

编译原理课程报告

162210107蔡蕾



2024-11-18

南京航空航天大学

计算机科学与技术学院

目录

[总任务 2](#_Toc183546090)

[上机练习1 2](#_Toc183546091)

[上机练习2 3](#_Toc183546092)

[系统设计 3](#_Toc183546093)

[系统结构 3](#_Toc183546094)

[types类（项目初始准备） 4](#_Toc183546095)

[词法分析器 6](#_Toc183546096)

[引言 6](#_Toc183546097)

[词法分析器的基本原理 7](#_Toc183546098)

[词法分析器模块设计 9](#_Toc183546099)

[测试用例 13](#_Toc183546100)

[符号表 14](#_Toc183546101)

[引言 14](#_Toc183546102)

[符号表模块设计 15](#_Toc183546103)

[错误处理单元 22](#_Toc183546104)

[引言 22](#_Toc183546105)

[错误处理单元模块设计 23](#_Toc183546106)

[语法分析 24](#_Toc183546107)

[引言 24](#_Toc183546108)

[证明 25](#_Toc183546109)

[语法分析器模块设计 26](#_Toc183546110)

[测试用例 32](#_Toc183546111)

[语义分析 38](#_Toc183546112)

[中间代码生成 38](#_Toc183546113)

[解释器 38](#_Toc183546114)

[课设总结 38](#_Toc183546115)

# 总任务

## 上机练习1

**词法分析**

**一个PASCAL语言子集（PL/0）词法分析器的设计与实现**

**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**

**odd：判断表达式的奇偶性。**

**要求：**

1. **按照P45的算法思想，使用循环分支方法实现PL/0语言的词法分析器，该词法分析器能够读入使用PL/0语言书写的源程序，输出单词符号串及其属性到一中间文件中，具有一定的错误处理能力，给出词法错误提示（需要输出错误所在的行列）。**

## 上机练习2

**语法分析**

根据上机练习一给出的PL/0语言扩充的巴克斯范式语法描述，利用递归下降的语法分析方法，编写PL/0语言的语法分析程序。

要求：

1. 对给出的PL/0语言进行分析，证明其可以进行自上而下的语法分析；
2. 对block、proc、statement、condition、expression、term、factor进行分析，画出语法分析图，在此基础上描述这些子程序的设计思想；
3. 具有一定的语法错误处理能力；

# 系统设计

## 系统结构

在编译器的整体架构中，它可以分为前端（Front End）和后端（Back End）两个主要部分。每一部分又包含不同的处理阶段，依次完成从源代码到目标代码的翻译和优化。下图展示了编译器的流程及其各个组件的相互作用。

**前端：**编译器的前端主要负责分析源代码，确保其结构和语义正确，并将其转换为一种中间表示（Intermediate Representation）。前端的几个主要阶段包括：

**词法分析器（Lexical Analyzer）：** 词法分析器将源代码逐字符扫描，将其分解为最基本的构成单元，即词法单元（token）。这些 token 包含诸如关键字、标识符、常量、操作符等。经过词法分析后，源代码被转换为一个记号流（Token Stream），为后续的语法分析器提供输入。

**语法分析器（Syntax Analyzer）：** 语法分析器负责根据语言的语法规则，分析记号流，构建语法树（Syntax Tree）。这一步骤的目标是确保代码符合语言的语法规范，例如括号是否匹配、语句结构是否正确等。语法树是源代码的结构化表示，反映了程序的嵌套关系和控制结构。

**语义分析器（Semantic Analyzer）：** 语义分析器在语法树的基础上进行语义检查，确保程序的逻辑符合语言的语义规则。它检查变量的定义和使用是否一致、类型是否正确等问题。如果在这一阶段检测到语义错误，编译器将报告相应的错误。语义分析器输出的结果是一个带注解的语法树（Annotated Syntax Tree），为后续代码生成提供更多的上下文信息。

**中间代码生成器（Intermediate Code Generator）：** 这一阶段将带注解的语法树转换为中间代码表示（Intermediate Representation, IR）。中间代码是一种抽象的代码表示形式，独立于具体的机器结构。它提供了一种通用的方式来表示程序的逻辑，方便后续的优化和目标代码生成。

**后端：**编译器的后端主要负责优化中间代码并生成目标机器代码。后端的各个阶段包括：

**代码优化器（Code Optimizer）：** 在这个阶段，编译器对中间代码进行优化，旨在提高程序的执行效率或减少资源消耗。例如，它可以通过消除冗余代码、减少内存使用或优化循环等方式改进程序的性能。优化后的中间代码将进一步减少代码的执行时间或空间占用。

**目标代码生成器（Code Generator）：** 最后一步，编译器将优化后的中间代码转换为特定平台的目标代码（Target Code），通常是二进制形式的机器代码或汇编代码。这一阶段的目标是生成可以在特定硬件上执行的程序。

**符号表与错误处理：**贯穿整个编译过程的是两个重要的支持模块：符号表（Symbol Table）和错误处理器（Error Handler）。

**符号表：**符号表在编译的各个阶段中维护着程序中变量、函数、常量等的符号信息。它记录每个符号的类型、作用域、内存地址等详细信息。在词法分析、语法分析、语义分析等阶段，符号表都起到了至关重要的作用，特别是在语义分析阶段，用于检查符号的正确性。

**错误处理器：**错误处理模块则负责在编译过程中发现并报告源代码中的错误。词法错误、语法错误、语义错误都会在相应的阶段被发现并报告给开发者。错误处理器不仅仅是在遇到问题时中止编译，它还会尽量恢复正常的编译过程，以发现更多的潜在问题。

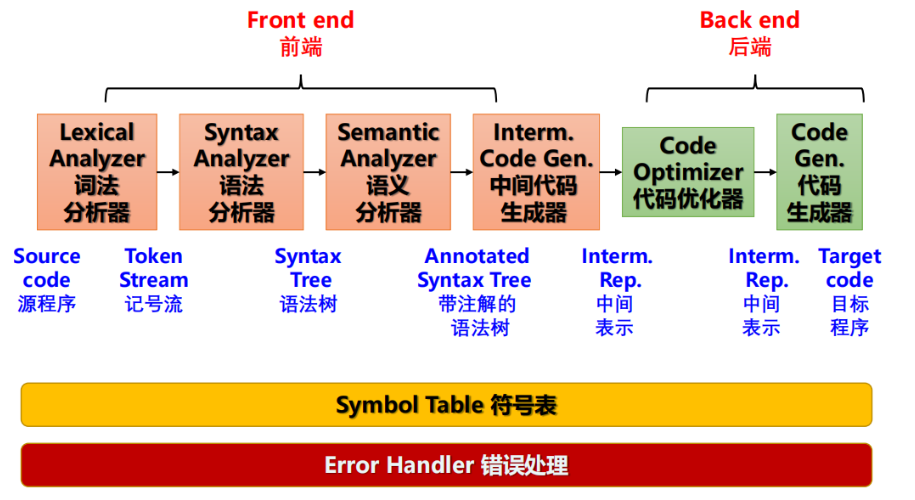


图1.编译器系统结构

## types类（项目初始准备）

在整个编译器设计中，types 模块的初始化部分是为了处理和管理编译过程中涉及的不同数据类型。在这部分中，types.cpp 负责一些关键的初始化和类型转换操作，例如从 wstring 到 int 的转换、Unicode 字符的读取与处理等。以下是对初始化部分的具体叙述。

**函数准备：**

**w\_str2int 函数：**该函数负责将宽字符串（wstring）转换为整数（int）。它首先检查输入的字符串是否为空，并遍历字符串的每一个字符，确保每个字符都是合法的数字字符。如果存在非法字符，它将返回错误并提示不合法的字符串。这一转换函数对于处理源代码中的数字常量非常重要，确保在编译过程中能够正确识别和转换数值。

int w\_str2int(wstring num\_str)

**ReadUnicode类：**

主要功能是读取 UTF-8 编码的源代码文件，将其内容转换为 Unicode 字符流，并存储到内部变量中以便后续处理。这是编译器前端处理源代码的基础操作之一，确保无论源代码包含何种语言的字符，编译器都能够正确识别和处理。

**类的成员变量:**

**wstring progm\_w\_str：**这是类中的一个主要成员变量，用于存储从源代码文件中读取的 Unicode 字符串。所有从源代码文件中读取的字符，都会被解析为宽字符（wchar\_t）并存入该变量中，供后续的词法分析器使用。

**类的成员函数:**

**InitReadUnicode 函数：**该函数用于初始化 ReadUnicode 对象，清空存储的 Unicode 字符串 progm\_w\_str。在编译过程中，需要反复处理源代码中的字符流，该函数确保在每次处理新文件时，能够重新初始化存储结构，避免残留数据影响后续的编译过程。

void ReadUnicode::InitReadUnicode()

**readFile2USC2 函数：**这是一个核心初始化函数，负责从 UTF-8 编码的文件中读取字符并将其转换为 UCS-2（Unicode 编码）。该函数会跳过 UTF-8 文件的 BOM 头，并根据不同的字节长度解析字符（如单字节、双字节、三字节等）。在解析过程中，它将每个字符存储为宽字符（wchar\_t），并最终生成完整的 Unicode 字符串。此外，该函数还处理了文件读取中的异常情况，如非法的 Unicode 范围或无效的 UTF-8 编码。文件读取完成后，Unicode 字符串存储在 progm\_w\_str 中，供后续词法分析和语法分析使用。

void ReadUnicode::readFile2USC2(const string filename)

**wchar\_t getProgmWStr(const size\_t nowPtr):**该函数根据提供的指针（nowPtr）返回 progm\_w\_str 中对应位置的字符。这对于在词法分析器中需要逐个读取和处理字符时非常有用。

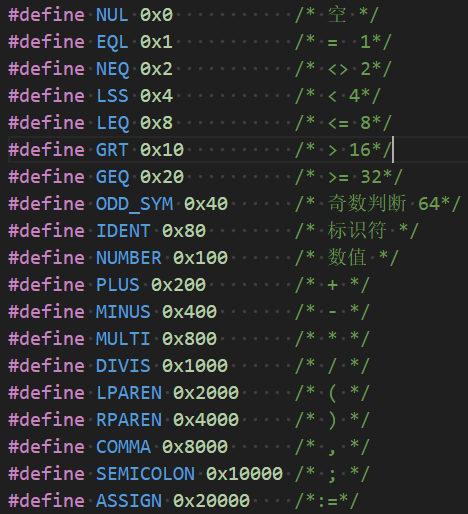
**bool isEmpty()：**该函数用于判断 progm\_w\_str 是否为空。它通常用于检查是否已经成功加载了文件数据或者在文件读取前进行判断。

**宏定义：**

**符号类型宏：** IDENT, NUMBER, PLUS, MINUS, TIMES, SLASH 等是符号类型的定义，表示词法分析器中各类词法单元的类型。例如，IDENT 表示标识符，NUMBER 表示数字常量，PLUS, MINUS 等分别表示运算符 +, -。这些宏定义帮助词法分析器在扫描源代码时，更清晰地标记每个识别出的 token 类型。

**运算符相关宏：** LSS, LEQ, GRT, GEQ, NEQ, EQL 定义了常见比较运算符的代号。它们用于标识 <, <=, >, >=, <>, == 这类运算符。在词法分析过程中，这些宏可以帮助快速标识和处理源代码中的运算符，简化代码中的运算符比较和处理逻辑。

**符号表相关宏：** RSV\_WORD\_MAX 表示保留字的最大数量，用于管理编译器中所有保留字的集合。通过 RSV\_WORD\_MAX，编译器能够确保在处理标识符时，可以有效地与语言中的保留字进行区分，避免关键字的误用。



## 词法分析器

### 引言

编译器的设计是计算机科学中的一个重要课题，其中词法分析是编译过程的第一步。词法分析器的主要任务是将源代码转换为一系列的词法单元（tokens），这些词法单元是编译器进行语法分析的基础。词法分析器的设计不仅影响到编译器的整体性能和准确性，还直接关系到编程语言的可解析性和灵活性。

在现代编程语言中，源代码通常由字符组成，这些字符包括字母、数字、操作符、标点符号等。词法分析器需要根据这些字符的组合规则，识别出有意义的成分，如关键字（如 if、while）、标识符（如变量名）、常量（如数字、字符串）和运算符（如 +、-）。此外，词法分析器还需要能够处理空白字符和注释，这些内容在语法分析中是不需要考虑的。

为了实现这一功能，词法分析器通常采用有限状态机（Finite State Machine, FSM）模型。状态机通过不同的状态和状态之间的转换，能够有效地识别出不同类型的词法单元。设计一个高效且准确的词法分析器，要求我们深入理解源语言的语法规则，以及如何利用状态机的机制来实现字符的分类和词法单元的生成。

本报告将详细介绍我们设计的词法分析器模块，重点讨论其实现思路、数据结构、主要函数（如 GetWord）的工作机制，以及如何通过状态机模型来进行词法分析的具体过程。通过对这些内容的分析，我们希望能够展示出词法分析器在编译器中的关键作用，并提供一个有效的解决方案，以支持后续的语法分析和代码生成。

### 词法分析器的基本原理

#### 词法分析器的作用

词法分析器是编译器的重要组成部分，负责将源代码中的字符流转换为一系列的词法单元（tokens），这些 tokens 为后续的语法分析提供基础。它通过逐字符扫描源代码，识别关键字、标识符、常量、运算符和界符，并使用有限状态机（FSM）模型进行有效识别。

词法分析器的主要任务是将字符流划分为有效的词法单元。标识符由字母开头，可以包含字母和数字，而关键字是语言的保留字。它还识别数字常量、运算符和界符，并将这些符号转换为相应的 tokens。在分析过程中，词法分析器跳过空白字符和注释，以专注于提取有效代码成分。

通过状态机模型，词法分析器能够根据当前字符和前一个字符的状态进行转换。当遇到字母时，状态机会进入标识符或关键字状态；遇到数字时，进入数字识别状态；遇到运算符或界符时，直接生成相应的 token。这样设计提高了分析器的灵活性和效率。

此外，词法分析器需处理错误情况，例如非法字符或不完整的词法单元。遇到未识别字符时，分析器应给出错误提示并尽量继续分析后续代码。

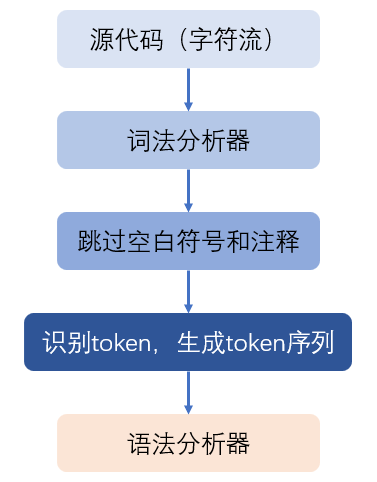


图2 词法分析器的简易工作原理

#### 状态机模型

词法分析器通常采用有限状态自动机（Finite State Machine, FSM）来实现。状态机的每个状态对应一种字符或字符序列的识别状态，状态转换根据输入字符的类型（字母、数字、操作符等）来进行。

**状态图描述**

下图展示了词法分析器中的状态机模型，各个状态代表不同的词法单元识别过程。自动机从初始状态开始，通过读取源代码的字符，根据字符类型进行状态转移，最终生成相应的词法单元。

**q0（初始状态）：**这是自动机的初始状态，也是默认状态。每次从源代码读取一个字符后，首先进入该状态。它会根据输入字符的类型，决定进入哪个状态进行处理。

**状态转移：**遇到字母时，进入标识符或关键字的识别状态 q1；遇到数字时，进入数字识别状态 q2；遇到小于号 < 时，进入 q4，以判断是否是复合运算符 <= 或 <或<>；遇到大于号 > 时，进入 q5，判断是否为复合运算符 >= 或 >；遇到操作符（如 +, -, \*, /）时，进入 q6，识别单字符操作符；遇到赋值符号: 时，进入 q3,进一步判断后续是否有=。

**q1（标识符/关键字状态）：**用于识别标识符或关键字。当读取到字母后，进入该状态，继续识别字母或数字的组合，直到遇到非字母或非数字字符。

**状态转移：**若下一个字符是字母或数字，保持在 q1 状态，继续拼接字符。若遇到非空格换行符，报错。若遇到换行空格字符，则结束标识符或关键字的识别，调用查表函数 Reserve() 判断是否为关键字，识别结束后，返回 q0，准备识别下一个词法单元。

**q2（数字识别状态）**：用于识别数字常量。当读取到数字时，进入该状态，并继续读取后续的数字字符，直到遇到非数字字符。

**状态转移：**若下一个字符是数字，保持在 q2 状态，继续拼接数字；若遇到非数字字符，结束数字的识别，判断是否出错，最终都会返回 q0。

**q3（赋值操作符状态）：**用于识别赋值操作符 =。当读取到等号 = 时，进入该状态。

**状态转移：**若下一个字符是等号，则生成赋值 token，并返回 q0。其他情况则进入出错处理。

**q4（小于号状态）：**用于处理小于号 <，并判断是否为复合运算符（如 <= 或 <>）。

**状态转移：**如果下一个字符是等号 =, 则进入 q8，识别复合运算符 <=，生成 token;如果下一个字符是大于号 >, 则进入 q11，识别不等于操作符 <>，生成 token;如果没有其他符号，则识别为 < 操作符，返回 q0。

**q5（大于号状态）：**用于处理大于号 >，并判断是否为复合运算符（如 >= 或 >）。

**状态转移：**如果下一个字符是等号 =, 则进入 q9，识别复合运算符 >=，生成 token;如果没有其他符号，则识别为 > 操作符，返回 q0。

**q6（操作符状态）：**用于识别单字符操作符或界符（如 +, -, \*, /, (, )）。

**状态转移：**识别单字符操作符后，生成对应的 token，并返回 q0。

**q7（赋值操作符终止状态）：**完成对赋值符号 = 的识别，生成 token。

**状态转移：**生成 token 后，返回 q0。

**q8（小于等于状态）：**完成对小于等于符号 <= 的识别。

**状态转移：**识别复合运算符 <= 后，生成对应 token，并返回 q0。

**q9（大于等于状态）：**完成对大于等于符号 >= 的识别。

**状态转移：**识别复合运算符 >= 后，生成对应 token，并返回 q0。

**q11（不等于状态）：**完成对不等于符号 <> 的识别。

**状态转移：**识别复合运算符 <> 后，生成对应 token，并返回 q0。

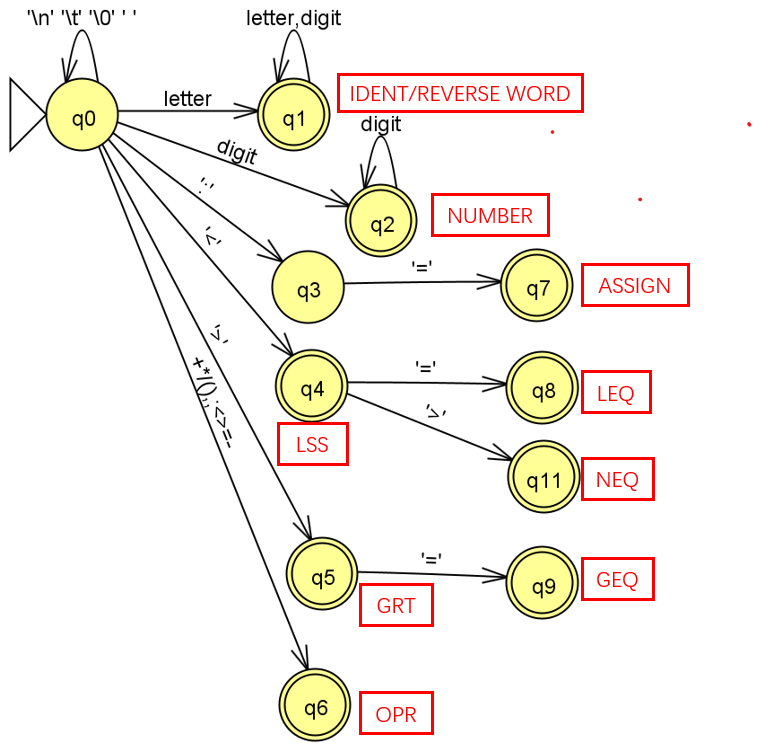


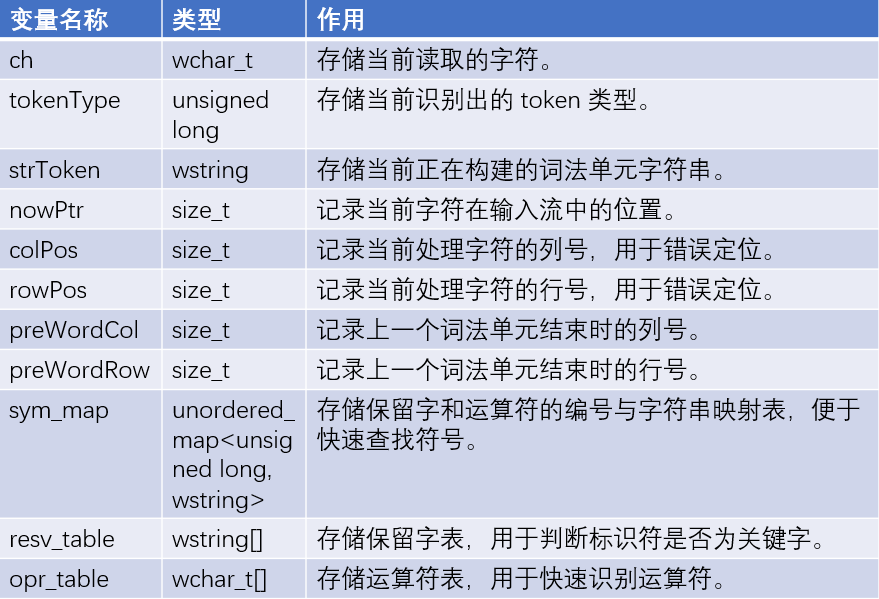
图3.词法分析器状态转换图

### 词法分析器模块设计

#### Lexer类

lexer 类是词法分析器的核心部分，负责从源代码中逐字符读取并识别词法单元（token）。它的主要任务是将源代码转化为一系列的 tokens，这些 tokens 作为后续语法分析器的输入。通过成员变量，lexer 类记录当前处理的字符、位置以及构建中的词法单元；通过成员函数，它完成字符的读取、词法单元的识别、以及错误处理。

成员变量：用于存储当前正在处理的字符、词法单元字符串、行列位置等信息，方便在识别过程中定位错误并构建词法单元。



成员函数：实现了词法单元的识别逻辑，例如判断字符是否是字母、数字，是否为运算符等，最终构建出完整的 token。



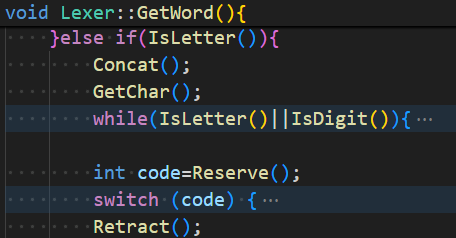
#### GetWord 函数实现

在 lexer 类中，核心函数 GetWord() 模拟了自动机的状态转换过程，用于识别源代码中的词法单元。GetWord() 函数依据当前读取的字符，决定进入不同的状态，识别标识符、数字、操作符等。其流程与状态图密切对应。

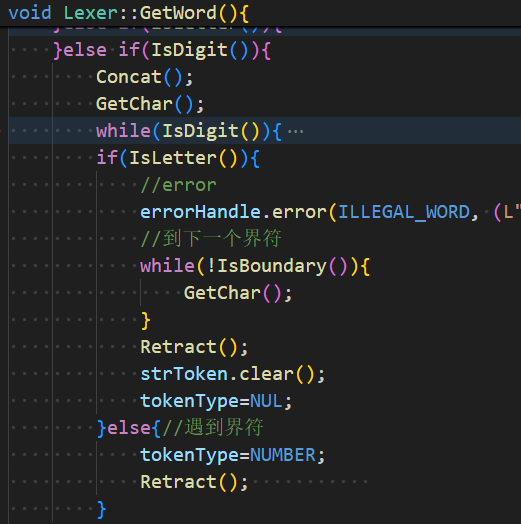
**初始状态 q0：**GetWord() 函数首先调用 GetBC() 函数跳过空白字符，再根据GetChar（）获取到的字符跳过\n,\0，进入 q0 状态。此时，依据第一个有效字符类型决定进入不同的状态。



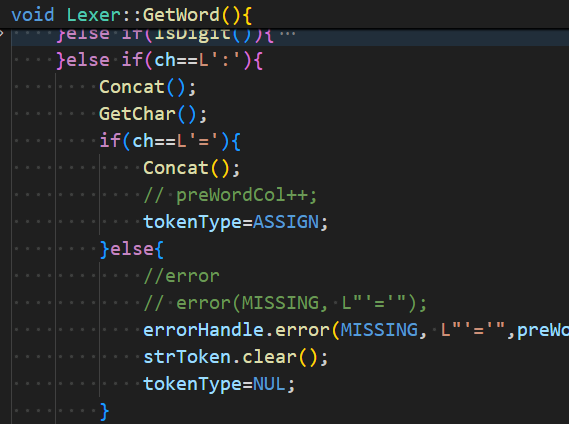
**标识符和关键字识别 q1：**当进入 q1 状态后，GetWord() 函数调用 Concat() 拼接当前字符，继续读取下一个字符，直到遇到非字母或数字字符为止。完成标识符识别后，通过 Reserve() 判断是否为关键字。



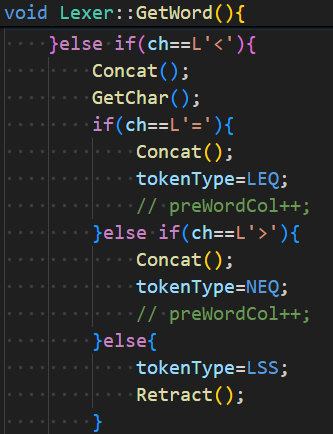
**数字识别 q2：**在 q2 状态下，GetWord() 函数继续读取数字字符并拼接，直到遇到非数字字符，如果非数字为字母则报错结束数字识别，否则为number类型，并生成数字类型的 token。



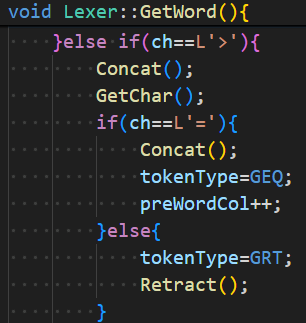
**赋值符识别 q3：**当遇到赋值符号 ：，自动进入 q3，再根据下一个字符是否是=判断是否为赋值符号，并生成相应的 token。



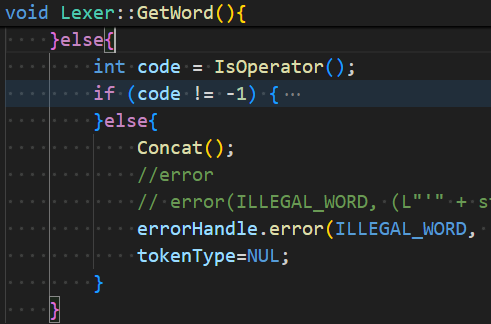
**比较操作符识别 q4, q8，q11：**当遇到<，自动进入 q4，读取下一个字符。如果是 = 或 >，则进入 q8 或 q11 状态，生成相应的 token；如果是空格或者换行，生成相应q4的token。



**比较操作符识别 q5， q9：**当遇到>,自动进入 q5，读取下一个字符。如果是 = ，则进入 q9 状态，生成相应的 token；如果是空格或者换行，生成相应q5的token。

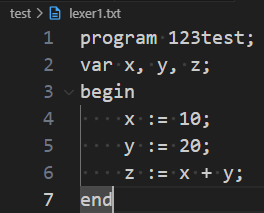
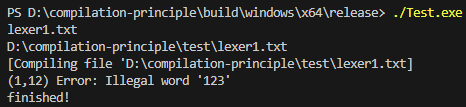


其他操作符识别q6：当遇到其他操作符,自动进入 q6，查表后判断是哪种操作符生成相应的token，如果都不是则报错。

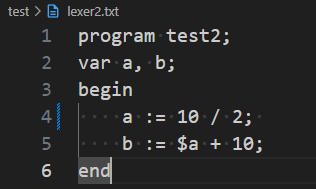
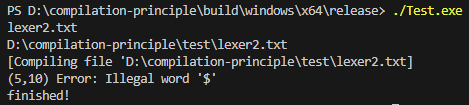


### 测试用例

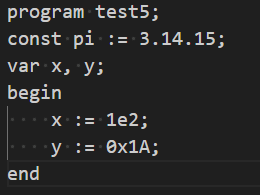
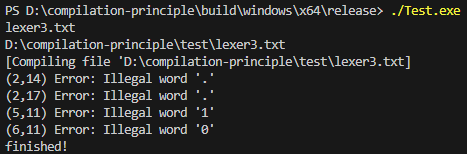
#### lexer1.txt标识符以数字开头。

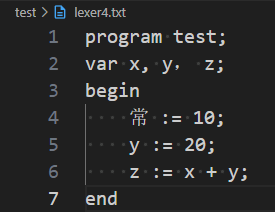
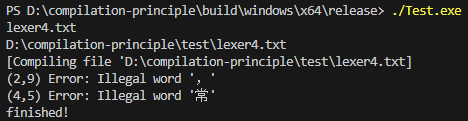
#### lexer2.txt使用了 PL/0 中未定义的符号$。

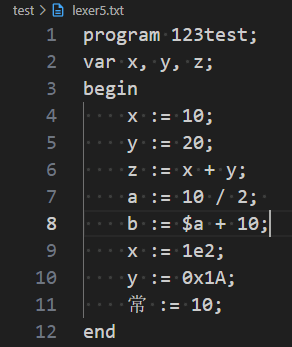
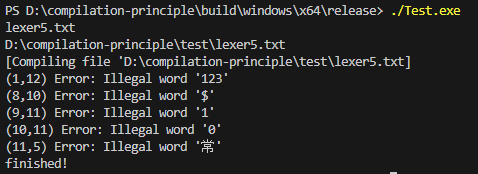
#### lexer3.txt数字中包含非法字符或格式错误。

#### Lexer4.txt中文字符

#### lexer5.txt 综合测试

## 符号表

### 引言

在编译器设计中，符号表是用于管理和记录源代码中所有标识符（如变量、常量、过程等）信息的重要数据结构。它通过维护标识符的名称、类型、作用域等属性，支持编译过程中标识符的查重、作用域管理以及类型检查等关键任务，确保程序的正确性和一致性。

本设计中的符号表采用**层次化的链式符号表**结构，支持多层嵌套的作用域管理。符号表中的每个表项不仅包含标识符的基本信息，还通过链表的方式将同一作用域内的符号连接起来，并且每个表项都包含一个回溯指针 previous，用于指向该作用域内之前定义的符号表项。这个 previous 指针使得编译器在处理嵌套作用域时，可以高效地在同一作用域内回溯到之前的定义。

符号表的核心机制之一是使用 display 数组管理各个作用域的层次结构。display 记录了当前各层作用域内最后一个符号表项的地址。每当进入一个新的作用域（如过程或函数）时，display 指向该新作用域中的第一个符号表项，并在退出时恢复到上一个作用域的状态。通过 display 和 previous 的结合，符号表可以实现快速的局部查找和回溯查找。

这种符号表的设计不仅支持作用域的嵌套管理，还能够高效处理局部变量与全局变量的查找与区分，避免了在嵌套作用域中重复查找的低效操作。在符号查找过程中，编译器从当前层次的 display 指向的符号开始，沿着 previous 指针进行同层回溯，如果未找到匹配项，再向上层 display 继续查找，直至找到匹配的符号或到达全局作用域。

符号表通过 display 数组和 previous 链接的结合，实现了高效的作用域嵌套管理，特别适用于类似 Pascal 语言的符号表结构。这种设计为编译器的语义分析和后续代码生成提供了坚实的基础，确保了符号信息的正确性与查找效率。

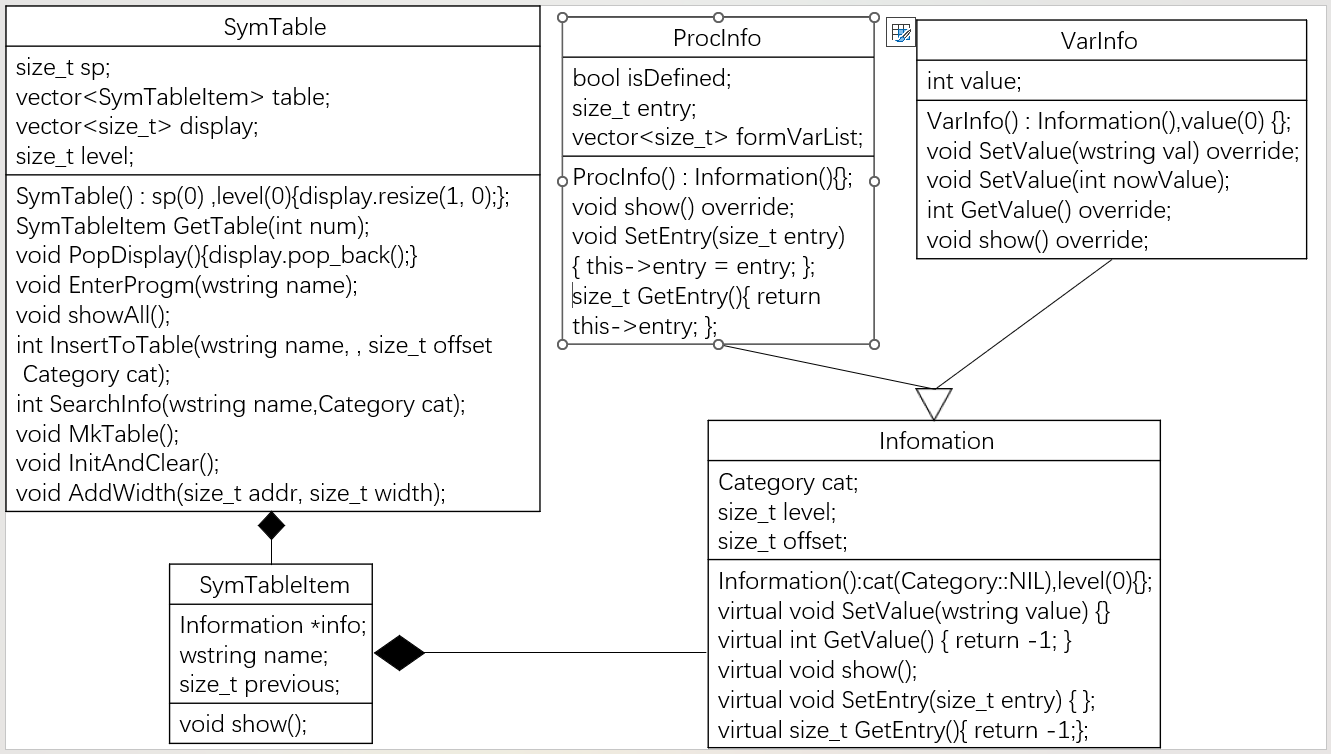
### 符号表模块设计

符号表是编译器用于存储和管理Token信息的重要数据结构。在编译过程中，符号表提供对Token的快速查找和作用域管理功能。SymTable 类是符号表的实现核心，主要功能包括Token的插入、查找、进入新作用域（过程）和清空操作。它通过一个全局的符号表数组（table）、嵌套层次管理数组（display）和回溯指针（previous）来高效管理多层嵌套的作用域。

#### Information，VarInfo，ProcInfo，SymTable，SymTableItem类

**Information、VarInfo 和 ProcInfo 类**在符号表设计中负责管理Token的具体信息，它们之间存在继承关系，并在符号表中分别承担不同的角色。这种设计不仅提高了符号表的灵活性和扩展性，还支持在编译过程中对变量和过程进行精确的语义分析和作用域控制。下面详细介绍这些类之间的关系及其在符号表中的作用。

**SymTable 类**及其相关实现是符号表的核心，用于管理程序中的标识符信息，包括变量、过程、常量等。通过维护符号表，编译器能够对标识符进行高效的插入、查找、作用域管理以及多层嵌套的过程处理。**SymTableItem 类**表示符号表中的一个项，包含符号的名称、类型以及指向前一个符号的指针 previous，以形成链表结构。它还包含一个 info 指针，用于指向符号的详细信息，如变量或过程的具体信息。



类图展示了符号表系统的类结构和继承关系。类图中包括了五个类：SymTable、SymTableItem、ProcInfo、VarInfo和Information，它们相互之间有不同的关系。

**SymTable类**是符号表的核心，负责管理符号表项的插入、查询、创建等操作，包含多个私有成员，如sp、table、display和level，这些用于管理符号表堆栈指针和层级信息。

**SymTableItem类**用于表示符号表中的项，其中包含指向Information类的指针、名字和前一个项的索引。该类支持显示表项的详细信息。

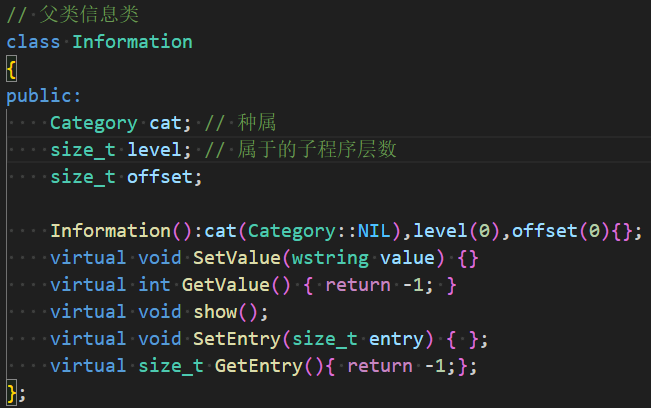
**Information是**一个基类，表示符号表项的通用信息，包含cat（类别）和level（层级）成员，并定义了一些虚函数，如SetValue、GetValue和show，使其成为其他信息类的父类。

**ProcInfo**继承自Information，表示过程信息，扩展了符号表项以包含是否定义、入口点、形式变量列表等，用于描述程序的过程或函数。

**VarInfo**也继承自Information，表示变量信息，包含变量的值，并重写了基类的SetValue、GetValue和show方法，用于操作变量的值。

* Information类

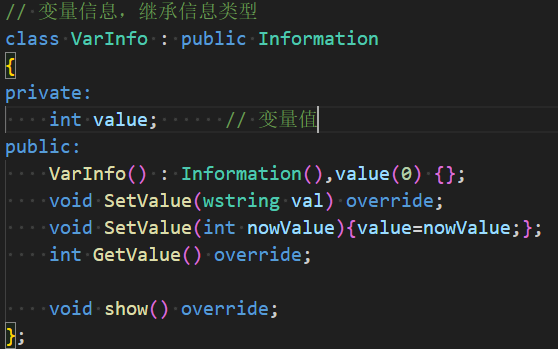
Information 是一个基类，用于表示符号表中所有标识符的通用信息。它包含了基本的属性，如标识符的类别（Category）和所在的作用域层级（level）。Information 类是一个抽象概念，具体的标识符信息由其派生类 VarInfo 和 ProcInfo 来实现。它提供了虚函数 SetValue、GetValue 和 show等函数供子类重写，以实现对不同类别标识符的操作。



* VarInfo类

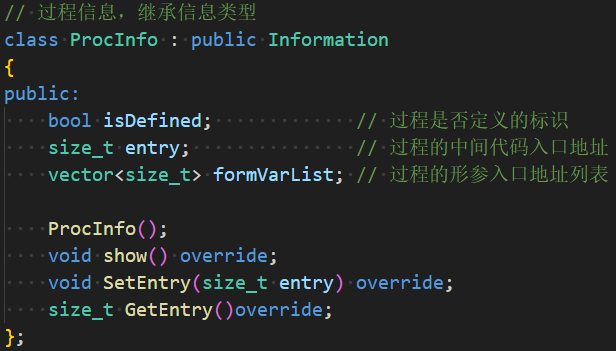
VarInfo 是 Information 类的派生类，专门用于存储变量（VAR 类型）的信息。除了继承自 Information 的通用属性外，VarInfo 还包含特有的属性 value，用于存储变量的值。

VarInfo 重写了基类的虚函数 SetValue、GetValue 和 show，实现对变量值的具体操作，如设置和获取变量的数值，以及显示变量信息。



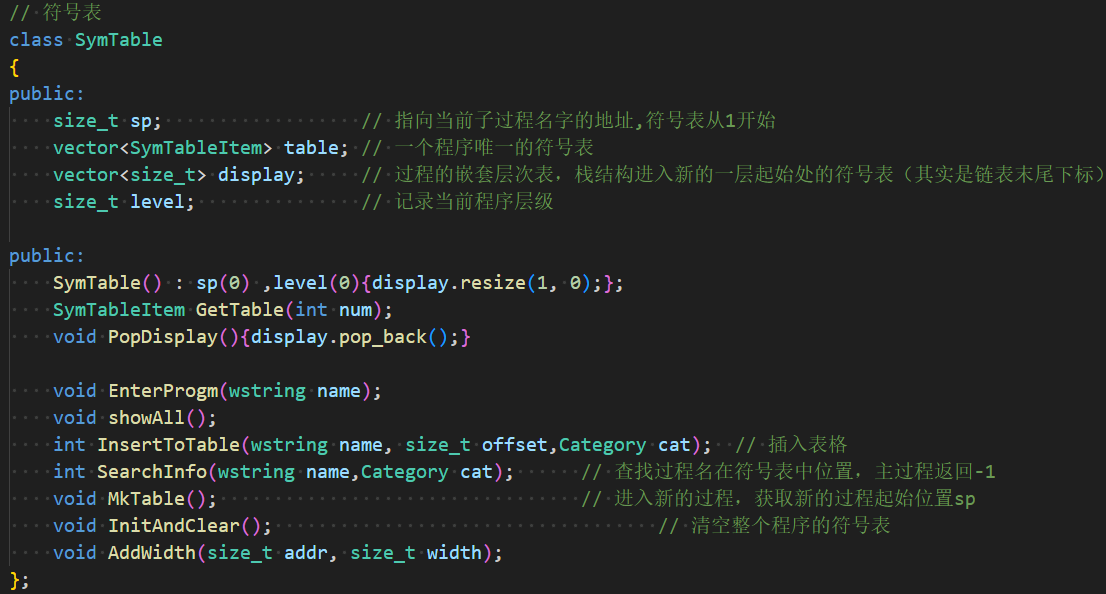
* ProcInfo类

ProcInfo 类是 Information 类的派生类，用于表示符号表中与过程（或函数）相关的信息。它扩展了基类 Information 的功能，添加了与过程定义和参数管理相关的属性。**isDefined：**表示过程是否已被定义的标志位，类型为 bool。用于记录过程是否已经有了完整的定义。如果 isDefined 为 true，表示过程已经被完整定义；如果为 false，则表示过程只有声明，没有定义。**entry：**过程的中间代码入口地址，类型为 size\_t。记录过程的中间代码开始的地址位置，用于在编译过程中定位过程的代码段。在生成和调用过程的代码时，编译器需要根据这个入口地址跳转到相应的位置执行过程。**formVarList：**过程的形参列表，类型为 vector<size\_t>。这个列表存储了过程形参在符号表中的位置（即符号表项的索引）。通过该列表可以访问每个形参的具体信息，如类型、值等，支持编译器对过程参数的管理和类型检查。



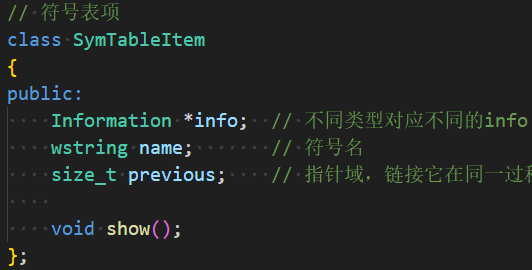
* SymTable类

**sp：**符号表指针，指向当前子过程符号表的首地址。符号表中的符号是线性存储的，sp 记录了每个过程（或函数）进入时符号表的起始位置。**table：**符号表项（SymTableItem）的集合，用于存储所有的标识符信息。每个标识符（变量、过程等）对应一个符号表项，存放在 table 中。**display：**层次表，用于记录过程的嵌套层次。display[level] 指向当前层次符号表中最后一个插入的符号表项。进入新的子过程时，会更新 display 数组以标记新的嵌套层次。**level：**记录当前的嵌套层次（作用域深度）。每当进入一个新的子过程，level 会递增，表示进入了更深的作用域。



* SymTableItem类

SymTableItem 类表示符号表中的一个项，包含符号的名称、类型以及指向前一个符号的指针 previous，以形成链表结构。它还包含一个 info 指针，用于指向符号的详细信息，如变量或过程的具体信息。



#### 管理各个作用域的层次结构display

在多层嵌套的编程语言中（如 Pascal、C 等），作用域的嵌套管理是符号表的一个关键功能。display 数组用于记录各个作用域中符号表的最后一个符号的位置，便于符号查找和作用域切换。具体来说，display[level] 记录了当前层级（level）中的最新符号表项的位置。通过使用 display 数组，编译器可以快速确定当前作用域的符号范围，并进行高效的符号查找。每次进入一个新的作用域（如过程或函数），都会更新 display 以指向新的作用域的符号表起始位置。当离开当前作用域时，display 则恢复到上一层的状态。

在 SymTable 类中，display 被定义为一个 vector<size\_t> 类型的数组，其元素是符号表项的索引。具体定义如下：

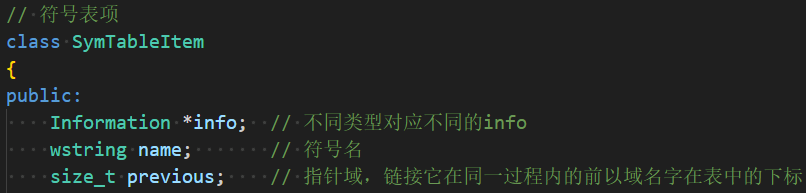


其中，display[i] 表示第 i 层作用域中最新的符号表项的下标。这个数组在符号表的初始化时设置为大小为1，并用0填充，表示全局作用域还未有符号。

#### 回溯指针previous

previous 指针在符号表设计中起着关键作用，用于实现符号表项之间的回溯链接。它帮助编译器在同一作用域内有效地查找和管理标识符，支持作用域嵌套结构的符号管理。previous 是符号表中每个符号表项的成员变量，用于指向同一作用域中上一个定义的符号表项。这种链接结构形成了一个链表，使得同一作用域内的所有符号表项按照插入顺序链接在一起，从而可以高效地回溯查找符号。

在符号表的实现中，每个符号表项（SymTableItem）都有一个 previous 指针，用于链接同一层次内的符号表项。具体定义如下：



previous 存储了当前符号表项在符号表数组 table 中的前一个符号项的索引。当 previous 为 0 时，表示这是当前作用域中的第一个符号表项（没有更早的符号）。

#### SearchInfo函数实现

SearchInfo 函数用于在符号表中查找指定的标识符，并返回其在符号表中的位置。它的作用是在编译过程中，根据标识符的名称和类别，找到最接近当前作用域的声明，确保符号的正确引用。这个函数的实现结合了 display 和 previous 两个数据结构，支持在嵌套的多层作用域中高效地查找符号。

在查找标识符时，编译器会首先从 display[level] 开始查找，然后沿着 previous 链回溯，直到找到目标符号或查找完当前层级的所有符号。如果在当前层次找不到符号，则逐层向上回溯，依次检查更高层次的 display。

* **工作流程**

SearchInfo 函数通过从当前层次（level）向上回溯到全局层次（level == 0），逐层查找符号表中的符号项，直到找到匹配的标识符或遍历完所有层次。具体步骤如下：

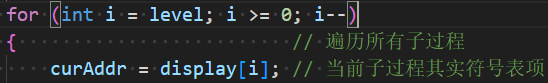
**检查是否在全局层级并且符号表为空**

如果当前层级为 0 且 display[0] == 0，意味着全局层级没有任何符号定义，可以直接返回 -1，表示未找到该符号。



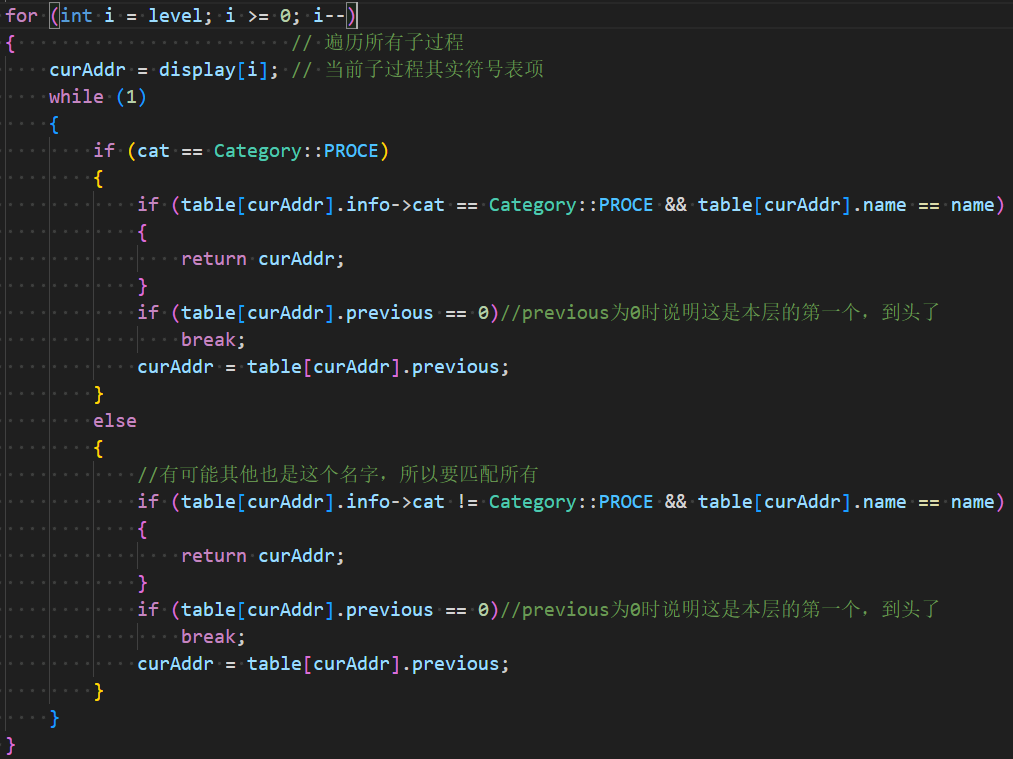
**从当前层级开始逐层查找**

从当前层级（level）向上遍历，逐层向全局层级（level == 0）回溯。每次循环，i 表示当前正在查找的层次。display[i] 记录了层级 i 的最新符号表项的位置，即最近插入的符号表项。



**在当前层级内沿着 previous 链进行查找**

通过 curAddr = display[i] 获取当前层级的符号表项位置，从这个位置开始沿着 previous 链进行查找。previous 指针链接了同一作用域中的所有符号表项，通过不断更新 curAddr = table[curAddr].previous，可以依次访问当前层级中的每个符号。如果找到一个与目标标识符 name 和类别 cat 相匹配的符号，则立即返回其在符号表中的位置（curAddr）。如果当前层级未找到，则向上一层继续查找如果遍历完当前层级的所有符号表项仍未找到匹配的符号，则进入下一次循环，继续在上一层级（i-1）查找，直到到达全局层级。



**查找失败时返回 -1**

如果所有层级都没有找到匹配的符号，返回 -1 表示查找失败。

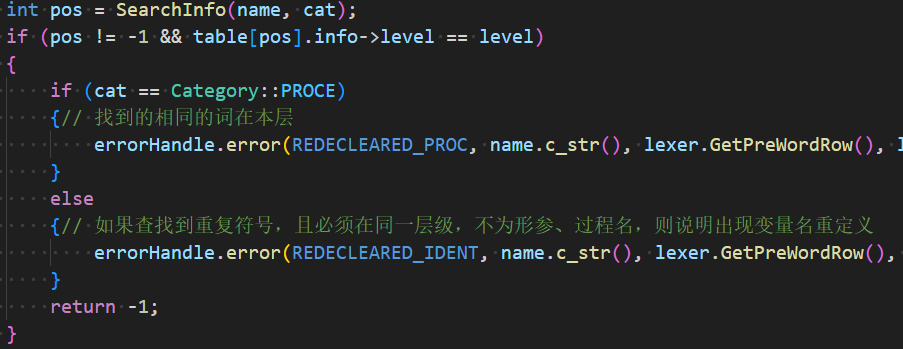
#### InsertToTable函数实现

InsertToTable 函数用于将新的符号（变量、常量、过程等）插入到符号表中。它负责处理符号的插入、查重检查以及更新符号表的结构。这一函数结合了 display 和 previous 的使用，以管理符号表的层次结构并确保符号的唯一性。接下来，我们详细介绍 InsertToTable 函数的实现原理和具体操作。

* **工作流程**

**查重检查**

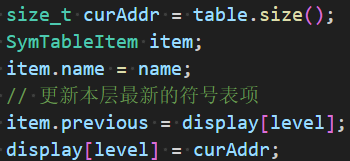
插入符号之前，编译器需要确保在当前作用域中没有重复定义的符号。这一步是通过 SearchInfo 函数实现的，SearchInfo 从当前层级开始，沿着 previous 链逐步回溯，查找是否有同名符号。



调用 SearchInfo 函数，在当前层次的符号表中查找是否已经存在相同名称的符号。如果 SearchInfo 返回的 pos 不为 -1，并且该符号的层次（table[pos].info->level）与当前层次相同（level），则表示在当前作用域中存在相同名称的符号，说明符号重定义。如果符号是过程（Category::PROCE），则报告过程重定义错误；如果符号是其他类别（如变量、常量等），则报告标识符重定义错误。如果发生重定义，返回 -1，表示插入失败，函数执行结束。

**创建新的符号表项**

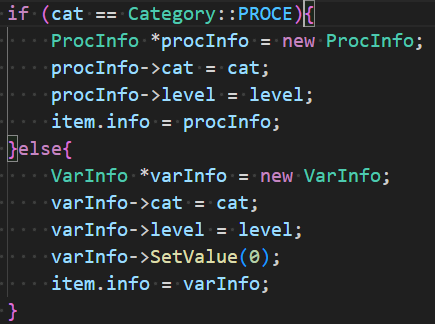
如果符号未重定义，则可以创建新的符号表项。每个符号在符号表中对应一个 SymTableItem，包含符号的名称、信息（info）以及 previous 指针。



 获取当前符号的地址：通过 table.size() 获取符号表当前的大小，即新符号插入的位置（即符号表的末尾）。创建符号表项：声明一个新的 SymTableItem 对象 item，并设置其 name 属性为要插入的符号名称。设置 previous 指针：previous 指针用于链接当前层次的符号链。item.previous 设置为 display[level]，表示新插入的符号的 previous 指针指向当前层次的最后一个符号项。更新 display：display[level] 更新为新插入符号的位置 curAddr，即当前层次的最新符号表项。这样后续查找时会优先从新符号开始查找。

**设置符号信息（info 字段）**

符号的具体信息（如类型、值等）存储在符号表项的 info 字段中。不同类别的符号（如变量、常量、过程等）对应不同的 info 类型，例如，变量使用 VarInfo，过程使用 ProcInfo。



处理过程符号：如果符号的类别是过程（Category::PROCE），则创建 ProcInfo 对象。将符号的类别（cat）和层次（level）信息赋值给 ProcInfo 对象，然后将其赋值给 item.info。处理变量符号：如果符号是变量、常量或其他非过程类型，则创建 VarInfo 对象，并初始化其值为 0。将符号的类别和层次信息赋值给 VarInfo 对象，最后将其赋值给 item.info。这种设计支持符号表项的多态处理，不同类别的符号可以有各自的属性和行为。

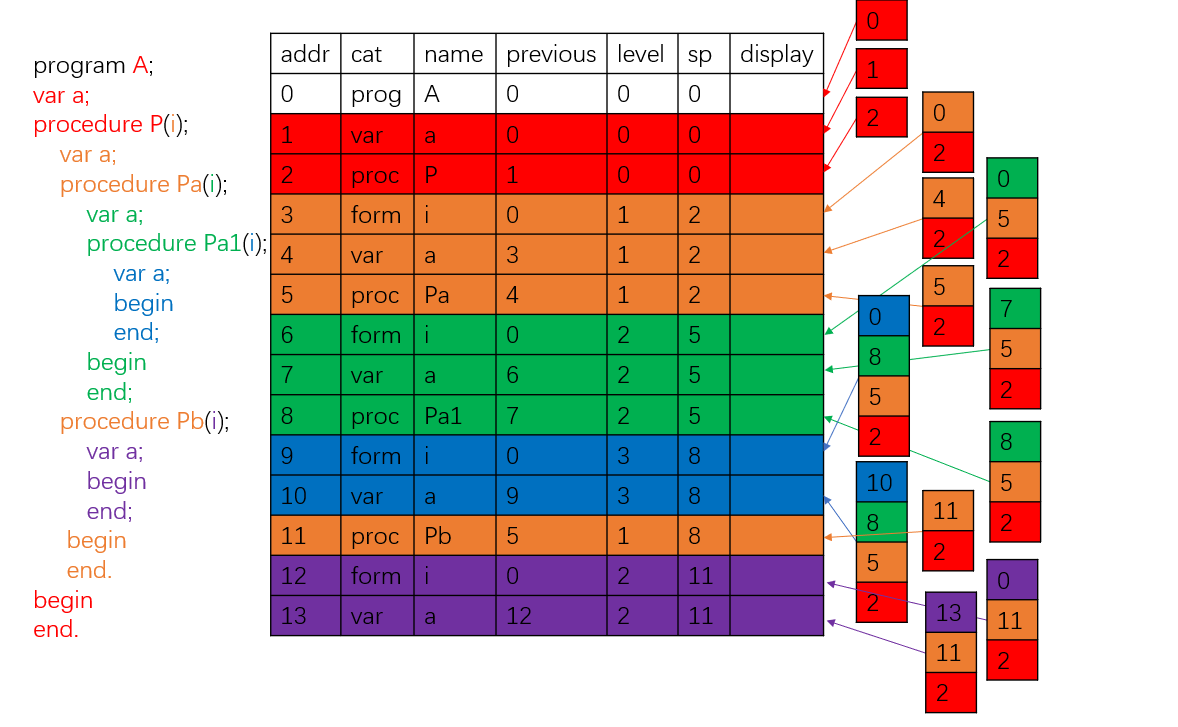
**插入符号表**

最后一步是将新创建的符号表项插入到符号表中，并返回新符号的位置。



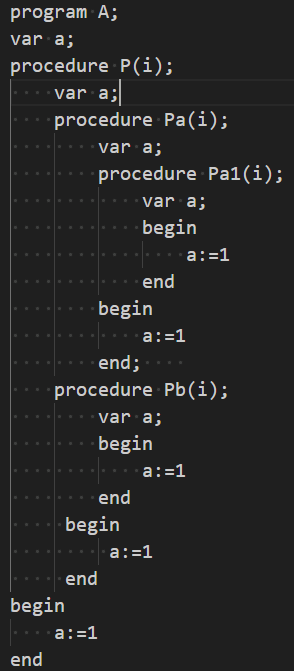
使用 table.push\_back(item) 将新创建的符号表项 item 插入到符号表的末尾。返回新符号的地址 curAddr，表示插入成功。

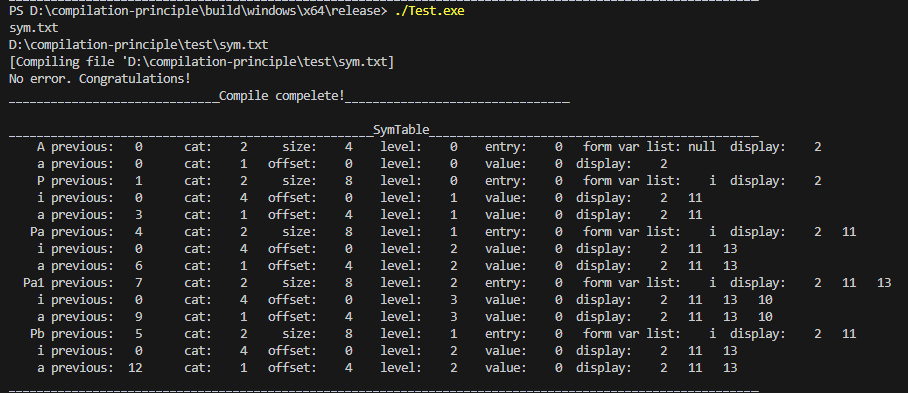
#### 程序符号表流程总览



最左边是待执行的程序源代码，中间是符号表本体，最右边是符号表的 display 表。符号表中的颜色对应源代码中的颜色，不同的颜色区分了不同的作用域。符号表通过维护每个符号的名称、类别、层级、前一个符号项的位置（previous）以及在不同作用域中的层次（display 和 sp）来管理程序中的标识符。display 数组用于记录每个作用域的最新符号表项位置，确保符号的查找优先在当前作用域进行，而 previous 则链接了同一层次内的符号项，形成链式结构。在嵌套的子过程或作用域中，display 和 previous 的结合保证了符号表能够有效管理局部和全局作用域，支持符号的插入、查找和回溯。

代码测试：





## 错误处理单元

### 引言

在编译器设计中，错误处理单元是不可或缺的组件之一。它负责识别和处理编译过程中出现的各种错误，包括词法错误、语法错误和语义错误。由于编译过程的复杂性，用户在编写代码时可能会遇到多种类型的错误，及时、准确地处理这些错误对于提升编译器的用户体验至关重要。

错误处理单元的设计旨在提供清晰的反馈，以便用户能够迅速理解问题所在并进行修正。通过合理的错误分类和详细的信息输出，错误处理单元能够帮助用户有效地排查代码中的问题，并提高编程的效率。此外，良好的错误处理机制也增强了编译器的鲁棒性，减少了因错误导致的编译中断。

在本部分，将详细介绍错误处理单元的结构、主要功能及其与其他编译器组件之间的关系，以展示其在整个编译过程中的重要性。

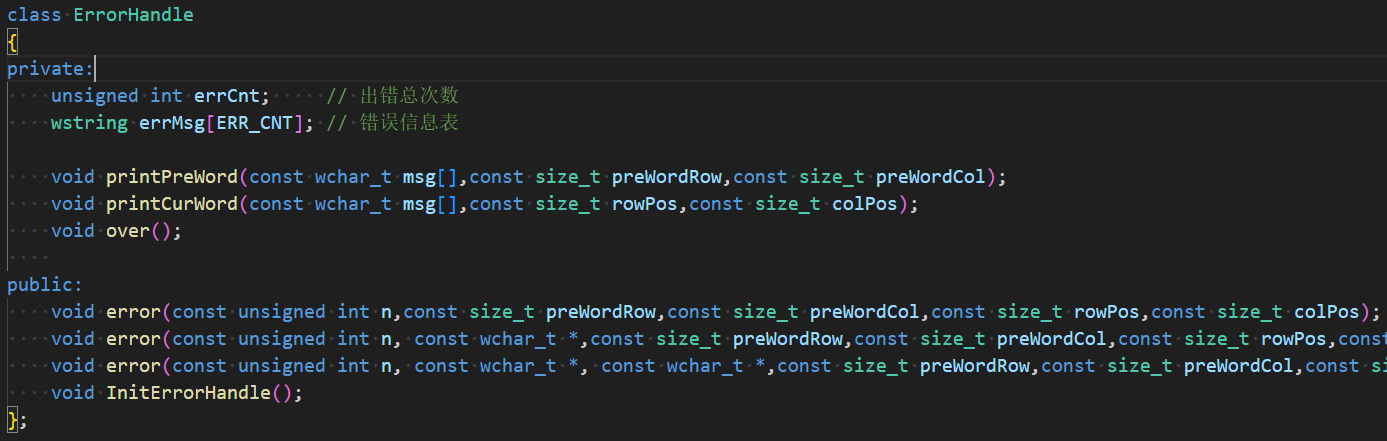
### 错误处理单元模块设计

#### 错误定义



#### ErrorHandle类

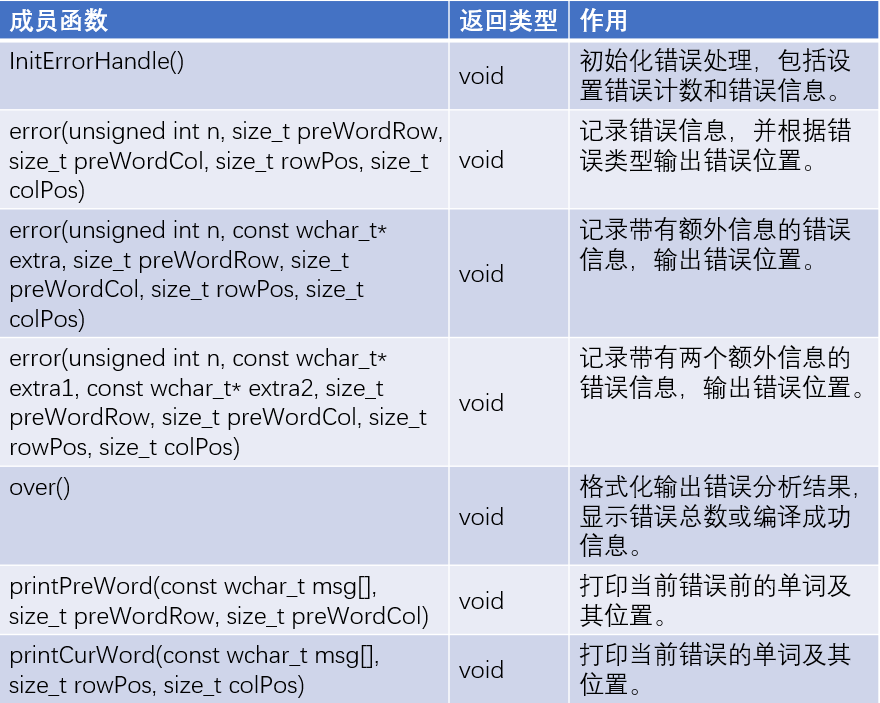
ErrorHandle 类是编译器中的一个关键组件，主要用于管理和处理编译过程中发生的各种错误。其设计目标是提供一个清晰而有效的错误记录与报告机制，以帮助用户快速识别和修复源代码中的问题。



成员变量：



成员函数：



## 语法分析

### 引言

语法分析器是编译器和解释器中的核心模块之一，其主要任务是将词法分析器生成的Token按照语言的语法规则组织成符合特定结构的语法树。通过对输入程序的结构化解析，语法分析器能够验证程序的正确性并为后续阶段（如语义分析或代码生成）提供基础。

本项目的语法分析器采用递归下降解析法实现，结合 parser.cpp 和 parser.hpp 提供的功能模块，能够解析简单的上下文无关文法。本项目代码结构清晰，具有以下特点：

**模块化设计：**将解析逻辑划分为多个函数，每个函数对应一个语法规则，便于维护和扩展。

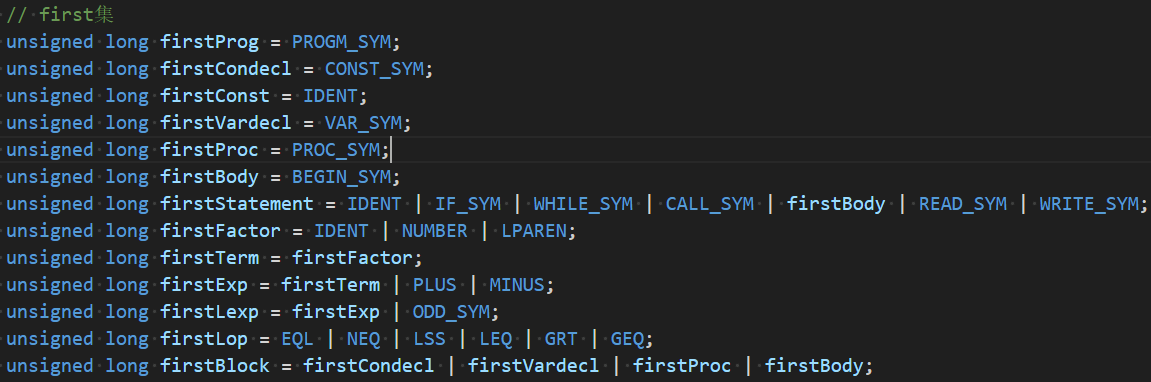
**错误处理机制：**内置了基本的错误检测和恢复功能，确保在输入不符合预期时提供明确的错误信息。

**高效解析：**通过对文法规则的精确设计和递归调用的优化，解析器能够快速处理输入。

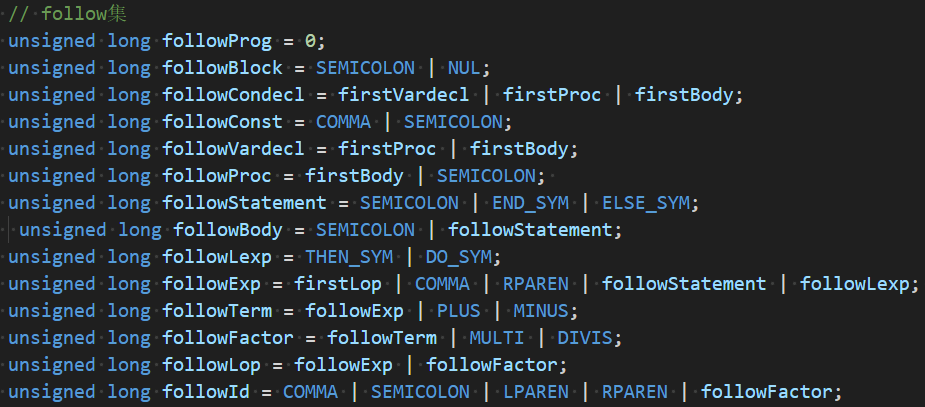
语法分析器在整个编译流程中扮演承上启下的角色，既需要准确识别程序的语法结构，又需为后续的语义分析提供可靠的支持。本文结合具体实现代码，详细说明了该语法分析器的设计理念和技术细节，并展示其在实际解析任务中的应用效果。

### 证明

所有非终结符的first集



所有非终结符的follow集



（1）自上而下语法分析（如LL(1)分析）要求文法不能有左递归，并且需要左因子化。逐步分析PL/0文法的各个规则：

文法规则中不存在直接左递归，如：

<prog> → program <id>; <block> 中，左递归不存在。

<block> → [<condecl>][<vardecl>][<proc>]<body> 中，非终结符在扩展时没有直接回到<block>。

其他规则类似，可以确认文法中没有直接左递归

间接左递归也可以通过进一步展开推导检查，结合FIRST集和FOLLOW集，可证明不存在间接左递归。

（2）对于文法的每一个非终结符的各个产生式的候选首符集两两不相交.

问题主要出现在多条产生式前缀相同时。例如：

<statement> → <id> := <exp>

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

| while <lexp> do <statement>

| call <id>([<exp>{,<exp>}])

| read (<id>{,<id>})

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

| <body>

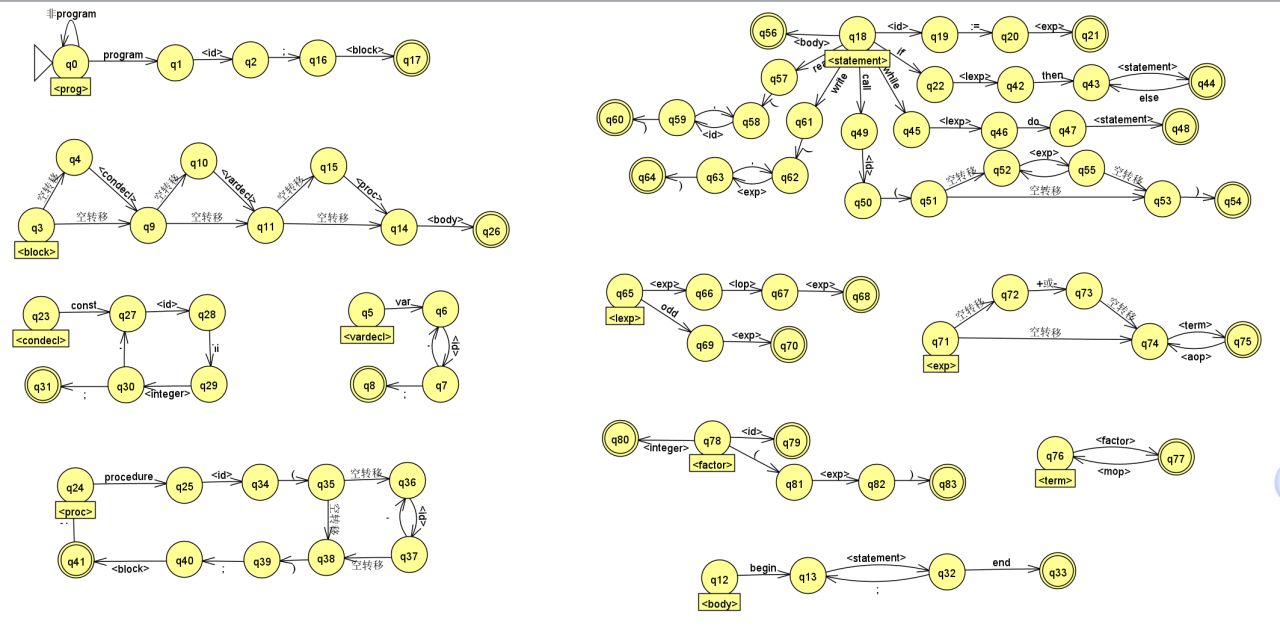
这些规则不会引发歧义，因为其开头的FIRST集合是互不相交的（例如if、while、call等是明确区分的终结符）。

（3）对文法中的每个非终结符A，若它存在某个候选首符集包含空，其first集和follow集不相交

（4）文法无二义性。**唯一性推导：**对任何输入串，从起始符号<prog>到终结符串，推导过程是唯一的。PL/0文法通过明确的关键字（如program、begin、if、while等）区分了不同规则，避免了二义性。**控制流清晰：**通过if-then-else等规则，可以根据FIRST集判断具体分支，例如then和else的关键字可以唯一确定控制流。

因此，PL/0文法可以进行LL(1)自上而下语法分析。

### 语法分析器模块设计



语法分析图

#### void block();

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

block 函数是语法分析器中的核心组成部分之一，专门负责解析 PL/0 语言的块结构。一个块结构可以包含常量定义（const）、变量定义（var）、过程声明（procedure），以及语句体（body）。其设计旨在逐步分解和解析每个部分，确保程序逻辑和结构的正确性。

block 函数采用递归下降解析法。函数通过judge检查当前记号是否属于 firstBlock，假如进函数时不符合则报错，往后直到找到属于firstBlock和followBlock才跳出judge。假如是firstBlock则进入if，否则说明是followBlock，直接退出block。

if具体实现中，通过按顺序检查 lexer.GetTokenType() 是否匹配 firstCondecl、firstVardecl 和 firstProc 来动态解析各部分内容，并在最终调用 body() 解析复合语句部分。

函数的设计注重错误处理，例如在缺少必要符号时提供明确错误提示并跳过无效部分，保证解析流程的鲁棒性和连续性。这种模块化和容错设计为复杂语法结构的高效解析和未来扩展奠定了基础。



#### void proc();

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

proc 函数是语法分析器的重要部分之一，专门负责解析 PL/0 语言中的过程声明。过程声明包含过程的定义以及嵌套的其他过程，结构复杂但层次清晰。其设计目标是通过逐步解析过程名称、参数列表、过程体和嵌套过程，确保过程声明的正确性并支持过程嵌套。

proc 函数采用递归下降解析法，通过判断当前记号是否属于 PROC\_SYM（过程声明的 FIRST 集）来决定是否进入解析流程。若当前记号不匹配 procedure，则通过 judge 结合 proc的FOLLOW 集进行错误恢复，以确保在错误发生时尽可能继续后续解析。

假如一开始属于PROC\_SYM则进入if进行后续判断，proc 通过调用 lexer.GetWord() 获取下一个token，通过 lexer.GetTokenType() 确定当前记号类型是否为 IDENT，表示过程的名称。接着，获取下一个token检查是否为左括号 (，表示开始解析过程参数列表。如果是左括号，我们读取它并进入参数列表的解析逻辑。在参数列表的解析过程中，首先检查下一个token是否是标识符 IDENT，表示参数的名称。如果是标识符，我们读取该标识符，并检查后续是否有逗号，如果有则继续解析下一个标识符。如果没有逗号或标识符，错误处理机制会提供提示并进行错误恢复。当所有参数解析完毕后，检查是否遇到右括号 ) 来结束参数列表。如果缺少右括号，同样会给出错误提示。如果参数列表为空，即 ()，会直接跳过不再进行参数解析。解析完参数列表后，proc 函数接着检查是否有分号 ; 来结束参数列表和过程体之间的分隔符。如果存在分号，调用 lexer.GetWord() 继续读取并进入过程体的解析，调用 block() 函数解析过程体（<block>）。block 函数将会处理常量定义、变量定义、嵌套过程以及语句部分。过程体解析完毕后，如果当前记号为分号 ;，说明可能还有其他嵌套的过程声明。此时通过递归调用 proc() 来解析后续的嵌套过程，直到没有更多的过程声明为止。如果在任何一个步骤中遇到缺失或非法的符号，proc 函数会通过 errorHandle.error() 报告错误，给出错误的详细信息（如缺失的符号或非法的过程声明格式），并跳过错误部分继续解析后续内容。

函数设计注重错误处理，例如在过程名称、参数列表或分号缺失时，提供明确的错误提示并尝试恢复，避免因单个错误中断整个过程的解析。此外，proc 的递归设计使其能够自然地支持多层嵌套的过程结构，从而满足 PL/0 语言复杂过程声明的需求。

#### void statement();

**<statement> → <id> := <exp>**

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

**|while <lexp> do <statement>**

**|call <id>（[<exp>{,<exp>}]）**

**|<body>**

**|read (<id>{，<id>})**

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

statement 函数解析 PL/0 中的各种语句，包括赋值语句、条件语句（if）和循环语句（while）。对于赋值语句，首先检查标识符（IDENT），然后解析赋值符号（:=）及右侧的表达式。对于 if 语句，解析条件表达式、then 和可选的 else 语句。对于 while 语句，解析条件表达式和 do 后的语句体。如果当前语句不符合预期，errorHandle 会报告错误并继续解析后续语句。如果当前记号不符合任何语句类型，judge 函数会进行错误恢复。

#### void constA();

**<const> → <id> := <integer>**

constA 函数是语法分析器中的一个关键部分，负责解析 PL/0 语言中的常量声明。常量声明由一个标识符 <id> 和一个整数值 <integer> 组成，语法结构简单且易于理解。constA 函数的设计目标是通过逐步解析常量名称和其值，确保常量声明的正确性，并且在遇到错误时能够进行适当的错误报告和恢复。

constA 函数采用递归下降解析法，首先通过判断当前token是否为标识符 IDENT 来决定是否进入解析过程。如果当前token不是标识符 IDENT，则通过 judge 函数结合 FOLLOW 集进行错误恢复，确保即使在遇到错误时，语法分析能够继续进行。如果是IDENT，则获取下一个token。在解析完常量名称后，接着检查当前记号是否为赋值符号 :=。如果是，我们继续读取该记号，进入常量赋值部分。如果赋值符号缺失，则通过错误处理报告，并跳过错误部分。常量的值必须是一个整数（NUMBER）。constA 函数通过 lexer.GetTokenType() 检查当前记号是否为一个数字常量。如果是数字，调用 lexer.GetWord() 获取该数字并继续解析。如果当前记号不是数字常量，语法分析器会报告错误，指明缺少合法的整数值，并尝试恢复。

如果在解析过程中发现常量声明的名称、赋值符号或整数缺失，constA 函数会通过 errorHandle.error() 报告错误，并给出详细的错误信息（例如，缺少 <id> 或 :=）。错误处理不仅仅报告错误，还尽量继续解析后续部分，避免解析过程中断。

#### void condecl();

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

condecl 函数负责解析 PL/0 语言中的常量声明部分。常量声明由一个或多个常量组成，每个常量包含一个标识符 <id> 和一个整数值 <integer>，并且常量之间由逗号分隔。condecl 的设计目标是通过逐个解析常量声明，确保常量定义的正确性，并支持多个常量的连续声明。

首先，condecl 函数检查当前记号是否为 const 关键字。若是，函数通过 lexer.GetWord() 读取该记号并进入常量声明的解析部分。如果当前记号不是 const，则通过错误恢复机制报告错误，并跳过无效部分。接着，函数检查下一个记号是否符合常量声明的语法规则，属于 firstConst 集。若是，调用 constA() 函数解析第一个常量声明（<id> := <integer>）。如果当前记号是逗号（,），表示后续有更多常量声明。condecl 函数会读取逗号，并继续解析下一个常量。如果接下来的记号符合常量声明的规则，调用 constA() 继续解析。如果不是，错误处理机制会报告缺失的常量声明。解析完所有常量后，condecl 函数检查是否有分号（;）来结束常量声明。如果缺少分号，错误处理机制会报告错误并跳过。

#### void vardecl();

**<vardecl> → var <id>{,<id>};**

vardecl 函数负责解析 PL/0 语言中的变量声明部分。变量声明由一个或多个变量组成，每个变量是一个标识符 <id>，并且变量之间由逗号分隔。vardecl 函数的设计目标是通过逐个解析变量声明，确保变量声明的正确性，并支持多个变量的连续声明。

首先，vardecl 函数检查当前记号是否为 var 关键字，表示开始解析变量声明部分。如果是 var，则通过 lexer.GetWord() 读取并进入后续变量的解析。如果当前记号不是 var，则通过错误恢复机制报告错误，并跳过无效部分。在解析完 var 后，vardecl 函数检查下一个记号是否为标识符 IDENT，表示变量名称。如果是标识符，调用 lexer.GetWord() 读取并继续解析。如果不是标识符，报告错误并跳过。如果当前记号是逗号（,），表示后续有更多变量声明。vardecl 函数继续解析后续的变量。如果接下来的记号是标识符，继续解析下一个变量。否则，错误处理机制报告缺失的变量名。解析完所有变量后，vardecl 函数检查是否有分号（;）来结束变量声明。如果缺少分号，错误处理机制会报告错误并跳过。

#### void term();

**<term> → <factor>{<mop><factor>}**

term 函数负责解析乘法和除法运算。首先，它解析第一个因子（<factor>），然后继续检查是否存在乘法（\*）或除法（/）操作符。如果存在，它会继续解析下一个因子并将运算符与因子连接起来，形成一个完整的乘法或除法表达式。错误处理机制会确保在遇到连续运算符（如 \*\* 或 //）时报错，并处理缺失因子的情况。若当前符号不符合 term 的开头规则，judge 函数会进行错误恢复。



#### void factor();

**<factor>→<id>|<integer>|(<exp>)**

factor 函数负责解析表达式中的基本单元，包括标识符、数字常量和括号表达式。首先，它检查当前记号是否是标识符（IDENT）或数字常量（NUMBER），并解析这些常量。如果是括号表达式，函数会解析括号内的完整表达式，并确保括号的配对正确。如果当前记号不符合因子的规则，judge 函数会进行错误恢复。factor 的设计确保了对基本表达式单元的准确解析。



#### void prog();

**<prog> → program <id>；<block>**

prog 函数解析 PL/0 语言中的程序结构。首先，它检查当前记号是否为 program 关键字，若是，则解析程序名称。解析完程序名称后，检查是否有分号（;），如果有，获取下一个token后调用 block 函数解析程序体。如果在 program 后缺少程序名或分号，或其他语法错误发生，errorHandle 会报告错误。若当前记号不符合程序开头符号，judge 函数会进行错误恢复。

#### void body();

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

body 函数负责解析复合语句块（begin ... end）。首先，检查当前记号是否是 begin，如果是，开始解析 begin 块。接着，解析语句部分，每条语句之间由分号（;）分隔。statement 函数用于解析单条语句。如果遇到分号，继续解析后续语句。最后，检查是否有 end 来结束 begin 块。如果缺少 end，则报告错误。若当前记号不符合复合语句的开头，judge 函数会进行错误恢复。

#### void lexp();

**<lexp> → <exp> <lop> <exp>|odd <exp>**

lexp 函数解析逻辑表达式。首先，它检查当前记号是否符合表达式的开头符号（firstExp），如果符合，则解析左侧的表达式。接着，检查是否存在逻辑操作符（如 =, <>, <, <=, >, >=）。如果有逻辑操作符，函数会解析右侧的表达式。如果逻辑操作符缺失或存在其他错误，错误处理机制会报告缺失并尝试继续解析。若当前记号是 odd 操作符，则解析该操作符并继续解析表达式。若符号不符合期望的逻辑表达式起始符号，judge 函数会进行错误恢复。



#### void exp();

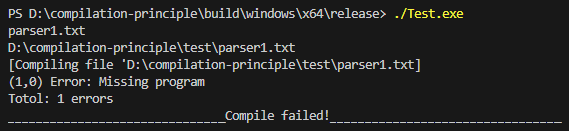
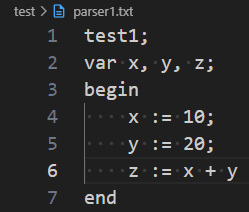
**<exp> → [+|-]<term>{<aop><term>}**

exp 函数负责解析加法和减法运算。首先，它检查当前符号是否有加号或减号（+ 或 -），若是则获取下一个token。然后，它进入term解析term。如果遇到加号或减号运算符，它将继续解析接下来的 term。如果输入不属于firstExp，judge 函数将用于错误恢复，确保语法分析器能够继续解析后续部分。

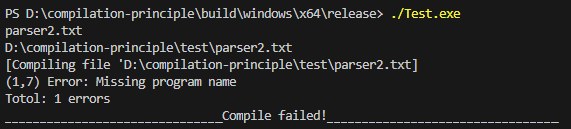
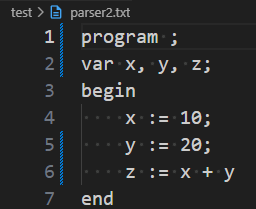


### 测试用例

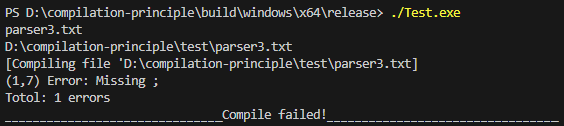
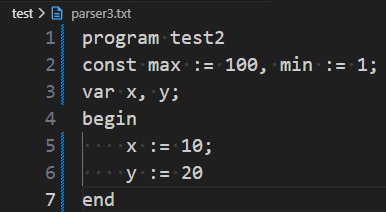
#### parser1.txt <program>： 程序开头缺少 program 关键字。



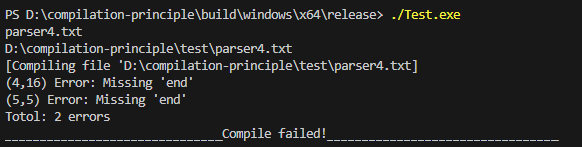
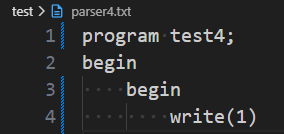
#### parser2.txt <program>： 程序开头缺少 id。



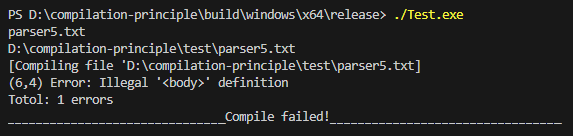
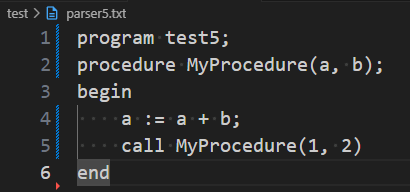
#### parser3.txt <program>： 程序开头分号缺失。



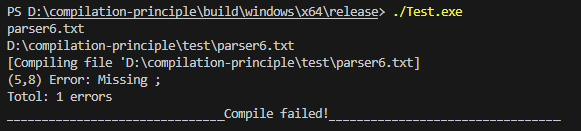
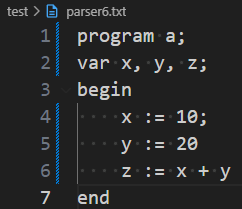
#### parser4.txt <body>： 嵌套的 begin-end 块未正确闭合。



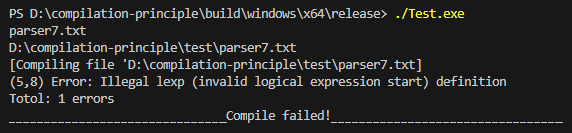
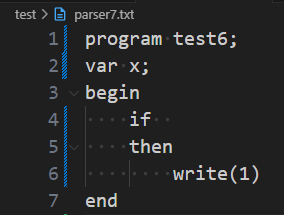
#### parser5.txt <body>： 过程定义缺少 begin-end 块。



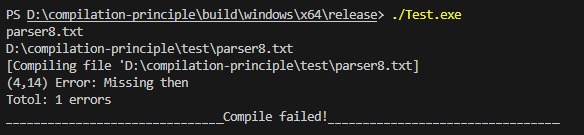
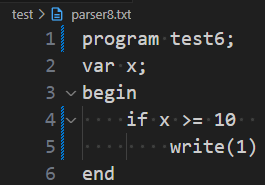
#### parser6.txt <body>： 多个statement缺少分号。



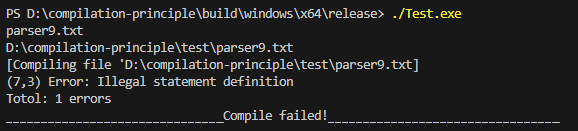
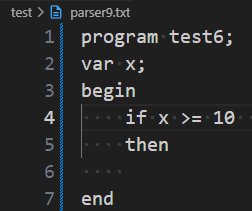
#### parser7.txt <if>： if后lexp缺失。



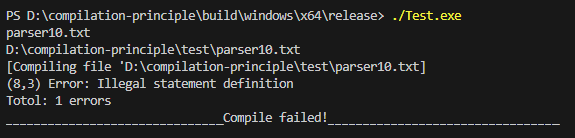
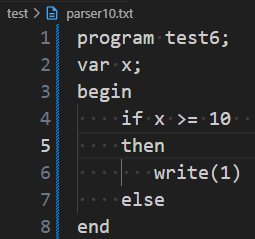
#### parser8.txt <if>： then缺失。



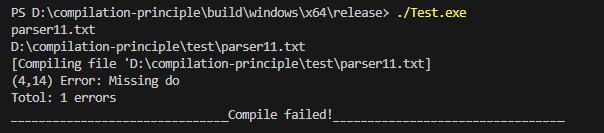
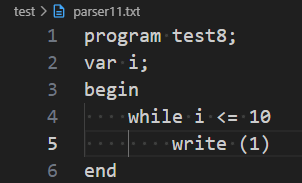
#### parser9.txt <if>： then后面statement缺失。



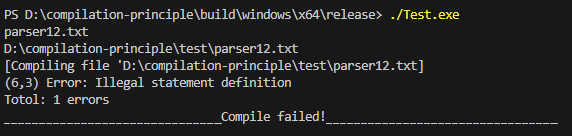
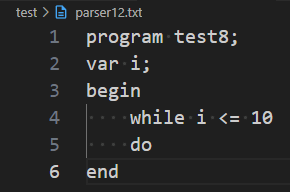
#### parser10.txt <if>： else后面statement缺失。



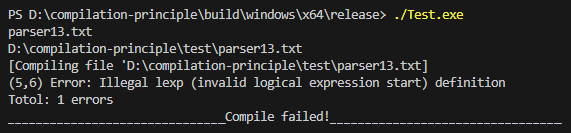
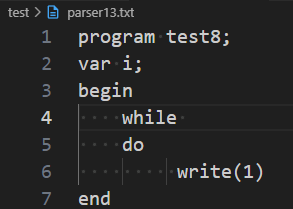
#### parser11.txt <while>： 缺少 do。



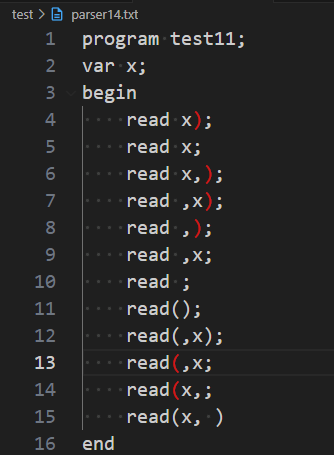
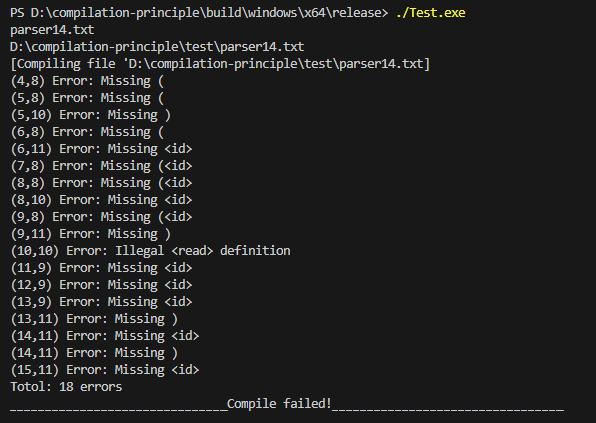
#### parser12.txt <while>： do后缺少 statement。



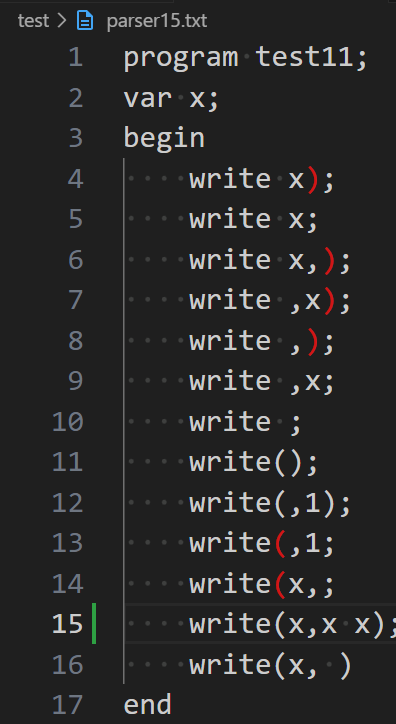
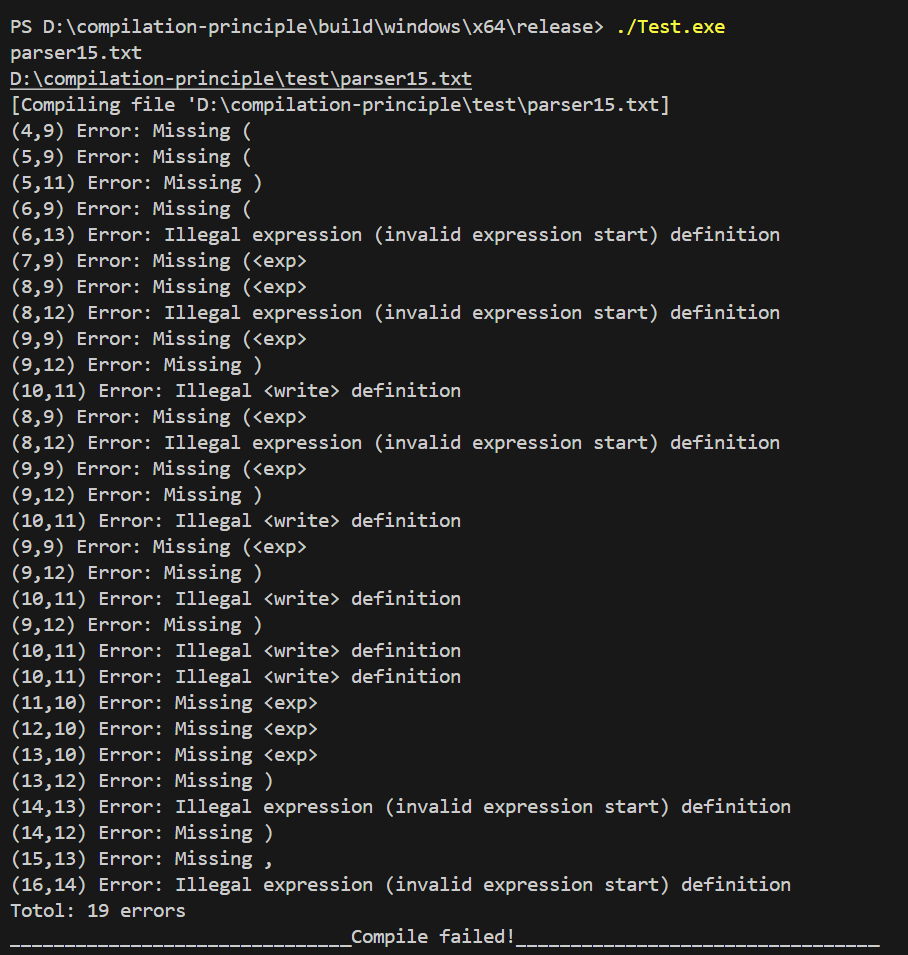
#### parser13.txt <while>： while后缺少lexp。



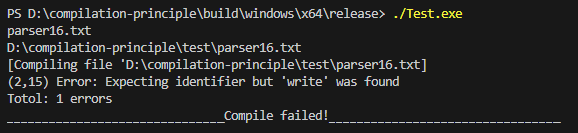
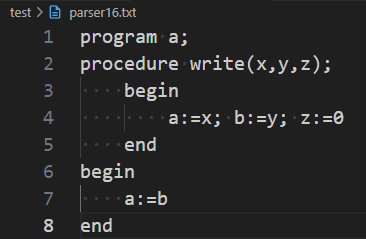
#### parser14.txt <read>： read使用不符合 PL/0 的语法规则。

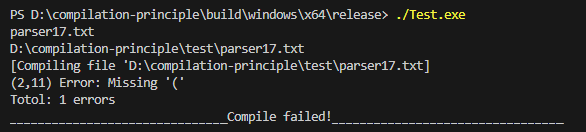
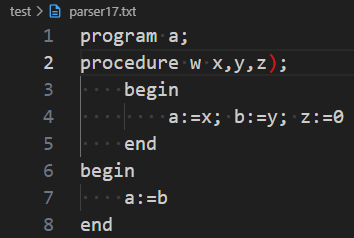
#### parser15.txt <write>： write使用不符合 PL/0 的语法规则。

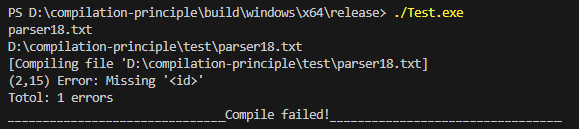
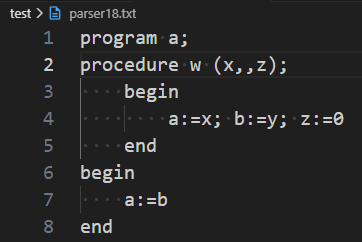
#### parser16.txt <procedure>： procedure后id缺失或不符合规范。



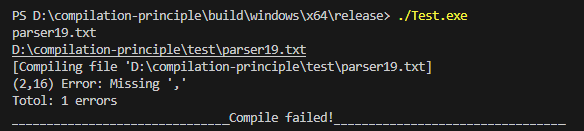
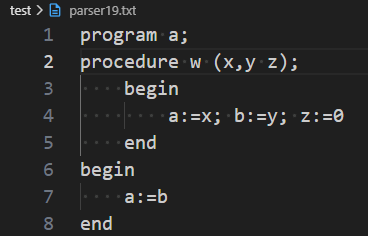
#### parser17.txt <procedure>： procedure id后括号不符合规范。



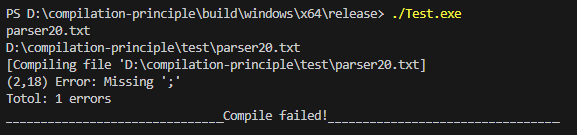
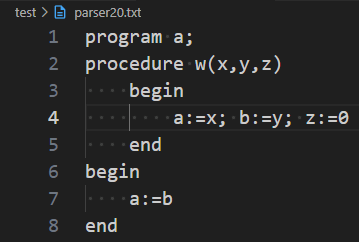
#### parser18.txt <procedure>： procedure id后括号内，多余。



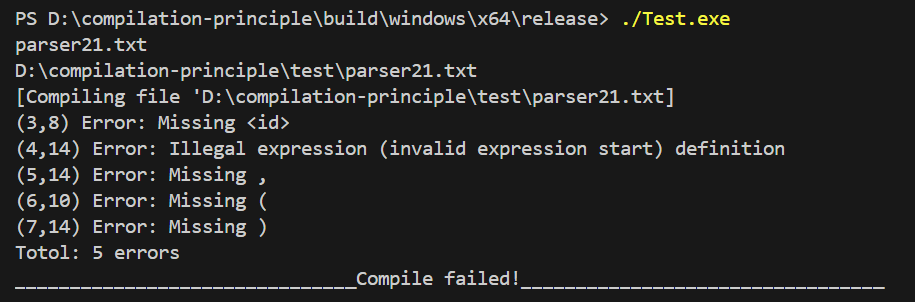
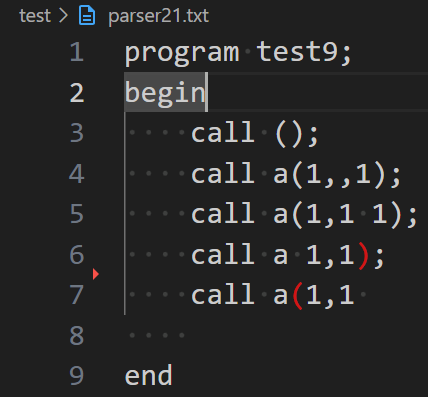
#### parser19.txt <procedure>： procedure id后括号内，缺少。



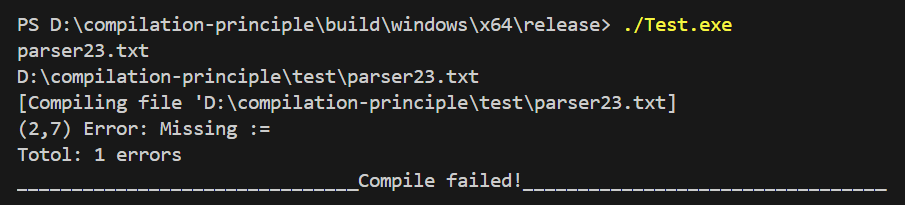
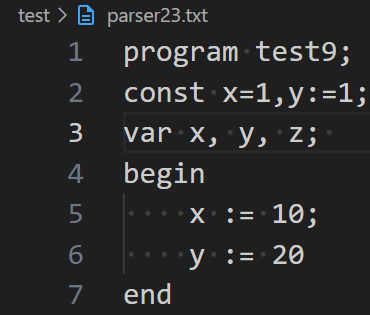
#### parser20.txt <procedure>： procedure声明后缺少;。



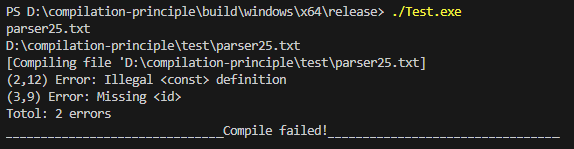
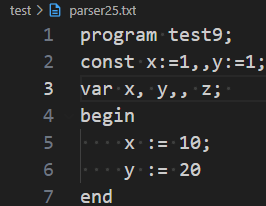
#### parser21.txt <call>： call使用不符合 PL/0 的语法规则。



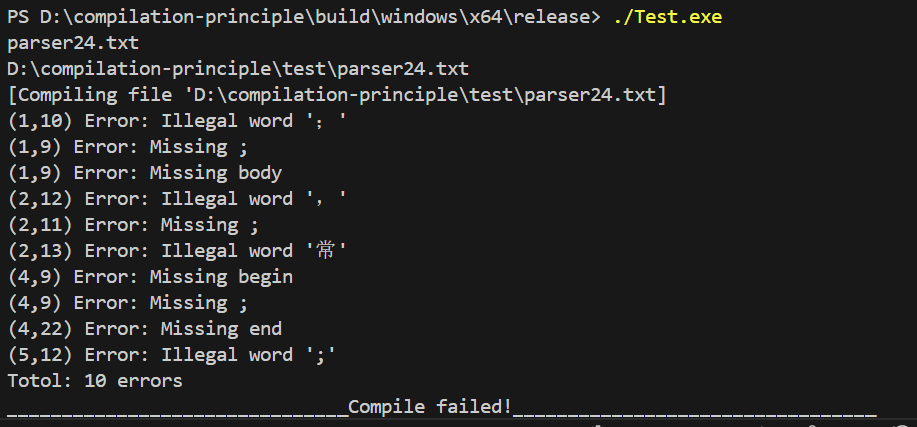
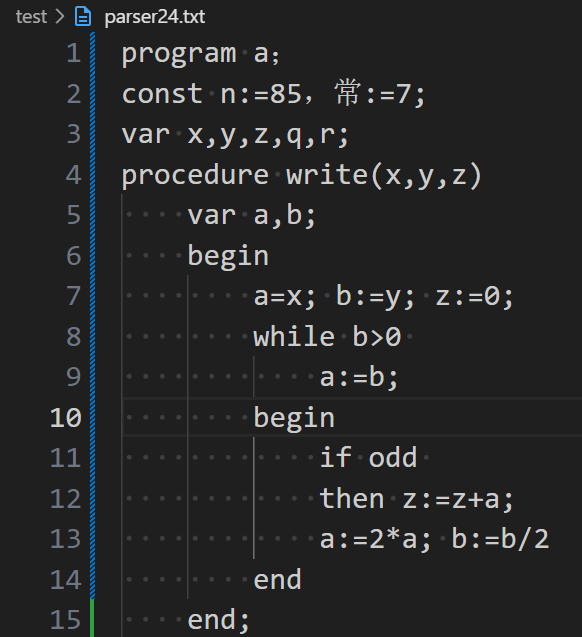
#### parser22.txt <const>： :=缺少错误。



#### parser23.txt <,>： const和var时,多余。



#### parser24.txt 综合测试。



## 语义分析

### 引言

语义分析是该 PL0 编译器项目的核心部分之一，主要任务是对源代码进行深层次的分析，并为代码生成中间代码所需的参数做好准备。语义分析的目的是确定每个语法单元（如变量、常量、语句等）在程序中的意义，以及如何在符号表中处理这些信息。

### 核心思想

确定中间代码的参数：op（操作码）参数的确定；L（层级）参数的确定；a（地址）参数的确定

#### op 参数的确定

op 参数的作用是确定每条中间代码指令的操作码，它与代码的具体语法单元（例如表达式、语句等）相关。

表达式相关：对于 <factor>、<term>、<exp> 这样的表达式部分，我们需要确定操作码，比如计算表达式时需要 opr 和 lod 指令。opr 用于执行各种运算操作，lod 用于加载变量的值。

语句相关：对于 call、read、write 等语句，需要生成相应的操作码，如 cal（调用过程）、red（读取输入）、wrt（写出输出）。

条件相关：对于 if、while 语句等条件控制语句，我们需要生成 opr（操作码）、jpc（跳转指令条件假）、jmp（无条件跳转指令）等操作码。

#### L（层级）参数的确定

L 参数主要用于跨活动记录访问内存单元，在生成中间代码时，层级（L）用来标识一个变量或过程在符号表中的位置。

LOD 和 STO 指令：L 参数用来表示当前活动记录的层级。在 lod（加载）和 sto（存储）指令中，L 参数用于确定变量所在的层级，用以访问该层级的内存单元。

参数传递：当 sto 指令中的 L 为 -1 时，表示它是传递形式参数的特殊指令，用来为过程或函数传递参数。

符号表查找：在符号表中，程序中的变量和过程有对应的层级信息，可以通过符号表查找到相关的 L 值。

#### a（地址）参数的确定

a 参数在语义分析中有四种不同的形式，分别是：

偏移量：变量或常量的相对地址，计算出来的偏移量通常在符号表中可以找到。

绝对地址：在生成中间代码时，某些常量、变量或符号的地址是固定的，常常是中间代码的下标。

常量的值：常量值直接用于中间代码中，例如在表达式计算时，常量的值会直接传递给操作码 LIT。

操作码的值：某些指令（例如 opr 指令）使用操作码来执行具体的操作，a 参数存储的就是这些操作码的值。

#### 符号表管理与语义分析

符号表（Symbol Table）在语义分析中扮演着非常重要的角色，它存储了程序中所有标识符的信息（如变量、常量、函数等）。在 PL0 编译器中，符号表的操作包括：

插入（enter）：将标识符添加到符号表中。

查找（lookUp）：根据标识符查找符号表，获取相应的信息。

更新：在符号表中更新标识符的相关信息（如位置、类型、值等）。

在编译过程中，符号表不仅用于管理变量和常量，还用于管理过程和函数的入口。每次遇到一个标识符时，语义分析都会检查该标识符是否已在符号表中声明。如果标识符未定义或有冲突，则会报错。

#### PL0 语言翻译模式和语义分析的结合

PL0 编译器根据文法规则逐步分析源代码并生成中间代码。在翻译模式中，属性信息用于生成中间代码，并通过符号表来处理变量、常量、过程等的详细信息。例如：

变量赋值：<statement> → <id> := <exp>

在语义分析中，当遇到赋值语句时，首先查找 <id> 是否已定义（通过符号表中的查找），然后检查其类型是否为常量，如果是常量，则不能赋值。接着，分析 exp 的值并生成相应的 STO（存储）指令，将计算结果存储到符号表中的变量地址。

条件语句：<statement> → if <lexp> then <statement1> [else <statement2>]

对于条件语句，语义分析需要检查条件表达式 <lexp> 是否有效，生成相应的跳转指令 JPC（条件跳转指令）。然后，生成 JMP（无条件跳转）指令，回到语句后面。通过 backpatch 回填跳转指令的目标地址，确保代码执行的正确性。

函数调用：<statement> → call <id>

对于过程调用，语义分析需要查找符号表，确保函数或过程已在符号表中定义。如果定义正确，则生成相应的调用指令 CAL。

#### 中间代码生成

通过语义分析得到的操作码 op、层级 L 和地址 a 参数，最终生成中间代码。这些中间代码用于表达程序的执行逻辑，并为后续的机器码生成或虚拟机解释执行打下基础。例如：

LIT：用于将常量加载到寄存器。LOD：用于加载变量的值。STO：用于将值存储到变量。JMP：无条件跳转。JPC：条件跳转。OPR：执行算术或逻辑运算。

语义分析是 PL0 编译器的关键部分，它通过符号表的管理和语法分析的结果，确定中间代码的参数（如操作码、层级和地址）。通过这种方式，编译器能够根据源代码的语义生成正确的中间代码，并为后续的代码优化、机器码生成等阶段打下基础。

### 翻译模式

翻译模式（Translation Schemes）是编译原理中用于将源语言的语法结构转换为中间代码的一种方法。在编译过程中，每个语法规则通过一个具体的翻译模式映射到相应的中间代码或其他中间表示形式。翻译模式不仅描述了语法规则如何转换，还包括了如何通过符号表等工具管理语义信息。

#### 全局属性

全局属性存储编译过程中需要的全局信息，包括：

glo\_offset：全局偏移量，单位是字节。用于计算全局变量或常量的内存地址。

nextquad：下一条中间代码的入口地址，用于管理中间代码生成的位置。

lexer.GetStrToken()：最近一次分析出的词法单元字符串。

symTable.level：当前分析的子过程的层级。

symTable.sp：当前正在分析的子过程的入口地址。

这些全局变量帮助编译器管理内存、控制代码生成过程以及处理语法分析过程中产生的信息。

#### 程序翻译模式

<prog> → program <id> {

symTable.MkTable()

symTable.enterProgm(lexer.GetStrToken())

block.entry = emit(JMP, 0, \) // 预留空地址，为代码块生成入口地址

} <block>

这个规则描述了程序的基本结构：

program <id> 表示程序声明的开始，<id> 是程序名。

symTable.MkTable()：创建符号表。

symTable.enterProgm(lexer.GetStrToken())：将程序名插入符号表，表示程序的开始。

emit(JMP, 0, \)：生成一条跳转指令，留待后续回填（在处理代码块时确定入口地址）。

这部分翻译模式表明，程序的翻译需要处理符号表，并生成跳转指令，以便正确解析代码块。

#### 块翻译模式

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

size\_t cur\_proc = symTable.sp;

ProcInfo \*cur\_info = (ProcInfo \*)symTable.table[cur\_proc].info;

symTable.AddWidth(cur\_proc, glo\_offset);// 计算过程占用内存

} [<proc>]{

body.entry = pcodelist.emit(alloc, 0, cur\_info->offset / UNIT\_SIZE + ACT\_PRE\_REC\_SIZE + symTable.level + 1);

   size\_t target = cur\_info->entry;// 将过程入口地址回填至过程的跳转语句

   pcodelist.backpatch(target, entry);// 过程体开始，过程已定义

   cur\_info->isDefined = true;

}<body>

一个 block（代码块）可以包含常量声明、变量声明和过程声明。翻译步骤如下：

symTable.AddWidth(cur\_proc, glo\_offset)：计算当前过程的内存占用。glo\_offset 表示内存的全局偏移量，sp 是当前过程的入口地址。

block.cur\_proc = symTable.sp：将当前过程的入口地址保存到 block.cur\_proc 中。

该翻译模式表明，块的内容包括常量、变量、过程的声明，需要计算变量和过程占用的内存空间，并确定代码块的入口地址。

#### 常量声明

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

<const> → <id> {

id.sym\_entry = symTable.InsertToTable(lexer.GetStrToken(), 0, CST); // 进入符号表

} := <integer> {

symTable.table[symTable.table.size() - 1].info->SetValue(lexer.GetStrToken());// 赋值给常量

}

常量声明由 const 开头，后面跟着常量的定义。翻译模式如下：

symTable.InsertToTable(lexer.GetStrToken(), 0, CST)：将常量名插入符号表，CST 表示常量类型。

symTable.table[symTable.table.size() - 1].info->SetValue(lexer.GetStrToken())：将常量的值赋给符号表中的常量条目。

常量在符号表中不会占用内存（0），而其值在编译时就已确定。setValue 使常量值直接存储到符号表中。

#### 变量声明

<vardecl> → var <id> {

symTable.InsertToTable(lexer.GetStrToken(), glo\_offset, Category::VAR);

   glo\_offset += 4;

} { ,<id> {

symTable.InsertToTable(lexer.GetStrToken(), glo\_offset, Category::VAR);

  glo\_offset += 4;

}} ;

变量声明通过 var 关键字开始。每个变量插入符号表，偏移量逐个增加。翻译过程如下：

symTable.InsertToTable(lexer.GetStrToken(), glo\_offset, Category::VAR)：将变量名插入符号表，类型为变量（VAR），glo\_offset 是内存的偏移量，通常按 4 字节对齐（假设每个变量占 4 字节）。

glo\_offset += 4：更新全局偏移量。

每个变量的内存位置通过更新 glo\_offset 来管理，保证变量在内存中的位置是唯一且正确的。

#### 过程声明

<proc> → procedure <id1> {

symTable.MkTable();

   id1.sym\_entry = symTable.InsertToTable(lexer.GetStrToken(), 0, Category::PROCE);// 在符号表中插入过程

block.entry = emit(JMP, 0, \) // 预留过程体入口地址

} （{

symTable.display.push\_back(0);

   symTable.level++;

}[<id>{

id2.sym\_entry = symTable.InsertToTable(lexer.GetStrToken(), glo\_offset, Category::FORM);

   glo\_offset += 4;

id1.sym\_entry->formVarList.push\_back(form\_var);

}{,<id>{

id.sym\_entry-> symTable.InsertToTable(lexer.GetStrToken(), glo\_offset, Category::FORM);

   glo\_offset += 4;

id1.sym\_entry->formVarList.push\_back(form\_var);

}}])<block>{

// 本层数据结束，记得回到上一层

   pcodelist.emit(opr, 0, OPR\_RETURN);

   symTable.display.pop\_back();

   symTable.level--;

} {;<proc>}

过程声明中，过程的定义包含符号表的插入操作，表示过程的开始和结束位置，同时通过跳转指令生成过程体入口的预留地址。

#### 因子翻译

<factor> → <id> {

int pos = symTable.SearchInfo(lexer.GetStrToken(), Category::VAR);

cur\_info = (VarInfo \*)symTable.table[pos].info;

if (cur\_info->cat == Category::CST)

   {

       int val = cur\_info->GetValue();

       pcodelist.emit(lit, cur\_info->level, val);

   }

   else

  pcodelist.emit(load, cur\_info->level, cur\_info->offset / UNIT\_SIZE + ACT\_PRE\_REC\_SIZE + cur\_info->level + 1);

} | <integer> {

pcodelist.emit(lit, 0, w\_str2int(lexer.GetStrToken())); // 常量直接加载

}|(<exp>)

因子的翻译模式负责处理变量和常量的加载。对常量使用 LIT 指令，加载常量值；对变量使用 LOD 指令，加载变量值。

#### 赋值语句

<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)

}

赋值语句将表达式 <exp> 的值存储到 <id> 表示的变量中：首先检查 <id> 是否为常量，如果是常量则报错。使用 pcodelist.emit(store, cur\_info->level, cur\_info->offset / UNIT\_SIZE + ACT\_PRE\_REC\_SIZE + cur\_info->level + 1) 将表达式的值存储到符号表中记录的变量位置。

#### 条件语句

<statement> → if <lexp> then{

entry\_jpc = pcodelist.emit(jpc, 0, 0);

} <statement1> [else {

entry\_jmp = pcodelist.emit(jmp, 0, 0);

//将else入口地址回填至jpc

   pcodelist.backpatch(entry\_jpc, pcodelist.code\_list.size());

}<statement2>{

//有else，则将if外入口地址回填至jmp

   pcodelist.backpatch(entry\_jmp, pcodelist.code\_list.size());

}] {

lexp.false\_entry = emit(JPC, 0, \) // 生成条件跳转指令

backpatch(lexp.false\_entry, nextquad) // 回填跳转地址

}

条件语句会根据表达式 <lexp> 的值决定是否执行 then 部分的语句：生成条件跳转指令 JPC，如果条件为假则跳转。使用 backpatch 回填跳转地址，确保条件判断后的代码顺利执行。

#### 循环语句

<statement> → while <M><lexp>{

lexp.false\_entry = emit(JPC, 0, \)

} do <statement>{

emit(JMP, 0, M.quad)

backpatch(lexp.false\_entry, nextquad)

}

循环语句生成 JPC 跳转指令，如果条件为假则跳出循环，否则通过 JMP 跳回循环条件处重新判断。

#### 函数调用

<statement> → 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)

}}]）

函数调用通过查找符号表确认过程是否存在，若存在则生成 CAL 指令，表示调用过程。

#### read

<statement>->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 } })

#### M和N的翻译

<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和exp

<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**

<term> → <factor>{<mop>{

if(mop==’\*’) 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); }}

## 中间代码生成

## 解释器

# 课设总结