2 所示的状态转换图进行分析，直到程序结束。由于标识符和关键字可能会有重复的部分，想要识别出正确的符号，必须进行“超前搜索”，超前搜索到界符，才会停止当前符号的识别。

2) 读取到非法字符时调用错误处理单元.

实现流程图如图 3.4 所示。



3.4 词法分析器流程图

3.2.3. 语法制导翻译

由于语义分析采用递归下降的一遍扫描方式实现，会在语法分析的过程中计算属性，执行语义规则，因此无法将语义分析单元拆分为一个独立的模块。

3.2.3.1. 语法分析

主要功能：

1) 为每一个非终结符实现递归的子程序

由于给定文法没有左递归与回溯，各个非终结符的候选式也没有公共左因子，所以当面临某一输入的终结符，当前非终结符选择的候选式是唯一确定的，所以读取到对应产生式的左因子便进入对应的产生式。若在产生式中遇到终结符，就查看其是否与文法 中的符号相匹配，若遇到非终结符，则调用其子程序继续分析。

2) 错误恢复

语法分析应当最大限度地分析出所有存在的语法错误，便于用户修改源程序。因此，若在语法分析过程中遇到了不符合文法的情况，调用错误处理单元后，应当尽可能地将分析恢复至正常，不会影响程序其余部分的分析。

为了实现这一功能，我的大致思路是：遇到错误时选定某些可以恢复分析的同步符号，比如当前非终结符的 follow 集、分号等，继续读取直到遇到这些符号时停止，然后继续正常的分析流程。

3.2.3.2. 语义分析

主要功能：

1) 提供目标代码的参数

语义分析的目的其实就是为了确定产生目标代码的参数。

首先是 op 参数的确定。op 的确定应当根据具体分析的语句功能来确定例如与表达式相关的<factor>、<term>、<exp>需要 opr 和 lod 指令计算并存储表达式的值， 而<statement>中的 call、read、write 语句则需要执行相应的 cal、red、wrt 指令，与条件相关的 if else 、while 语句则需要 opr、jpc、jmp 指令的参与。

接着是 L 层级参数的确定。需要提供 L 参数的指令只有 lod 、sto 、cal 三条，其中 lod 和 sto 指令的 L 是为了跨活动记录访问内存单元服务的，而对于 sto 当 L 为-1 时，sto 是一条传递形参的特殊指令。无论是过程还是变量符号的层级，都可以在符号表中直接查找到。

最后是 a 参数的确定。a 参数的含义有 4 种，偏移量（相对地址）、绝对地址（目标代码下标）、常量的值、opr 指令的操作码。对于前三种可以在符号表中直接查找到，而 opr 的操作码同 op 一样，也能根据具体分析的语句直接确定。

3.2.3.3. 目标代码生成

基本介绍：

定义了目标代码的结构，实现了目标代码的存储、生成、回填。

主要功能：

1) 结构

定义了一个结构体 OneCode 目标代码的具体形式：

op	L	a

op 段代表伪操作码，L 段代表调用层的层级 A 段代表相对地址或值。

由于目标代码的运行单元使用 display 表进行跨不同层级间的查找，所以只需知道被查找单元所在的层级便可直接访问对应的活动记录，因此目标代码的第二参数为层级而不是层差。

2) 存储

使用数组存储。

3) 生成(emit)

根据指定参数生成 OneCode 对象，然后直接写入 Code 数组。

4) 回填(backpatch)

使用 current 计数器 ，通过下标访问直接修改指定目标代码。

3.2.3.4. 翻译模式

详情见附件。

3.2.4. 错误单元处理

基本介绍：

错误处理单元接收来自词法分析器、语法分析器、语义分析过程中的错误信息，处理后将其输出至控制台。

主要功能：

1) 准确地告知错误出现的位置：

准确地告知错误出现的位置： 错误处理单元首先需要能够准确地定位到错误发生的位置，通常包括行号和列号。当词法分析器、语法分析器或语义分析器检测到错误时，错误处理单元会提供详细的位置信息。

2) 错误类型的描述：

错误处理单元会描述错误的类型，如“缺少关键字”、“非法字符”、“语法错误”等。通过清晰的错误类型描述，编译器能够给出更具体、易懂的错误信息，减少开发人员对错误原因的猜测。

3.2.5. 解释器

基本介绍：

符本模块对应代码中的 Interpreter.py 部分。主要实现了一个简单的解释器，能够执行由给定指令（以文本格式存储在 code.txt 文件中的操作码）描述的虚拟机程序。

本程序采用的活动记录结构如图 3.5 所示。

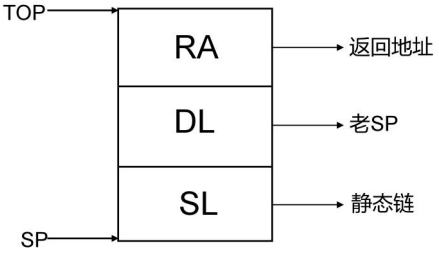


图 3.5 活动记录结构

其中:

- RA 表示返回地址
- DL 表示调用者的活动记录首地址，即老 SP
- SL 表示保存该过程直接外层的活动记录首地址，即静态链

当存在调用关系时，假设的调用关系及对应的活动记录结构变化如图 3.6 所示：

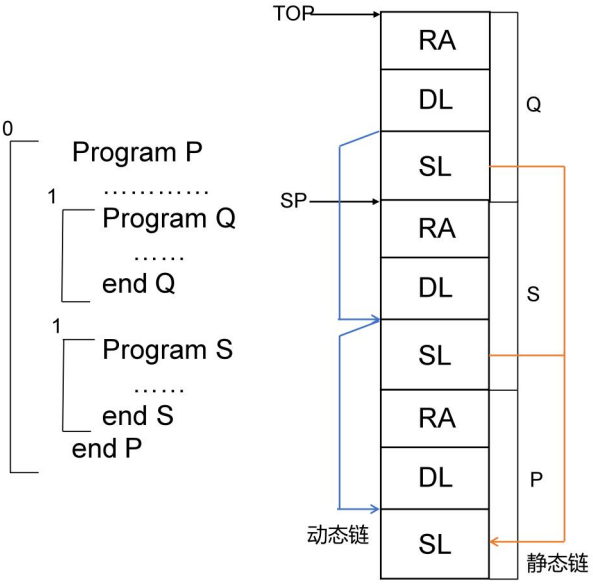


图 3.6 P 调用 Q、S 的活动记录变化图

给定的假想目标机代码及其具体含义如表 3.1 所示：

表 3.1 假想机代码及其具体含义

指令	含义
LIT 0，a	取常量 a 放入数据栈栈顶
OPR 0，a	执行运算，a 表示执行某种运算
LOD L，a	取变量（相对地址为 a，层差为 L）放到数据栈的栈顶
STO L，a	将数据栈栈顶的内容存入变量（相对地址为 a，层次差为 L）
CAL L，a	调用过程（转子指令）（入口地址为 a，层次差为 L）
INT 0，a	数据栈栈顶指针增加 a
JMP 0，a	无条件转移到地址为 a 的指令
JPC 0，a	条件转移指令，转移到地址为 a 的指令
RED L，a	读数据并存入变量（相对地址为 a，层次差为 L）
WRT 0，0	将栈顶内容输出

每条指令的含义通过 F、L 和 A 字段进行编码，分别表示操作码、层差和地址。

在此基础上，本解释器对指令 OPR 0，a 扩展为以下内容：

表 3.2 OPR 0,X 详细说明

指令	含义
OPR 0，0	返回调用点并退回
OPR 0，1	取反
OPR 0，2	相加
OPR 0，3	相减
OPR 0，4	相乘
OPR 0，5	相除
OPR 0，6	判断奇偶
OPR 0，7	等于
OPR 0，8	不等于
OPR 0，9	小于
OPR 0，10	大于
OPR 0，11	大于等于
OPR 0，12	小于等于

程序的核心是通过栈（stack）和静态链来模拟指令的执行流程。

程序首先读取指令并按行解析， get_sl(B, level)函数根据当前的 B（基址寄存器）和层差 level 来计算静态链的位置，从而获取相应的变量，add_code()函数将每行指令转换为操作字典后存储到 code 列表中。解释器依次执行每条指令，根据不同的 F（操作码）执行对应的操作，直到遇到终止条件（P == 0）

主要功能：

1) 栈管理与静态链：

栈管理：程序使用一个栈（stack）来模拟内存操作，栈中的数据包括常量、变量、函数调用的返回地址等。栈操作的主要功能是入栈和出栈，控制数据存取。

静态链：通过静态链（get_sl 函数）来访问不同层级的变量，保证在多层函数调用的情况下能够正确找到变量的位置。B（基址寄存器）和 level（层差）被用来定位静态链。

2) 指令集执行：

a) 控制流指令：

- JMP: 直接跳转到指定地址。
- JPC: 条件跳转，如果栈顶值为 0，则跳转。

b) 栈操作指令：

- INT: 在栈上开辟新的空间。
- LOD: 从栈中加载数据。
- STO: 将栈顶数据存储到指定位置。
- LIT: 将常量推入栈中。

c) 算术运算（OPR 指令）：

包括加法、减法、乘法、除法、取反、奇偶判断、比较操作（如等于、不等于、大于、小于等），具体内容见表 3.4。

d) 输入输出：

- RED: 从用户输入中读取一个整数，并将其存储到栈中。
- WRT: 输出栈顶的值。

3.2.6. 界面/可视化设计

可视化模块的设计目的是通过图形化界面展示 PL/0 程序的运行过程，直观地呈现解释器的核心执行逻辑和数据栈的变化。该模块基于 PyQt5 实现，包含数据栈的动态可视化、伪代码指令的展示、程序运行控制和用户交互功能等核心部分。以下对各个子模块进行详细说明。

3.2.6.1. 数据栈可视化

主要功能：

数据栈是解释器执行过程中最重要的运行时结构之一。可视化模块通过动态绘制的方式，实时展示数据栈的内容、栈单元的变化以及静态链、动态链和返回地址等重要信息。用户可以通过观察数据栈的变化，直观了解伪代码执行过程中变量的加载与存储、函数调用的上下文切换等操作。

实现方法：

数据栈的可视化使用 PyQt5 的 QGraphicsScene 和 QGraphicsView 实现。栈中的每个单元通过矩形框表示，框内的文字显示当前单元的值。

对于特殊的栈单元，例如静态链、动态链和返回地址，界面会通过文字标注予以说明。

当解释器执行一条伪代码指令后，系统会触发界面更新逻辑，重新绘制栈单元并标注最新的内容。

详细说明：

在绘制栈单元时，每个单元的高度和宽度固定，排列在一个垂直列表中，栈底在下方，栈顶在上方。

例如，当执行 LOD 指令时，新的变量值会被加载到栈顶；当执行 CAL 指令时，系统会在栈中为函数调用分配静态链、动态链和返回地址。这些变化会在界面上动态展示，帮助用户清晰地理解程序的执行逻辑。此外，栈单元的索引也会在矩形框的左侧显示，便于用户定位具体的栈位置。

3.2.6.2. 目标代码展示

主要功能：

目标代码展示模块用于直观地显示编译生成的伪代码列表，帮助用户理解程序的控制流和指令执行逻辑。通过表格形式展示伪代码，用户可以清楚地看到每条指令的操作码、层次差和目标地址等详细信息。

实现方法：

目标代码展示模块使用 PyQt5 的 QTableWidget 或 tkinter 的 Treeview 控件实现表格显示。每条伪代码作为表格中的一行，列包括操作码 (F)、层次差 (L) 和目标地址/常量 (A)。当用户点击“显示目标代码”按钮时，系统会读取已生成的伪代码列表，将其逐行插入表格中进行展示。

详细说明：

例如，对于如下的伪代码：

- JMP 0 10：无条件跳转到地址 10；
- LOD 1 4：从上一层加载偏移地址为 4 的变量值；
- OPR 0 2：执行栈顶的两数加法。

这些指令会被逐行插入到表格中，用户可以通过滚动查看全部伪代码。此外，当前正在执行的指令会在界面上高亮标注，帮助用户快速定位程序的执行位置。

3.2.6.3. 程序运行控制柜

主要功能：

程序运行控制模块提供对解释器的运行管理功能，支持单步执行和整体运行两种模式。用户可以自由选择执行方式，通过单步调试了解每条指令对数据栈的影响，或者选择整体运行快速完成程序的测试。

实现方法：

运行控制模块主要通过按钮事件绑定实现。当用户点击“运行”按钮时，系统会启动一个定时器，循环调用解释器的单步执行逻辑，实现连续的指令执行；当用户点击“下一条”按钮时，系统会暂停运行并逐步执行一条指令。所有的按钮事件均绑定到解释器的运行逻辑，并在每次执行后触发界面更新，包括数据栈的内容、目标代码的状态等。

详细说明：

在单步调试模式下，用户可以逐条执行伪代码，并在每次执行后观察数据栈的变化。例如，在执行 STO 指令时，栈顶的值会被存储到指定的内存位置；在执行 CAL 指令时，新的栈帧会被分配。在整体运行模式下，用户只需点击“运行”按钮，程序会自动执行所有指令并在结束后显示最终结果。通过这种方式，运行控制模块大大提升了用户的操作灵活性。

3.2.6.4. 用户交互设计

主要功能：

用户交互模块为用户提供输入输出功能，使程序能够与用户进行动态交互。在程序执行过程中，当遇到输入指令时，系统会暂停运行并提示用户输入；当遇到输出指令时，系统会将结果显示在界面的输出区域。

实现方法：

输入功能通过一个输入框和按钮实现，当程序执行到 RED 指令时，系统会弹出输入提示，要求用户在输入框中填写数据并点击确认按钮。确认后，系统会将输入值存储到解释器的栈中并恢复运行。输出功能通过一个文本框实现，当执行到 WRT 指令时，解释器会将栈顶的值提取出来并实时更新到输出框中。

详细说明：

例如，当程序执行到如下伪代码：

- RED：提示用户输入一个整数，并将其存储到栈中；
- WRT：从栈顶取出一个值并打印。

用户可以在界面中填写输入数据，并通过点击按钮确认输入；程序的输出结果则会直接显示在界面下方的输出区域。通过这种设计，用户可以与程序进行动态交互，进一步增强了系统的可用性。

3.2.6.5. 界面设计与布局

主要功能：

界面设计模块将系统的各个功能模块划分为不同的区域，包括数据栈展示区、目标代码显示区、运行控制按钮区以及输入输出交互区。通过合理的布局设计，保证用户能够方便地进行操作并观察程序的运行结果。

实现方法：

界面布局使用 PyQt5 的布局管理器实现，将窗口划分为多个区域：

- 左侧：数据栈展示区，动态显示栈的变化。
- 右侧上方：目标代码展示区，显示伪代码指令列表。
- 右侧下方：输入输出区域，用于与用户交互。
- 顶部：运行控制按钮区，包括“运行”、“下一条”、“暂停”等按钮。

详细说明：

例如，在程序运行过程中，左侧的栈区域会随着指令的执行动态更新，而右侧的目标代码展示区会高亮当前正在执行的指令。用户可以通过顶部按钮自由控制程序的执行，输入框和输出框则实现了用户与程序的交互。

3.3. 系统运行流程

系统运行流程如图 3.7 所示。

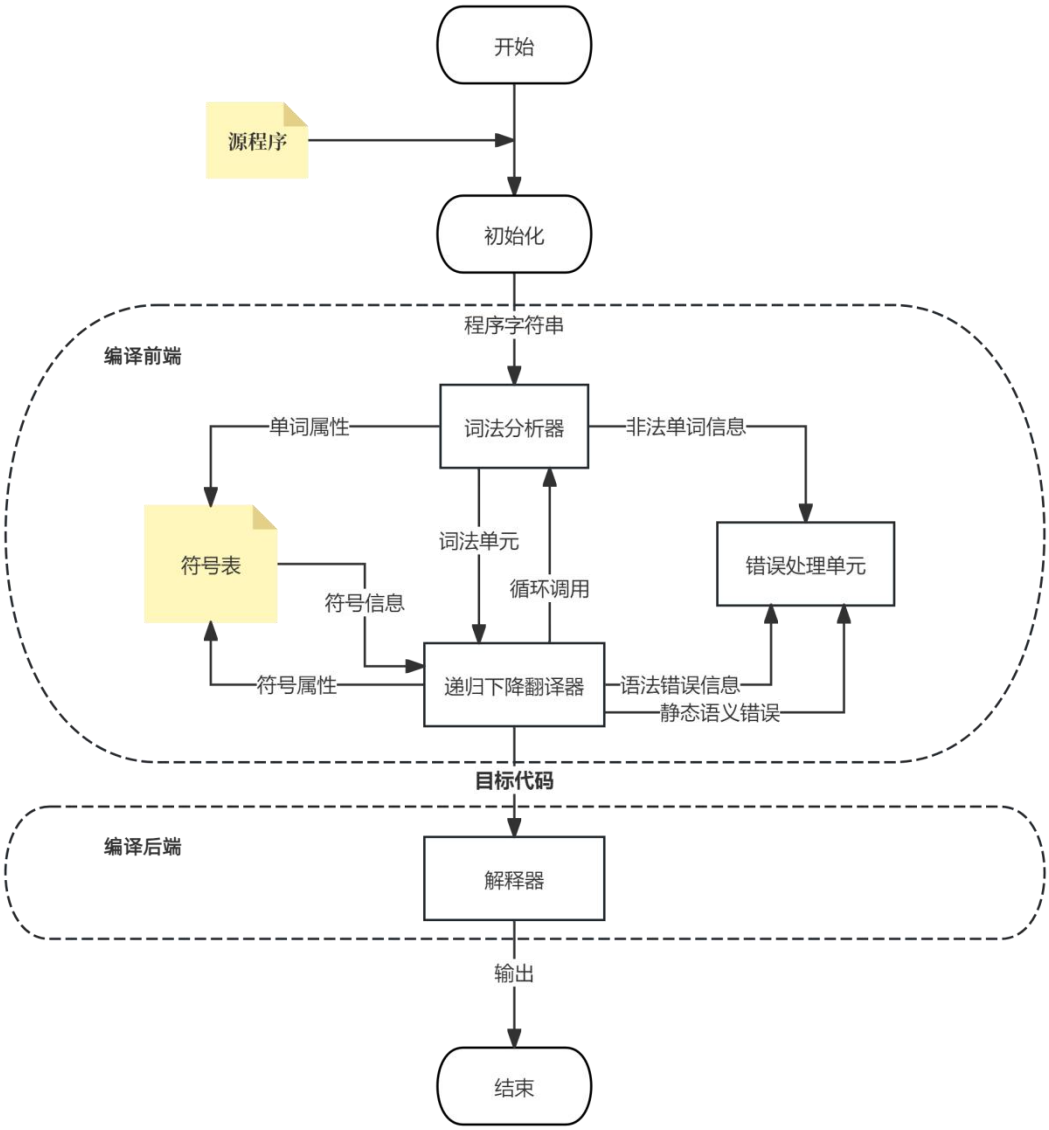


图 3.7 系统运行流程示意图

4. 系统实现

4.1. 系统主要函数说明

在本节中将按照第 3 节所介绍的顺序介绍系统实现。注意，这里仅展示相对重要的函数。

4.1.1. 符号表

表 4.1 符号表主要函数

函数声明	说明
void recordTable(string n, string t, int l = 0, int v = 0, int a = 0);	将符号登入符号表
vector<int> find(string n)	查找 n 在符号表中的位置
vector<int> find2(string n)	查找过程 n 在符号表中的位置
int len()	返回符号表的大小

符号表类实现思路：

• TableItem 类：用于表示符号表中的一个条目。包含符号的名称、类型、所在层次、值（如常数的值）和地址（如变量或过程的存储地址）。

• Table 类：表示符号表，使用 map<int, vector<TableItem>> 存储符号，其中 int 为层级，vector<TableItem> 为该层级中的所有符号项。提供了符号的录入、查找功能，并能跟踪符号的层次结构

4.1.2. 词法分析器

表 4.2 词法分析器主要函数

函数名	主要功能	输入	输出	实现思想
WorldAnalyse	构造函数、初始化	输入文件、输出文件	无	打开输入文件并初始化输出文件指针
Getxx	从输入文件读取有效字符。	无	无	递归处理无效字符，直到遇到有效字符
Isxx	判断当前字符是否为 xx 类型	当前字符	Bool	判断
Retract	撤回上一读取的字符，返回到上一个位置	无	无	通过 fseek 撤回字符，如果是换行符，则更新行号
Reserve	检查 strtoken 是否为保留字，如果是则返回相应编号	Strtoken	Int	顺序查找 Strtoken 是否为保留字，并返回相应编号
oneWordAnalyse	分析一个词法单元，识别关键字、标识符、常量符号等。	无	返回当前分析的词法单元的详细信息	通过读取并判断其类型，处理标识符、常量、符号并输出

4.1.3. 语法制导翻译,错误处理单元

4.1.3.1. Analyse

主要功能：语法、语义分析并判断是否错误。

输入：无直接输入。

输出：无

4.1.3.2. Prog

主要功能：分析 program 关键字和相关部分的语法，处理变量名和语法错误。

输入：通过词法分析器获取当前分析单词。

输出：根据分析结果输出语法错误信息。

实现思想：逐一检查是否产生错误信息（缺少 program、重复定义 id 等）调用 block 进一步解析。

4.1.3.3. Block

主要功能：处理程序中的 block 部分，包括常量声明、变量声明、过程声明和主体的语法分析及代码生成。

输入：通过词法分析器获取当前分析单词。

输出：输出语法分析和代码生成过程中的错误信息，并通过 code.emit()开辟空间和生成返回指令。

实现思想：

1. 初始化语句条数并记录当前位置，生成 JMP 指令用于跳转。
2. 依次检查 const、var、procedure 的声明，并调用相应的处理函数进行处理。
3. 回填开头的跳转指令，使得执行顺序从代码块的 body 部分开始。
4. 生成空间分配指令，调用 body() 解析实际的代码块内容。
5. 执行结束后生成返回指令，退出当前代码块。
6. 遇到程序结束符号 . 或错误时，跳过不合法的部分，进入下一个合法部分。

4.1.3.4. Condelcl

主要功能：处理常量声明部分，支持多个常量的声明，每个常量使用逗号分隔，并检查语法错误。

输入：通过词法分析器获取当前分析单词。

输出：输出语法错误信息。

实现思想：

1. 解析第一个常量声明，通过 _const() 处理。
2. 如果存在多个常量，判断逗号分隔并继续处理，检查是否遗漏逗号。
3. 检查常量声明的结束符号，若没有分号，报错。
4. 若遇到合法终结符（如 var, procedure, begin），停止当前分析并同步

4.1.3.5. _const

主要功能：处理单个常量的声明，检查常量的标识符、赋值符号、常量值，并记录到符号表中。

输入：通过词法分析器获取当前分析单词。

输出：输出语法错误信息。

实现思想：

1. 判断常量声明的标识符是否合法，确保是有效的标识符。
2. 检查是否存在赋值符号 :=，如果没有则报错。
3. 检查赋值符号后面是否是整数常量，若不是则报错。
4. 如果一切合法，将常量记录到符号表中。

4.1.3.6. Vardecl

主要功能：处理变量声明部分，支持多个变量的声明，检查标识符是否合法并记录到符号表。

输入：通过词法分析器获取当前分析单词。

输出：输出语法错误信息。

实现思想：

1. 解析第一个变量声明，通过检查标识符是否合法。
2. 支持多个变量声明，变量之间用逗号隔开，缺少逗号时报错。
3. 检查变量声明结束符号，若没有分号，则报错。
4. 处理变量声明时，若遇到错误或非法字符，进行同步跳过

4.1.3.7. proc

主要功能：处理过程声明，检查过程标识符、参数、过程体等，确保语法正确并记录到符号表中。

输入：通过词法分析器获取当前分析单词。

输出：输出语法错误信息。

实现思想：

1. 解析过程的关键字 procedure。
2. 处理过程名和参数，检查过程定义的合法性。
3. 解析参数列表，处理参数的合法性。
4. 进入过程体前，记录过程的起始地址，并增加当前空间。
5. 处理过程体（block()）。
6. 处理过程结束后，回收过程参数并进行同步。

4.1.3.8. body

主要功能：解析 begin 和 end 之间的语句，处理语句的顺序与语法错误。

输入：通过词法分析器获取当前分析单词。

输出：输出语法错误信息。

实现思想：

1. 检查是否以 begin 关键字开始。
2. 处理多个语句，确保每个语句之间正确分隔（使用分号 ;）。
3. 确保代码块以 end 关键字结束。
4. 在解析过程中进行错误同步，以确保能够继续分析

4.1.3.9. statement

主要功能:解析各种类型的语句，并进行错误处理与语法同步。

输入：通过词法分析器获取当前分析单词。

输出：输出语法、语义错误信息。

实现思想:

1. 检查当前词汇是否属于语句的开始。
2. 识别并调用相应的语句处理函数。
3. 处理语法错误，提供同步机制，确保能够继续解析后续部分

4.1.3.10. statement_id

主要功能：解析并处理赋值语句，验证变量的有效性，确保赋值符号和右侧表达式的正确性。

输入：通过词法分析器获取当前分析单词。

输出：语法、语义错误信息。通过 code.emit() 输出目标代码，用于将计算结果存

储到变量的地址。

实现思想：

1. 检查当前单词是否为有效的变量（id）。
2. 确认赋值符号 := 是否正确。
3. 解析赋值表达式并将结果存储到变量的地址。（生成目标代码 STO）

4.1.3.11. statement_if

主要功能：解析 if 语句，包括条件表达式和可选的 else 分支，生成目标代码，处理条件跳转。

输入：通过词法分析器获取当前分析单词。

输出：输出语法错误信息。通过 code.emit() 生成目标代码，实现条件跳转。

实现思想：

1. 检查是否是 if 语句的开头。
2. 解析条件表达式，将其计算结果放置在栈顶。
3. 处理 then，生成条件跳转代码。
4. 处理可选的 else 分支，生成跳转代码并回填跳转位置。

4.1.3.12. statement_while

主要功能：解析 while 循环，包括条件表达式和循环体，生成目标代码，处理条件跳转，模拟循环行为。

输入：通过词法分析器获取当前分析单词。

输出：语法、语义错误信息。通过 code.emit() 生成目标代码，实现循环的条件判断和跳转。

实现思想：

1. 检查是否是 while 语句的开头。
2. 解析循环条件表达式，并将其结果放入栈顶。
3. 生成条件跳转代码，判断条件是否为真。
4. 解析循环体语句。
5. 生成跳转代码，将程序控制流跳回到循环开始位置。

4.1.3.13. statement_call

主要功能：解析和处理过程调用语句，包括验证过程名、检查参数传递的正确性、生成目标代码（CAL 指令）。

输入：通过词法分析器获取当前分析单词。

输出：语法、语义错误信息。通过 code.emit() 生成目标代码，进行过程调用。

实现思想：

1. 检查 call 关键字是否存在，确保是过程调用语句。
2. 解析过程名，确保过程已定义且类型为过程。
3. 解析函数的参数列表，检查参数数量和格式。
4. 使用 CAL 指令生成调用过程的目标代码。

4.1.3.14. statement_read

主要功能：解析 read 语句并生成相应的目标代码。它检查 read 后面的变量，确保它们是合法的，并生成读取指令。

输入：通过词法分析器获取当前分析单词。

输出：语法、语义错误信息。通过 code.emit() 生成目标代码，模拟 read 操作。

实现思想：

1. 检查 read 语句的正确性，包括关键字、括号和变量。

2. 查找符号表中对应的变量，确保它们是定义过且合法的变量。
3. 对每个变量生成读取指令 (RED 和 STO)，并将值存储到栈中。
4. 处理逗号分隔符，确保每个变量都合法。
5. 对语法错误进行报告，并在需要时进行同步。

4.1.3.15. statement_write

主要功能：解析 write 语句并生成相应的目标代码。将栈顶的表达式值输出，并处理多个值的输出。

输入：通过词法分析器获取当前分析单词。

输出：语法、语义错误信息。通过 code.emit() 生成目标代码，模拟 write 操作。

实现思想：

1. 检查 write 语句的语法，包括关键字和括号。
2. 解析括号内的表达式，并生成输出指令。
3. 处理多个表达式的输出，并确保语法正确。
4. 处理右括号，并进行错误报告和同步。

4.1.3.16. exp

主要功能：解析表达式并生成相应的目标代码。处理加法、减法和取反操作，并依赖 term() 来解析与加法、减法相关的操作。

输入：通过词法分析器获取当前分析单词。

输出：语法错误信息。通过 code.emit() 生成目标代码。

实现思想：

1. 处理表达式的符号（+ 和 -）。
2. 解析乘除等优先级更高的运算（通过 term()）。
3. 处理连续的加法和减法操作。
4. 生成目标代码，模拟运算。

4.1.3.17. lexp

主要功能：解析逻辑表达式（如 odd 运算和比较运算符），并生成相应的目标代码。

输入：通过词法分析器获取当前分析单词。

输出：语法错误信息。通过 code.emit() 生成目标代码，模拟逻辑运算。

实现思想：

1. 判断是否为 odd 运算符，如果是，则进行处理。
2. 解析比较运算符（=, <, >, <= 等）。
3. 生成相应的目标代码。
4. 错误处理和语法修正。

4.1.3.18. term

主要功能：解析乘法和除法表达式。

输入：通过词法分析器获取当前分析单词。

输出：语法错误信息。通过 code.emit() 生成目标代码，模拟乘法和除法操作。

实现思想：

1. 解析第一个因子。
2. 处理后续的乘法或除法运算符。
3. 生成相应的目标代码。

4.1.3.19. factor

主要功能：解析因子，支持常量、变量和括号中的表达式。

输入：通过词法分析器获取当前分析单词。

输出：通过 code.emit() 生成目标代码，模拟对常量、变量或表达式的访问。

实现思想：

1. 判断当前词汇类型：常量、变量或数字。
2. 处理左括号情况，递归解析表达式。
3. 根据词汇生成对应的目标代码。

4.1.4. 解释器

4.1.4.1. get_sl

主要功能：根据当前基址寄存器 B 和层次差 L 查找目标静态链的基址。

输入：当前基址寄存器 B 和层次差 L。

输出：目标静态链的基址。

实现思想：

1. 从当前基址寄存器 B 开始，通过递归下降逐层遍历静态链，直到 L 减为 0。
2. 静态链的计算逻辑基于 PL/0 的语言特性，即函数的定义层级确定了其静态链。
3. 每次递归更新基址为栈中存储的上一层基址，最终返回目标基址。

4.1.4.2. add_code

主要功能：根据指令类型、层次差和目标地址动态生成目标代码，并存储到 code 列表中。

输入：指令类型 F，层次差 L，目标地址或常量值 A。

输出：无直接输出，目标代码存储在 code 列表中。

实现思想：

1. 将目标代码以字典的形式组织，包含 F（操作码）、L（层次差）和 A（目标地址）。
2. 动态调用 add_code 方法，将伪代码指令依次添加到 code 列表中，确保伪代码的存储顺序与逻辑执行顺序一致。
3. 为不同的操作码提供统一的存储格式，为解释器执行提供支持。

4.1.4.3. interpreter

主要功能：逐条解释执行目标代码，完成程序的动态运行，包括变量加载与存储、函数调用与返回、条件与无条件跳转等功能。

输入：已生成的目标代码 code。

输出：程序的执行结果（通过终端输出或数据栈的改变）。

实现思想：

1. 初始化寄存器和指令：解释器启动时初始化基址寄存器 B、栈顶指针 T 和指令指针 P，并加载第一条目标代码。
2. 循环执行指令：在 P 不为 0 的情况下，逐条解释执行目标代码。
3. 指令处理逻辑：

- 跳转类指令（JMP、JPC）：
  - JMP：直接修改指令指针 P 为目标地址，实现无条件跳转。
  - JPC：若栈顶值为 0，则跳转到目标地址，否则继续执行下一条指令。
- 变量类指令（LOD、STO、LIT）：
  - LOD：根据层次差 L 和偏移地址 A 从目标栈帧加载变量值到栈顶。
  - STO：将栈顶值存储到目标栈帧的指定位置。
  - LIT：将常量值 A 压入栈顶。
- 函数调用指令（CAL）：

- 在栈中存储静态链、动态链和返回地址，更新基址寄存器 B 和指令指针 P。
- 算术逻辑指令（OPR）：
  - 根据操作码 A 执行对应的算术或逻辑运算（如加法、减法、取反等）。
  - 当 A 为 0 时，执行函数返回操作，恢复调用点的运行环境。
- 输入输出指令（RED、WRT）：
  - RED：提示用户输入一个整数并将其存储到栈顶。
  - WRT：将栈顶的值输出到终端并移除栈顶值。

4. 程序结束：当执行到 OPR 0 0 时，结束程序运行。

4.1.5. 界面/可视化设计

4.1.5.1. display

主要功能：该函数绘制当前栈的状态，包括栈帧信息、指令信息、静态链、动态链、返回地址等内容。用于在界面上实时展示当前执行的状态。

输入：无显式输入，函数从 self.interpreter 获取当前的栈状态、指令信息、以及其他执行数据。

输出：绘制栈的信息和指令状态；绘制栈的信息和指令状态。

实现思想：

1. 该函数遍历栈中的每一个元素，绘制栈帧并显示其中的内容（包括静态链、动态链、返回地址等）。
2. 获取当前指令并将其信息显示出来，若程序执行完毕，显示结束信息。
3. 如果有提示信息（例如 info），则更新提示框内容；同时，如果 output 有栈顶输出，也会显示在界面上。

4.1.5.2. open_file

主要功能：打开文件对话框，读取用户选择的 txt 文件内容，并将其显示在界面中的文本框里。

输入：无显式输入，用户通过文件对话框选择文件。

输出：显示文件内容在 text_edit 中，格式化后的内容将按行号显示；清空场景、数据编辑框和输出编辑框。

实现思想：

1. 通过 QFileDialog 打开文件对话框，用户选择文件后，读取文件内容。
2. 对文件内容进行格式化，将每一行的内容添加行号，并将其显示在 text_edit 中。
3. 同时清空界面上的其他区域，如图形场景和输出框。

4.1.5.3. complier

主要功能：启动编译过程，调用外部 C++程序来生成目标代码，并将编译结果显示在输出框中。

输入：用户在文件对话框中选择的文件路径作为输入。

输出：将编译结果显示在 output_edit 中；根据目标代码生成解释器需要的栈、代码信息，并添加到 code 中。

实现思想：

1. 通过 subprocess.Popen 调用 C++程序进行编译，传入用户选择的文件路径。
2. 编译完成后，读取并显示编译结果。如果编译成功，进一步解析生成的目标代码，并存储在 code 列表中，准备后续的程序解释执行。
3. 初始化并创建一个解释器，启动执行过程。

4.1.5.4. begin

主要功能：初始化程序的执行状态，并启动解释器。该函数开始执行程序代码，并启动定时器来控制每一步执行。

输入：无显式输入，通过 self.interpreter 中的状态初始化。

输出：启动执行过程，初始化栈、指令指针、基址等信息；更新界面显示当前的执行状态。

实现思想：

1. 将 newbase 清空并初始化，重置解释器的栈、指令指针、基址等。
2. 获取当前指令并执行，更新界面上的图形和文本显示。
3. 启动定时器，设置每 1 毫秒执行一次下一步操作。

4.1.5.5. show_code

主要功能：通过 Tkinter 库创建窗口，展示目标代码表格。

输入：无显式输入，数据从 code 列表获取。

输出：弹出一个窗口，显示目标代码的操作表格。

实现思想：

1. 使用 Tkinter 的 Treeview 组件，创建一个表格显示目标代码的内容。
2. 表格的列包含操作码 F、L 值 L、A 值 A，每一条目标代码对应一行。
3. 启动一个 Tkinter 事件循环，保持窗口的交互性。

4.1.5.6. handle_next

主要功能：处理“下一条”按钮的点击事件，执行程序的下一步操作，并更新界面。

输入：无显式输入，依赖于解释器的当前状态。

输出：更新界面上的栈帧信息、指令信息；如果程序执行完毕，停止定时器并启用“开始”按钮。

实现思想：

1. 如果“下一步”按钮没有被禁用，则继续执行解释器的下一步操作。
2. 每次执行后，清空当前的界面，并重新绘制更新后的栈和指令信息。

如果程序已经结束，停止定时器并启用“开始”按钮，允许用户重新开始执行。

4.2. 系统代码

4.2.1. 符号表

```
class TableItem
{
public:
    TableItem(string n, string t, int l, int v, int a) : name(n), type(t), level(l), val(v), addr(a) {};
    string name;
    string type;
    int level; // 层次
    int val;
    int addr;
};
class Table
{
public:
    map<int, vector<TableItem>> table; // key 为 level，value 用一个 TableItem 组就可以了，因为同一时刻至多每层只定义了一个过程
    Table() {}
    void recordTable(string n, string t, int l = 0, int v = 0, int a = 0)
    {
        table[l].push_back(TableItem(n, t, l, v, a));
    }
    // 查找 name 在符号表的位置
    vector<int> find(string n)
    {
        // 先从自身 level 找起
        for (int i = level; i >= 0; i--)
        {
            for (int j = table[i].size() - 1; j >= 0; --j)
            {
                if (table[i][j].name == n)
                    return {i, j};
            }
        }
        return {-1, -1};
    }
    vector<int> find2(string n)
    {
        // 查找过程，可以找子过程，所以要这么做
        for (int i = 0;; i++)
        {
            if (!table.count(i))
```

```
            break;
        for (int j = table[i].size() - 1; j >= 0; --j)
        {
            if (table[i][j].name == n)
                return {i, j};
        }
    }
    return {-1, -1};
  }
  int len()
  {
    return table.size();
  }
};
```

4.2.2. 词法分析器

```
class WordAnalyse
{
public:
    string strToken = "";
    char ch;
    int code;
    int value;
    int worldNum = 31;
    int constNum = 0;
    int line = 1;
    int column = 1;
    int lastcolumn = 1;
    FILE *input;
    FILE *output = fopen("D:\\cppcode\\complier_course_design\\WordAnalyse.txt", "w");
    char output2[30];

    map<string, int> world = {
        {"program", 1}, {"const", 2}, {"var", 3}, {"procedure", 4}, {"begin", 5}, {"end", 6}, {"if", 7}, {"then", 8}, {"else", 9}, {"while", 10}, {"do", 11}, {"call", 12}, {"read", 13}, {"write", 14}, {"odd", 15}, {":=", 16}, {";", 17}, {",", 18}, {"=", 19}, {"<>", 20}, {"<", 21}, {"<=", 22}, {">", 23}, {">=", 24}, {"+", 25}, {"-", 26}, {"*", 27}, {"/", 28}, {"(", 29}, {")", 30}, {".", 31}};
    map<long, int> number = {};

    WordAnalyse(string in = "input.txt", string ou = "output.txt", string ou2 = "wordAnalyse.txt")
    {
        // cout << in << endl;
        input = fopen(in.c_str(), "r");
        // output = fopen(ou.c_str(),"w");
        // output2 = fopen(ou2.c_str(),"w");
    }
    void close()
    {
        fclose(input);
    }
    void GetChar()
    {
        // cout<<"Getchar   ";
        // cout<<input.tellg();
        ch = fgetc(input);
        // cout<<line<<"   "<<column;
```

```
38.       // printf("   %c \n",ch);
39.       column++;
40.    }
41.
42.    void GetBC()
43.    {
44.       if (ch == ' ' || ch == ' ')
45.       {
46.          GetChar();
47.          GetBC();
48.       }
49.    }
50.
51.    void Concat()
52.    {
53.       strToken += ch;
54.    }
55.
56.    bool IsDigit()
57.    {
58.       return ch >= '0' && ch <= '9';
59.    }
60.    bool IsLetter()
61.    {
62.       return (ch >= 'a' && ch <= 'z') || (ch >= 'A' && ch <= 'Z');
63.    }
64.    bool IsSymbol()
65.    {
66.       return ch == '=' || ch == '+' || ch == '-' || ch == '*' || ch == '/' ||
67.             ch == '>' || ch == '<' || ch == ':' || ch == ',' || ch == ';' || ch == '(' || ch == ')' || ch == '.
    ';
68.    }
69.    int Reserve()
70.    {
71.       // cout<<"reserve   "+strToken<<endl;
72.       if (world[strToken])
73.       {
74.          return world[strToken];
75.       }
76.       return 0;
77.    }
78.    void Retract()
79.    {
80.       // cout<<"retract   "<<endl;;
```

```
81.         if (feof(input))
82.             return;
83.         if (ch == '\n')
84.         {
85.             line++;
86.             column = 0;
87.             return;
88.         }
89.         column--;
90.         // cout<<input.tellg()<<"  ";
91.         fseek(input, -1, SEEK_CUR);
92.         // cout<<input.tellg()<<endl;
93.     }
94.     void test()
95.     {
96.         if (input == NULL)
97.             return;
98.         GetChar();
99.         // cout<<input.tellg()<<"  ";
100.          fseek(input, -1, SEEK_CUR);
101.          // cout<<input.tellg()<<endl;
102.          GetChar();
103.          // cout<<input.tellg()<<"  ";
104.          fseek(input, -1, SEEK_CUR);
105.          // cout<<input.tellg()<<endl;
106.      }
107.      void InsertId()
108.      {
109.          ++worldNum;
110.          world[strToken] = worldNum;
111.      }
112.      void InsertConst()
113.      {
114.          ++constNum;
115.          // cout<<strToken<<endl;
116.          number[stoi(strToken)] = constNum;
117.      }
118.
119.      void handle()
120.      {
121.          Retract();
122.          strToken.clear();
123.      }
124.
```

```
125.  string oneWordAnalyse()
126.  {
127.      sprintf(output2, "");
128.      GetChar();
129.      GetBC();
130.      // cout<<"Analyse  ";
131.      // cout<<input.tellg()<<"   "<<ch<<endl;
132.
133.      if (IsLetter())
134.      {
135.          while (IsLetter() || IsDigit())
136.          {
137.              Concat();
138.              GetChar();
139.          }
140.          code = Reserve();
141.          // 标识符
142.          if (code == 0)
143.          {
144.              InsertId();
145.              // output<<"(id,"+strToken +','+ to_string(worldNum) +")"<<endl;
146.              fprintf(output, "(id,%s,%d)\n", strToken.c_str(), worldNum);
147.              sprintf(output2, "nid %s %d %d", strToken.c_str(), line, column - 1);
148.              // cout<<"(id,"+strToken +','+ to_string(worldNum) +")"<<endl;
149.          }
150.          // 保留字
151.          else
152.          {
153.              // output<<"("+ strToken+","+ to_string(world[strToken]) +")"<<endl;
154.              if (world[strToken] <= 31)
155.              {
156.                  fprintf(output, "(reserved word,%s,%d)\n", strToken.c_str(), world[strToken]
    );
157.                  sprintf(output2, "word %s %d %d", strToken.c_str(), line, column - 1);
158.              }
159.              else
160.              {
161.                  fprintf(output, "(id,%s,%d)\n", strToken.c_str(), world[strToken]);
162.                  sprintf(output2, "id %s %d %d", strToken.c_str(), line, column - 1);
163.              }
164.
165.              // cout<<"("+ strToken+","+ to_string(world[strToken]) +")"<<endl;
166.          }
167.      }
```

```
168.        else if (IsDigit())
169.        {
170.            while (IsDigit())
171.            {
172.                Concat();
173.                GetChar();
174.            }
175.            if (ch == '\n')
176.            {
177.                InsertConst();
178.                fprintf(output, "(integer,%s,%d)\n", strToken.c_str(), constNum);
179.                sprintf(output2, "integer %s %d %d", strToken.c_str(), line, column - 1);
180.                // cout<<"(integer,"+ strToken+","+to_string(constNum) +")"<<endl;
181.                handle();
182.                return output2;
183.            }
184.            // 处理小数点
185.            Retract();
186.            GetChar();
187.            if (ch == '.')
188.            {
189.                Concat();
190.                GetChar();
191.                while (IsDigit())
192.                {
193.                    Concat();
194.                    GetChar();
195.                }
196.                InsertConst();
197.                fprintf(output, "(float,%s,%d)\n", strToken.c_str(), constNum);
198.                sprintf(output2, "float %s %d %d", strToken.c_str(), line, column - 1);
199.                // cout<<"(float,"+ strToken+","+to_string(constNum) +")"<<endl;
200.            }
201.            // 形如 111aaa
202.            else if (IsLetter())
203.            {
204.                sprintf(output2, "integer %s %d %d", strToken.c_str(), line, column - 1);
205.                string tmp;
206.                while (IsLetter() || IsDigit())
207.                {
208.                    if (IsLetter())
209.                        tmp += ch;
210.                    Concat();
211.                    GetChar();
```

```
212.                 }
213.                 if (world.count(tmp))
214.                     sprintf(output2, "word %s %d %d", tmp.c_str(), line, column - 1);
215.                 // fprintf(output,"Error: 非法标志符 %s 出现
    在 行:%d, 列:%d.\n",strToken.c_str(),line,column-1);
216.                 printf("词法错误: 非法标志符 %s 出现
    在 行:%d, 列:%d.\n", strToken.c_str(), line, column - 1);
217.                 generate = false;
218.             }
219.             else
220.             {
221.                 InsertConst();
222.                 fprintf(output, "(integer,%s,%d)\n", strToken.c_str(), constNum);
223.                 sprintf(output2, "integer %s %d %d", strToken.c_str(), line, column - 1);
224.                 // cout<<"(integer,"+ strToken+","+to_string(constNum) +")"<<endl;
225.             }
226.         }
227.         else if (IsSymbol())
228.         {
229.             // cout<<"else if IsSymbol "<<(char)ch<<endl;
230.             while (IsSymbol())
231.             {
232.                 Concat();
233.                 GetChar();
234.             }
235.             if (strToken == "+")
236.             {
237.                 fprintf(output, "(aop,%s)\n", strToken.c_str());
238.                 sprintf(output2, "aop %s %d %d", strToken.c_str(), line, column - 1);
239.             }
240.             else if (strToken == "-")
241.             {
242.                 fprintf(output, "(aop,%s)\n", strToken.c_str());
243.                 sprintf(output2, "aop %s %d %d", strToken.c_str(), line, column - 1);
244.             }
245.             else if (strToken == "*")
246.             {
247.                 fprintf(output, "(mop,%s)\n", strToken.c_str());
248.                 sprintf(output2, "mop %s %d %d", strToken.c_str(), line, column - 1);
249.             }
250.             else if (strToken == "/")
251.             {
252.                 fprintf(output, "(mop,%s)\n", strToken.c_str());
253.                 sprintf(output2, "mop %s %d %d", strToken.c_str(), line, column - 1);
```

```
254.        }
255.        else if (strToken == "=")
256.        {
257.            fprintf(output, "(lop,%s)\n", strToken.c_str());
258.            sprintf(output2, "lop %s %d %d", strToken.c_str(), line, column - 1);
259.        }
260.        else if (strToken == "<>")
261.        {
262.            fprintf(output, "(lop,%s)\n", strToken.c_str());
263.            sprintf(output2, "lop %s %d %d", strToken.c_str(), line, column - 1);
264.        }
265.        else if (strToken == "<")
266.        {
267.            fprintf(output, "(lop,%s)\n", strToken.c_str());
268.            sprintf(output2, "lop %s %d %d", strToken.c_str(), line, column - 1);
269.        }
270.        else if (strToken == "<=")
271.        {
272.            fprintf(output, "(lop,%s)\n", strToken.c_str());
273.            sprintf(output2, "lop %s %d %d", strToken.c_str(), line, column - 1);
274.        }
275.        else if (strToken == ">")
276.        {
277.            fprintf(output, "(lop,%s)\n", strToken.c_str());
278.            sprintf(output2, "lop %s %d %d", strToken.c_str(), line, column - 1);
279.        }
280.        else if (strToken == ">=")
281.        {
282.            fprintf(output, "(lop,%s)\n", strToken.c_str());
283.            sprintf(output2, "lop %s %d %d", strToken.c_str(), line, column - 1);
284.        }
285.        else if (strToken == ":=")
286.        {
287.            fprintf(output, "(lop,%s)\n", strToken.c_str());
288.            sprintf(output2, "lop %s %d %d", strToken.c_str(), line, column - 1);
289.        }
290.        else if (strToken == "(")
291.        {
292.            fprintf(output, "(lpar,%s)\n", strToken.c_str());
293.            sprintf(output2, "lpar %s %d %d", strToken.c_str(), line, column - 1);
294.        }
295.        else if (strToken == ")")
296.        {
297.            fprintf(output, "(rpar,%s)\n", strToken.c_str());
```

```
298.             sprintf(output2, "rpar %s %d %d", strToken.c_str(), line, column - 1);
299.         }
300.         else if (strToken == ",")
301.         {
302.             fprintf(output, "(comma,%s)\n", strToken.c_str());
303.             sprintf(output2, "comma %s %d %d", strToken.c_str(), line, column - 1);
304.         }
305.         else if (strToken == ";")
306.         {
307.             fprintf(output, "(semicolon,%s)\n", strToken.c_str());
308.             sprintf(output2, "semicolon %s %d %d", strToken.c_str(), line, column - 1);
309.         }
310.         else if (strToken == ".")
311.         {
312.             fprintf(output, "(period,%s)\n", strToken.c_str());
313.             sprintf(output2, "period %s %d %d", strToken.c_str(), line, column - 1);
314.         }
315.         else if (strToken == ");")
316.         {
317.             fprintf(output, "(rpar,%s)\n", strToken.c_str());
318.             sprintf(output2, "rpar ) %d %d semicolon ; %d %d", line, column - 2, line, col
    umn - 1);
319.             // fprintf(output,"(semicolon,%s)\n",strToken.c_str());
320.         }
321.         else if (strToken == "()")
322.         {
323.             fprintf(output, "(lpar,%s)\n", strToken.c_str());
324.             sprintf(output2, "lpar ) %d %d rpar ) %d %d", line, column - 2, line, column -
    1);
325.             // fprintf(output,"(rpar,%s)\n",strToken.c_str());
326.         }
327.         else if (strToken == "();")
328.         {
329.             fprintf(output, "(lpar,%s)\n", strToken.c_str());
330.             sprintf(output2, "lpar ( %d %d rpar ) %d %d semicolon ; %d %d", line, colum
    n - 3, line, column - 2, line, column - 1);
331.             // fprintf(output,"(rpar,%s)\n",strToken.c_str());
332.         }
333.         else
334.         {
335.             if (strToken != "\n")
336.             {
337.                 if (strToken != ");")
338.                 {
```

```
339.                     printf("词法错误: 疑似运算符 '%s' 出现在 行:%d, 列:%d，请检查是否缺失符号\n", strToken.c_str(), line, column - 1);
340.                     generate = false;
341.                     // fprintf(output,"Error: 疑似运算符 '%s' 出现在 行:%d, 列:%d，请检查是否缺失符号\n",strToken.c_str(),line,column-1);
342.                 }
343.             }
344.         }
345.     }
346.     else
347.     {
348.         if (strToken != "\n" && strToken != "")
349.         {
350.             printf("词法错误: 疑似运算符 '%s' 出现在 行:%d, 列:%d，请检查是否缺失符号\n", strToken.c_str(), line, column - 1);
351.             generate = false;
352.             // fprintf(output,"Error: 疑似运算符 '%s' 出现在 行:%d, 列:%d，请检查是否缺失符号\n",strToken.c_str(),line,column-1);
353.         }
354.         GetChar();
355.     }
356.     handle();
357.     return output2;
358. }
359. string analyse()
360. {
361.     if (!input)
362.     {
363.         sprintf(output2, "ProEnd # %d %d", line, column - 1);
364.         return output2;
365.     }
366.     if (!feof(input))
367.     {
368.         return oneWordAnalyse();
369.         // cout<<input.tellg()<<(char)input.get()<<endl;
370.     }
371.     sprintf(output2, "ProEnd # %d %d", line, column - 1);
372.     return output2;
373. }
374. };
```

4.2.3. 语法制导翻译,错误处理单元

```
class GrammarAnalyse
{
public:
    int nowNum = 0;
    int allNum = 0;
    bool wrong = false;   // 语法错误
    bool wrong2 = false;  // 语义错误
    vector<OneWord> words; // 存储单词
    vector<error> errors;
    WordAnalyse word_analyse;
    fstream input;

    GrammarAnalyse(string in) : word_analyse(in)
    {
        allNum += 2;
        string input = word_analyse.analyse();
        vector<string> res;
        Stringsplit(input, ' ', res);
        words.push_back(OneWord(res[0], res[1], res[2], res[3]));
        input = word_analyse.analyse();
        res.clear();
        Stringsplit(input, ' ', res);
        // cout << input << endl;
        // debug;
        words.push_back(OneWord(res[0], res[1], res[2], res[3]));
    }
    void printError(string info, int row, int column) // 语法错误
    {
        generate = false;
        wrong = true;
        // if(info.find("同步")!=info.npos) return;
        printf("语法错误：row:%2d,column:%2d ", row, column);
        printf(info.c_str());
        errors.push_back(error(info, row, column));
        printf("\n");
    }
    void printError2(string info, int row, int column) // 语义错误
    {
        generate = false;
        wrong2 = true;
        // if(info.find("同步")!=info.npos) return;
        printf("语义错误：row:%2d,column:%2d ", row, column);
```

```
43.         printf(info.c_str());
44.         errors.push_back(error(info, row, column));
45.         printf("\n");
46.     }
47.     void close()
48.     {
49.         word_analyse.close();
50.     }
51.     map<string, set<string>> first_set = {
52.         {"<prog>", {"program"}},
53.         {"<block>", {"const", "var", "procedure", "begin"}},
54.         {"<condecl>", {"const"}},
55.         {"<const>", {"id", "nid"}},
56.         {"<vardecl>", {"var"}},
57.         {"<proc>", {"procedure"}},
58.         {"<body>", {"begin"}},
59.         {"<statement>", {"nid", "id", "if", "while", "call", "begin", "read", "write"}},
60.         {"<lexp>", {"odd", "+", "-", "nid", "id", "(", "integer"}},
61.         {"<exp>", {"odd", "+", "-", "nid", "id", "(", "integer"}},
62.         {"<term>", {"id", "nid", "(", "integer"}},
63.         {"<factor>", {"id", "nid", "(", "integer"}},
64.     };
65.     map<string, set<string>> follow_set = {
66.         {"<prog>", {"#"}},
67.         {"<block>", {"#", ";", "begin"}},
68.         {"<condecl>", {"var", "procedure", "begin"}},
69.         {"<const>", {",", ";"}},
70.         {"<vardecl>", {"procedure", "begin"}},
71.         {"<proc>", {"begin", ";"}},
72.         {"<body>", {"#", "else", "end", ";", ".", "begin"}},
73.         {"<statement>", {"else", "end", ";"}},
74.         {"<lexp>", {"then", "do"}},
75.         {"<exp>", {"else", "end", "then", "do", ",", ")", ";", "=", "<>", "<", "<=", ">", ">="}}
    ,
76.         {"<term>", {"+", "-", "else", "end", "then", "do", ",", ")", ";", "=", "<>", "<", "<=", "
    >", ">="}},
77.         {"<factor>", {"*", "/", "+", "-", "else", "end", "then", "do", ",", ")", ";", "=", "<>", "<
    ", "<=", ">", ">="}},
78.         {"id", {",", ":=", ";", "(", ")", "*", "/", "+", "-", "else", "end", "then", "do", ",", ")", ";",
     "=", "<>", "<", "<=", ">", ">="}},
79.         {"integer", {
80.                     "*",
81.                     "/",
82.                     "+",
```

```
                "-",
                "else",
                "end",
                "then",
                "do",
                ",",
                ")",
                ";",
                "=",
                "<>",
                "<",
                "<=",
                ">",
                ">=",
            }},
};

  void nextWord()
  {
      if (words[nowNum].type == "ProEnd")
          return;
      allNum++;
      nowNum++;
      string input = word_analyse.analyse();
      vector<string> res;
      Stringsplit(input, ' ', res);
      for (int i = 0; i < res.size(); i += 4)
      {
       words.push_back(OneWord(res[i], res[i + 1], res[i + 2], res[i + 3]));
      }
      // if (words[nowNum].type != "ProEnd") cout << words[nowNum].name << " " <
    < words[nowNum].type<<" "<<words[nowNum].row<<" " << words[nowNum].column
    << endl;
  }
  void analyse()
  {
      prog();
      if (!wrong)
      {
          cout << "语法分析结束,程序无语法错误" << endl;
          if (!wrong2)
              cout << "语义分析结束,代码已经生成" << endl;
          else
              cout << "语义分析结束,程序出现语义错误" << endl;
```

```
125.        }
126.        if (wrong)
127.            cout << "语法分析结束,程序出现语法错误" << endl;
128.    }
129.    void prog()
130.    {
131.        // cout << nowNum << endl;
132.        // debug;
133.        if (words[nowNum].name != "program")
134.        {
135.            if (words[nowNum].type == "nid")
136.            {
137.                printError("缺少 program 关键字
    ", words[nowNum].column, words[nowNum].row);
138.                //--nowNum;
139.            }
140.            else if (words[nowNum + 1].name == "program")
141.            {
142.                printError("出现错误单词
    " + words[nowNum].name, words[nowNum].column, words[nowNum].row);
143.            }
144.            else if (words[nowNum + 1].type == "nid")
145.            {
146.                printError("program 关键字拼写错误
    ", words[nowNum].column, words[nowNum].row);
147.            }
148.            else
149.            {
150.            }
151.        }
152.        nextWord();
153.        if (words[nowNum].type != "nid")
154.        {
155.            if (words[nowNum].name == ";")
156.            {
157.                printError("缺少 id", words[nowNum].column, words[nowNum].row);
158.                --nowNum;
159.            }
160.            else if (words[nowNum + 1].type == "nid")
161.            {
162.                printError("出现错误单词
    " + words[nowNum].name, words[nowNum].column, words[nowNum].row);
163.            }
164.            else if (words[nowNum + 1].name == ";")
```

```
165.            {
166.                printError("使用关键字作为 id 名或者是错误的
    id", words[nowNum].column, words[nowNum].row);
167.            }
168.            else
169.            {
170.            }
171.        }
172.        nextWord();
173.        while (words[nowNum].type == "nid" || words[nowNum].type == "id")
174.        {
175.            if (words[nowNum].type == "id")
176.                printError("id  " + words[nowNum].name + "  重复定义
    ", words[nowNum].column, words[nowNum].row);
177.            printError("program 后不可有多个
    id", words[nowNum].column, words[nowNum].row);
178.            //!
179.            nextWord();
180.        }
181.        if (words[nowNum].name != ";")
182.        {
183.            if (first_set["<block>"].count(words[nowNum].name)) // 出现在 block 的 first 集
    中，说明缺失;
184.            {
185.                --nowNum;
186.                printError("缺失分号", words[nowNum].column, words[nowNum].row);
187.            }
188.            else if (first_set["<block>"].count(words[nowNum + 1].name)) // 下一个单词是
    block 的 first 集，说明这个符号错了
189.            {
190.                printError("错误的符号 " + words[nowNum].name + " 应该为分号
    ", words[nowNum].column, words[nowNum].row);
191.            }
192.        }
193.        nextWord();
194.        // <prog> → program <id>；<block>
195.        block();
196.    }
197.    void block()
198.    {
199.        // space 第一次进入 block 为 3，之后按照规矩来 SL DL RA
200.        // 记录当前语句条数，便于回填
201.        int current = code.len(); // 下一条生成的地址，便于回填 653 行
202.        code.emit("JMP", 0, 0);
```

```
203.
204.        if (words[nowNum].name == "const")
205.            condecl();
206.        if (words[nowNum].name == "var")
207.            vardecl();
208.        if (words[nowNum].name == "procedure")
209.        {
210.            int nowSpace = space; // 保存现在的空间
211.            proc();
212.            space = nowSpace; // 恢复
213.        }
214.        // 进入 body 前需要回填开头的 jump 指令，因为代码是顺序生成的，但是我们
    执行的时候是 body 先开始的，所以要跳到 body 这里
215.        code.code[current].a = code.len(); // 跳入到 body 这里
216.        code.emit("INT", 0, space);       // 开辟空间
217.        body();
218.        code.emit("OPR", 0, 0); // 返回调用点并退栈
219.        if (words[nowNum].name == ".")
220.            return;
221.        while (!follow_set["<block>"].count(words[nowNum].name))
222.        {
223.            printError("正在同步<block>,跳
    过 " + words[nowNum].name, words[nowNum].column, words[nowNum].row);
224.            nextWord();
225.        }
226.    }
227.    void condecl()
228.    {
229.        nextWord(); // 第一个词语肯定是 const
230.        _const();
231.        while (words[nowNum].name == "," || words[nowNum].type == "nid")
232.        {
233.            if (words[nowNum].name == ",")
234.            {
235.                nextWord();
236.                _const();
237.            }
238.            else
239.            {
240.                printError("const 中缺少逗号间隔。
    ", words[nowNum].column, words[nowNum].row);
241.                _const();
242.            }
243.        }
```

```
244.        if (words[nowNum].name == ";")
245.        {
246.            nextWord();
247.            return;
248.        }
249.        else
250.        {
251.            printError("缺失分号", words[nowNum].column, words[nowNum].row);
252.        }
253.        while (!follow_set["<condecl>"].count(words[nowNum].name))
254.        {
255.            printError("正在同步<condecl>,跳
    过 " + words[nowNum].name, words[nowNum].column, words[nowNum].row);
256.            nextWord();
257.        }
258.        // 现在指向的是 var procedure 或者 begin
259.    }
260.    void _const()
261.    {
262.        if (words[nowNum].type != "nid" && words[nowNum].type != "id")
263.        {
264.            if (words[nowNum].name == ":=")
265.            {
266.                printError("缺少 id", words[nowNum].column, words[nowNum].row);
267.                --nowNum;
268.            }
269.            else if (words[nowNum + 1].name == ":=")
270.            {
271.
272.                printError("错误的 id" + words[nowNum].name + ",可能使用了关键字作为
    标识符？ ", words[nowNum].column, words[nowNum].row);
273.            }
274.            nextWord();
275.            nextWord();
276.            nextWord();
277.        }
278.        else
279.        {
280.            string constname = words[nowNum].name;
281.            nextWord();
282.            if (words[nowNum].name != ":=")
283.            {
284.                if (words[nowNum].type == "integer")
285.                {
```

```
286.                 printError("缺少赋值符号
    ", words[nowNum].column, words[nowNum].row);
287.                 --nowNum;
288.             }
289.             else if (words[nowNum + 1].type == "integer")
290.             {
291.                 printError("错误的赋值符号
    " + words[nowNum].name, words[nowNum].column, words[nowNum].row);
292.             }
293.             nextWord();
294.             nextWord();
295.         }
296.         else
297.         {
298.             nextWord();
299.             if (words[nowNum].type != "integer")
300.             {
301.                 if (words[nowNum].name == "," || words[nowNum].name == ";")
302.                 {
303.                     printError("缺少整数", words[nowNum].column, words[nowNum].row);
304.                 }
305.                 else if (words[nowNum + 1].name == "," || words[nowNum + 1].name == ";
    ")
306.                 {
307.                     printError("有错误的整数
    " + words[nowNum].name, words[nowNum].column, words[nowNum].row);
308.                 }
309.             }
310.             else
311.             {
312.                 table.recordTable(constname, "CONSTANT", level, stoi(words[nowNum].n
    ame));
313.             }
314.             nextWord();
315.         }
316.     }
317.
318.     while (!follow_set["<const>"].count(words[nowNum].name) && words[nowNum].
    type != "nid")
319.     {
320.         printError("正在同步<const>,跳
    过 " + words[nowNum].name, words[nowNum].column, words[nowNum].row);
321.         nextWord();
322.     }
```

```
323.    }
324.    void vardecl()
325.    {
326.        nextWord(); // 第一个词语肯定是 var
327.        if (words[nowNum].type != "nid" && words[nowNum].type != "id")
328.        {
329.            if (words[nowNum].name == ",")
330.            {
331.                --nowNum;
332.                printError("缺少 id", words[nowNum].column, words[nowNum].row);
333.            }
334.            else if (words[nowNum + 1].name == ",")
335.            {
336.                printError("有错误的 id" + words[nowNum].name + "可能使用了保留字作为 id？", words[nowNum].column, words[nowNum].row);
337.            }
338.        }
339.        else
340.        {
341.            // 是 id
342.            table.recordTable(words[nowNum].name, "VARIABLE", level, 0, space);
343.            // cout<<words[nowNum].name<<" "<<level<<endl;
344.            space++; // 空间增加
345.        }
346.        nextWord();
347.        while (words[nowNum].type == "nid" || words[nowNum].type == "id")
348.        {
349.            // if (words[nowNum].type == "id") printError("id" + words[nowNum].name + "重复定义", words[nowNum].column, words[nowNum].row);
350.            printError("var 中缺少逗号", words[nowNum].column, words[nowNum].row);
351.            nextWord();
352.        }
353.        while (words[nowNum].name == ",")
354.        {
355.            nextWord();
356.            if (words[nowNum].name == ";")
357.                break;
358.            if (words[nowNum].type != "nid" && words[nowNum].type != "id")
359.            {
360.                if (words[nowNum].name == ",")
361.                {
362.                    printError("缺少 id", words[nowNum].column, words[nowNum].row);
363.                    --nowNum;
364.                }
```

```
365.             else if (words[nowNum + 1].name == ",")
366.             {
367.                 printError("有错误的 id" + words[nowNum].name + "可能使用了保留字
    作为 id？", words[nowNum].column, words[nowNum].row);
368.             }
369.         }
370.         else
371.         {
372.             // 是 id
373.             // cout<<words[nowNum].name<<" "<<space<<endl;
374.             table.recordTable(words[nowNum].name, "VARIABLE", level, 0, space);
375.             // cout << words[nowNum].name << " " << level << endl;
376.             space++; // 空间增加
377.         }
378.         nextWord(); // 似乎有点粗糙，万一目标只少了一个，也会直接开始同步,试
    着解决一下
379.         while (words[nowNum].type == "nid" || words[nowNum].type == "id")
380.         {
381.             if (words[nowNum].type == "id")
382.                 printError("id" + words[nowNum].name + "重复定义
    ", words[nowNum].column, words[nowNum].row);
383.             printError("缺少逗号", words[nowNum].column, words[nowNum].row);
384.             nextWord();
385.         }
386.     }
387.     if (words[nowNum].name == ";")
388.     {
389.         nextWord(); // 正常结束
390.         return;
391.     }
392.     while (!follow_set["<vardecl>"].count(words[nowNum].name))
393.     {
394.         printError("正在同步<vardecl>,跳
    过 " + words[nowNum].name, words[nowNum].column, words[nowNum].row);
395.         nextWord();
396.     }
397. }
398. void proc()
399. {
400.     space = 3;
401.     string tmp;
402.     int proca = 0;
403.     int vnmum = 0;
404.     if (words[nowNum].name != "procedure")
```

```
405.        {
406.            if (words[nowNum].type == "id")
407.            {
408.                --nowNum;
409.                printError("缺少 procedure 关键字
    ", words[nowNum].column, words[nowNum].row);
410.            }
411.            else if (words[nowNum + 1].type == "id")
412.            {
413.                printError("错误的关键字" + words[nowNum].name + "应该为
    procedure", words[nowNum].column, words[nowNum].row);
414.            }
415.        }
416.        nextWord();
417.        if (words[nowNum].type != "nid" && words[nowNum].type != "id")
418.        {
419.            if (words[nowNum].name == "(")
420.            {
421.                --nowNum;
422.                printError("缺少 id", words[nowNum].column, words[nowNum].row);
423.            }
424.            else if (words[nowNum + 1].name == "(")
425.            {
426.                printError("有错误的 id" + words[nowNum].name + "可能使用了保留字作
    为 id？", words[nowNum].column, words[nowNum].row);
427.            }
428.        }
429.        else
430.        {
431.            // 是 id，填写符号表
432.            tmp = words[nowNum].name;
433.            proca = table.table[level].size();
434.            //! 为了方便回填，记录当前过程的地址
435.            table.recordTable(words[nowNum].name, "PROCEDURE", level, 0, code.len());
    // 新的过程的地址 是 下一个写入的代码位置，也就是它自己的第一条代码地址
436.        }
437.        nextWord();
438.        while (words[nowNum].type == "nid" || words[nowNum].type == "id")
439.        {
440.            // if (words[nowNum].type == "id") printError("id  " + words[nowNum].name +
    "  重复定义", words[nowNum].column, words[nowNum].row);
441.            printError("procedure 后不可有多个
    id", words[nowNum].column, words[nowNum].row);
442.            nextWord();
```

```
443.        }
444.        if (words[nowNum].name != "(")
445.        {
446.
447.            if (words[nowNum].name == ")" || words[nowNum].type == "nid" || words[now
    Num].type == "id")
448.            {
449.                printError("缺少 ( ", words[nowNum].column, words[nowNum].row);
450.                --nowNum; // 回退一个单词
451.            }
452.            else if (words[nowNum].name == "," || words[nowNum + 1].name == ")" || word
    s[nowNum + 1].type == "nid")
453.            {
454.                printError("应该为 ( ， 出现了错误的单词： " + words[nowNum].name + "
    可能使用了保留字作为 id？ ", words[nowNum].column, words[nowNum].row);
455.            }
456.            // 不能全部跳完，要继续语法分析
457.            while (!first_set["<body>"].count(words[nowNum].name))
458.            {
459.                printError("正在同步<proc>,跳
    过 " + words[nowNum].name, words[nowNum].column, words[nowNum].row);
460.                nextWord();
461.            }
462.        }
463.        else
464.        {
465.            // 为('
466.            nextWord();
467.            if (words[nowNum].type == "nid" || words[nowNum].type == "id")
468.            {
469.                // 是 id,填写表
470.                table.recordTable(words[nowNum].name, "VARIABLE", level + 1, 0, space);
    // 特别注意这里，因为形参实际是下一级的
471.                space++;                                        // 空间增加
472.                nextWord();
473.                while (words[nowNum].type == "nid" || words[nowNum].type == "id")
474.                {
475.                    printError("缺少逗号", words[nowNum].column, words[nowNum].row);
476.                    nextWord();
477.                }
478.                while (words[nowNum].name == ",")
479.                {
480.                    nextWord();
481.                    if (words[nowNum].name == ")")
```

```
482.                 break;
483.             if (words[nowNum].type != "nid" && words[nowNum].type != "id")
484.             {
485.                 if (words[nowNum].name == ",")
486.                 {
487.                     printError("缺少 id", words[nowNum].column, words[nowNum].row);
488.                     --nowNum;
489.                 }
490.                 else if (words[nowNum + 1].name == ",")
491.                 {
492.                     printError("有错误的 id" + words[nowNum].name + "可能使用了保
    留字作为 id？", words[nowNum].column, words[nowNum].row);
493.                 }
494.             }
495.             else
496.             {
497.                 // 是 id,填写表
498.                 table.recordTable(words[nowNum].name, "VARIABLE", level + 1, 0, sp
    ace); // 因为形参实际是下一级的
499.                 space++;                                    // 空间增加
500.             }
501.             nextWord(); // 似乎有点粗糙，万一目标只少了一个，也会直接开始同步,
    试着解决一下
502.             // cout << words[nowNum].name << endl;
503.             while (words[nowNum].type == "nid" || words[nowNum].type == "id")
504.             {
505.                 if (words[nowNum].type == "id")
506.                     printError("id" + words[nowNum].name + "重复定义
    ", words[nowNum].column, words[nowNum].row);
507.                 printError("缺少逗号", words[nowNum].column, words[nowNum].row);
508.                 nextWord();
509.             }
510.         }
511.     }
512.
513.     if (words[nowNum].name != ")")
514.     {
515.         if (words[nowNum].name == ";")
516.         {
517.             --nowNum;
518.             printError("缺少 ) ", words[nowNum].column, words[nowNum].row);
519.         }
520.         else if (words[nowNum + 1].name == ";")
521.         {
```

```
522.                 printError("应该为 ) ， 出现了错误的单词：
    " + words[nowNum].name, words[nowNum].column, words[nowNum].row);
523.             }
524.         }
525.         nextWord();
526.         if (words[nowNum].name != ";")
527.         {
528.             if (first_set["<block>"].count(words[nowNum].name))
529.             {
530.                 --nowNum;
531.                 printError("缺少 ; ", words[nowNum].column, words[nowNum].row);
532.             }
533.             else if (first_set["<block>"].count(words[nowNum + 1].name))
534.             {
535.                 printError("应该为 ; ， 出现了错误的单词：
    " + words[nowNum].name, words[nowNum].column, words[nowNum].row);
536.             }
537.         }
538.     }
539.     table.table[level][proca].val = space - 3;
540.     // 在进入 block 前需要做的工作
541.     level += 1;
542.     nextWord(); // 指向 block 第一个词
543.     block();
544.     // 结束后恢复
545.     level -= 1;
546.
547.     // 销毁该层变量
548.     // 确保单一作用域 防止重复定义
549.     int t = table.table[level + 1].size();
550.     for (int i = 0; i < t; i++)
551.     {
552.         table.table[level + 1].pop_back();
553.     }
554.
555.     // 是分号继续 proc
556.     if (words[nowNum].name == ";") // 需要再思考一番 是 if 还是 while
557.     {
558.         nextWord();
559.         proc();
560.     }
561.     while (!follow_set["<proc>"].count(words[nowNum].name))
562.     {
563.         if (words[nowNum].type == "ProEnd")
```

```
564.                 return;
565.             printError("正在同步<proc>,跳过 " + words[nowNum].name, words[nowNum].column, words[nowNum].row);
566.             nextWord();
567.         }
568.     }
569.     void body()
570.     {
571.         if (words[nowNum].name != "begin")
572.         {
573.             if (first_set["<statement>"].count(words[nowNum].name) || first_set["<statement>"].count(words[nowNum].type))
574.             {
575.                 --nowNum;
576.                 printError("缺少 begin 关键字", words[nowNum].column, words[nowNum].row);
577.             }
578.             else if (first_set["<statement>"].count(words[nowNum + 1].name))
579.             {
580.                 printError("错误的关键字" + words[nowNum].name + "应该为begin", words[nowNum].column, words[nowNum].row);
581.             }
582.         }
583.         nextWord();
584.         statement();
585.
586.         while (words[nowNum].name == ";" || (first_set["<statement>"].count(words[nowNum].name) || first_set["<statement>"].count(words[nowNum].type)))
587.         {
588.             if (first_set["<statement>"].count(words[nowNum].name) || first_set["<statement>"].count(words[nowNum].type))
589.             {
590.                 --nowNum;
591.                 if (words[nowNum].name != ";")
592.                     printError("缺少;", words[nowNum].column, words[nowNum].row);
593.             }
594.             nextWord();
595.             statement();
596.         }
597.         // 处理 end
598.
599.         if (words[nowNum].name != "end")
600.         {
601.             if (follow_set["<body>"].count(words[nowNum].name))
```

```
602.            {
603.                --nowNum;
604.                printError("<body>缺少 end 关键字
    ", words[nowNum].column, words[nowNum].row);
605.            }
606.            else if (follow_set["<statement>"].count(words[nowNum + 1].name))
607.            {
608.                printError("<body>错误的关键字" + words[nowNum].name + "应该为
    end", words[nowNum].column, words[nowNum].row);
609.            }
610.        }
611.        nextWord();
612.        while (!follow_set["<body>"].count(words[nowNum].name))
613.        {
614.            printError("正在同步<body>,跳
    过 " + words[nowNum].name, words[nowNum].column, words[nowNum].row);
615.            nextWord();
616.        }
617.    }
618.    void statement()
619.    {
620.        if (words[nowNum].name == "#")
621.            return; // 防止忘记写 end
622.        if (words[nowNum].name == "end")
623.        {
624.            printError("不应该有分号", words[nowNum].column, words[nowNum].row);
625.            return; // 防止加了;，继续 statment 但是有 end
626.        }
627.
628.        while (!(first_set["<statement>"].count(words[nowNum].name) || first_set["<state
    ment>"].count(words[nowNum].type) || (follow_set["<statement>"].count(words[nowNu
    m].name) || follow_set["<statement>"].count(words[nowNum].type))))
629.        {
630.            printError("正在同步到<statement>开头，单词‘" + words[nowNum].name + "’
    可能多余", words[nowNum].column, words[nowNum].row);
631.            nextWord();
632.        }
633.        if (words[nowNum].type == "id" || words[nowNum].type == "nid" || words[nowNu
    m + 1].name == ":=")
634.        {
635.            // TEST;
636.            statement_id();
637.        }
638.        else if (words[nowNum].name == "if")
```

```
639.        {
640.            statement_if();
641.        }
642.        else if (words[nowNum].name == "while")
643.        {
644.            statement_while();
645.        }
646.        else if (words[nowNum].name == "call")
647.        {
648.            statement_call();
649.        }
650.        else if (words[nowNum].name == "begin")
651.        {
652.            body();
653.        }
654.        else if (words[nowNum].name == "read")
655.        {
656.            statement_read();
657.        }
658.        else if (words[nowNum].name == "write")
659.        {
660.            statement_write();
661.        }
662.        else if (first_set["<lexp>"].count(words[nowNum + 1].name))
663.        {
664.            statement_if();
665.        }
666.        else if (words[nowNum + 1].name == "(")
667.        {
668.            if (words[nowNum + 2].type == "id" || words[nowNum + 2].type == "nid")
669.            {
670.                statement_read();
671.            }
672.            else
673.            {
674.                statement_write();
675.            }
676.        }
677.        while (!(follow_set["<statement>"].count(words[nowNum].name) || follow_set["<s
    tatement>"].count(words[nowNum].type) || first_set["<statement>"].count(words[nowNu
    m].name) || first_set["<statement>"].count(words[nowNum].type)))
678.        {
679.            if (words[nowNum].name == "#")
680.                return; // 防止忘记写 end
```

```
681.            if (words[nowNum].name == "end")
682.            {
683.                printError("不应该有分号", words[nowNum].column, words[nowNum].row);
684.                return; // 防止加了;，继续 statment 但是有 end
685.            }
686.
687.            printError("正在同步<statement>,跳
    过 " + words[nowNum].name, words[nowNum].column, words[nowNum].row);
688.            nextWord();
689.        }
690.    }
691.    void statement_id()
692.    {
693.        int l, index;
694.        if (words[nowNum].type != "id")
695.        {
696.            if (words[nowNum].type == "nid")
697.            {
698.                printError("id " + words[nowNum].name + " 尚未定义
    ", words[nowNum].column, words[nowNum].row);
699.            }
700.            else if (words[nowNum].name == ":=")
701.            {
702.                --nowNum;
703.                printError("缺少 id", words[nowNum].column, words[nowNum].row);
704.            }
705.            else if (words[nowNum + 1].name == ":=")
706.            {
707.                printError("有错误的 id" + words[nowNum].name + "可能使用了保留字作
    为 id？ ", words[nowNum].column, words[nowNum].row);
708.            }
709.        }
710.        else
711.        {
712.            // 确实是 id
713.            l = table.find(words[nowNum].name)[0], index = table.find(words[nowNum].na
    me)[1]; // 第 i 层 第 j address
714.            if (index == -1)
715.            {
716.                printError2(words[nowNum].name + "没有定义
    ", words[nowNum].column, words[nowNum].row);
717.            }
718.            else if (table.table[l][index].type != "VARIABLE")
719.            {
```

```
720.                printError2(words[nowNum].name + "不是变量
    ", words[nowNum].column, words[nowNum].row);
721.            }
722.        }
723.        nextWord();
724.        if (words[nowNum].name != ":=")
725.        {
726.            if (first_set["<exp>"].count(words[nowNum].name) || first_set["<exp>"].count(
    words[nowNum].type))
727.            {
728.                printError("缺少赋值符号", words[nowNum].column, words[nowNum].row);
729.                --nowNum;
730.            }
731.            else if (first_set["<exp>"].count(words[nowNum + 1].name) || first_set["<exp>"].
    count(words[nowNum + 1].type))
732.            {
733.                printError("错误的赋值符号
    " + words[nowNum].name, words[nowNum].column, words[nowNum].row);
734.            }
735.            else
736.            {
737.                printError("错了，但不知道发生了什么
    " + words[nowNum].name, words[nowNum].column, words[nowNum].row);
738.            }
739.        }
740.        nextWord();
741.        exp();
742.        // 合法变量我们应该把 exp 产生的值存起来
743.        if (index != -1)
744.        {
745.            code.emit("STO", level - table.table[l][index].level, table.table[l][index].addr);
746.            // 容易证明的是只要到了正确 level 那么 addr 肯定是正确的，这是由于 addr
    生成的规律决定的
747.            // 为什么层差就是正确的地址是因为，在 call 当中会给定层差，而我们生成
    call 时传递的层差就是对的
748.        }
749.    }
750.    void statement_if()
751.    {
752.        if (words[nowNum].name != "if")
753.        {
754.            if (first_set["<lexp>"].count(words[nowNum].name))
755.            {
756.                --nowNum;
```

```
757.                printError("缺少 if", words[nowNum].column, words[nowNum].row);
758.            }
759.            else if (first_set["<lexp>"].count(words[nowNum + 1].name))
760.            {
761.                printError("有错误的关键字" + words[nowNum].name + "，应该为
    if", words[nowNum].column, words[nowNum].row);
762.            }
763.        }
764.        nextWord();
765.        // 交给 lexp 把一个判断操作结果值放到栈顶
766.        lexp();
767.        if (words[nowNum].name != "then")
768.        {
769.            if (first_set["<statement>"].count(words[nowNum].name))
770.            {
771.                --nowNum;
772.                printError("缺少 then", words[nowNum].column, words[nowNum].row);
773.            }
774.            else if (first_set["<statement>"].count(words[nowNum + 1].name))
775.            {
776.                printError("有错误的关键字" + words[nowNum].name + "，应该为
    then", words[nowNum].column, words[nowNum].row);
777.            }
778.        }
779.        int current = code.len(); // 为了之后回填
780.        code.emit("JPC", 0, 0);  // 非真跳到 else 或者 if 的后面
781.        nextWord();
782.        statement();
783.        code.code[current].a = code.len(); // 跳到 if 后面，没有 else 就不用加一
784.        if (words[nowNum].name == "else")
785.        {
786.            // 一样的回填，为了 if 为真时不在运行这里的 statement
787.            code.code[current].a = code.len() + 1; // 让 JPC 跳到 else 开头,因为还会生成下
    面 JMP 代码，所以要加 1，
788.            current = code.len();
789.            code.emit("JMP", 0, 0); // 这句实际上是 if 为真执行过后为了不执行 statement
    的
790.            nextWord();
791.            statement();
792.            code.code[current].a = code.len();
793.        }
794.    }
795.    void statement_while()
796.    {
```

```
797.        // 保存 code 开始语句,便于循环
798.        int whileStart = code.len();
799.        if (words[nowNum].name != "while")
800.        {
801.            if (first_set["<lexp>"].count(words[nowNum].name))
802.            {
803.                --nowNum;
804.                printError("缺少 while", words[nowNum].column, words[nowNum].row);
805.            }
806.            else if (first_set["<lexp>"].count(words[nowNum + 1].name))
807.            {
808.                printError("有错误的关键字" + words[nowNum].name + "，应该为
    while", words[nowNum].column, words[nowNum].row);
809.            }
810.        }
811.        // 依旧是 lexp 生成一个值放在栈顶
812.
813.        nextWord();
814.        lexp();
815.        if (words[nowNum].name != "do")
816.        {
817.            if (first_set["<statement>"].count(words[nowNum].name))
818.            {
819.                --nowNum;
820.                printError("缺少关键字
    do", words[nowNum].column, words[nowNum].row);
821.            }
822.            else if (first_set["<statement>"].count(words[nowNum + 1].name))
823.            {
824.                printError("有错误的关键字" + words[nowNum].name + "，应该为
    do", words[nowNum].column, words[nowNum].row);
825.            }
826.        }
827.        int current = code.len();
828.        code.emit("JPC", 0, 0);
829.        nextWord();
830.        statement();
831.        code.emit("JMP", 0, whileStart);  // 回到最初的起点
832.        code.code[current].a = code.len(); // 跳出循环
833.    }
834.    void statement_call()
835.    {
836.        int l, index;
837.        int callNum = 0;
```

```
838.        if (words[nowNum].name != "call")
839.        {
840.            if (words[nowNum].type == "id" || words[nowNum].type == "nid")
841.            {
842.                --nowNum;
843.                printError("缺少关键字
    call", words[nowNum].column, words[nowNum].row);
844.            }
845.            else if (words[nowNum + 1].type == "id" || words[nowNum + 1].type == "nid")
846.            {
847.                printError("有错误的关键字" + words[nowNum].name + "，应该为
    call", words[nowNum].column, words[nowNum].row);
848.            }
849.        }
850.        nextWord();
851.        if (words[nowNum].type != "id" && words[nowNum].type != "nid")
852.        {
853.            if (words[nowNum].name == "(")
854.            {
855.                --nowNum;
856.                printError("缺少 id", words[nowNum].column, words[nowNum].row);
857.            }
858.            else if (words[nowNum + 1].name == "(")
859.            {
860.                printError("有错误的 id" + words[nowNum].name + "可能使用了保留字作
    为 id？", words[nowNum].column, words[nowNum].row);
861.            }
862.        }
863.        else
864.        {
865.            l = table.find2(words[nowNum].name)[0], index = table.find(words[nowNum].n
    ame)[1];
866.            if (index == -1)
867.            {
868.                printError2(words[nowNum].name + "没有定义
    ", words[nowNum].column, words[nowNum].row);
869.            }
870.            else if (table.table[l][index].type != "PROCEDURE")
871.            {
872.                printError2(words[nowNum].name + "不是函数
    ", words[nowNum].column, words[nowNum].row);
873.            }
874.        }
875.        nextWord();
```

```
876.        if (words[nowNum].name != "(")
877.        {
878.            if (first_set["<exp>"].count(words[nowNum].name))
879.            {
880.                printError("缺少(", words[nowNum].column, words[nowNum].row);
881.                --nowNum;
882.            }
883.            else if (first_set["<exp>"].count(words[nowNum + 1].name))
884.            {
885.                printError("错误的符号" + words[nowNum].name + "应该为
    (", words[nowNum].column, words[nowNum].row);
886.            }
887.        }
888.        nextWord();
889.        int parameter = 3; // 从第三个开始传递形式参数
890.        // exp 最终会有一个值在栈顶的
891.        if (first_set["<exp>"].count(words[nowNum].name) || first_set["<exp>"].count(wor
    ds[nowNum].type))
892.        {
893.            exp();
894.            callNum += 1;
895.            code.emit("STO", -1, parameter++); //-1 代表传参
896.        }
897.        while (first_set["<exp>"].count(words[nowNum].name))
898.        {
899.            printError("缺少逗号", words[nowNum].column, words[nowNum].row);
900.            exp();
901.        }
902.        while (words[nowNum].name == ",")
903.        {
904.            nextWord();
905.            exp();
906.            callNum += 1;
907.            code.emit("STO", -1, parameter++); //-1 代表传参
908.            while (first_set["<exp>"].count(words[nowNum].name) || first_set["<exp>"].cou
    nt(words[nowNum].type))
909.            {
910.                printError("缺少逗号", words[nowNum].column, words[nowNum].row);
911.                nextWord();
912.            }
913.        }
914.        if (words[nowNum].name != ")")
915.        {
916.            if (follow_set["<statement>"].count(words[nowNum].name))
```

```
917.            {
918.                printError("缺少)", words[nowNum].column, words[nowNum].row);
919.                --nowNum;
920.            }
921.            else if (follow_set["<statement>"].count(words[nowNum + 1].name))
922.            {
923.                printError("错误的符号" + words[nowNum].name + "应该
    为)", words[nowNum].column, words[nowNum].row);
924.            }
925.        }
926.        // 确实是过程
927.        if (table.table[l][index].type == "PROCEDURE")
928.        {
929.            if (callNum != table.table[l][index].val)
930.            {
931.                char t[10];
932.                sprintf(t, "%d", table.table[l][index].val);
933.                printError2(table.table[l][index].name + "参数数目应该为
    " + t, words[nowNum].column, words[nowNum].row);
934.            }
935.            code.emit("CAL", level - table.table[l][index].level, table.table[l][index].addr);
936.        }
937.        else
938.        {
939.            wrong2 = true;
940.            printError2(words[nowNum].name + "不是过程名
    ", words[nowNum].column, words[nowNum].row);
941.        }
942.
943.        nextWord();
944.    }
945.    void statement_read()
946.    {
947.        int l, index;
948.        if (words[nowNum].name != "read")
949.        {
950.            if (words[nowNum].name == "(")
951.            {
952.                --nowNum;
953.                printError("缺少关键字
    read", words[nowNum].column, words[nowNum].row);
954.            }
955.            else if (words[nowNum + 1].name == "(")
956.            {
```

```
957.                printError("有错误的关键字" + words[nowNum].name + "，应该为
    read", words[nowNum].column, words[nowNum].row);
958.            }
959.        }
960.        nextWord();
961.        if (words[nowNum].name != "(")
962.        {
963.            if (words[nowNum].type == "id" || words[nowNum].type == "uid")
964.            {
965.                --nowNum;
966.                printError("缺少(", words[nowNum].column, words[nowNum].row);
967.            }
968.            else if (words[nowNum + 1].type == "id" || words[nowNum + 1].type == "uid")
969.            {
970.                printError("有错误的单词" + words[nowNum].name + "，应该为
    (", words[nowNum].column, words[nowNum].row);
971.            }
972.        }
973.        nextWord();
974.        if (words[nowNum].type != "id" && words[nowNum].type != "nid")
975.        {
976.            if (words[nowNum].name == "," || words[nowNum].name == ")")
977.            {
978.                --nowNum;
979.                printError("缺少 id", words[nowNum].column, words[nowNum].row);
980.            }
981.            else if (words[nowNum + 1].name == "," || words[nowNum + 1].name == ")")
982.            {
983.                printError("有错误的 id" + words[nowNum].name + "可能使用了保留字作
    为 id？", words[nowNum].column, words[nowNum].row);
984.            }
985.            else
986.            {
987.                printError("read 中非法的单词
    ", words[nowNum].column, words[nowNum].row);
988.            }
989.        }
990.        else
991.        {
992.            // 是 id 的话准备读入
993.            l = table.find(words[nowNum].name)[0], index = table.find(words[nowNum].na
    me)[1];
994.            if (index == -1)
995.            {
```

```
996.                printError2(words[nowNum].name + "没有定义
    ", words[nowNum].column, words[nowNum].row);
997.            }
998.            else if (table.table[l][index].type != "VARIABLE")
999.            {
1000.                printError2(words[nowNum].name + "不是变量
    ", words[nowNum].column, words[nowNum].row);
1001.               }
1002.               else
1003.               {
1004.                  // 确实找到了定义
1005.                  // cout<<words[nowNum].name<<" "<<l<<" "<<table.table[l][index].
    addr<<endl;
1006.                  code.emit("RED", 0, 0);                                    // 把一个数
    据读入栈顶
1007.                  code.emit("STO", level - table.table[l][index].level, table.table[l][index].a
    ddr); // 把栈顶数据存好
1008.               }
1009.            }
1010.            nextWord();
1011.            while (words[nowNum].type == "nid" || words[nowNum].type == "id")
1012.            {
1013.               // if (words[nowNum].type == "id") printError("id" + words[nowNum].na
    me + "重复定义", words[nowNum].column, words[nowNum].row);
1014.               printError("缺少逗号", words[nowNum].column, words[nowNum].row);
1015.               nextWord();
1016.            }
1017.            while (words[nowNum].name == ",")
1018.            {
1019.               nextWord();
1020.               if (words[nowNum].name == ")")
1021.                  break;
1022.               if (words[nowNum].type != "id" && words[nowNum].type != "nid")
1023.               {
1024.                  if (words[nowNum].name == "," || words[nowNum].name == ")")
1025.                  {
1026.                     --nowNum;
1027.                     printError("缺少 id", words[nowNum].column, words[nowNum].row);
1028.                  }
1029.                  else if (words[nowNum + 1].name == ",")
1030.                  {
1031.                     printError("有错误的 id" + words[nowNum].name + "可能使用了保
    留字作为 id？", words[nowNum].column, words[nowNum].row);
1032.                  }
```

```
1033.                 else
1034.                 {
1035.                     printError("read 中非法的单词
    ", words[nowNum].column, words[nowNum].row);
1036.                 }
1037.             }
1038.             else
1039.             {
1040.                 // 是 id 的话准备读入
1041.                 l = table.find(words[nowNum].name)[0], index = table.find(words[nowN
    um].name)[1];
1042.                 if (index == -1)
1043.                 {
1044.                     printError2(words[nowNum].name + "没有定义
    ", words[nowNum].column, words[nowNum].row);
1045.                 }
1046.                 else if (table.table[l][index].type != "VARIABLE")
1047.                 {
1048.                     printError2(words[nowNum].name + "不是变量
    ", words[nowNum].column, words[nowNum].row);
1049.                 }
1050.                 else
1051.                 {
1052.                     // 确实找到了定义
1053.                     // cout<<words[nowNum].name<<" "<<l<<" "<<table.table[l][level]
    .addr<<endl;
1054.                     code.emit("RED", 0, 0);                                    // 把一个
    数据读入栈顶
1055.                     code.emit("STO", level - table.table[l][index].level, table.table[l][index]
    .addr); // 把栈顶数据存好
1056.                 }
1057.             }
1058.             nextWord(); // 似乎有点粗糙，万一目标只少了一个，也会直接开始同步,
    试着解决一下
1059.             while (words[nowNum].type == "nid" || words[nowNum].type == "id")
1060.             {
1061.                 if (words[nowNum].type == "id")
1062.                     printError("id" + words[nowNum].name + "重复定义
    ", words[nowNum].column, words[nowNum].row);
1063.                 printError("缺少逗号", words[nowNum].column, words[nowNum].row);
1064.                 nextWord();
1065.             }
1066.         }
1067.         if (words[nowNum].name != ")")
```

```
1068.             {
1069.                 if (follow_set["<statement>"].count(words[nowNum].name))
1070.                 {
1071.                     printError("缺少)", words[nowNum].column, words[nowNum].row);
1072.                     --nowNum;
1073.                 }
1074.                 else if (follow_set["<statement>"].count(words[nowNum + 1].name))
1075.                 {
1076.                     printError("错误的符号" + words[nowNum].name + "应该
    为)", words[nowNum].column, words[nowNum].row);
1077.                 }
1078.                 else
1079.                 {
1080.                     printError("错误的符号" + words[nowNum].name + "应该
    为)", words[nowNum].column, words[nowNum].row);
1081.                 }
1082.             }
1083.             nextWord();
1084.             while (!(follow_set["<statement>"].count(words[nowNum].name) || follow_se
    t["<statement>"].count(words[nowNum].type)))
1085.             {
1086.                 printError("正在同步<statement>,跳
    过 " + words[nowNum].name, words[nowNum].column, words[nowNum].row);
1087.                 nextWord();
1088.             }
1089.             nextWord();
1090.         }
1091.         void statement_write()
1092.         {
1093.             int l, index;
1094.             if (words[nowNum].name != "write")
1095.             {
1096.                 if (words[nowNum].name == "(")
1097.                 {
1098.                     --nowNum;
1099.                     printError("缺少关键字
    write", words[nowNum].column, words[nowNum].row);
1100.                 }
1101.                 else if (words[nowNum + 1].name == "(")
1102.                 {
1103.                     printError("有错误的关键字" + words[nowNum].name + "，应该为
    write", words[nowNum].column, words[nowNum].row);
1104.                 }
1105.             }
```

```
1106.             nextWord();
1107.             if (words[nowNum].name != "(")
1108.             {
1109.                 if (first_set["<exp>"].count(words[nowNum].name))
1110.         {
1111.             --nowNum;
1112.             printError("缺少(", words[nowNum].column, words[nowNum].row);
1113.         }
1114.         else if (first_set["<exp>"].count(words[nowNum + 1].name))
1115.         {
1116.             printError("有错误的单词" + words[nowNum].name + "，应该为
    (", words[nowNum].column, words[nowNum].row);
1117.         }
1118.     }
1119.     nextWord();
1120.             exp();
1121.             // exp 以后会有一个数字在栈顶的，要输出的就是这个
1122.             code.emit("WRT", 0, 0);
1123.             while (first_set["<exp>"].count(words[nowNum].name))
1124.             {
1125.                 printError("缺少逗号", words[nowNum].column, words[nowNum].row);
1126.                 exp();
1127.             }
1128.             while (words[nowNum].name == ",")
1129.             {
1130.                 nextWord();
1131.                 exp();
1132.                 // TEST;
1133.                 // exp 以后会有一个数字在栈顶的，要输出的就是这个
1134.                 code.emit("WRT", 0, 0);
1135.                 while (first_set["<exp>"].count(words[nowNum].name))
1136.                 {
1137.                     printError("缺少逗号", words[nowNum].column, words[nowNum].row);
1138.                     nextWord();
1139.                 }
1140.             }
1141.             if (words[nowNum].name != ")")
1142.             {
1143.                 if (follow_set["<statement>"].count(words[nowNum].name))
1144.                 {
1145.                     printError("缺少)", words[nowNum].column, words[nowNum].row);
1146.                     --nowNum;
1147.                 }
1148.                 else if (follow_set["<statement>"].count(words[nowNum + 1].name))
```

```
1149.                 {
1150.                     printError("错误的符号" + words[nowNum].name + "应该
    为)", words[nowNum].column, words[nowNum].row);
1151.                 }
1152.             }
1153.             nextWord();
1154.         }
1155.         void exp()
1156.         {
1157.             bool flag = false;
1158.             if (words[nowNum].name == "+" || words[nowNum].name == "-")
1159.             {
1160.                 if (words[nowNum].name == "-")
1161.                     flag = true;
1162.                 nextWord();
1163.             }
1164.             term(); // 应该细化判断
1165.             if (flag)
1166.             {
1167.                 // 取反
1168.                 code.emit("OPR", 0, 1);
1169.             }
1170.             while (words[nowNum].name == "+" || words[nowNum].name == "-")
1171.             {
1172.                 int tmp = nowNum;
1173.                 nextWord();
1174.                 term();
1175.                 if (words[tmp].name == "+")
1176.                 {
1177.                     code.emit("OPR", 0, 2); // 相加
1178.                 }
1179.                 else
1180.                 {
1181.                     code.emit("OPR", 0, 3); // 相减
1182.                 }
1183.
1184.                 while (first_set["<term>"].count(words[nowNum].name))
1185.                 {
1186.                     printError("缺少
    <aop>", words[nowNum].column, words[nowNum].row);
1187.                     term();
1188.                 }
1189.             }
1190.             // while(!follow_set["<exp>"].count(words[nowNum].name)){
```

```
1191.             //  printError("正在同步<exp>,跳过 "+words[nowNum].name,words[nowNum].column,words[nowNum].row);
1192.             //  nextWord();
1193.             // }
1194.         }
1195.         void lexp()
1196.         {
1197.             if (words[nowNum].name == "odd")
1198.             {
1199.                 nextWord();
1200.                 // exp 会把一个要判断的放到栈顶的
1201.                 exp();
1202.                 code.emit("OPR", 0, 6); // 判断奇偶
1203.             }
1204.             else if (first_set["<exp>"].count(words[nowNum + 1].name) || first_set["<exp>"].count(words[nowNum + 1].type))
1205.             {
1206.                 printError("错误关键字" + words[nowNum].name + "，应该为odd", words[nowNum].column, words[nowNum].row);
1207.                 nextWord();
1208.                 exp();
1209.             }
1210.             else
1211.             {
1212.                 if (!(first_set["<exp>"].count(words[nowNum + 1].name) || first_set["<exp>"].count(words[nowNum + 1].type)))
1213.                 {
1214.                     if (words[nowNum].name == "=" || words[nowNum].name == "<>" || words[nowNum].name == "<" || words[nowNum].name == "<=" || words[nowNum].name == ">" || words[nowNum].name == ">=")
1215.                     {
1216.                         printError("缺少<exp>内容", words[nowNum].column, words[nowNum].row);
1217.                     }
1218.                     else
1219.                     {
1220.                         // TEST;
1221.                         exp();
1222.                     }
1223.                 }
1224.                 else
1225.                 {
1226.                     exp();
1227.                 }
```

```
1228.                 if (!(words[nowNum].name == "=" || words[nowNum].name == "<>" || wor
      ds[nowNum].name == "<" || words[nowNum].name == "<=" || words[nowNum].name ==
      ">" || words[nowNum].name == ">="))
1229.                 {
1230.                     if (first_set["<exp>"].count(words[nowNum].name) || first_set["<exp>"].
      count(words[nowNum].type))
1231.                     {
1232.                         --nowNum;
1233.                         printError("缺少<lop>符号
      ", words[nowNum].column, words[nowNum].row);
1234.                     }
1235.                     else if (first_set["<exp>"].count(words[nowNum + 1].name) || first_set["
      <exp>"].count(words[nowNum + 1].type))
1236.                     {
1237.                         printError("有错误的单词" + words[nowNum].name + "，应该为
      <lop>符号", words[nowNum].column, words[nowNum].row);
1238.                     }
1239.                     else
1240.                     {
1241.                         printError("有错误的单词" + words[nowNum].name + "，应该为
      <lop>符号", words[nowNum].column, words[nowNum].row);
1242.                     }
1243.                     // 虽然出错了但还是得继续语法分析的
1244.                     nextWord();
1245.                     exp();
1246.                 }
1247.                 else
1248.                 {
1249.                     string sign = words[nowNum].name; // 必定是比较符号之一，因为在
      else 中
1250.                     nextWord();
1251.                     exp();
1252.                     if (sign == "=")
1253.                     {
1254.                         code.emit("OPR", 0, 7);
1255.                     }
1256.                     else if (sign == "<>")
1257.                     {
1258.                         code.emit("OPR", 0, 8);
1259.                     }
1260.                     else if (sign == "<")
1261.                     {
1262.                         code.emit("OPR", 0, 9);
1263.                     }
```

```
1264.                 else if (sign == "<=")
1265.                 {
1266.                     code.emit("OPR", 0, 12);
1267.                 }
1268.                 else if (sign == ">")
1269.                 {
1270.                     code.emit("OPR", 0, 10);
1271.                 }
1272.                 else if (sign == ">=")
1273.                 {
1274.                     code.emit("OPR", 0, 11);
1275.                 }
1276.             }
1277.         }
1278.
1279.         while (!follow_set["<lexp>"].count(words[nowNum].name))
1280.         {
1281.             printError("正在同步<lexp>,跳
    过 " + words[nowNum].name, words[nowNum].column, words[nowNum].row);
1282.             nextWord();
1283.         }
1284.     }
1285.     void term()
1286.     {
1287.         factor();
1288.         // while(first_set["<factor>"].count(words[nowNum].name)||first_set["<fact
    or>"].count(words[nowNum].type)){
1289.         //  printError("缺少<mop>",words[nowNum].column,words[nowNum].row);
1290.         //  factor();
1291.         // }
1292.         while (words[nowNum].name == "*" || words[nowNum].name == "/")
1293.         {
1294.             int tmp = nowNum;
1295.             nextWord();
1296.             factor();
1297.             if (words[tmp].name == "*")
1298.             {
1299.                 code.emit("OPR", 0, 4); // 相乘
1300.             }
1301.             else
1302.             {
1303.                 code.emit("OPR", 0, 5); // 相除
1304.             }
1305.         }
```

```
1306.         }
1307.         void factor()
1308.         {
1309.             int l, index;
1310.             if (words[nowNum].type == "nid" || words[nowNum].type == "id" || words[no
    wNum].type == "integer")
1311.             {
1312.                 l = table.find(words[nowNum].name)[0], index = table.find(words[nowNu
    m].name)[1];
1313.                 if (index == -1)
1314.                 {
1315.                     if (words[nowNum].type == "integer")
1316.                     {
1317.                         // 虽然没有找到，不过发现是一个数字
1318.                         code.emit("LIT", 0, stoi(words[nowNum].name));
1319.                     }
1320.                     else
1321.                     {
1322.                         printError2(words[nowNum].name + "没有定义
    ", words[nowNum].column, words[nowNum].row);
1323.                     }
1324.                 }
1325.                 else
1326.                 {
1327.                     if (table.table[l][index].type == "CONSTANT") // 常量
1328.                     {
1329.                         code.emit("LIT", 0, table.table[l][index].val);
1330.                     }
1331.                     else if (table.table[l][index].type == "VARIABLE") // 变量
1332.                     {
1333.                         code.emit("LOD", level - table.table[l][index].level, table.table[l][inde
    x].addr);
1334.                     }
1335.                     else
1336.                     {
1337.                         printError2(words[nowNum].name + "为过程名
    ", words[nowNum].column, words[nowNum].row);
1338.                     }
1339.                 }
1340.                 nextWord();
1341.                 return;
1342.             }
1343.             else if (words[nowNum].name == "(")
1344.             {
```

```
1345.                 // 左括号进入表达式
1346.                 nextWord();
1347.                 exp();
1348.                 if (words[nowNum].name != ")")
1349.                 {
1350.                     if (follow_set["<factor>"].count(words[nowNum].name))
1351.                     {
1352.                         printError("缺少)", words[nowNum].column, words[nowNum].row);
1353.                         --nowNum;
1354.                     }
1355.                     else if (follow_set["<factor>"].count(words[nowNum + 1].name))
1356.                     {
1357.                         printError("错误的符号" + words[nowNum].name + "应该
    为)", words[nowNum].column, words[nowNum].row);
1358.                     }
1359.                 }
1360.                 nextWord();
1361.             }
1362.         }
1363.     };
```

4.2.4. 解释器

```
stack = [0 for i in range(0, 8000)] # 数据栈 前三个 0 是主函数的 SL DL RA
code = [] # 存放代码


# 根据当前 B 的值和 level 层差获取 SL 的值
# 因为生成代码的时候就是给定层差，而每个层差的上一层就是定义它的，所以是可行的
def get_sl(B, level):
    res_B = B
    # 由当前的 basic 去找最近的直接外层，B 是沿着静态链找的，因为要找最新的
    # 只要保证每一个静态链都是对的，就可以这么找到了，而递推计算，当 level 为 0 时，静态链就是自身，level 为 1 时，会去找 0……
    while level > 0:
        res_B = stack[res_B]
        level -= 1
    return res_B


def add_code(f, l, a):
    operation = dict()
    operation["F"] = f
    operation["L"] = l
    operation["A"] = a
    code.append(operation)


def interpreter():
    B = 0 # 基址寄存器
    T = 0 # 栈顶寄存器
    I = None # 存放要执行的代码
    P = 0 # 下地址寄存器
    # 开始执行
    I = code[P]
    P += 1
    while P != 0: # 因为退出程序有一个 OPR 0 0 的，所以看到 P 要到 0 就说明退出了
        if I["F"] == "JMP": # 直接跳转到对应指令
            P = I["A"]
        elif I["F"] == "JPC":
            if stack[T] == 0: # 栈顶值为 0 才跳转
                P = I["A"]
            T -= 1 # 无论是否跳转都要去除栈顶的值
        elif I["F"] == "INT":
```

```
            T += I["A"] - 1  # 开辟空间
        elif I["F"] == "LOD":
            T += 1
            stack[T] = stack[
                get_sl(B, I["L"]) + I["A"]
            ] # 到了那层 找到真正基地址 再加 addr
        elif I["F"] == "STO":
            if I["L"] == -1:
                stack[T + I["A"]] = stack[T]
                T -= 1
            else:
                stack[get_sl(B, I["L"]) + I["A"]] = stack[T]
                T -= 1
        elif I["F"] == "LIT":
            T += 1
            stack[T] = I["A"]
        elif I["F"] == "CAL": # 函数调用
            T += 1
            stack[T] = get_sl(B, I["L"]) # 静态链
            stack[T + 1] = B # 动态链
            stack[T + 2] = P # 返回地址-当前的下一条地址
            B = T # 新的基地址
            P = I["A"] # 注意 pro 生成时填写的都是绝对地址 code.len
        elif I["F"] == "OPR":
            if I["A"] == 0: # 函数返回
                T = B - 1
                P = stack[T + 3]
                B = stack[T + 2]
            elif I["A"] == 1: # 取反操作
                stack[T] = -stack[T]
            elif I["A"] == 2: # 加法
                T -= 1
                stack[T] = stack[T] + stack[T + 1]
            elif I["A"] == 3: # 减法
                T -= 1
                stack[T] = stack[T] - stack[T + 1]
            elif I["A"] == 4: # 乘法
                T -= 1
                stack[T] = stack[T] * stack[T + 1]
            elif I["A"] == 5: # 除法
                T -= 1
                stack[T] = int(stack[T] // stack[T + 1])
            elif I["A"] == 6: # odd 奇偶
                stack[T] = int(stack[T] % 2)
```

```
85.             elif I["A"] == 7:  # ==
86.                 T -= 1
87.                 stack[T] = int(stack[T] == stack[T + 1])
88.             elif I["A"] == 8:  # <>
89.                 T -= 1
90.                 stack[T] = int(stack[T] != stack[T + 1])
91.             elif I["A"] == 9:  # <
92.                 T -= 1
93.                 stack[T] = int(stack[T] < stack[T + 1])
94.             elif I["A"] == 10:  # >
95.                 T -= 1
96.                 stack[T] = int(stack[T] > stack[T + 1])
97.             elif I["A"] == 11:  # >=
98.                 T -= 1
99.                 stack[T] = int(stack[T] >= stack[T + 1])
100.            elif I["A"] == 12:  # <=
101.                T -= 1
102.                stack[T] = int(stack[T] <= stack[T + 1])
103.        elif I["F"] == "RED":
104.            T += 1
105.            stack[T] = int(input("输入： "))
106.        elif I["F"] == "WRT":
107.            print("栈顶输出： ", stack[T])
108.            T -= 1
109.        # print(P, stack[:T+8], T)
110.        I = code[P]  # 获取下一条指令
111.    # print(I)
112.        if P == 0:  # 因为退出程序有一个 OPR 0 0 的，所以看到 P 要到 0 就说明退出
    了
113.            break
114.    P += 1  # 默认 P+1 获取下一条指令 除非跳转
```

4.2.5. 界面/可视化设计

```
from PyQt5.QtWidgets import *
from PyQt5.QtGui import *
from PyQt5.QtCore import *
import tkinter as tk
from tkinter import ttk
import sys
import subprocess

stack = [0 for i in range(0, 8000)] # 数据栈 前三个 0 是主函数的 SL DL RA
code = [] # 存放代码

newbase = [0]
setfile = False
output_edit = None
font = QFont()
font.setPointSize(12) # 设置字体大小为 12


def add_code(f, l, a):
    operation = dict()
    operation["F"] = f
    operation["L"] = l
    operation["A"] = a
    code.append(operation)

    # 由当前的 basic 去找最近的直接外层，B 是沿着静态链找的，因为要找最新的
    # 只要保证每一个静态链都是对的，就可以这么找到了，而递推计算，当 level 为 0 时，静态链就是自身，level 为 1 时，会去找 0……


def get_sl(B, level):
    res_B = B
    while level > 0:
        res_B = stack[res_B]
        level -= 1
    return res_B


class Interpreter:
    def __init__(self):
        self.B = 0 # 基址寄存器
        self.T = 0 # 栈顶寄存器
        self.P = 0 # 下地址寄存器
```

```
        # 开始执行
        self.I = code[self.P]
        self.P += 1
        global window

    def start(self):
        self.step()  # 显式调用 step() 方法开始执行

    def end(self):
        return self.P == 0

    #  根据当前 B 的值和 level 层差获取 SL 的值
    # 因为生成代码的时候就是给定层差，而每个层差的上一层就是定义它的，所以是可行的

    def step(self):
        if (
            not self.end()
        ): # 因为退出程序有一个 OPR 0 0 的，所以看到 self.P 要到 0 就说明退出了
            if self.I["F"] == "JMP":  # 直接跳转到对应指令
                self.P = self.I["A"]
            elif self.I["F"] == "JPC":
                if stack[self.T] == 0:  # 栈顶值为 0 才跳转
                    self.P = self.I["A"]
                self.T -= 1  # 无论是否跳转都要去除栈顶的值
            elif self.I["F"] == "INT":
                self.T += self.I["A"] - 1  # 开辟空间
            elif self.I["F"] == "LOD":
                self.T += 1
                stack[self.T] = stack[
                    get_sl(self.B, self.I["L"]) + self.I["A"]
                ] # 到了那层 找到真正基地址 再加 addr
            elif self.I["F"] == "STO":
                if self.I["L"] == -1:
                    stack[self.T + self.I["A"]] = stack[self.T]
                    self.T -= 1
                else:
                    # print(get_sl(self.B, self.I['L']))
                    stack[get_sl(self.B, self.I["L"]) + self.I["A"]] = stack[self.T]
                    self.T -= 1
            elif self.I["F"] == "LIT":
                self.T += 1
                stack[self.T] = self.I["A"]
            elif self.I["F"] == "CAL":  # 函数调用
```

```
86.             self.T += 1
87.             stack[self.T] = get_sl(self.B, self.I["L"])  # 静态链
88.             stack[self.T + 1] = self.B  # 动态链
89.             stack[self.T + 2] = self.P  # 返回地址-当前的下一条地址
90.             self.B = self.T  # 新的基地址
91.             self.P = self.I["A"]  # 注意 pro 生成时填写的都是绝对地址 code.len
92.             newbase.append(self.B)
93.         elif self.I["F"] == "OPR":
94.             if self.I["A"] == 0:  # 函数返回
95.                 self.T = self.B - 1
96.                 self.P = stack[self.T + 3]
97.                 self.B = stack[self.T + 2]
98.                 newbase.pop()
99.             elif self.I["A"] == 1:  # 取反操作
100.                stack[self.T] = -stack[self.T]
101.            elif self.I["A"] == 2:  # 加法
102.                self.T -= 1
103.                stack[self.T] = stack[self.T] + stack[self.T + 1]
104.            elif self.I["A"] == 3:  # 减法
105.                self.T -= 1
106.                stack[self.T] = stack[self.T] - stack[self.T + 1]
107.            elif self.I["A"] == 4:  # 乘法
108.                self.T -= 1
109.                stack[self.T] = stack[self.T] * stack[self.T + 1]
110.            elif self.I["A"] == 5:  # 除法
111.                self.T -= 1
112.                stack[self.T] = int(stack[self.T] // stack[self.T + 1])
113.            elif self.I["A"] == 6:  # odd 奇偶
114.                stack[self.T] = int(stack[self.T] % 2)
115.            elif self.I["A"] == 7:  # ==
116.                self.T -= 1
117.                stack[self.T] = int(stack[self.T] == stack[self.T + 1])
118.            elif self.I["A"] == 8:  # <>
119.                self.T -= 1
120.                stack[self.T] = int(stack[self.T] != stack[self.T + 1])
121.            elif self.I["A"] == 9:  # <
122.                self.T -= 1
123.                stack[self.T] = int(stack[self.T] < stack[self.T + 1])
124.            elif self.I["A"] == 10:  # >
125.                self.T -= 1
126.                stack[self.T] = int(stack[self.T] > stack[self.T + 1])
127.            elif self.I["A"] == 11:  # >=
128.                self.T -= 1
129.                stack[self.T] = int(stack[self.T] >= stack[self.T + 1])
```

```
            elif self.I["A"] == 12:  # <=
                self.T -= 1
                stack[self.T] = int(stack[self.T] <= stack[self.T + 1])
        elif self.I["F"] == "RED":
            self.T += 1
            # stack[self.T] = int(input("输入： "))
            window.display_TEXT("")
            output_edit.setText("请输入数字")
            # print("输入数字！")
        elif self.I["F"] == "WRT":
            # print("栈顶输出：", stack[self.T])
            window.output_STACK(str(stack[self.T]))
            self.T -= 1
        # print(self.P, stack[:T+8], T)
        self.I = code[self.P]  # 获取下一条指令
        # print(self.I)
        if (
            self.P == 0
        ):  # 因为退出程序有一个 OPR 0 0 的，所以看到 self.P 要到 0 就说明退出了
            return
        self.P += 1  # 默认 self.P+1 获取下一条指令 除非跳转


class MainWindow(QMainWindow):
    def __init__(self):
        super().__init__()
        self.file_name = None
        self.output_file_name = "D:\cppcode\complier_course_design\ouputerror.txt"
        self.setWindowTitle("PL/0 COMPLIER MADE BY NUAA")
        self.setFixedSize(1200, 960)

        self.scene = QGraphicsScene(self)
        self.view = QGraphicsView(self.scene, self)
        self.view.setGeometry(0, 0, 400, 960)
        self.interpreter = None

        self.start_button = QPushButton("运行", self)
        self.start_button.move(230, 10)
        self.start_button.clicked.connect(self.begin)

        self.next_button = QPushButton("下一条", self)
        self.next_button.move(10, 50)
        self.next_button.clicked.connect(self.handle_next)

```

```
174.        self.text_edit = QTextEdit(self)
175.        self.text_edit.setFont(font)
176.        self.text_edit.setFixedSize(400, 450)  # 固定大小
177.        self.text_edit.move(400, 30)
178.        self.text_edit.setReadOnly(True)
179.
180.        self.label3 = QLabel("源代码", self)
181.        self.label3.setFont(
182.            QFont("微软雅黑", 12, QFont.Bold)
183.        ) # 设置字体为微软雅黑，大小 12，粗体
184.        self.label3.move(400, 0)
185.        self.label3.setFixedSize(400, 30)  # 固定大小
186.        self.label3.setWordWrap(True)
187.        self.label3.setStyleSheet("background-color: lightblue;")
188.
189.        self.label2 = QLabel("问题", self)
190.        self.label2.move(400, 480)
191.        self.label2.setFont(
192.            QFont("微软雅黑", 12, QFont.Bold)
193.        ) # 设置字体为微软雅黑，大小 12，粗体
194.        self.label2.setFixedSize(400, 30)  # 固定大小
195.        self.label2.setWordWrap(True)
196.        self.label2.setStyleSheet("background-color: lightblue;")
197.
198.        self.output_edit = QTextEdit(self)
199.        self.output_edit.setFont(font)
200.        self.output_edit.setFixedSize(400, 450)  # 固定大小
201.        self.output_edit.move(400, 510)
202.        self.output_edit.setReadOnly(True)
203.        global output_edit
204.        output_edit = self.output_edit
205.
206.        self.label4 = QLabel("目标代码", self)
207.        self.label4.move(800, 0)
208.        self.label4.setFont(
209.            QFont("微软雅黑", 12, QFont.Bold)
210.        ) # 设置字体为微软雅黑，大小 12，粗体
211.        self.label4.setFixedSize(400, 30)  # 固定大小
212.        self.label4.setWordWrap(True)
213.        self.label4.setStyleSheet("background-color: lightblue;")
214.
215.        self.code_edit = QTextEdit(self)
216.        self.code_edit.setFont(font)
217.        self.code_edit.setFixedSize(400, 450)  # 固定大小
```

```
        self.code_edit.move(800, 30)
        self.code_edit.setReadOnly(True)

        self.label5 = QLabel("词法单元", self)

        self.label5.setFixedSize(400, 30)  # 固定大小
        self.label5.move(800, 480)
        self.label5.setFont(
            QFont("微软雅黑", 12, QFont.Bold)
        ) # 设置字体为微软雅黑，大小 12，粗体
        self.label5.setWordWrap(True)
        self.label5.setStyleSheet("background-color: lightblue;")

        self.word_edit = QTextEdit(self)
        self.word_edit.setFont(font)
        self.word_edit.setFixedSize(400, 450)  # 固定大小
        self.word_edit.move(800, 510)
        self.word_edit.setReadOnly(True)

        self.open_button = QPushButton("打开文件", self)
        self.open_button.move(10, 10)
        self.open_button.clicked.connect(self.open_file)

        self.com_button = QPushButton("编译", self)
        self.com_button.move(120, 10)
        self.com_button.clicked.connect(self.complier)

        self.stop_button = QPushButton("暂停", self)
        self.stop_button.move(10, 150)
        self.stop_button.clicked.connect(self.handle_stop)

        self.label = QLabel("", self)
        self.label.move(280, 10)

        self.label2 = QLabel("栈顶输出： ", self)
        self.label2.setFont(QFont("Arial", 10))
        self.label2.move(10, 100)
        self.label2.resize(400, 60)
        self.label2.setWordWrap(True)

        self.data_edit = QLineEdit(self)
        self.data_edit.move(120, 50)

        self.input_button = QPushButton("输入", self)
```

```
        self.input_button.move(230, 50)
        self.input_button.clicked.connect(self.handle_input)

        self.send_time = QTimer(self)

        self.info = ""
        self.output = ""

    def display(self):
        pen = QPen(QColor(0, 100, 0))
        font = QFont("Arial", 10)
        y = 2000
        for i in range(self.interpreter.T + 1):
            self.scene.addRect(50, y, 40, 25, pen)
            # 绘制栈中内容
            text = self.scene.addText(str(stack[i]), font)
            text.setDefaultTextColor(QColor(0, 0, 0))
            text.setPos(53, y)
            # 绘制标号
            index = self.scene.addText(str(i), font)
            index.setDefaultTextColor(QColor(0, 0, 0))
            if i < 10:
                index.setPos(25, y)
            else:
                index.setPos(22, y)
            if i in newbase:
                sl = self.scene.addText("静态链：", font)
                sl.setDefaultTextColor(QColor(0, 0, 0))
                sl.setPos(-50, y)
                dl = self.scene.addText("动态链：", font)
                dl.setDefaultTextColor(QColor(0, 0, 0))
                dl.setPos(-50, y - 40)
                ra = self.scene.addText("返回地址：", font)
                ra.setDefaultTextColor(QColor(0, 0, 0))
                ra.setPos(-63, y - 80)
            y -= 40
        # 绘制指令信息
        # 这里-1 是在最后会自动加一,
        if not self.interpreter.end():
            order = (
                "将要执行 "
                + str(self.interpreter.P - 1)
                + ": "
                + self.interpreter.I["F"]
```

```
            + " "
            + str(self.interpreter.I["L"])
            + " "
            + str(self.interpreter.I["A"])
        )
        text2 = self.scene.addText(order, font)
        text2.setDefaultTextColor(QColor(0, 0, 0))
        text2.setPos(100, 2000)
    else:
        output_edit.setText("程序运行结束")
    # 绘制提示
    if self.info != "":
        self.label.setText(self.info)
        self.info = ""
    else:
        self.label.setText("")
    # 绘制栈顶输出
    if self.output != "":
        self.label2.setText("栈顶输出： " + self.output)

    else:
        self.label2.setText("栈顶输出： ")

def open_file(self):
    # 打开文件对话框，选择 txt 文件
    options = QFileDialog.Options()
    self.file_name, _ = QFileDialog.getOpenFileName(
        self, "打开文件", "", "Text Files (*.txt);;All Files (*)", options=options
    )
    if self.file_name:
        try:
            # 读取文件内容并显示在 text_edit 中
            with open(self.file_name, "r", encoding="utf-8") as file:
                file_content = file.readlines()
                formatted_content = ""
                for idx, line in enumerate(file_content):
                    formatted_content += f"{idx + 1:3}: {line}"  # 给每行添加行号
                self.text_edit.setText(
                    formatted_content
                ) # 将格式化后的内容显示在 text_edit 中
        except Exception as e:
            self.text_edit.setText(f"无法打开文件: {e}")
    self.scene.clear()
    self.data_edit.clear()
```

```
        self.output_edit.clear()
        self.word_edit.clear()
        self.code_edit.clear()
        # print(file_name)

    def complier(self):
        code.clear()
        # print(self.file_name)
        process = subprocess.Popen(
            ["./main"], # C++ 可执行文件
            stdin=subprocess.PIPE, # 启用标准输入管道
            stdout=subprocess.PIPE, # 启用标准输出管道
            stderr=subprocess.PIPE,
        )
        stdout, stderr = process.communicate(input=self.file_name.encode())
        with open(self.output_file_name, "r") as file:
            file_content = file.read()
            self.output_edit.setText(file_content)
        origin = []
        with open("code.txt", "r") as f:
            origin = f.readlines()
        for i in range(len(origin)):
            origin[i] = origin[i][:-1]
            x = origin[i].split()
            add_code(x[0], int(x[1]), int(x[2]))
        self.interpreter = Interpreter()
        if stderr.decode() == "1":
            self.begin()
            self.show_code()
            self.show_word()

    def display_TEXT(self, s):
        # 必须先输入才可以继续下一步
        self.next_button.setEnabled(False)
        self.info = s

    def output_STACK(self, s):
        self.output = self.output + s + " "
        # print(self.output)

    def begin(self):
        global newbase
        newbase.clear()
        newbase.append(0)
```

```
394.         self.output = ""
395.         self.interpreter.B = 0  # 基址寄存器
396.         self.interpreter.T = 0  # 栈顶寄存器
397.         self.interpreter.P = 0  # 下地址寄存器
398.         # 开始执行
399.         self.interpreter.I = code[self.interpreter.P]
400.         self.interpreter.P += 1
401.         self.interpreter.step()
402.         self.scene.clear()
403.         self.output_edit.clear()
404.         self.display()
405.
406.         self.timeIntervals = 1
407.         self.send_time.start(self.timeIntervals)
408.         # 给 QTimer 设定一个时间，每到达这个时间一次就会调用一次该方法
409.         self.send_time.timeout.connect(self.handle_next)
410.
411.     def handle_next(self):
412.         # print(self.interpreter.T)
413.         if not self.next_button.isEnabled():
414.             return
415.         if not self.interpreter.end():
416.             self.data_edit.clear()
417.             self.interpreter.step()
418.             self.scene.clear()
419.             self.display()
420.         else:
421.             self.send_time.stop()
422.             self.start_button.setEnabled(True)
423.
424.     def handle_input(self):
425.         data = self.data_edit.text()
426.         if data != "":
427.             stack[self.interpreter.T] = int(data)
428.             self.scene.clear()
429.             self.display()
430.             self.data_edit.clear()
431.             self.next_button.setEnabled(True)
432.
433.     def handle_stop(self):
434.         if self.send_time.isActive():
435.             self.send_time.stop()
436.         else:
437.             self.send_time.start()
```

```
438.
439.     def show_code(self):
440.         with open("code.txt", "r", encoding="utf-8") as file:
441.             file_content = file.readlines()
442.             formatted_content = ""
443.             for idx, line in enumerate(file_content):
444.                 formatted_content += f"{idx:3}: {line}"  # 给每行添加行号
445.             self.code_edit.setText(
446.                 formatted_content
447.             ) # 将格式化后的内容显示在 text_edit 中
448.
449.     def show_word(self):
450.         with open("WordAnalyse.txt", "r", encoding="utf-8") as file:
451.             file_content = file.readlines()
452.             formatted_content = ""
453.             for idx, line in enumerate(file_content):
454.                 formatted_content += f"{idx + 1:3}: {line}"
455.             self.word_edit.setText(
456.                 formatted_content
457.             ) # 将格式化后的内容显示在 text_edit 中
458.
459.
460. if __name__ == "__main__":
461.     app = QApplication(sys.argv)
462.     window = MainWindow()
463.     window.show()
464.     sys.exit(app.exec_())
```

5. 系统测试

5.1. 错误测试 1

语法语义错误测试。

5.1.1. 错误程序

```
1.  progrom NUAA nuaa nuAA;
2.  const a:=123,b:=213 c:=132;
3.  var d e f g
4.  procedure (get ha sa ,a);
5.  begin
6.   g:=1
7.  end
8.  begin
9.   d:=2;
10.  e=1
11.  f:=3;
12.  read(x+y);
13.  while m>n do
14.   n := m+1
15.  if a(b then then then
16.   f=7
17.  else
18.   e:=8;
19. end
```

5.1.2. 结果分析

图 5.1 测试 1 程序执行结果

由执行结果可见，系统能正确识别出所有错误，满足“遇到的错误给出在源程序中出错的位置和错误性质”，同时在识别到相应错误后，能够继续执行下去。

5.2. 错误测试 2

参数传递测试。

5.2.1. 错误程序

```
1.  program huiwen;
2.  var a,b,numSum,lenreturn,i,flag;
3.  procedure sum(a,b);
4.  begin
5.     numSum:=a+b;
6.     a:=99;
7.     b:=99;
8.     write(numSum)
9.  end;
10. procedure sum2();
11. begin
12.    write(a+b)
13. end;
14. procedure isHuiWen(x);
15. var left,right,tmp;
16. procedure len(x);
17. var sum;
18. begin
19.    sum:=1;
20.    while x>0 do
21.  begin
22.   sum:=sum*10;
23.   x:=x/10
24.  end;
25.  lenreturn:=sum/10
26. end
27. begin
28.    if x<10 then flag:=1 else
29.  begin
30.   call len(x);
31.   right:=x-x/10*10;
32.   left:=x/lenreturn;
33.   if left<>right then flag:=0 else
34.   begin
35.      tmp:=x-left*lenreturn-right;
36.    call isHuiWen(tmp/10)
37.   end
38.  end
39. end
40. begin
```

```
    i:=1;
    read(a,b);
    call sum(a);
    write(a,b);
    call sum2(a,b);
while i<=numSum do
    begin
        call isHuiWen(i);
        if flag=1 then write(i);
        i:=i+1
    end
end
```

5.2.2. 结果分析

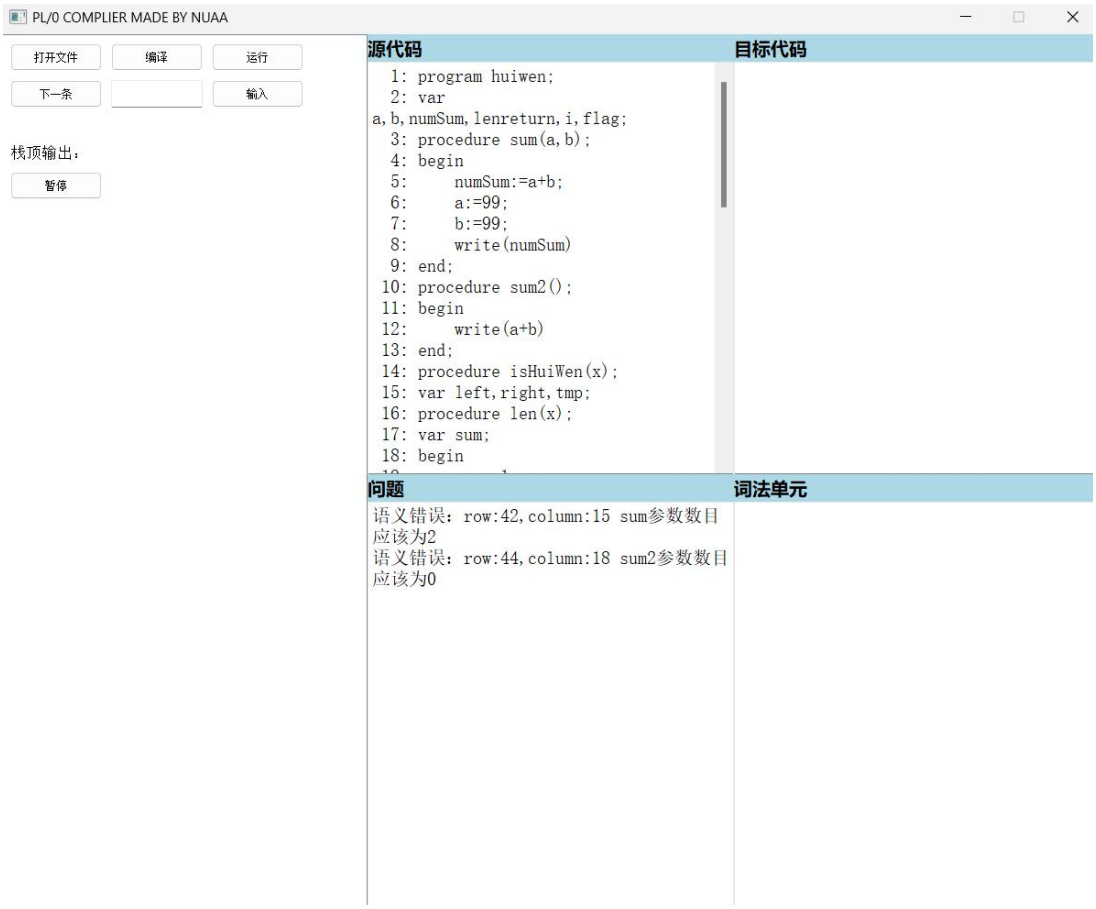


图 5.2 测试 2 程序执行结果

由上述结果可看出，程序能够正确识别到参数传递错误，并且识别为语义错误。

5.3. 求和测试

求 1~10 所有自然数之和。

5.3.1. 测试程序

```
1.  program id;
2.  const m:=1, n:=10;
3.  var x,y;
4.  procedure addALL();
5.     var a,b,c;
6.     begin
7.        a:=x; b:=y;c:=0;
8.        while b>=m do
9.           begin
10.             c:=c+b;
11.           b:=b-1
12.          end;
13.       if c>0 then
14.          write(c)
15. end
16. begin
17.    x:=m; y:=n;
18.    call addALL()
19. end
```

5.3.2. 结果分析

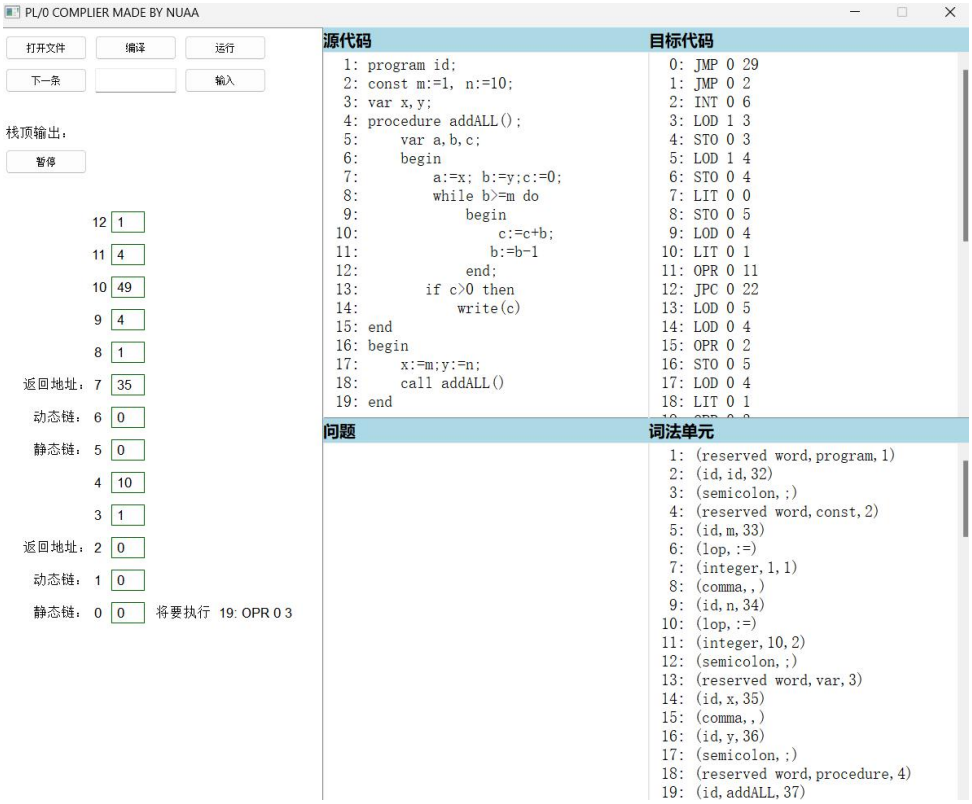


图 5.3 测试 3 程序运行过程图

栈顶最终输出 55，由上述结果可知，该系统能正确执行输出正确结果，并且能够显示目标代码，并且能够很清晰地看见目标代码的执行过程。

5.4. 过程嵌套测试

5.4.1. 测试程序

```
1.  program a;
2.  var x;
3.  procedure B(n);
4.      procedure C(n);
5.          procedure E(n);
6.              procedure F(n);
7.              begin
8.                  x := n-1
9.              end
10.         begin
11.             x := n-1;
12.             call F(x)
13.         end
14.     begin
15.         x:= n-1;
16.         call E(x)
17.     end;
18.     procedure D(n);
19.     begin
20.         x:= n-1;
21.         call C(x)
22.     end
23. begin
24.     x:= n-1;
25.     call D(x)
26. end
27.
28. begin
29.     x := 5;
30.     call B(x);
31.     write(x)
32. end
```

5.4.2. 结果分析

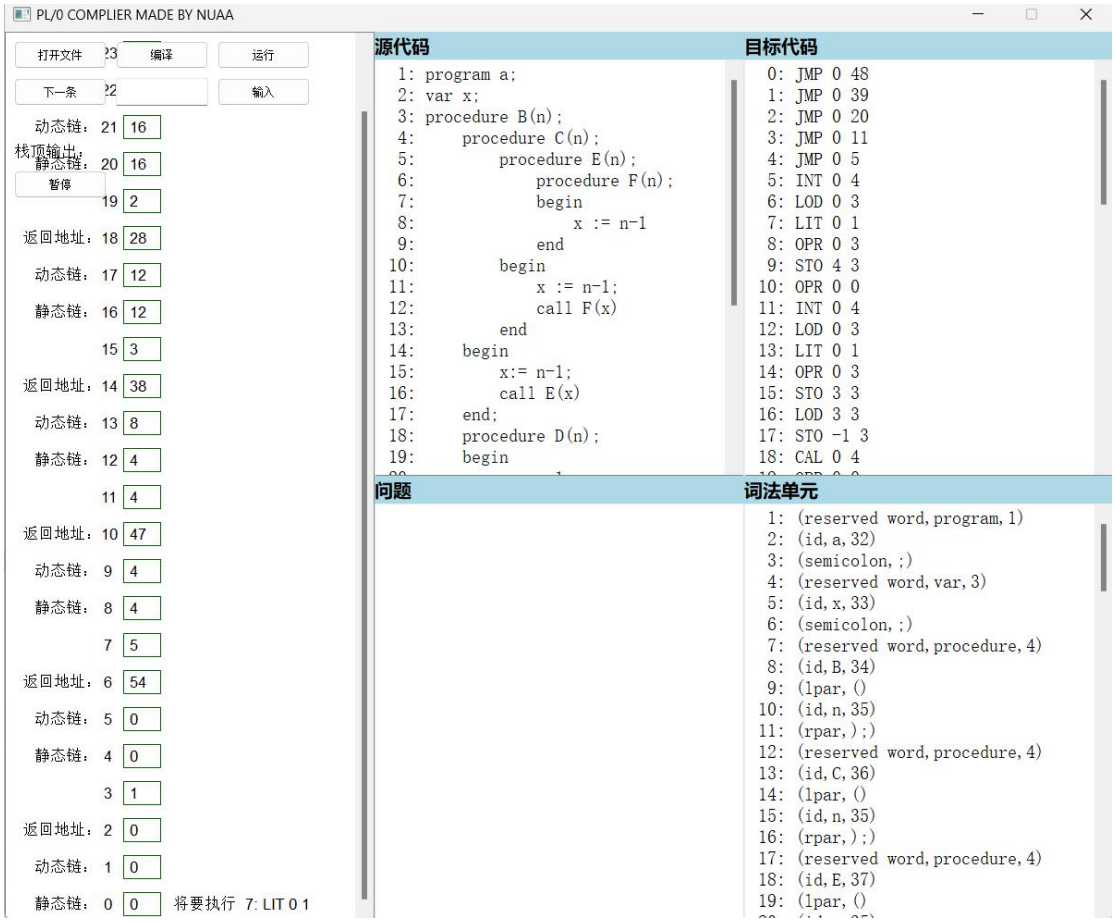


图 5.4 测试 4 程序运行过程图

栈顶最终输出 0.，由上述结果可知，该系统能正确执行输出正确结果，说明该系统能正确处理过程嵌套的情况。

5.5. 递归测试

5.5.1. 测试程序

```
1.  program fibonacci;
2.  const index:=10;
3.  var return,i;
4.  procedure fib(x);
5.  var sum;
6.  begin
7.     sum:=0;
8.     if x<2 then
9.        return:=x
10.    else
11.    begin
12.       call fib(x-1);
13.       sum:=sum+return;
14.       call fib(x-2);
15.       sum:=sum+return;
16.       return:=sum
17.    end
18. end
19. begin
20.    i:=1;
21.    while i<=index do
22.    begin
23.       call fib(i);
24.       write(return);
25.       i:=i+1
26.    end
27. end
```

5.5.2. 结果分析

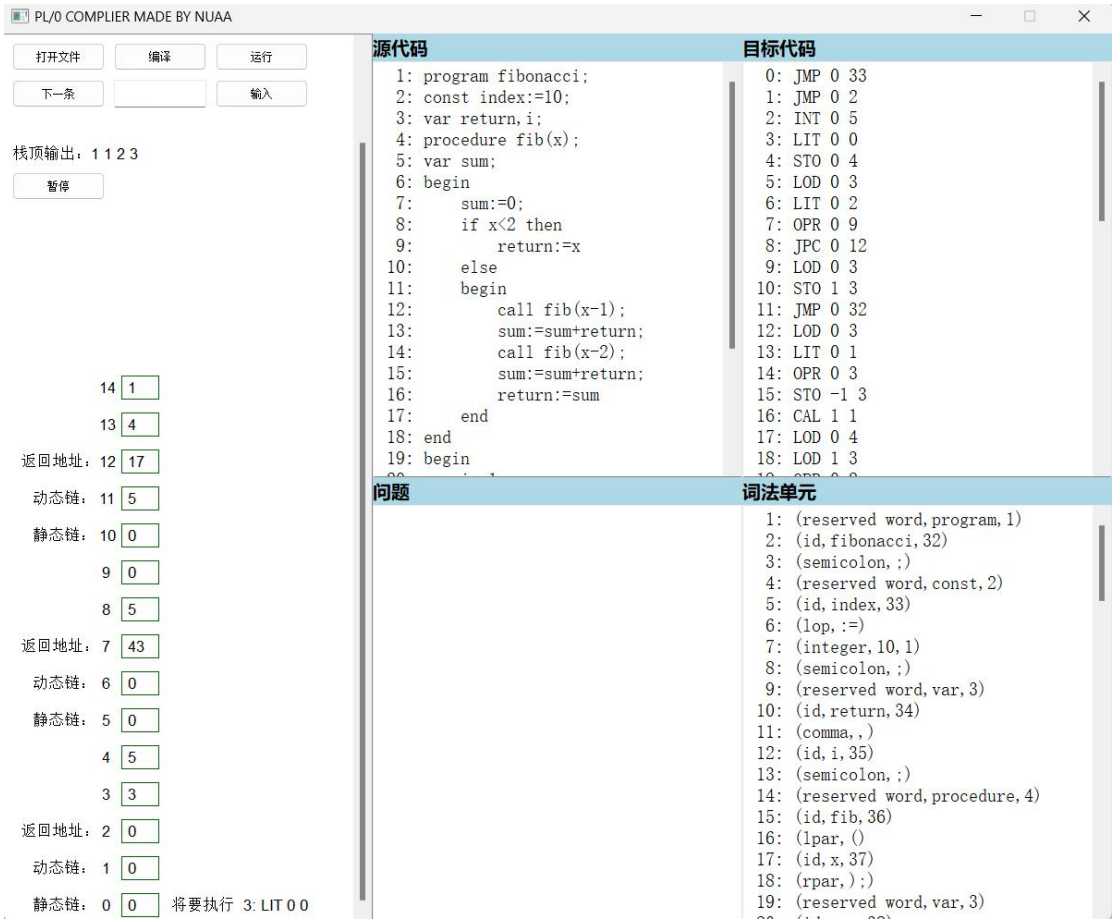


图 5.5 测试 5 程序运行过程图

栈顶最终输出 1 1 2 3 5 8 13 21 34 55.，由上述结果可知，该系统能正确执行输出正确结果，说明该系统能正确处理递归的情况。

5.6. 多功能测试

过程调用、递归调用、循环、条件判断、作用域测试。

5.6.1. 测试程序

```
program right;
const a:=5;
var x,y,b,c,k,fibreturn,return2;
  procedure fib(x);
  var sum;
  begin
    if x<=2 then
      fibreturn:=1
    else
    begin
      sum:=0;
      call fib(x-1);
      sum:=sum+fibreturn;
      call fib(x-2);
      sum:=sum+fibreturn;
      fibreturn:=sum
    end
  end;
  procedure test1();
    procedure in();
    begin
      y:=y+a
    end
  begin
    y:=y+a;
    call in()
  end;
  procedure judge(x);
  begin
    if odd x+1 then
      return2:=1
    else
      return2:=0
  end;
  procedure test2(m,k);
  var i,sum;
  begin
    i:=0;
    sum:=0;
    while i<k do
```

```
41.         begin
42.             sum:=sum+m;
43.             i:=i+1
44.         end;
45.         write(sum);
46.         call judge(sum)
47.     end;
48.     procedure test3();
49.         procedure in1();
50.         var x;
51.         begin
52.             x:=3
53.         end;
54.         procedure in2();
55.         begin
56.             write(x)
57.         end
58.     begin
59.         call in1();
60.         call in2()
61.     end
62. begin
63.     read(x);
64.     call fib(x);
65.     write(fibreturn);
66.     read(y);
67.     call test1();
68.     write(y);
69.     read(b,c,k);
70.     call test2(b+c,k);
71.     write(return2);
72.     x:=99;
73.     call test3()
74. end
```

5.6.2. 结果分析

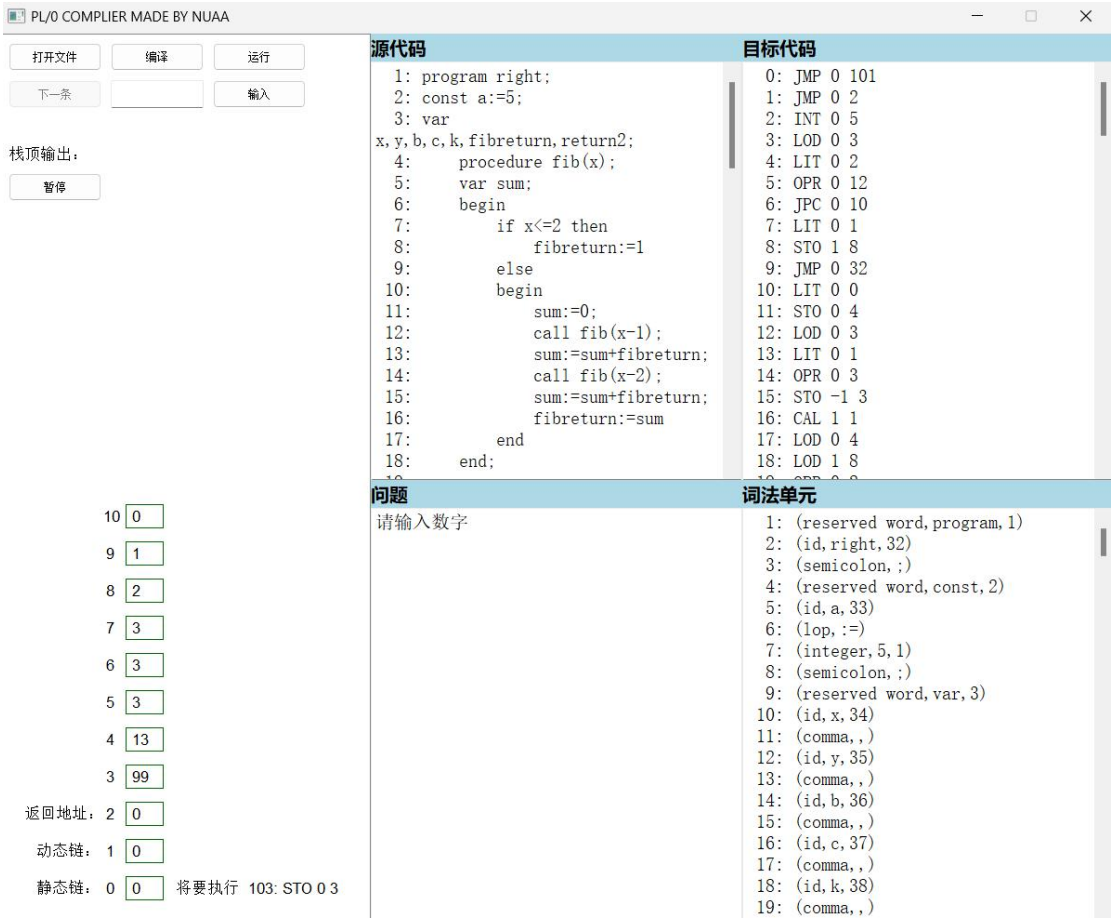


图 5.6 程序测试结果

在输入栏均输入 3，可以看到所有 test 均得到正确的结果，说明了系统的正确性。

5.7. While_if 嵌套测试

5.7.1. 测试程序

```
program id;
const m:=0;
var x,b;
begin
    read(b);
    while b>=m do
        begin
            if b>m then
                x:=x+1;
            b:=b-1
        end;
    write(x)
End
```

5.7.2. 结果分析

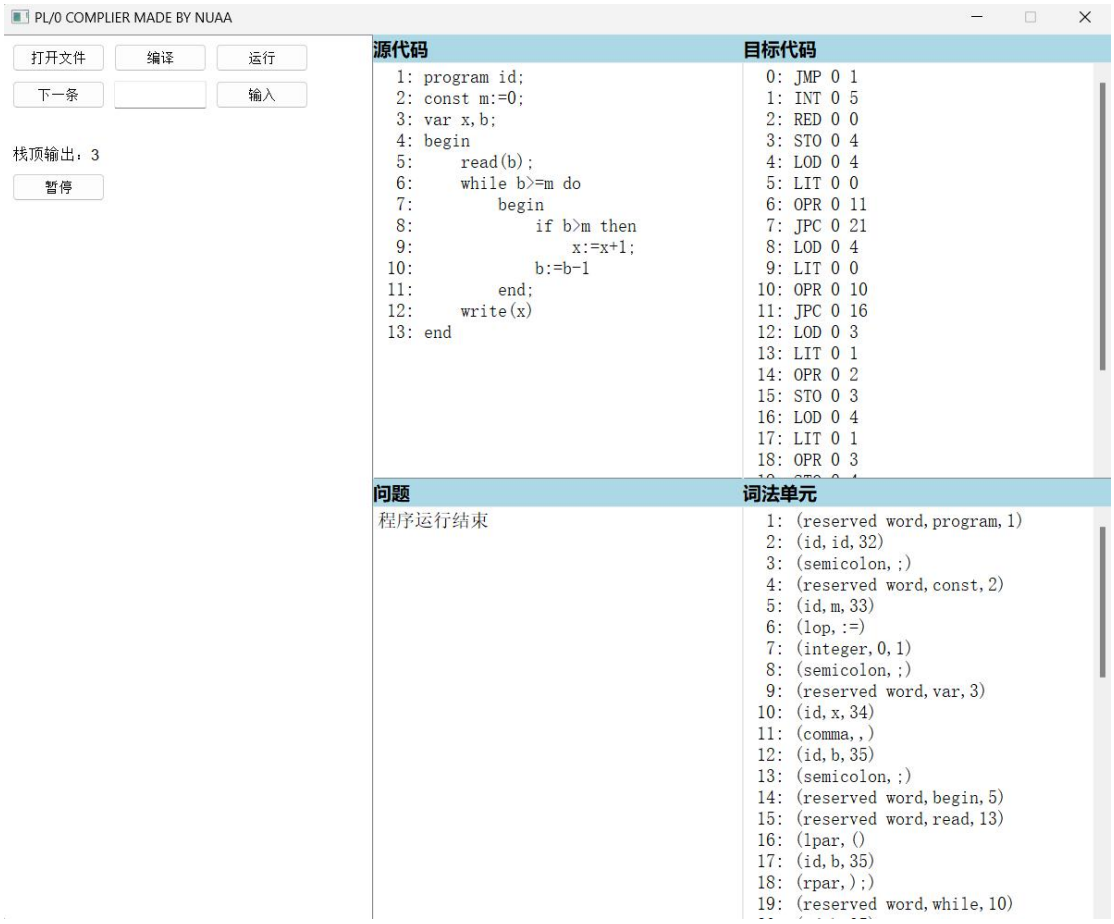


图 5.7 测试程序 7 运行结果

由上述结果可得，系统能正确处理 while_if 嵌套的情况，证明了系统的正确性。

5.8. 更多测试用例

更多测试用例见附件。

6. 课程设计心得

6.1. 162230217 陈梓鹏课程设计心得

这次课设可以算是我大学生涯以来最特殊也最难忘的一次课设了。

为何特殊？由于课设时间和学校的金工实习冲突，基本上是白天在厂里干活，然后拖着疲惫的身体在线上和组员们讨论，实现系统。并且由于个人的原因，在这期间突发急性肠胃炎....非常特殊。

为何难忘？因为这次课设不仅让我收获了许多专业知识，而且每天如同着魔一般，在白天厂里干活的时候一直思考着如何实现，这样实现会造成什么问题，尤其是在符号表的设计的时候，也是想了很久，疲惫不堪，却充实而快乐。

谈谈我的收获吧！

代码能力：虽然参加过很多程序设计的比赛，但编译器的设计是一个非常复杂的工程，涉及到多个模块的协同工作。在这个过程中，我学会了如何将一个大问题拆解成多个小模块，并实现模块化的代码结构。每个模块都有明确的职责，代码的高内聚和低耦合成为了我设计时的原则，这样将可以使每个模块有各自的职能，便于维护调试。以及对于数据结构和算法的设计，尤其是在符号表的设计的时候决定采用分层的思想，效率高且保证正确。

以及最重要的，写注释的重要性，不写注释，不仅折磨过两天忘记自己写了什么东西的自己，还折磨阅读自己代码的人。

调试与错误处理能力：程序员最害怕的就是调 bug，更害怕的是调别人的 bug。这次课设我们小组分工明确，各自主要负责对应的模块，但也免不了各自给各自写了 bug。我们小组之间一旦有人自己的 bug 实在解决不了，就会请求别人的帮助来调试，以一种新的视角来调试，虽然看别人的代码很痛苦，但效率非常高，这也是我们的团队协作能力。印象最深刻的是，在程序回填的时候，仅仅是一个变量忘记赋值，导致了错误，这个小小的 bug 就耗费了我一个晚上的时间去调试。

新技能：学习到了 python 也可以调用 C++程序,版本控制工具，画图工具，界面设计...

文献调研：在系统实现的过程中，遇到了各类困难，也是通过网上大量搜集资料，解决问题。例如编程实现时语法错误，翻译模式设计，编程环境配置...在这期间也看到了许多有意思的学术研究，虽然仅仅是点开粗略地阅读了一番...

团队协作的重要性：虽然有组长组员之分，但我们团队的每个组员都在尽全力地发挥着各自的作用，这也使得在我生病（上文提到），金工实习压力，组员有多门考试压力的情况下，能够顺利完成课设任务。

最后，我想感谢一下我们的组员，在我写编译器期间，我和他们进行了许多技术上的交流，他们给了我很多帮助与启发，我编译器的完成离不开他们的帮助。感谢杨老师的耐心，给出了我们更多的时间去实现。希望自己再接再厉，努力提升自己的代码水平，争取完成更加有水平的项目。

6.2. 162230203 李萌课程设计心得

在本次的课程设计中，我们小组实现了一个简易的 PL/0 语言的编译器，让我认识到编译原理不仅仅是理论知识的学习，更多的是如何将这些抽象的理论知识应用到实际的编译器设计中。本次课设，我们完成了语法、词法、语义分析，目标代码的生成以及虚拟执行，涉及到了书上的几乎所有所学章节内容，这对于期末考试的复习也有很大的

益处。

1. 从理论到实践的转变

编译原理课设给我最大的感受就是它与课本内容是切实相关的。通过理论课程的学习，再加上杨老师时时刻刻地叮嘱和强调，我们在学习课本知识及相关算法的基础上，也对于代码逻辑更加清晰，这对于最后的代码编写和 debug 是及其有用的，只有理清自己的逻辑才能写出更加完整可靠的代码。

2. 构建可视化页面

在这次设计中，我们还增加了可视化界面来展示编译器的各个阶段和结果，这有助于自身对编译器的理解以及老师的最后验收。我们利用 visual.py 调用 main.exe 函数，最后生成可视化界面，成功显示了源代码、目标代码、栈内容等信息，还对错误信息进行输出。可视化界面的实现并不简单，尤其是在如何有效地展示编译过程中的各个环节时，我们遇到了不少技术难题。为了完成这个部分，我们查阅了大量的资料，研究了如何利用 Python 与其他工具（如 PyQt、Tkinter 等）配合实现可视化界面。同时，也学习了如何将编译器的输出信息与图形化界面进行联动。

3. 对整体结构框架的清晰认识

通过本次课程设计，我对编译器的整体结构框架有了更加清晰的认识。每一个阶段的工作并不是孤立的，它们相互配合，共同完成编译任务。例如，词法分析和语法分析紧密相关，语法树的构建依赖于词法分析的结果；而语义分析又依赖于语法树的正确性，符号表的设计也必须与语法分析的结果密切配合。这种各部分相互依赖、相互协作的结构框架让我更加理解编译器设计的复杂性，同时也提升了我对整体工作流程的把控能力。

4. 解释器与假想目标机

这一块的完成充分利用了第九章的知识，用老师给定的假想目标机的指令和活动记录栈，再加上我们自己扩展的部分指令，形成了目标代码并虚拟执行。解释器主要是对于不同的指令部分进行相应的操作，直到完成所有指令的执行。我们还针对动态链、静态链、返回地址、其他栈的内容做了可视化输出，帮助了我们深入理解编译过程中的内存管理和函数调用机制，同时也为调试和验证程序的正确性提供了直观的帮助。

5. 团队合作与沟通

在本次课程设计中，我们小组的成员分工合作，互相帮助，大家共同讨论并解决遇到的问题。通过这样的团队协作，我学会了如何在项目中与他人协作，如何分配任务和进度，以及如何在项目进展过程中及时沟通解决问题。在与队友共同编写代码、设计各个模块时，我们互相借鉴思路，提出改进建议，使得编译器的设计和实现更加完善。

总之，这次课程设计让我真正体会到了编译原理的魅力与挑战，通过将理论知识应用到实际编译器的设计中，我不仅增强了自己的编程能力，也提升了对计算机底层原理的理解。

最后，感谢杨老师一学期的帮助和耐心讲解。杨老师通过生动的例子和清晰的讲解，使我们能够更加深入地理解编译原理。在课堂上，杨老师不仅强调了理论知识和考试的关键点，还经常提醒我们将理论与课设实践相结合，帮助我们将复杂的编译原理知识转化为可操作的技能。这种理论与实践的紧密结合，让我们不仅掌握了编译原理的基本概念，还能够将这些概念应用到实际的编译器设计中。

6.3. 162230205 杨雨涵课程设计心得

这次编译原理课程设计让我受益匪浅。作为一门理论性极强的学科，编译原理在课堂上更多强调的是形式化的文法、语法分析方法、语义分析与目标代码生成等抽象

概念。在学习过程中，我常常感到这些理论知识虽然逻辑清晰，但与实际应用的联系却显得遥远。然而，在这次课程设计中，通过亲自动手实现解释器和可视化功能，我深刻体会到这些理论的实际价值，也进一步加深了对编译原理核心内容的理解。

在整个实验中，我主要负责解释器和可视化模块的实现，同时也参与了其他模块的辅助设计工作。解释器模块可以说是编译原理理论知识的直接体现。在实现过程中，我们需要逐条解析和执行目标代码，这实际上就是课堂上学习的语法制导翻译的延续。从理论知识出发，我理解了目标代码的结构以及它与目标代码执行的关系；而在实践中，设计和实现一个能够正确执行目标代码的解释器，需要我深入思考栈的操作逻辑、指令的功能实现以及函数调用的上下文切换等具体细节。在课堂上，我们经常提到栈帧的概念，但在代码实现中，我需要考虑的不仅仅是栈帧的定义，还包括如何高效地通过静态链和动态链来实现跨层级的变量访问。通过实现 get_sl 函数，我进一步理解了静态链的递归查找过程，并成功地将理论知识转化为程序逻辑。在设计过程中，我也多次反复调试，修正了由于层次差计算错误导致的变量查找问题。这种理论和实践结合的过程，不仅让我对解释器的运行机制有了更加清晰的认识，也让我对课堂上学到的理论知识有了更深刻的感悟。

可视化模块则是一个全新的挑战。相比解释器模块，它对编译原理的理论知识依赖较少，但对用户体验和动态展示的要求更高。在课堂上，我们学习了目标代码的生成规则和运行逻辑，但如何让这些目标代码的执行过程变得直观明了，则是这次课程设计赋予我的额外任务。通过使用 PyQt5 框架，我们组设计了一个可视化界面，能够动态展示数据栈的内容、目标代码的执行状态以及输入输出的交互过程。实现过程中，我遇到了许多未曾预料的难题，比如如何实时刷新栈的展示内容、如何用高亮标记当前正在执行的指令、如何设计交互功能使用户能够控制程序的执行节奏等。每一个问题的解决都需要不断实验与调整，同时也让我学会如何将复杂的逻辑清晰地呈现给用户。特别是在动态展示数据栈的内容时，课堂上的栈操作抽象知识成为了最重要的指导思想，让我能够设计出一套既直观又符合实际运行逻辑的可视化方法。

在实验的后期，我也参与了其他模块的调试与改进工作，比如语法分析模块的递归下降子程序以及符号表的实现与作用域管理。这些模块在功能上紧密联系，任何一个细节的错误都可能导致系统运行的不完整或错误。这让我意识到，编译原理这门课的知识体系虽然模块化清晰，但在实际系统设计中却是一个高度集成的整体。任何一个模块都需要准确地与其他模块配合才能保证整个编译器的正确性和效率。

这次实验让我深刻认识到理论和实践的结合是多么重要。在编译原理的课堂上，很多概念看似抽象甚至难以理解，但当我们通过代码将它们实现时，原本枯燥的理论就变得生动起来。这种从理论到实践的转化过程，让我对编译器的运行机制有了更全面的理解，也让我意识到编程的逻辑性和严谨性对一个复杂系统的重要性。此外，这次实验也让我对编译器的整体架构有了更清晰的认识，从词法分析到语法分析，再到语义分析与代码生成，最后到目标代码的执行，每一步都是理论知识在实践中的具体体现。

当然，这次实验中我也意识到自己的不足之处。在开发初期，由于对系统整体需求分析不够深入，导致了一些功能设计的返工。例如，在解释器模块的初版设计中，我没有考虑到函数调用栈的边界条件，这导致在执行一些复杂函数调用时程序会崩溃。经过深入的调试和同组同学的讨论，我逐步完善了解释器的边界处理机制。

此外，我还发现自己在时间管理和代码规范性方面还有提升的空间。例如在实验初期，有时为了快速完成功能，会忽略代码的可读性和结构性，而后期调试时却因此增加了不必要的困难。这些经验让我明白，理论和实践结合不仅仅是实现功能，还包

括如何以高效、规范的方式实现功能。

最后的最后，我要感谢杨老师，没有理论课上夯实的基础也就没有实践中的收获；也要感谢我的同伴们，没有他们我无法完成这么大型的课程设计。

6.4. 162230202 吴静柔课程设计心得

在本次编译原理课程设计中，我承担了词法分析器部分代码的实现工作，期末周写课程设计的经历对我来说是一次极具价值的挑战与成长！

在词法分析器的开发过程中，我面临诸多难题：处理复杂逻辑时，要依据状态转换图准确识别各类词法单元，这需要细致入微的字符判断与状态管理，像标识符和关键字的“超前搜索”，其逻辑复杂，容易出错。面对非法字符等错误情况，必须准确判断错误类型并及时传递错误信息，稍有疏忽就可能导致错误处理不当。为提升效率，代码优化也面临挑战，需精心设计字符读取和字符串拼接等操作，避免不必要的开销。

我也取得了显著的收获。在专业知识方面，对编译原理中词法分析的理论理解更为深刻，学会编写清晰、高效、可维护的代码，能够处理复杂逻辑和边界情况，确保程序稳定准确。团队协作也让我感悟颇多，与队友沟通协作共同解决模块协同问题，深刻体会到团队合作的强大力量。回顾整个过程，我也意识到存在一些不足。错误处理信息的详细程度和友好度有待提高，以便用户更好理解错误原因。代码优化方面仍有进步空间，可探索更高效算法和数据结构来提升词法分析效率。

展望未来，我将继续学习相关知识，学习先进技术和算法，并积极参与项目实践，为未来从事更复杂的编译工作筑牢基础。同时，我也会将此次课设经验运用到其他编程项目中，持续提高编程水平和问题解决能力。

最后，我要衷心感谢杨老师在课程中的详细讲授，为我们提供了坚实的理论基础和实践指导！老师的指导让我们在编译原理的学习道路上少走了许多弯路，也让我有信心和能力应对课程设计中的各种挑战。

7. 参考文献

[1] 《程序设计语言编译原理（第 3 版）》 陈火旺等

[2] 编译原理课设尝试（一）——PL/0 编译器分析 | Chenfan Blog (jcf94.com)

[3] The Absolute Minimum Every Software Developer Absolutely, Positively Must Know About Unicode and Character Sets (No Excuses!) OCTOBER 8, 2003 by JOEL SPOLSKY

[4] StackOverflow：Printing unicode characters in PowerShell via a C++ program

[5] C++ 字符串string、字符char、宽字符数组wstring、宽字符wchar_t 互相转换（2021.4.20）_jing_zhong 的博客-CSDN 博客_string 怎么转换 wchar_t

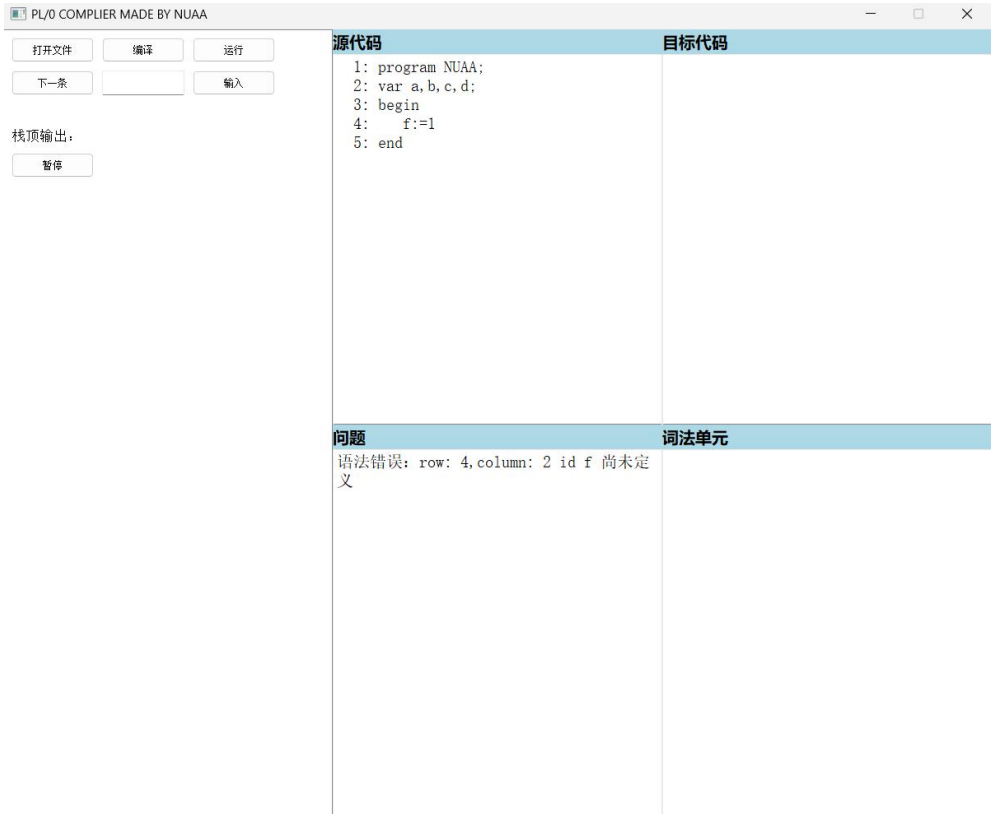
[6] 实战中遇到的C++流文件重置的一个大陷阱： 为什么ifstream 的seekg 函数无效？_涛 歌依旧的博客-CSDN 博客_c++ seekg 不起效

[7] 下载安装MinGW-w64 详细步骤（c/c++的编译器gcc 的windows 版，win10 真实可用） _jjxcsdn 的博客-CSDN 博客_mingw-w64

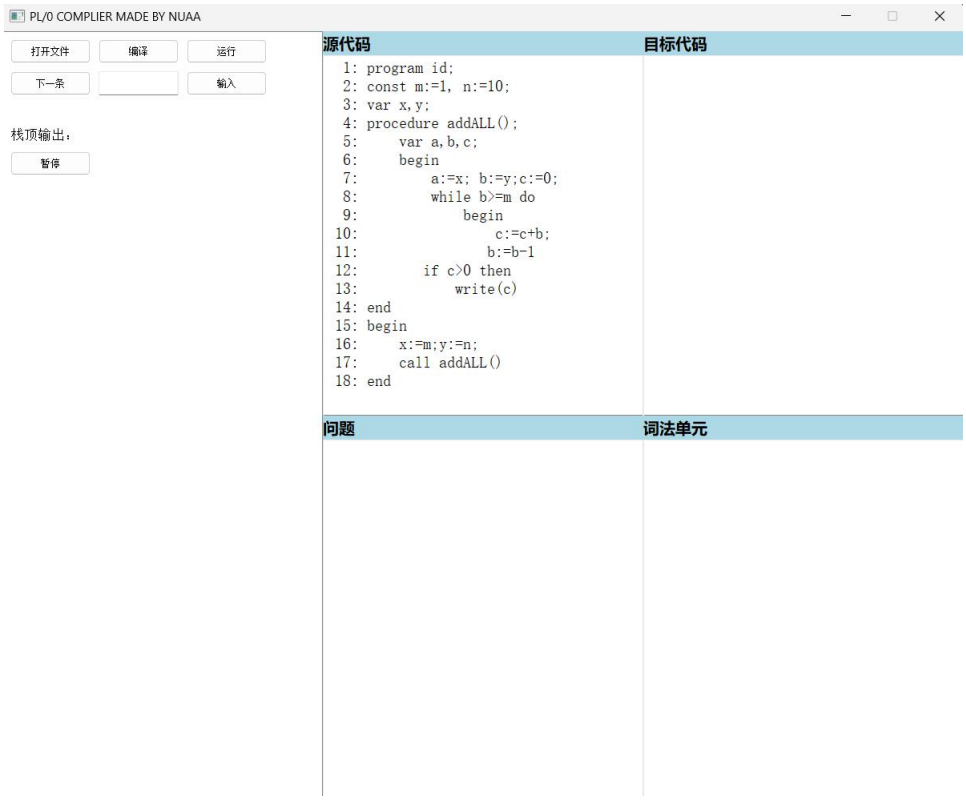
[8] Python 调用 C/C++的两种方法 - 知乎

8. 附件

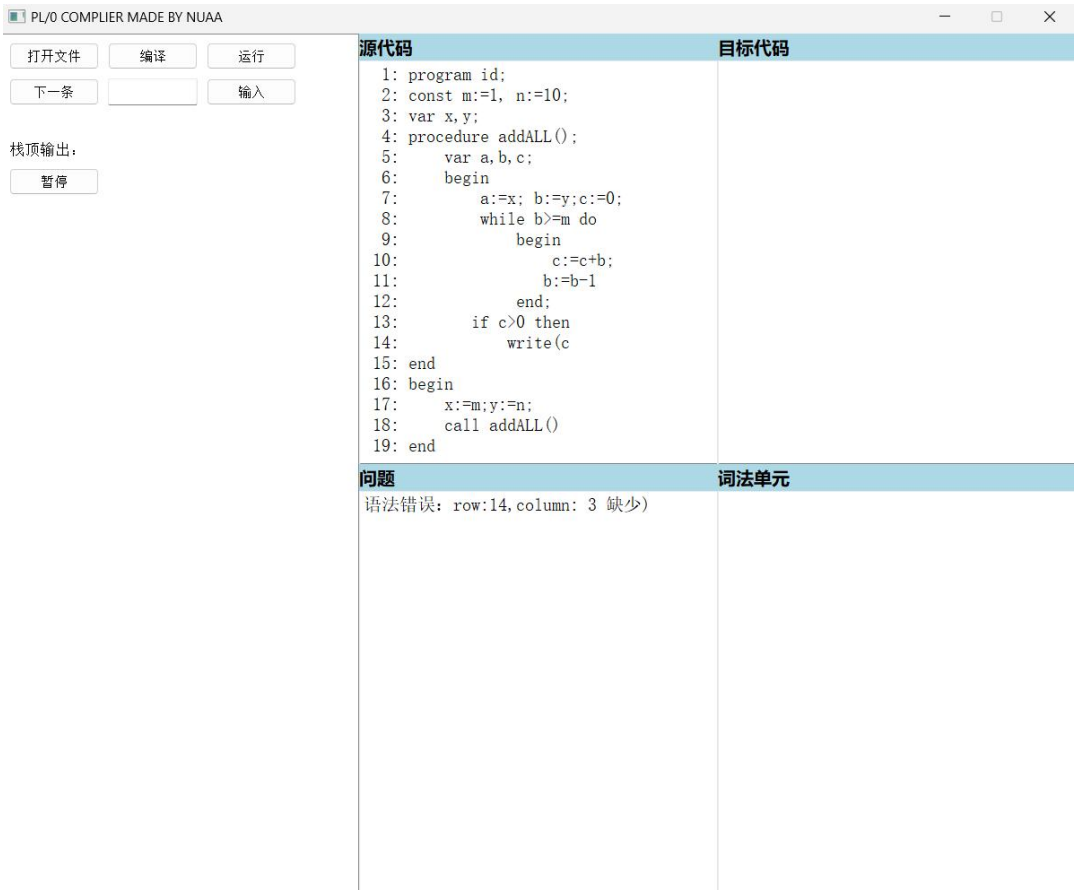
1.变量未定义



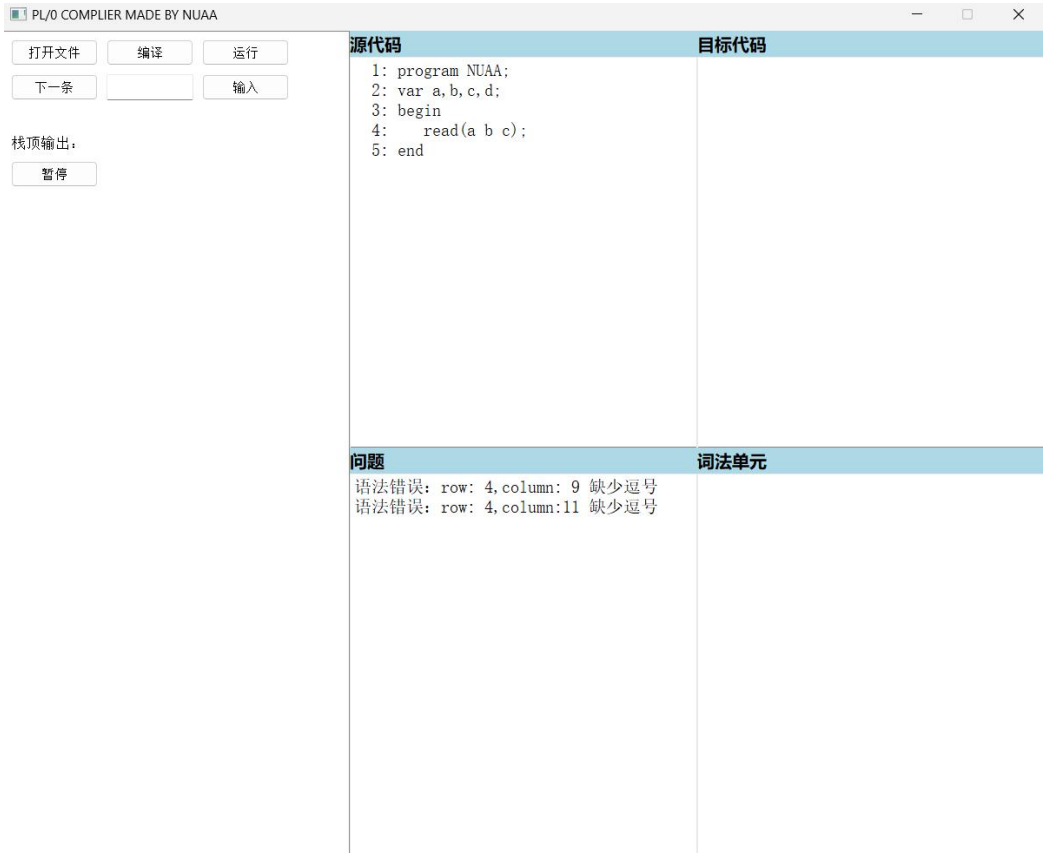
2.begin-end 不匹配



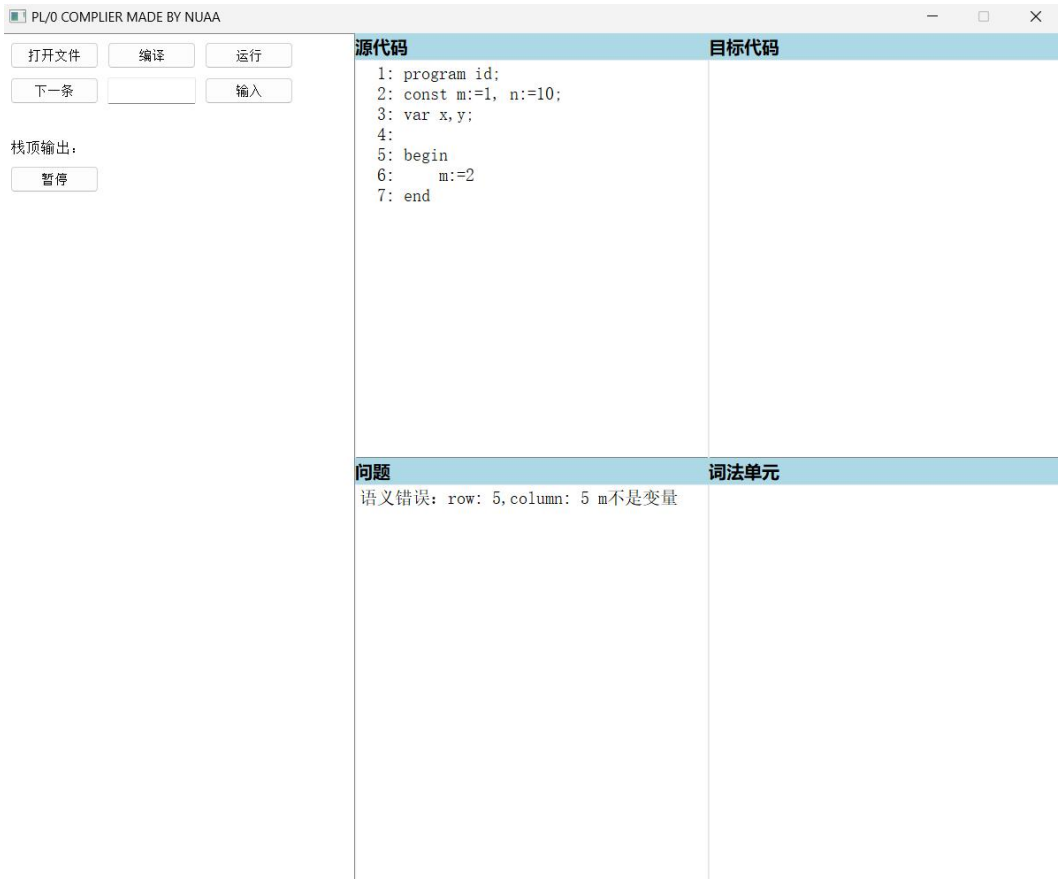
3.write/read 缺少右括号



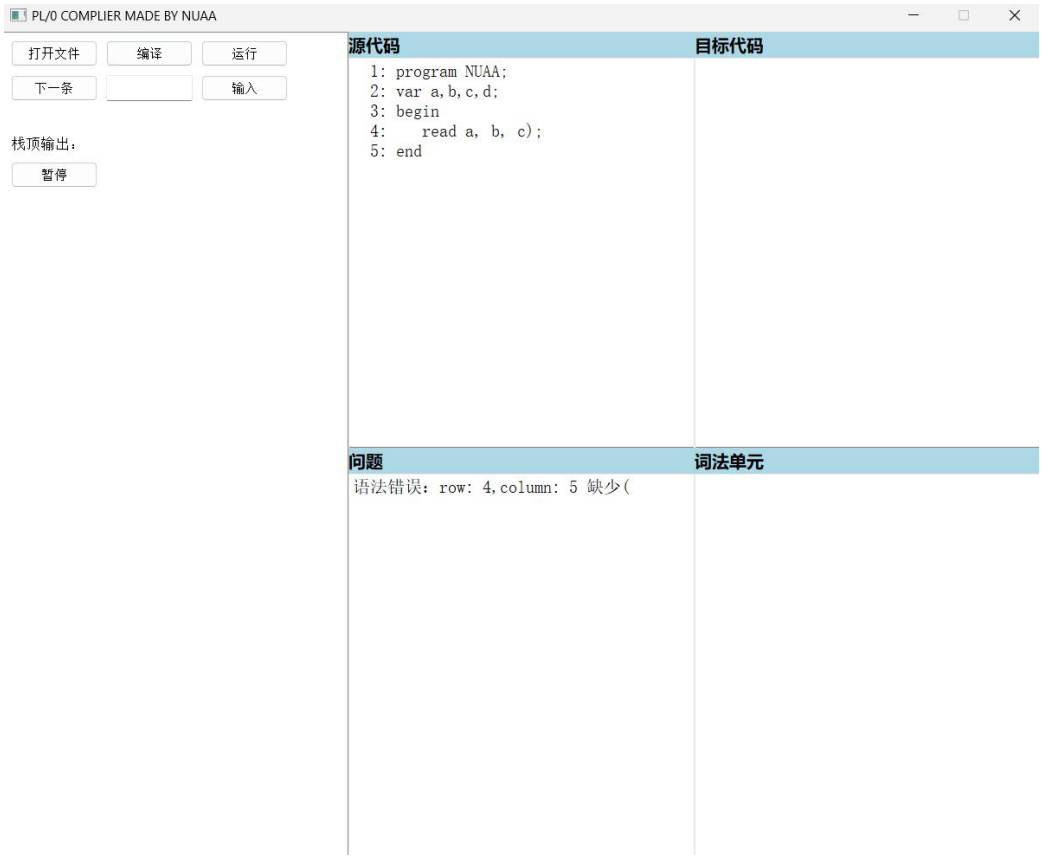
4.参数未用逗号隔开



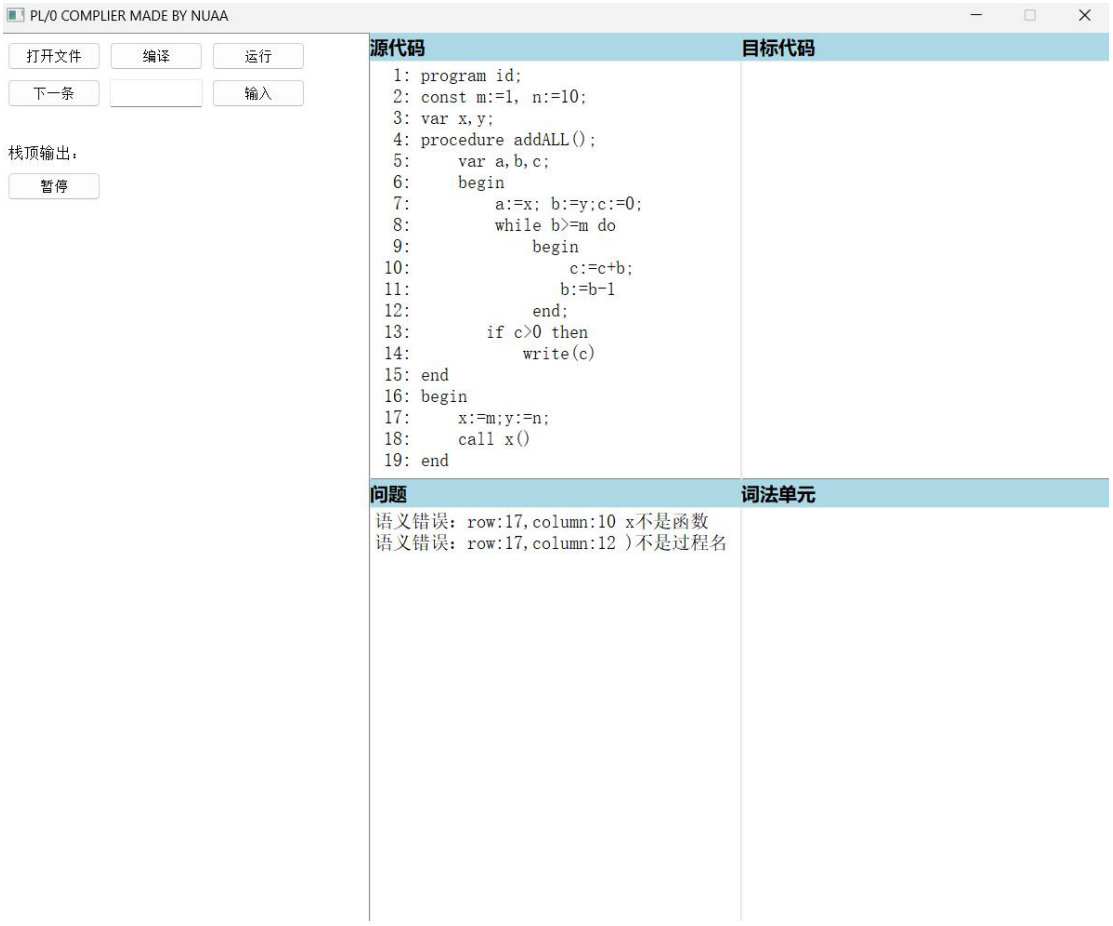
5.赋值左边是 const



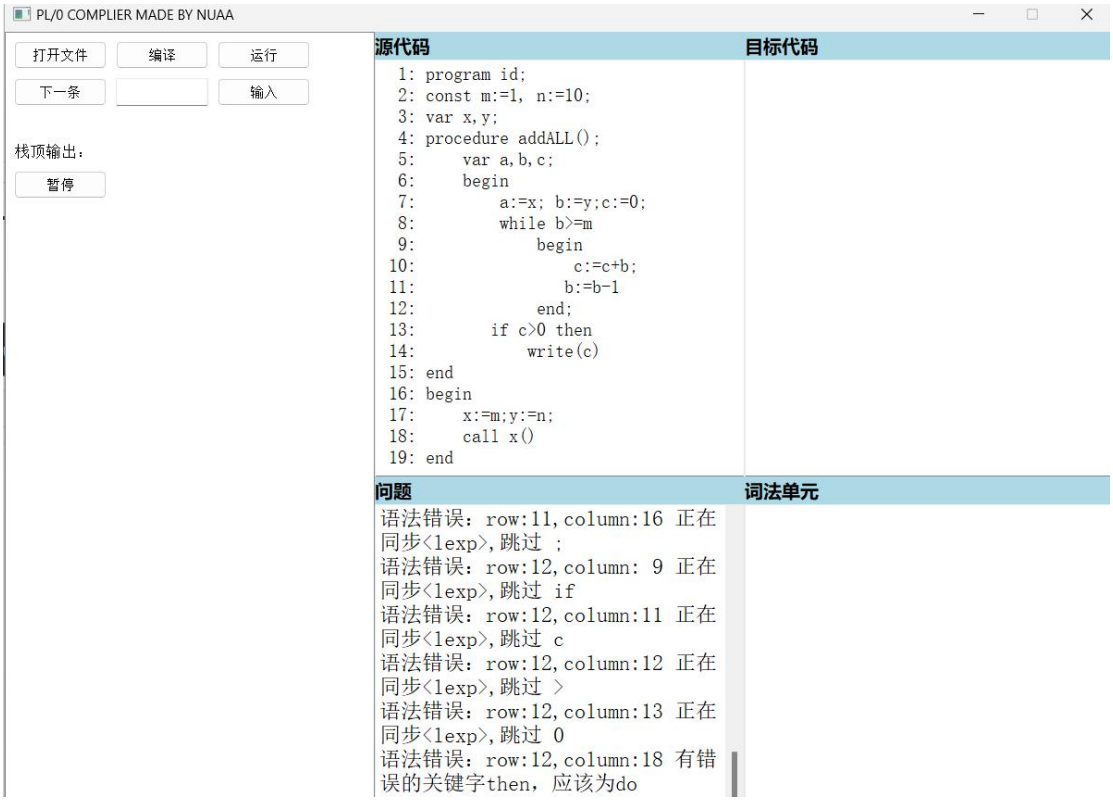
6.write/read 缺少左括号



7.call 调用错误



8.while 缺少 do



9 缺少初值

10.翻译模式

说明：下面的翻译模式中的属性不是和代码一一对应的，比如左边终结符的继承属性，我是采用符号表记录的，是对代码的简化表示，用于展示翻译的基本思路。

表 8.1 全局属性说明

全局属性	说明
Block.entry	全局偏移量
Nextquad	下一条目标代码入口
Strtoken	当前分析出的词法单元
Level	当前分析的子过程的层级
Sp	当前正在分析的子过程的入口

<prog> → **program**

<id>{ mkTable()

enterProgm(strToken)

block.entry = emit(JMP, 0, \) // 为 block 的继承属性赋值 }

;<block>

<block> → **[<condecl>][<vardecl>]**{

addWidth(sp, glo_offset) // 子过程变量声明结束，计算过程占用内存

block.cur_proc = sp }

[<proc>]{

body.entry = emit(INT, 0, block.cur_proc->width)

backpatch(block.entry, body.entry) // 将过程体的目标代码入口地址回填，block.entry 是已经定义的继承属性

block.cur_proc->isDefined = true // 即将进入过程体，标识改为过程体已定义 }

<body>

<condecl> → **const <const>{,<const>};**

<const> → **<id>**{ id.sym_entry = enter(strToken, 0, CST) } // 右值不占用内存 **:=<integer>**{ setValue(id.sym_entry, integer) } // 右值直接登入符号表

<vardecl> → **var <id>**{

enter(strToken, glo_offset, VAR)

glo_offset += 1 }

{,<id>{

enter(strToken, glo_offset, VAR)

glo_offset += 1 **};**

<proc> → **procedure**

<id1>{ mkTable()

id1.sym_entry = enterProc(strToken)

block.entry = emit(JMP, 0, \) } // 为 block 的继承属性赋值 **([<id2>**{

id2.sym_entry = enter(strToken, glo_offset, VAR)

glo_offset += 1

id1.sym_entry->form_var_list.add(id2.sym_entry) } // 在符号表中为过程添加形参入口

{,<id3>{

enter(strToken, glo_offset, VAR)

glo_offset += 1

id1.sym_entry->form_var_list.add(id3.sym_entry) } **}])**

;<block>{

emit(OPR, 0, OPR_RETURN)

display.pop()// 子过程结束，层级减一，display 表弹栈

level-- }

{;<proc>}

<body> → begin <statement>{;<statement>}end

<statement> → <id>{

id.sym_entry = lookUpVar(strToken)

if (id.sym_entry == -1)

error } **:= <exp>** {

if(id.sym_entry->cat!=CST) // 常量不可被赋值

emit(STO,id.sym_entry->level, id.place) }

|if <lexp> then { lexp.false_entry = emit(JPC, 0, \) } // 条件为假时,跳转到 else 处或 if 语句外 **<statement1>[else <N> <statement2>**{ backpatch(N.entry, nextquad) }**]**

{// 如果没有 else 才执行下面的翻译模式

backpatch(lexp.false_entry, nexquad) }// 将 if 语句外的入口地址回填至 JPC

|while <M><lexp>{ lexp.false_entry = emit(JPC, 0, \)} **do**

<statement>{ emit(JMP, 0, M.quad)

backpatch(lexp.false_entry, nextquad) }

|call <id>{

id.sym_entry = lookUpProc(strToken)

if (id.sym_entry == -1) error }

([<exp1>{

i = 0

emit(STO, -1, 3 + id.sym_entry->level + 1 + i) }

{,<exp2>{

i += 1

emit(STO, -1, 3 + id.sym_entry->level + 1 + i) }**}])**

|read (<id1>{

id1.sym_entry = lookUpVar(strToken)

if (id1.sym_entry == -1) error

if(id1.sym_entry->cat != CST)

begin

emit(RED, 0, 0)

emit(STO, id1.sym_entry->level, id1.place)

end }

{,<id2>{

id2.sym_entry = lookUpVar(strToken)

if (id2.sym_entry == -1) error

if(id2.sym_entry->cat != CST)

begin

emit(RED, 0, 0)

emit(STO,id2.sym_entry->level, id2.place)

end } **})**

|write (<exp>{ emit(WRT, 0, 0) }**{,<exp>}**{ emit(WRT, 0, 0) }**)**

|<body>

<M>→ε{ M.quad = nextquad }

<N>→ε{

N.entry = emit(JMP, 0 ,\) // 无条件跳转语句，跳转至 if 语句外

backpatch(lexp.false_entry, nextquad) } // 将 else 入口地址回填至 JPC

<lexp> → <exp> <lop> <exp>{

switch(lop):

case '<': emit(OPR, 0, OPR_LSS); break;

case '<=': emit(OPR, 0, OPR_LEQ); break;

case '>': emit(OPR, 0, OPR_GRT); break;

case '>=': emit(OPR, 0, OPR_GEQ); break;

case '<>': emit(OPR, 0, OPR_NEQ); break;

case '=': emit(OPR, 0, OPR_EQL); break;

defalut: break; }

| odd <exp>{ emit(OPR, 0 ,OPR_ODD) }

<exp> → <aop1>{
if(aop1==’-‘) aop1.flag = 1;
else aop1.flag =
0; } <term>{
if(aop1.flag == 1) emit(OPR, 0 , OPR_NEGTIVE) } // 第一个 aop 为负号,产生取反代码 {<aop2> {
if(aop2==’-‘) aop2.flag = 1;
else aop2.flag =
0; } <term>{
if(aop2.flag == 1) // 根据加减号产生相应代码
emit(OPR, 0, OPR_SUB);
else
emit(OPR, 0 ,OPR_ADD); }}

<term> →
<factor>{<mop>{ if(mo
p==’*’) mop.flag = 0;
else mop.flag =
1; } <factor>{
if(mop.flag == 0) // 根据乘除号产生相应代
码 emit(OPR, 0, OPR_MULTI);
else
emit(OPR, 0 ,OPR_DIVIS); }}

<factor>→<id>{
id.sym_entry = lookUpVar(strToken)
if (id.sym_entry == -1) error
if(id.sym_entry->cat == CST) // 读取到常量符号，产生 LIT 代码，否则为变量，产生 LOD
emit(LIT, id.sym_entry->level, id.sym_entry->value)
else
emit(LOD, id.sym_entry->level, id.place) }
|<integer>{ emit(LIT, 0, w_str2int(strToken)) }
|(<exp>)