编译技术课程实验报告

实验名称：	大作业 PL/0 编译器的设计与实现
实验日期：	2024-5-26
实验地点：	西部片区 4 号楼 208

学号：	33920212204567
姓名：	任宇
专业年级：	软工 2021 级
学年学期：	2023-2024 学年第二学期

一、 实验目的

掌握计算机语言的词法分析和语法分析程序设计以及属性文法应用的实现方法。

二、 实验内容

设计并实现一个 PL/0 语言的编译器，能够将 PL/0 语言翻译成 P-code 语言。

三、 实验环境

- PC 微机 Windows10 操作系统
- 开发环境为 VS2022

四、 实验过程

1) PL/0 语言编译系统构成

PL/0 语言编译系统由编译程序和解释程序两部分组成，分别成为 PL/0 编译程序和类 P-code 解释程序。本次大作业完成的部分为 PL/0 编译程序。使用 T 形图表示一个编译程序涉及的三个方面的语言，即源语言、目标语言和编译程序的实现语言。T 形图的左上角表示源语言，右上角表示目标语言，底部表示书写语言，PL/0 编译程序的 T 形图如图 1 所示。

图 1 PL/0 编译程序 T 形图

PL/0 编译程序将 PL/0 源程序翻译成类 P-code 目标程序，源语言为 PL/0，目标语言为类 P-code。PL/0 编译程序使用 C++语言书写。

2) PL/0 语言语法的 EBNF 描述

表 1.1 PL/0 语言语法的 EBNF 描述

PL/0 语法单位	ENBF 描述
<程序>	::= <分程序>.
<分程序>	::= [<常量说明部分>][变量说明部分>]{<过程说明部分>}<语句>
<常量说明部分>	::= const<常量定义>{,<常量定义>};
<常量定义>	::= <id>=<integer>
<变量说明部分>	::= var<id>{,<id>};
<过程说明部分>	::= <过程首部><分程序>{;<过程说明部分>};
<过程首部>	::= procedure<id>;
<语句>	::= <赋值语句>|<条件语句>|<当型循环语句>|<过程调用语句>|<读语句>|<写语句>|<复合语句>|<重复语句>|<空语句>
<赋值语句>	::= <id>:=<表达式>
<复合语句>	::= begin<语句>{;<语句>}end
<空语句>	::= ε
<条件>	::= <表达式><关系运算符><表达式>|odd<表达式>

<表达式>	::= [+|-]<项>{<加法运算符><项>}
<项>	::= <因子>{<乘法运算符><因子>}
<因子>	::= <id>|<integer>|'('<表达式>')'
<加减运算符>	::= +|-
<乘除运算符>	::= *|/
<关系运算符>	::==|<>|<|<=|>|>=
<条件语句>	::=if<条件> then <语句>
<过程调用语句>	::= call<id>
<当型循环语句>	::= while<条件> do <语句>
<读语句>	::= read'('<id>{,<id>}')'
<写语句>	::= write'('<表达式>{,<表达式>}')'

无符号整数<integer>是由一个或多个数字组成的序列。数字为0,1,2,……,9。

标识符<id>是字母开头的字母数字序列。字母包括大小写字母:a,b,……,z,A,B,……,Z。

3) 类P-code语言说明

类P-code语言可以看作类P-code虚拟机的汇编语言。类P-code虚拟机是一种简单的纯栈式结构的机器，它有一个栈式存储器，有4个控制寄存器。类P-code程序运行期间的数据存储和算术及逻辑运算都在栈顶进行。类P-code虚拟机的指令格式形如：

F L A

它由3个部分构成，其含义如下：

F：指令的操作码：

L：若起作用，则表示引用层与声明层之间的层次差；若不起作用，则置为0。

A: 不同的指令含义不同。

表1.2 类P-code虚拟机指令系统

指令分类	指令格式	指令功能
过程调用相关指令	INT 0 A	在栈顶开辟A个存储单元
	OPR 0 0	结束被调用过程，返回调用点并退栈
	CAL L A	调用地址为A的过程，调用过程与被调用过程层差为L
存取指令	INT 0 A	立即数A存入t所指单元，t+1
	LOD L A	将层差L、偏移量为A的存储单元的值取到栈顶，t+1
	STO L A	将栈顶的值存入层差为L、偏移量为A的单元，t-1
一元运算和比较指令	OPR 0 1	求栈顶元素的相反数，结果值留在栈顶
	OPR 0 6	栈顶内容为奇数则变为1，若为偶数则变为0
二元运算指令	OPR 0 2	次栈顶与栈顶的值相加，结果存入次栈顶，t-1
	OPR 0 3	次栈顶的值减去栈顶的值，结果存入次栈顶，t-1
	OPR 0 4	次栈顶的值乘以栈顶的值，结果存入次栈顶，t-1
	OPR 0 5	次栈顶的值除以栈顶的值，结果存入次栈顶，t-1
二元比较指令	OPR 0 8	次栈顶与栈顶内容若相等，则将0存于次栈顶，t-1
	OPR 0 9	次栈顶与栈顶内容若不相等，则将0存于次栈顶，t-1
	OPR 0 10	次栈顶内容若小于栈顶，则将0存于次栈顶，t-1
	OPR 0 11	次栈顶内容若大于等于栈顶，则将0存于次栈顶，t-1
	OPR 0 12	次栈顶内容若大于栈顶，则将0存于次栈顶，t-1
	OPR 0 13	次栈顶内容若小于等于栈顶，则将0存于次栈顶，t-1

转移指令	JMP 0 A	无条件转移至地址 A
	JPC 0 A	若栈顶为 0，则转移至地址 A，t-1
输入输出指令	OPR 0 15	栈顶的值输出至控制台屏幕，t-1
	OPR 0 15	控制台屏幕输出一个换行
	OPR 0 16	从控制台读入一行输入，置入栈顶，t+1

4) PL/0 语言编译程序的语法图描述

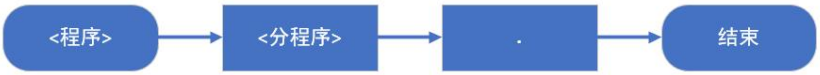

图 2 程序语法描述图

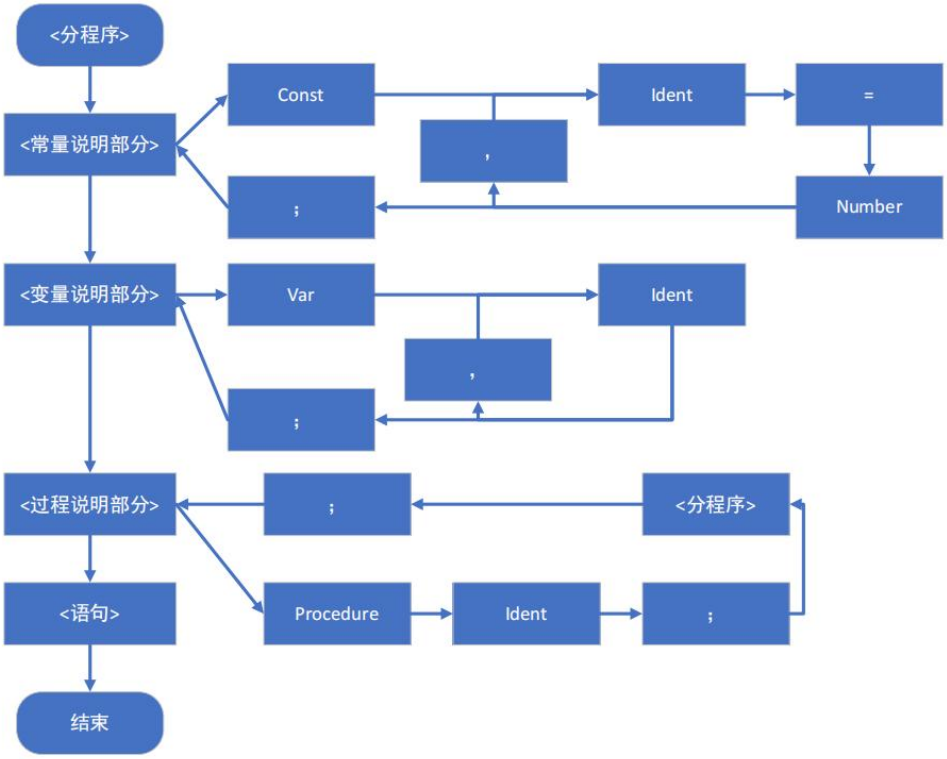

图 3 分程序语法描述图

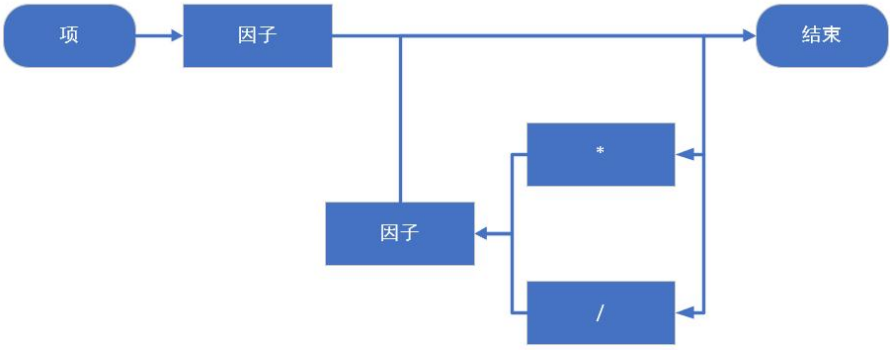

图 4 项语法描述图

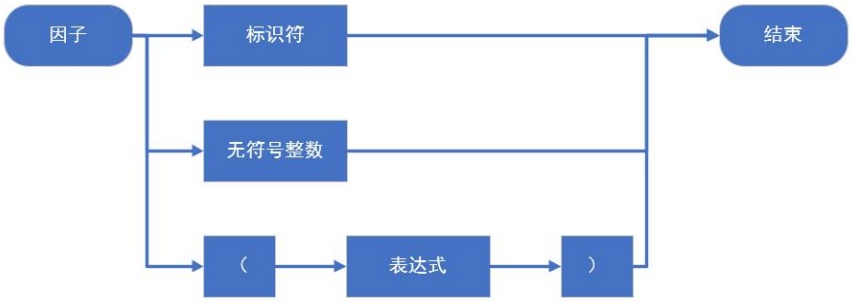

图 5 因子语法描述图

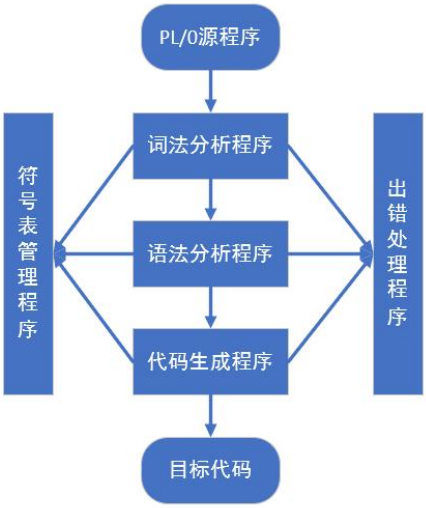

图 6 PL/0 编译程序和解释执行过程

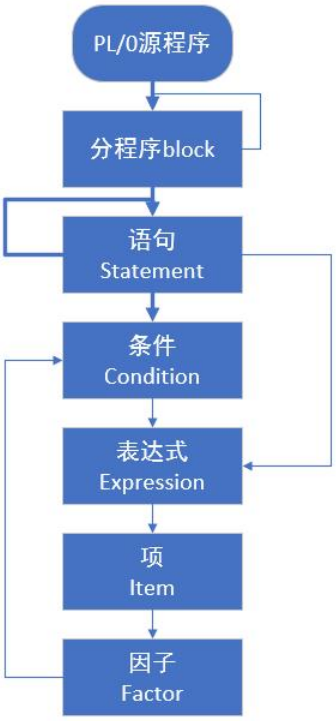

图 7 语法调用关系图

5) 词法分析

词法分析是编译器前端的重要组成部分，其目的是将源代码转换成一系列的标记（Token），这些标记将被语法分析器使用以构建抽象语法树（AST）。在本 PL/0 编译器项目中，词法分析的设计涉及以下几个文件：

a) PL0Token.h

PL0Token 类用于表示词法分析生成的标记。每个标记包含以下三个属性：

- type：标记的类别，使用 PL0SymType 枚举类型。
- line：标记所在的行号，用于错误处理和调试。
- value：标记的值，主要用于标识符和常量。

```
16  class PL0Token {
17  private:
18      PL0SymType st; // token的类别
19      int line; // token所在行，错误处理使用
20      std::string value; // token的值，只有标识符和常量有值
21
22  public:
23      // 构造函数
24      PL0Token(PL0SymType _st, int _line, const std::string& _value)
25          : st(_st), line(_line), value(_value) {}
26
27      // getter和setter
```

PL0SymType 枚举类型定义了所有可能的标记类别。这些类别包括关键字（如 BEGIN、END）、操作符（如 ADD、SUB）、标识符、常量等。

```
6   // 枚举类型SymType
7   enum class PL0SymType {
8       BEG, END, IF, THEN, ELS, CON, PROC, VAR, DO, WHI, CAL, REA, WRI, ODD, REP, UNT,
9       EQU, LES, LESE, LARE, LAR, NEQE, ADD, SUB, MUL, DIV,
10      SYM, CONST,
11      CEQU, COMMA, SEMIC, POI, LBR, RBR,
12      COL,
13      END_OF_FILE // 修改后的EOF
14  };
```

b) PL0LexicalAnalyzer.h

PL0LexicalAnalyzer 类用于执行词法分析的主要逻辑。它读取源代码文件，将其转换为字符流，并逐字符解析以生成标记。其主要功能包括：

- 初始化和读取源代码文件。
- 逐字符扫描并识别标记。
- 处理关键字、标识符、常量和各种操作符。
- 管理当前解析位置和行号。

词法分析过程如下：

① 初始化：PL0LexicalAnalyzer 类的构造函数读取源代码文件，将其内容存储在 buffer 中，并初始化相关变量。

② 字符扫描：doAnalysis 方法逐字符扫描 buffer，调用 analysis 方法解析每个标记，并将生成的标记存储在 allToken 向量中。

③ 标记解析：analysis 方法通过判断当前字符类型（字母、数字、操作符等）来生成相应的标记，并根据需要调用 getChar 获取下一个字符或 retract 回退字符。

④ 关键字和标识符处理：通过检查字符串 strToken 是否与关键字列表 keyWords 中的任何一个匹配来识别关键字。如果不是关键字，则将其视为标识符。

⑤ 常量处理：如果当前字符是数字，则解析为常量标记。

⑥ 操作符处理：识别和处理各种操作符，包括单字符操作符（如 +、-）和双字符操作符（如 <=、<>）。

void doAnalysis()：扫描整个源代码文件并生成所有标记。循环调用 analysis 方法，直到处理完整个文件。

```
48  private:
49      void doAnalysis() {
50          while (searchPtr < buffer.size()) {
51              allToken.push_back(analysis());
52          }
53      }
```

std::shared_ptr<PL0Token> analysis()：解析单个标记，返回一个 PL0Token 对象。处理空白字符、关键字、标识符、常量和各种操作符。

```
55  std::shared_ptr<PL0Token> analysis() {
56      strToken.clear();
57      getChar();
58      while ((ch == ' ' || ch == '\n' || ch == '\t' || ch == '\0') && searchPtr < buffer.size()) {
59          if (ch == '\n') {
60              line++;
61          }
62          getChar();
63      }
64      if (ch == '$' && searchPtr >= buffer.size()) { // 到达文件末尾
65          return std::make_shared<PL0Token>(PL0SymType::END_OF_FILE, line, "-1");
66      }
67      if (isLetter()) { // 首位为字母，可能为保留字或者变量名
68          while (isLetter() || isDigit()) {
69              strToken += ch;
70              getChar();
71          }
72          retract();
73          for (size_t i = 0; i < keyWords.size(); i++) {
74              if (strToken == keyWords[i]) { // 说明是保留字
75                  return std::make_shared<PL0Token>(static_cast<PL0SymType>(i), line, "-");
76              }
77          }
78          // 不是保留字，则为标识符，需要保存值
79          return std::make_shared<PL0Token>(PL0SymType::SYM, line, strToken);
80      }
81      else if (isDigit()) { // 首位为数字，即为整数
82          while (isDigit()) {
83              strToken += ch;
84              getChar();
85          }
86          retract();
87          return std::make_shared<PL0Token>(PL0SymType::CONST, line, strToken);
88      }
89      else if (ch == '=') { // 等号
90          return std::make_shared<PL0Token>(PL0SymType::EQU, line, "-");
91      }
92      else if (ch == '+') { // 加号
93          return std::make_shared<PL0Token>(PL0SymType::ADD, line, "-");
94      }
95      else if (ch == '-') { // 减号
96          return std::make_shared<PL0Token>(PL0SymType::SUB, line, "-");
97      }
98      else if (ch == '*') { // 乘号
99          return std::make_shared<PL0Token>(PL0SymType::MUL, line, "-");
100     }
101     else if (ch == '/') { // 除号
102         return std::make_shared<PL0Token>(PL0SymType::DIV, line, "-");
103     }
104     else if (ch == '<') { // 小于或不等于或小于等于
105         getChar();
```

```
104         else if (ch == '<') { // 小于或不等于或小于等于
105             getChar();
106             if (ch == '=') {
107                 return std::make_shared<PL0Token>(PL0SymType::LESE, line, "-");
108             }
109             else if (ch == '>') {
110                 return std::make_shared<PL0Token>(PL0SymType::NEQE, line, "-");
111             }
112             else {
113                 retract();
114                 return std::make_shared<PL0Token>(PL0SymType::LES, line, "-");
115             }
116         }
117         else if (ch == '>') { // 大于或大于等于
118             getChar();
119             if (ch == '=') {
120                 return std::make_shared<PL0Token>(PL0SymType::LARE, line, "-");
121             }
122             else {
123                 retract();
124                 return std::make_shared<PL0Token>(PL0SymType::LAR, line, "-");
125             }
126         }
127         else if (ch == ',') { // 逗号
128             return std::make_shared<PL0Token>(PL0SymType::COMMA, line, "-");
129         }
130         else if (ch == ';') { // 分号
131             return std::make_shared<PL0Token>(PL0SymType::SEMIC, line, "-");
132         }
133         else if (ch == '.') { // 点
134             return std::make_shared<PL0Token>(PL0SymType::POI, line, "-");
135         }
136         else if (ch == '(') { // 左括号
137             return std::make_shared<PL0Token>(PL0SymType::LBR, line, "-");
138         }
139         else if (ch == ')') { // 右括号
140             return std::make_shared<PL0Token>(PL0SymType::RBR, line, "-");
141         }
142         else if (ch == ':') { // 赋值号
143             getChar();
144             if (ch == '=') {
145                 return std::make_shared<PL0Token>(PL0SymType::CEQU, line, "-");
146             }
147             else {
148                 retract();
149                 return std::make_shared<PL0Token>(PL0SymType::COL, line, "-");
150             }
151         }
152         return std::make_shared<PL0Token>(PL0SymType::END_OF_FILE, line, "-");
153     }
154
155     void init() { ... }
158
```

6) 语法分析

a) PL0Pcode.h

PL0Pcode.h 文件定义了 PL0Pcode 类和 Operator 枚举类型，用于表示和管理 P-code 指令。该文件是生成和执行 P-code 指令的基础。

Operator 枚举类型：定义了所有可能的 P-code 操作码，用于表示各种操作指令。

- INT：为被调用的过程（包括主过程）在运行栈中开辟数据区。
- CAL：调用过程。
- LIT：将常量送到运行栈的栈顶。
- LOD：将变量送到运行栈的栈顶。
- STO：将运行栈的栈顶内容送入某个变量单元。
- JMP：无条件转移。
- JPC：条件转移。
- OPR：关系或算术运算。

```
enum class Operator {
    INT, CAL, LIT, LOD, STO, JMP, JPC, OPR
};
```

PL0Pcode 类：用于表示一条 P-code 指令，包含操作码和操作数。每条 P-code 指令由三个部分组成：

- F：操作码，使用 Operator 枚举类型。
- L：层次差，用于指示需要从哪个栈帧基地址访问数据。
- A：操作数，用于指示具体的值或地址。

```
class PL0Pcode {
private:
    Operator F;
    int L;
    int A;

public:
    PL0Pcode(Operator _F, int _L, int _A)
        : F(_F), L(_L), A(_A) {}

    // getter 和 setter
```

b) PL0Symbol.h

PL0Symbol 类用于表示符号表中的条目。符号表是编译器中用来存储和管理变量、常量和过程等符号信息的重要数据结构。在 PL/0 编译器中，符号表管理符号的声明和作用域信息，并在语法分析和代码生成过程中提供符号查询功能。

成员变量：

int type：表示符号的类型。可以是常量、变量或过程。

int value：表示常量的值或变量的初始值。对于过程，这个值通常未使用。

int level：表示符号的嵌套层次。用于支持嵌套的作用域。

int address：表示符号在其作用域中的地址，通常是相对于所在嵌套过程基地址的偏移量。

int size：表示符号所占的大小。对于常量和变量，通常为 1；对于过程，这个值通常未使用。

std::string name：符号的名称。用于标识变量、常量或过程的名称。

```
class PL0Symbol {
private:
    int type;          // 表示常量、变量或过程
    int value;         // 表示常量或变量的值
    int level;         // 嵌套层次
    int address;       // 相对于所在嵌套过程基地址的地址
    int size;          // 表示常量，变量，过程所占的大小
    std::string name;  // 变量、常量或过程名

public:
    // 构造函数
    PL0Symbol(int _type, int _value, int _level, int _address, int _size, const std::string& _name)
        : type(_type), value(_value), level(_level), address(_address), size(_size), name(_name) {}

    PL0Symbol(int _type, int _level, int _address, int _size, const std::string& _name)
        : type(_type), value(0), level(_level), address(_address), size(_size), name(_name) {}

    // setter 和 getter
```

c) PL0PcodeManager.h

PL0PcodeManager 类用于管理所有生成的 P-code 指令。该文件提供了生成和管理 P-code 指令的功能，是语法分析器和代码生成器的重要组成部分。

成员变量：

std::vector<std::shared_ptr<PL0Pcode>> allPcode：存储所有生成的 P-code 指令的向量。

```
8  class PL0PcodeManager {
9  private:
10     std::vector<std::shared_ptr<PL0Pcode>> allPcode;
11
```

成员方法：

int getPcodePtr() const：返回当前 P-code 指令的数量，即 P-code 指针的位置。

```
19     int getPcodePtr() const {
20         return allPcode.size();
21     }
```

void gen(const std::shared_ptr<PL0Pcode>& pcode)：接受一个 PL0Pcode 对象的共享指针，并将其添加到 allPcode 向量中。

```
23     void gen(const std::shared_ptr<PL0Pcode>& pcode) {
24         allPcode.push_back(pcode);
25     }
```

void gen(Operator L, int F, int A)：根据操作码、层次差和操作数生成一个新的 PL0Pcode 对象，并将其添加到 allPcode 向量中。

```
27     void gen(Operator L, int F, int A) {
28         allPcode.push_back(std::make_shared<PL0Pcode>(L, F, A));
29     }
30 };
```

d) PL0SymbolTable.h

PL0SymbolTable 类用于管理 PL/0 编译器中的符号表。符号表是编译器的重要组成部分，用于存储和管理源代码中声明的常量、变量和过程等符号信息。该类提供插入、查找和管理符号的方法，支持多层嵌套作用域。

成员变量：

- std::vector<std::shared_ptr<PL0Symbol>> allSymbol：存储符号表中所有符号的向量。
- const int con：表示常量类型的常量值。
- const int var：表示变量类型的常量值。
- const int proc：表示过程类型的常量值。
- int ptr：指示当前符号表的指针位置，表示符号的数量。

```
9  class PL0SymbolTable {
10 private:
11     std::vector<std::shared_ptr<PL0Symbol>> allSymbol;
12
13     const int con = 1; // 常量类型用1表示
14     const int var = 2; // 变量类型用2表示
15     const int proc = 3; // 过程类型用3表示
16
17     int ptr = 0;
```

主要方法：

插入符号：

- void enterConst(const std::string& name, int level, int value, int address)：向符号表中插入一个常量符号。

```
22  // 向符号表中插入常量
23  void enterConst(const std::string& name, int level, int value, int address) {
24      allSymbol.push_back(std::make_shared<PL0Symbol>(con, value, level, address, 0, name));
25      ptr++;
26  }
```

- void enterVar(const std::string& name, int level, int address)：向符号表中插入一个变量符号。

```
28  // 向符号表中插入变量
29  void enterVar(const std::string& name, int level, int address) {
30      allSymbol.push_back(std::make_shared<PL0Symbol>(var, level, address, 0, name));
31      ptr++;
32  }
```

- void enterProc(const std::string& name, int level, int address)：向符号表中插入一个过程符号。

```
34  // 向符号表中插入过程
35  void enterProc(const std::string& name, int level, int address) {
36      allSymbol.push_back(std::make_shared<PL0Symbol>(proc, level, address, 0, name));
37      ptr++;
38  }
```

查找符号：

- bool isNowExists(const std::string& name, int level) const：在符号表当前层查找符号是否存在。

```
40  // 在符号表当前层查找变量是否存在
41  bool isNowExists(const std::string& name, int level) const {
42      for (const auto& symbol : allSymbol) {
43          if (symbol->getName() == name && symbol->getLevel() == level) {
44              return true;
45          }
46      }
47      return false;
48  }
```

- bool isPreExists(const std::string& name, int level) const：在符号表之前层查找符号是否存在。

```
50  // 在符号表之前层查找符号是否存在
51  bool isPreExists(const std::string& name, int level) const {
52      for (const auto& symbol : allSymbol) {
53          if (symbol->getName() == name && symbol->getLevel() <= level) {
54              return true;
55          }
56      }
57      return false;
58  }
```

- std::shared_ptr<PL0Symbol> getSymbol(const std::string& name) const：按名称查找符号，并返回符号的共享指针。

```
60      // 按名称查找变量
61      std::shared_ptr<PLOSymbol> getSymbol(const std::string& name) const {
62          for (auto it = allSymbol.rbegin(); it != allSymbol.rend(); ++it) {
63              if ((*it)->getName() == name) {
64                  return *it;
65              }
66          }
67          return nullptr;
68      }
```

- int getLevelProc(int level) const：查找当前层所在的过程在符号表中的位置。

```
70      // 查找当前层所在的过程
71      int getLevelProc(int level) const {
72          for (int i = allSymbol.size() - 1; i >= 0; --i) {
73              if (allSymbol[i]->getType() == proc) {
74                  return i;
75              }
76          }
77          return -1;
78      }
```

获取符号表信息：

- std::vector<std::shared_ptr<PLOSymbol>> getAllSymbol() const：获取符号表中所有符号的向量。

```
80      std::vector<std::shared_ptr<PLOSymbol>> getAllSymbol() const {
81          return allSymbol;
82      }
```

e) PLOSyntaxAnalyzer.h

PLOSyntaxAnalyzer类的主要功能是对PL/0语言的源代码进行语法分析，生成相应的符号表和P-code（伪代码）指令，并进行错误处理和解释执行。

PLOSyntaxAnalyzer类的方法调用关系从编译入口compile()方法开始，首先调用program()方法处理整个程序。program()方法调用block()方法处理分程序，block()方法进一步调用conDeclare(),varDeclare()和proc()方法分别处理常量声明、变量声明和过程声明。在处理每个部分时，conDeclare(),varDeclare()和proc()方法会分别调用conHandle()方法和其他辅助方法来解析具体的声明和定义。

在解析复合语句时，block()方法调用body()方法，body()方法调用statement()方法来处理各类语句。在处理表达式和条件时，statement()方法进一步调用condition(),expression(),term()和factor()方法。整个过程中，通过调用errorHandle()方法来处理错误信息，并通过pcodeManager和symbolTable来生成相应的P-code和管理符号表。解释器方法interpreter()和interpreter(const std::vector<int>& input)用于执行生成的P-code，并输出结果。

成员变量

- lex：指向PLOLexAnalysis对象的智能指针，用于执行词法分析。

- allToken：保存词法分析结果的所有 Token 对象的向量。
- pcodeManager：指向 PL0PcodeManager 对象的智能指针，用于管理生成的 P-code。
- symbolTable：指向 PL0SymbolTable 对象的智能指针，用于管理符号表。
- errorMessage：保存错误信息的向量。
- errorHappen：布尔值，记录编译过程中是否发生错误。
- tokenPtr：当前 Token 的指针。
- level：嵌套层次。
- address：相对于所在嵌套过程基地址的地址。
- addIncrement：常量，用于地址增量。

```
15  class PL0SyntaxAnalyzer {
16  private:
17      std::shared_ptr<PL0LexAnalysis> lex;
18      std::vector<std::shared_ptr<PL0Token>> allToken; // 保存词法分析结果
19      std::shared_ptr<PL0PcodeManager> pcodeManager; // 保存生成的Pcode
20      std::shared_ptr<PL0SymbolTable> symbolTable; // 符号表管理
21      std::vector<std::string> errorMessage; // 保存错误信息
22
23      bool errorHappen = false; // 记录编译过程中是否发生错误
24      int tokenPtr = 0; // 指向当前token的指针
25
26      int level = 0;
27      int address = 0;
28      const int addIncrement = 1;
29
```

方法

- PL0SyntaxAnalyzer(const std::string& filename)：构造函数，初始化词法分析、P-code 管理和符号表管理。

```
31      PL0SyntaxAnalyzer(const std::string& filename) {
32          lex = std::make_shared<PL0LexAnalysis>(filename);
33          allToken = lex->getAllToken();
34
35          pcodeManager = std::make_shared<PL0PcodeManager>();
36
37          symbolTable = std::make_shared<PL0SymbolTable>();
38
39          errorMessage = std::vector<std::string>();
40      }
```

- bool compile()：编译入口函数，执行语法分析。

```
42      bool compile() {
43          program();
44          return (!errorHappen);
45      }
```

- void program()：处理 PL/0 程序的入口函数。

```
48      void program() {
49          block();
50          if (allToken[tokenPtr]->getSt() == PL0SymType::POI) {
51              tokenPtr++;
52              if (allToken[tokenPtr]->getSt() != PL0SymType::END_OF_FILE) {
53                  errorHandle(18, "");
54              }
55          }
56          else {
57              errorHandle(17, "");
58          }
59      }
```

- void block()：处理分程序，包括常量说明、变量说明和过程说明。

```
61  void block() {
62      int address_cp = address;
63      int start = symbolTable->getPtr();
64      int pos = 0;
65      address = 3;
66      if (start > 0) {
67          pos = symbolTable->getLevelProc(level);
68      }
69
70      int tmpPcodePtr = pcodeManager->getPcodePtr();
71      pcodeManager->gen(Operator::JMP, 0, 0);
72
73      if (allToken[tokenPtr]->getSt() == PL0SymType::CON) {
74          conDeclare();
75      }
76      if (allToken[tokenPtr]->getSt() == PL0SymType::VAR) {
77          varDeclare();
78      }
79      if (allToken[tokenPtr]->getSt() == PL0SymType::PROC) {
80          proc();
81      }
82
83      pcodeManager->getAllPcode()[tmpPcodePtr]->setA(pcodeManager->getPcodePtr());
84      pcodeManager->gen(Operator::INT, 0, address);
85      if (start != 0) {
86          symbolTable->getAllSymbol()[pos]->setValue(pcodeManager->getPcodePtr() - 1 - symbolTable->getAllSymbol()[pos]->getSize());
87      }
88
89      statement();
90      pcodeManager->gen(Operator::OPR, 0, 0);
91
92      address = address_cp;
93  }
```

- void conDeclare()：处理常量声明部分。

```
95  void conDeclare() {
96      if (allToken[tokenPtr]->getSt() == PL0SymType::CON) {
97          tokenPtr++;
98          conHandle();
99          while (allToken[tokenPtr]->getSt() == PL0SymType::COMMA || allToken[tokenPtr]->getSt() == PL0SymType::SYM) {
100             if (allToken[tokenPtr]->getSt() == PL0SymType::COMMA) {
101                 tokenPtr++;
102             }
103             else {
104                 errorHandle(23, "");
105             }
106             conHandle();
107         }
108         if (allToken[tokenPtr]->getSt() == PL0SymType::SEMIC) {
109             tokenPtr++;
110         }
111         else {
112             errorHandle(0, "");
113         }
114     }
115     else {
116         errorHandle(-1, "");
117     }
118 }
```

- void conHandle()：处理具体的常量定义。

```
120 void conHandle() {
121     std::string name;
122     int value;
123     if (allToken[tokenPtr]->getSt() == PL0SymType::SYM) {
124         name = allToken[tokenPtr]->getValue();
125         tokenPtr++;
126         if (allToken[tokenPtr]->getSt() == PL0SymType::EQU || allToken[tokenPtr]->getSt() == PL0SymType::CEQU) {
127             if (allToken[tokenPtr]->getSt() == PL0SymType::CEQU) {
128                 errorHandle(3, "");
129             }
```

```
                tokenPtr++;
                if (allToken[tokenPtr]->getSt() == PLOSymType::CONST) {
                    value = std::stoi(allToken[tokenPtr]->getValue());
                    if (symbolTable->isNowExists(name, level)) {
                        errorHandle(15, name);
                    }
                    symbolTable->enterConst(name, level, value, address);
                    tokenPtr++;
                }
            }
            else {
                errorHandle(3, "");
            }
        }
        else {
            errorHandle(1, "");
        }
    }

```

- void varDeclare()：处理变量声明部分。

```
void varDeclare() {
    std::string name;
    if (allToken[tokenPtr]->getSt() == PLOSymType::VAR) {
        tokenPtr++;
        if (allToken[tokenPtr]->getSt() == PLOSymType::SYM) {
            name = allToken[tokenPtr]->getValue();
            if (symbolTable->isNowExists(name, address)) {
                errorHandle(15, name);
            }
            symbolTable->enterVar(name, level, address);
            address += addIncrement;
            tokenPtr++;
            while (allToken[tokenPtr]->getSt() == PLOSymType::COMMA || allToken[tokenPtr]->getSt() == PLOSymType::SYM) {
                if (allToken[tokenPtr]->getSt() == PLOSymType::COMMA) {
                    tokenPtr++;
                }
                else {
                    errorHandle(23, "");
                }
                if (allToken[tokenPtr]->getSt() == PLOSymType::SYM) {
                    name = allToken[tokenPtr]->getValue();
                    if (symbolTable->isNowExists(name, address)) {
                        errorHandle(15, name);
                    }
                    symbolTable->enterVar(name, level, address);
                    address += addIncrement;
                    tokenPtr++;
                }
                else {
                    errorHandle(1, "");
                    return;
                }
            }
            if (allToken[tokenPtr]->getSt() != PLOSymType::SEMIC) {
                errorHandle(0, "");
                return;
            }
            else {
                tokenPtr++;
            }
        }
        else {
            errorHandle(1, "");
            return;
        }
    }
    else {
        errorHandle(-1, "");
        return;
    }
}
```

- void proc()：处理过程声明部分。

```
    void proc() {
        if (allToken[tokenPtr]->getSt() == PLOSymType::PROC) {
            tokenPtr++;
            int pos;
            if (allToken[tokenPtr]->getSt() == PLOSymType::SYM) {
                std::string name = allToken[tokenPtr]->getValue();
                if (symbolTable->isNowExists(name, level)) {
                    errorHandle(15, name);
                }
```

```
210                 pos = symbolTable->getPtr();
211                 symbolTable->enterProc(name, level, address);
212                 address += addIncrement;
213                 level++;
214                 tokenPtr++;
215                 if (allToken[tokenPtr]->getSt() == PLOSymType::SEMIC) {
216                     tokenPtr++;
217                 }
218                 else {
219                     errorHandle(0, "");
220                 }
221                 block();
222                 while (allToken[tokenPtr]->getSt() == PLOSymType::SEMIC || allToken[tokenPtr]->getSt() == PLOSymType::PROC) {
223                     if (allToken[tokenPtr]->getSt() == PLOSymType::SEMIC) {
224                         tokenPtr++;
225                     }
226                     else {
227                         errorHandle(0, "");
228                     }
229                     level--;
230                     proc();
231                 }
232             }
233             else {
234                 errorHandle(-1, "");
235                 return;
236             }
237         }
238     }
239
240     void body() {
```

- void body()：处理复合语句。

```
240     void body() {
241         if (allToken[tokenPtr]->getSt() == PLOSymType::BEG) {
242             tokenPtr++;
243             statement();
244             while (allToken[tokenPtr]->getSt() == PLOSymType::SEMIC || isHeadOfStatement()) {
245                 if (allToken[tokenPtr]->getSt() == PLOSymType::SEMIC) {
246                     tokenPtr++;
247                 }
248                 else {
249                     if (allToken[tokenPtr]->getSt() != PLOSymType::END) {
250                         errorHandle(0, "");
251                     }
252                 }
253                 if (allToken[tokenPtr]->getSt() == PLOSymType::END) {
254                     errorHandle(21, "");
255                     break;
256                 }
257                 statement();
258             }
259             if (allToken[tokenPtr]->getSt() == PLOSymType::END) {
260                 tokenPtr++;
261             }
262             else {
263                 errorHandle(7,    );
264                 return;
265             }
266         }
267         else {
268             errorHandle(6, "");
269             return;
270         }
271     }
```

- void statement()：处理语句，包括赋值语句、条件语句、当循环语句等。

```
void statement() {
    if (allToken[tokenPtr]->getSt() == PLOSymType::IF) {
        tokenPtr++;
        condition();
        if (allToken[tokenPtr]->getSt() == PLOSymType::THEN) {
            int pos1 = pcodeManager->getPcodePtr();
            pcodeManager->gen(Operator::JPC, 0, 0);
            tokenPtr++;
            statement();
            int pos2 = pcodeManager->getPcodePtr();
            pcodeManager->gen(Operator::JMP, 0, 0);
            pcodeManager->getAllPcode()[pos1]->setA(pcodeManager->getPcodePtr());
            pcodeManager->getAllPcode()[pos2]->setA(pcodeManager->getPcodePtr());
            if (allToken[tokenPtr]->getSt() == PLOSymType::ELS) {
                tokenPtr++;
                statement();
                pcodeManager->getAllPcode()[pos2]->setA(pcodeManager->getPcodePtr());
            }
        }
        else {
            errorHandle(8, "");
            return;
        }
    }
```

```
297         else if (allToken[tokenPtr]->getSt() == PLOSymType::WHI) {
298             int pos1 = pcodeManager->getPcodePtr();
299             tokenPtr++;
300             condition();
301             if (allToken[tokenPtr]->getSt() == PLOSymType::DO) {
302                 int pos2 = pcodeManager->getPcodePtr();
303                 pcodeManager->gen(Operator::JPC, 0, 0);
304                 tokenPtr++;
305                 statement();
306                 pcodeManager->gen(Operator::JMP, 0, pos1);
307                 pcodeManager->getAllPcode()[pos2]->setA(pcodeManager->getPcodePtr());
308             }
309             else {
310                 errorHandle(9, "");
311                 return;
312             }
313         }
314         else if (allToken[tokenPtr]->getSt() == PLOSymType::CAL) {
315             tokenPtr++;
316             std::shared_ptr<PLOSymbol> tmp;
317             if (allToken[tokenPtr]->getSt() == PLOSymType::SYM) {
318                 std::string name = allToken[tokenPtr]->getValue();
319                 if (symbolTable->isPreExists(name, level)) {
320                     tmp = symbolTable->getSymbol(name);
321                     if (tmp->getType() == symbolTable->getProc()) {
322                         pcodeManager->gen(Operator::CAL, level - tmp->getLevel(), tmp->getValue());
323                     }
324                     else {
325                         errorHandle(11, "");
326                         return;
327                     }
328                 }
329                 else {
330                     errorHandle(10, "");
331                     return;
332                 }
333                 tokenPtr++;
334             }
335             else {
336                 errorHandle(1, "");
337                 return;
338             }
339         }
340         else if (allToken[tokenPtr]->getSt() == PLOSymType::REA) {
341             tokenPtr++;
342             if (allToken[tokenPtr]->getSt() == PLOSymType::LBR) {
343                 tokenPtr++;
344                 if (allToken[tokenPtr]->getSt() == PLOSymType::SYM) {
345                     std::string name = allToken[tokenPtr]->getValue();
346                     if (!symbolTable->isPreExists(name, level)) {
347                         errorHandle(10, "");
348                         return;
349                     }
350                     else {
351                         std::shared_ptr<PLOSymbol> tmp = symbolTable->getSymbol(name);
352                         if (tmp->getType() == symbolTable->getVar()) {
353                             pcodeManager->gen(Operator::OPR, 0, 16);
354                             pcodeManager->gen(Operator::STO, level - tmp->getLevel(), tmp->getAddress());
355                         }
356                         else {
357                             errorHandle(12, "");
358                             return;
359                         }
360                     }
361                 }
362                 tokenPtr++;
363                 while (allToken[tokenPtr]->getSt() == PLOSymType::COMMA) {
364                     tokenPtr++;
365                     if (allToken[tokenPtr]->getSt() == PLOSymType::SYM) {
366                         std::string name = allToken[tokenPtr]->getValue();
367                         if (!symbolTable->isPreExists(name, level)) {
368                             errorHandle(10, "");
369                             return;
370                         }
371                         else {
372                             std::shared_ptr<PLOSymbol> tmp = symbolTable->getSymbol(name);
373                             if (tmp->getType() == symbolTable->getVar()) {
374                                 pcodeManager->gen(Operator::OPR, 0, 16);
375                                 pcodeManager->gen(Operator::STO, level - tmp->getLevel(), tmp->getAddress());
376 
377                             }
378                             else {
379                                 errorHandle(12, "");
380                                 return;
381                             }
382                         }
383                         tokenPtr++;
384                     }
385                     else {
386                         errorHandle(1, "");
387                         return;
388                     }
389                 }
390                 if (allToken[tokenPtr]->getSt() == PLOSymType::RBR) {
391                     tokenPtr++;
392                 }
393                 else {
```

```
                errorHandle(5, "");
                return;
            }
        }
        else {
            errorHandle(4, "");
            return;
        }
    }
    else if (allToken[tokenPtr]->getSt() == PLOSymType::WRI) {
        tokenPtr++;
        if (allToken[tokenPtr]->getSt() == PLOSymType::LBR) {
            tokenPtr++;
            expression();
            pcodeManager->gen(Operator::OPR, 0, 14);
            while (allToken[tokenPtr]->getSt() == PLOSymType::COMMA) {
                tokenPtr++;
                expression();
                pcodeManager->gen(Operator::OPR, 0, 14);
            }
            pcodeManager->gen(Operator::OPR, 0, 15);
            if (allToken[tokenPtr]->getSt() == PLOSymType::RBR) {
                tokenPtr++;
            }
            else {
                errorHandle(5, "");
                return;
            }
        }
        else {
            errorHandle(4, "");
        }
    }
    else if (allToken[tokenPtr]->getSt() == PLOSymType::BEG) {
        body();
    }
    else if (allToken[tokenPtr]->getSt() == PLOSymType::SYM) {
        std::string name = allToken[tokenPtr]->getValue();
        tokenPtr++;
        if (allToken[tokenPtr]->getSt() == PLOSymType::CEQU || allToken[tokenPtr]->getSt() == PLOSymType::EQU || allToken[tokenPtr]->
            if (allToken[tokenPtr]->getSt() == PLOSymType::EQU || allToken[tokenPtr]->getSt() == PLOSymType::COL) {
                errorHandle(3, "");
            }
            tokenPtr++;
            expression();
            if (!symbolTable->isPreExists(name, level)) {
                errorHandle(14, name);
                return;
            }
            else {
                std::shared_ptr<PLOSymbol> tmp = symbolTable->getSymbol(name);
                if (tmp->getType() == symbolTable->getVar()) {
                    pcodeManager->gen(Operator::STO, level - tmp->getLevel(), tmp->getAddress());
                }
                else {
                    errorHandle(13, name);
                    return;
                }
            }
        }
        else {
            errorHandle(3, "");
            return;
        }
    }
    else if (allToken[tokenPtr]->getSt() == PLOSymType::REP) {
        tokenPtr++;
        int pos = pcodeManager->getPcodePtr();
        statement();
        while (allToken[tokenPtr]->getSt() == PLOSymType::SEMIC || isHeadOfStatement()) {
            if (isHeadOfStatement()) {
                errorHandle(1, "");
            }
            else {
                tokenPtr++;
            }
            if (allToken[tokenPtr]->getSt() == PLOSymType::UNT) {
                errorHandle(22, "");
                break;
            }
            tokenPtr++;
            statement();
        }
        if (allToken[tokenPtr]->getSt() == PLOSymType::UNT) {
            tokenPtr++;
            condition();
            pcodeManager->gen(Operator::JPC, 0, pos);
        }
        else {
            errorHandle(19, "");
            return;
        }
    }
    else {
        errorHandle(1, "");
        return;
    }
}

void condition() {
    if (allToken[tokenPtr]->getSt() == PLOSymType::ODD) {
        pcodeManager->gen(Operator::OPR, 0, 6);
```

- void condition()：处理条件表达式。

```
492     void condition() {
493         if (allToken[tokenPtr]->getSt() == PL0SymType::ODD) {
494             pcodeManager->gen(Operator::OPR, 0, 6);
495             tokenPtr++;
496             expression();
497         }
498         else {
499             expression();
500             PL0SymType tmp = allToken[tokenPtr]->getSt();
501             tokenPtr++;
502             expression();
503             if (tmp == PL0SymType::EQU) {
504                 pcodeManager->gen(Operator::OPR, 0, 8);
505             }
506             else if (tmp == PL0SymType::NEQE) {
507                 pcodeManager->gen(Operator::OPR, 0, 9);
508             }
509             else if (tmp == PL0SymType::LES) {
510                 pcodeManager->gen(Operator::OPR, 0, 10);
511             }
512             else if (tmp == PL0SymType::LARE) {
513                 pcodeManager->gen(Operator::OPR, 0, 11);
514             }
515             else if (tmp == PL0SymType::LAR) {
516                 pcodeManager->gen(Operator::OPR, 0, 12);
517             }
518             else if (tmp == PL0SymType::LESE) {
519                 pcodeManager->gen(Operator::OPR, 0, 13);
520             }
521             else {
522                 errorHandle(2, "");
523             }
524         }
525     }
```

- void expression()：处理表达式。

```
527     void expression() {
528         PL0SymType tmp = allToken[tokenPtr]->getSt();
529         if (tmp == PL0SymType::ADD || tmp == PL0SymType::SUB) {
530             tokenPtr++;
531         }
532         term();
533         if (tmp == PL0SymType::SUB) {
534             pcodeManager->gen(Operator::OPR, 0, 1);
535         }
536         while (allToken[tokenPtr]->getSt() == PL0SymType::ADD || allToken[tokenPtr]->getSt() == PL0SymType::SUB)
537             tmp = allToken[tokenPtr]->getSt();
538             tokenPtr++;
539             term();
540             if (tmp == PL0SymType::ADD) {
541                 pcodeManager->gen(Operator::OPR, 0, 2);
542             }
543             else if (tmp == PL0SymType::SUB) {
544                 pcodeManager->gen(Operator::OPR, 0, 3);
545             }
546         }
547     }
```

- void term()：处理项。

```
549     void term() {
550         factor();
551         while (allToken[tokenPtr]->getSt() == PL0SymType::MUL || allToken[tokenPtr]->getSt() == PL0SymType::DIV) {
552             PL0SymType tmp = allToken[tokenPtr]->getSt();
553             tokenPtr++;
554             factor();
555             if (tmp == PL0SymType::MUL) {
556                 pcodeManager->gen(Operator::OPR, 0, 4);
557             }
558             else if (tmp == PL0SymType::DIV) {
559                 pcodeManager->gen(Operator::OPR, 0, 5);
560             }
561         }
562     }
```

- void factor()：处理因子。

```
564  void factor() {
565      if (allToken[tokenPtr]->getSt() == PLOSymType::CONST) {
566          pcodeManager->gen(Operator::LIT, 0, std::stoi(allToken[tokenPtr]->getValue()));
567          tokenPtr++;
568      }
569      else if (allToken[tokenPtr]->getSt() == PLOSymType::LBR) {
570          tokenPtr++;
571          expression();
572          if (allToken[tokenPtr]->getSt() == PLOSymType::RBR) {
573              tokenPtr++;
574          }
575          else {
576              errorHandle(5, "");
577          }
578      }
579      else if (allToken[tokenPtr]->getSt() == PLOSymType::SYM) {
580          std::string name = allToken[tokenPtr]->getValue();
581          if (!symbolTable->isPreExists(name, level)) {
582              errorHandle(10, "");
583              return;
584          }
585          else {
586              std::shared_ptr<PLOSymbol> tmp = symbolTable->getSymbol(name);
587              if (tmp->getType() == symbolTable->getVar()) {
588                  pcodeManager->gen(Operator::LOD, level - tmp->getLevel(), tmp->getAddress());
589              }
590              else if (tmp->getType() == symbolTable->getCon()) {
591                  pcodeManager->gen(Operator::LIT, 0, tmp->getValue());
592              }
593              else {
594                  errorHandle(12, "");
595                  return;
596              }
597          }
598          tokenPtr++;
599      }
600      else {
601          errorHandle(1, "");
602          return;
603      }
604  }
605
606  bool isHeadOfStatement() {
```

- bool isHeadOfStatement()：判断是否是语句的开头。

```
606  bool isHeadOfStatement() {
607      return (allToken[tokenPtr]->getSt() == PLOSymType::IF ||
608          allToken[tokenPtr]->getSt() == PLOSymType::WHI ||
609          allToken[tokenPtr]->getSt() == PLOSymType::CAL ||
610          allToken[tokenPtr]->getSt() == PLOSymType::REA ||
611          allToken[tokenPtr]->getSt() == PLOSymType::WRI ||
612          allToken[tokenPtr]->getSt() == PLOSymType::BEG ||
613          allToken[tokenPtr]->getSt() == PLOSymType::SYM ||
614          allToken[tokenPtr]->getSt() == PLOSymType::REP);
615  }
```

- void errorHandle(int k, const std::string& name)：处理错误信息。

```
617  void errorHandle(int k, const std::string& name) {
618      errorHappen = true;
619      std::string error;
620      switch (k) {
621      case -1:
622          error = "Error happened in line " + std::to_string(allToken[tokenPtr]->getLine()) + ": wrong token";
623          break;
624      case 0:
625          if (allToken[tokenPtr]->getSt() == PLOSymType::SYM) {
626              error = "Error happened in line " + std::to_string(allToken[tokenPtr]->getLine()) + ": Missing ; before " + allToken[tokenP
627          }
628          else {
629              error = "Error happened in line " + std::to_string(allToken[tokenPtr]->getLine()) + ": Missing ; before " + std::to_string(
630          }
631          break;
632      case 1:
633          error = "Error happened in line " + std::to_string(allToken[tokenPtr]->getLine()) + ": Identifier illegal";
634          break;
635      case 2:
636          error = "Error happened in line " + std::to_string(allToken[tokenPtr]->getLine()) + ": illegal compare symbol";
637          break;
638      case 3:
639          error = "Error happened in line " + std::to_string(allToken[tokenPtr]->getLine()) + ": Const assign must be =";
640          break;
641      case 4:
642          error = "Error happened in line " + std::to_string(allToken[tokenPtr]->getLine()) + ": Missing (";
```

```
            error = "Error happened in line " + std::to_string(allToken[tokenPtr]->getLine()) + ": Missing (";
            break;
        case 5:
            error = "Error happened in line " + std::to_string(allToken[tokenPtr]->getLine()) + ": Missind )";
            break;
        case 6:
            error = "Error happened in line " + std::to_string(allToken[tokenPtr]->getLine()) + ": Missing begin";
            break;
        case 7:
            error = "Error happened in line " + std::to_string(allToken[tokenPtr]->getLine()) + ": Missing end";
            break;
        case 8:
            error = "Error happened in line " + std::to_string(allToken[tokenPtr]->getLine()) + ": Missing then";
            break;
        case 9:
            error = "Error happened in line " + std::to_string(allToken[tokenPtr]->getLine()) + ": Missing do";
            break;
        case 10:
            error = "Error happened in line " + std::to_string(allToken[tokenPtr]->getLine()) + ": Not exist " + allToken[tokenPtr]->ge
            break;
        case 11:
            error = "Error happened in line " + std::to_string(allToken[tokenPtr]->getLine()) + ": " + allToken[tokenPtr]->getValue() +
            break;
        case 12:
            error = "Error happened in line " + std::to_string(allToken[tokenPtr]->getLine()) + ": " + allToken[tokenPtr]->getValue() +
            break;
        case 13:
            error = "Error happened in line " + std::to_string(allToken[tokenPtr]->getLine()) + ": " + name + " is not a varible";
            break;
        case 14:
            error = "Error happened in line " + std::to_string(allToken[tokenPtr]->getLine()) + ": not exist " + name;
            break;
        case 15:
            error = "Error happened in line " + std::to_string(allToken[tokenPtr]->getLine()) + ": Already exist " + name;
            break;
        case 16:
            error = "Error happened in line " + std::to_string(allToken[tokenPtr]->getLine()) + ": Number of parameters of procedure " +
            break;
        case 17:
            error = "Error happened in line " + std::to_string(allToken[tokenPtr]->getLine()) + ": Missing .";
            break;
        case 18:
            error = "Error happened in line " + std::to_string(allToken[tokenPtr]->getLine()) + ": too much code after .";
            break;
        case 19:
            error = "Error happened in line " + std::to_string(allToken[tokenPtr]->getLine()) + ": Missing until";
            break;
        case 20:
            error = "Error happened in line " + std::to_string(allToken[tokenPtr]->getLine()) + ": Assign must be :=";
            break;
        case 21:
            error = "Error happened in line " + std::to_string(allToken[tokenPtr]->getLine()) + ": ; is no need before end";
            break;
        case 22:
            error = "Error happened in line " + std::to_string(allToken[tokenPtr]->getLine()) + ": ; is no need before until";
            break;
        case 23:
            error = "Error happened in line " + std::to_string(allToken[tokenPtr]->getLine()) + ": Missing ,";
            break;
        }
        errorMessage.push_back(error);
    }

public:
    std::vector<std::string> getErrorMessage() const {
```

f) PLOCompiler.h

PLOCompiler 类是一个高层次的编译器接口类，用于协调词法分析、语法分析、符号表管理和 P-code 生成等编译过程的各个阶段。该类可以读取源代码文件，对其进行编译，并输出编译结果，包括符号表和 P-code 指令。

成员变量：

- std::shared_ptr<PLOSyntaxAnalyzer> gsa：用于语法分析和代码生成的主要分析器对象。
- std::vector<std::shared_ptr<PLOToken>> allToken：存储所有词法分析生成的标记。
- std::vector<std::shared_ptr<PLOSymbol>> allSymbol：存储所有符号表条目。
- std::vector<std::shared_ptr<PLOPcode>> allPcode：存储所有生成的 P-code 指令。
- std::vector<std::string> errors：存储编译过程中产生的错误信息。
- std::string consoleMessage：用于保存和显示编译信息的字符串。

- bool success：指示编译是否成功的标志。

```
class PLOCompiler {
private:
    std::shared_ptr<PLOSyntaxAnalyzer> gsa;
    std::vector<std::shared_ptr<PLOToken>> allToken;
    std::vector<std::shared_ptr<PLOSymbol>> allSymbol;
    std::vector<std::shared_ptr<PLOPcode>> allPcode;
    std::vector<std::string> errors;
    std::string consoleMessage;
    bool success = false;
```

主要方法：

- void compile(const std::string& filename)：compile 方法接受一个文件名作为参数，执行编译过程，包括语法分析、符号表管理和 P-code 生成，并输出编译结果。

```
void compile(const std::string& filename) {
    gsa = std::make_shared<PLOSyntaxAnalyzer>(filename);
    clean();
    if (success = gsa->compile()) {
        displayAllToken();
        displayAllSymbol();
        displayAllPcode();
        consoleMessage += "compile succeed!\n";
        std::cout << consoleMessage << std::endl;
    }
    else {
        displayErrorMessage();
        consoleMessage += "compile failed!";
        std::cout << consoleMessage << std::endl;
    }
}
```

- void clean()：用于在每次编译前清理所有的中间结果和信息。

```
void clean() {
    allToken.clear();
    allSymbol.clear();
    allPcode.clear();
    consoleMessage.clear();
    success = false;
}
```

- void displayErrorMessage()：用于显示编译过程中产生的错误信息。

```
void displayErrorMessage() {
    consoleMessage.clear();
    errors = gsa->getErrorMessage();
    for (const auto& error : errors) {
        consoleMessage += error + "\n";
    }
}
```

g) 补充：PLOInterpreter.h

PLOInterpreter 类提供了解释和执行 P-code 指令的功能。它通过模拟

PL/0 虚拟机的执行过程，对数据栈进行操作，实现了 PL/0 程序的运行。该类支持基本的算术运算、变量加载和存储、过程调用和跳转等功能。通过使用该类，可以执行由 PL/0 编译器生成的中间代码，实现对 PL/0 程序的解释执行。

PL0Interpreter 类的方法调用关系主要分为三个阶段：初始化阶段、执行阶段和结果获取阶段。在初始化阶段，设置需要执行的 P-code 指令；在执行阶段，解释和执行指令；在结果获取阶段，获取程序执行的输出结果。通过这些方法的调用关系，实现了对 PL/0 程序的解释执行。

首先创建 PL0Interpreter 对象并设置 P-code 指令。接着调用 interpreter 方法，开始解释执行 P-code 指令。interpreter 方法内部根据指令类型分别处理。对于加载、存储、过程调用等指令，调用 getBase 方法计算基地址。在解释执行完成后，调用 getOutput 方法获取输出结果。

成员变量：

- static const int STACK_SIZE = 1000：数据栈的大小。
- int dataStack[STACK_SIZE]：模拟运行栈，用于存储程序执行时的中间数据。
- std::vector<std::shared_ptr<PL0Pcode>> pcode：存储所有需要执行的 P-code 指令。
- std::vector<int> input：存储用户输入的数据。
- int inputPtr = 0：输入指针，用于读取用户输入。
- std::vector<std::string> output：存储程序执行的输出结果。

```
11  class PL0Interpreter {
12  private:
13      static const int STACK_SIZE = 1000;
14      int dataStack[STACK_SIZE];
15      std::vector<std::shared_ptr<PL0Pcode>> pcode;
16      std::vector<int> input;
17      int inputPtr = 0;
18      std::vector<std::string> output;
```

主要方法：

- void setAllPcode(const std::shared_ptr<PL0PcodeManager>& allPcode)：接受一个 PL0PcodeManager 对象的共享指针，并获取其中所有的 P-code 指令。

```
26  void setAllPcode(const std::shared_ptr<PL0PcodeManager>& allPcode) {
27      pcode = allPcode->getAllPcode();
28  }
```

- std::vector<std::string> getOutput() const：返回程序执行的输出结果。

```
30      std::vector<std::string> getOutput() const {
31          return output;
32      }
```

- void interpreter()：根据 P-code 指令，对数据栈进行操作，执行具体的指令操作。

```
34  void interpreter() {
35      int pc = 0; // 程序计数器，指向下一条指令
36      int base = 0; // 当前基地址
37      int top = 0; // 程序运行栈栈顶
38
39      do {
40          auto currentPcode = pcode[pc];
41          pc++;
42          switch (currentPcode->getF()) {
43          case Operator::LIT:
44              dataStack[top++] = currentPcode->getA();
45              break;
```

```
        case Operator::OPR:
            switch (currentPcode->getA()) {
            case 0:
                top = base;
                pc = dataStack[base + 2];
                base = dataStack[base];
                break;
            case 1:
                dataStack[top - 1] = -dataStack[top - 1];
                break;
            case 2:
                dataStack[top - 2] += dataStack[top - 1];
                top--;
                break;
            case 3:
                dataStack[top - 2] -= dataStack[top - 1];
                top--;
                break;
            case 4:
                dataStack[top - 2] *= dataStack[top - 1];
                top--;
                break;
            case 5:
                dataStack[top - 2] /= dataStack[top - 1];
                top--;
                break;
            case 6:
                dataStack[top - 1] %= 2;
                break;
            case 7:
                break;
            case 8:
                dataStack[top - 2] = dataStack[top - 2] == dataStack[top - 1];
                top--;
                break;
            case 9:
                dataStack[top - 2] = dataStack[top - 2] != dataStack[top - 1];
                top--;
                break;
            case 10:
                dataStack[top - 2] = dataStack[top - 2] < dataStack[top - 1];
                top--;
                break;
            case 11:
                dataStack[top - 2] = dataStack[top - 2] >= dataStack[top - 1];
                top--;
                break;
            case 12:
                dataStack[top - 2] = dataStack[top - 2] > dataStack[top - 1];
                top--;
                break;
            case 13:
                dataStack[top - 2] = dataStack[top - 2] <= dataStack[top - 1];
                top--;
                break;
            case 14:
                std::cout << dataStack[top - 1] << " ";
                break;
            case 15:
                std::cout << std::endl;
                break;
            case 16:
                std::cout << "please input a number" << std::endl;
                std::cin >> dataStack[top++];
                break;
            }
            break;
        case Operator::LOD:
            dataStack[top++] = dataStack[currentPcode->getA() + getBase(base, currentPcode->getL())];
            break;
        case Operator::STO:
            dataStack[currentPcode->getA() + getBase(base, currentPcode->getL())] = dataStack[--top];
            break;
        case Operator::CAL:
            dataStack[top] = base;
            dataStack[top + 1] = getBase(base, currentPcode->getL());
            dataStack[top + 2] = pc;
            base = top;
            pc = currentPcode->getA();
            break;
        case Operator::INT:
            top += currentPcode->getA();
            break;
        case Operator::JMP:
            pc = currentPcode->getA();
            break;
        case Operator::JPC:
            if (dataStack[--top] == 0) {
                pc = currentPcode->getA();
            }
            break;
        }
    } while (pc != 0);
```

- int getBase(int nowBp, int lev) const：用于获取层次差为 lev 的基地址。通过遍历链表，返回目标基地址。

```
142    int getBase(int nowBp, int lev) const {
143        int oldBp = nowBp;
144        while (lev > 0) {
145            oldBp = dataStack[oldBp + 1];
146            lev--;
147        }
148        return oldBp;
149    }
```

7） 测试

使用《编译原理》（第 3 版）上的测试程序：

输入样例：

```
const a=10;
var b,c;
procedure p;
    begin
    c:= b+a
end;
begin
    read(b);
    while b<>0 do
    begin
        call p;write(2*c);read(b)
    end
end.
```

输出的 P-code：

(0)jmp 0 8	转向主程序入口
(1)jmp 0 2	转向过程 p 入口
(2)int 0 3	过程 p 入口，为过程 p 开辟空间
(3)lod 1 3	取变量 b 的值到栈顶
(4)lit 0 10	取常数 10 到栈顶
(5)opr 0 2	次栈顶与栈顶相加
(6)sto 1 4	栈顶值送变量 c 中
(7)opr 0 0	退栈并返回调用点的下一条指令(16)
(8)int 0 6	主程序入口开辟 6 个栈空间
(9)opr 0 16	从命令行读入值并置于栈顶
(10)sto 0 3	将栈顶值存入变量 b 中
(11)lod 0 3	将变量 b 的值去至栈顶
(12)lit 0 0	常数值 0 进栈
(13)opr 0 9	次栈顶是否与栈顶不相等
(14)jpc 0 24	相等时转(24)
(15)cal 0 2	调用过程 p
(16)lit 0 2	常数值 2 进栈
(17)lod 0 4	将变量 c 的值取至栈顶

(18)opr 0 4	次栈顶与栈顶相乘(2*c)
(19)opr 0 14	栈顶值输出至屏幕
(20)opr 0 15	换行
(21)opr 0 16	从命令行读取值到栈顶
(22)sto 0 3	栈顶值送变量 b 中
(23)jmp 0 11	无条件转到循环入口(11)
(24)opr 0 0	结束退栈

程序运行结果：

```
Pcode Table:
Opcode (Operation)      Level Difference (L)    Address/Value (A)
5 (JMP)                 0                       8
5 (JMP)                 0                       2
0 (INT)                 0                       3
3 (LOD)                 1                       3
2 (LIT)                 0                       10
7 (OPR)                 0                       2
4 (STO)                 1                       4
7 (OPR)                 0                       0
0 (INT)                 0                       6
7 (OPR)                 0                       16
4 (STO)                 0                       3
3 (LOD)                 0                       3
2 (LIT)                 0                       0
7 (OPR)                 0                       9
6 (JPC)                 0                       24
1 (CAL)                 0                       2
2 (LIT)                 0                       2
3 (LOD)                 0                       4
7 (OPR)                 0                       4
7 (OPR)                 0                       14
7 (OPR)                 0                       15
7 (OPR)                 0                       16
4 (STO)                 0                       3
5 (JMP)                 0                       11
7 (OPR)                 0                       0
```

测试更多用例：

用例 1 输入：

test3.txt | test2.txt | **test1.txt** | PL0PcodeManager.h | PL0SyntaxAnalyzer.h | PL

```
1  const a = 45, b = 27;
2  var x, y, g, m;
3  procedure swap;
4      var temp;
5      begin
6          temp := x;
7          x := y;
8          y := temp
9      end;
10 procedure mod;
11     x := x - x / y * y;
12 begin
13     x := a;
14     y := b;
15     call mod;
16     while x <> 0 do
17     begin
18         call swap;
19         call mod
20     end;
21     g := y;
22     m := a * b / g;
23     write(g, m)
24 end.
```

用例 1 输出：

```
Pcode Table:
Opcode (Operation)      Level Difference (L)    Address/Value (A)
5 (JMP)                 0                       21
5 (JMP)                 0                       2
0 (INT)                 0                       4
3 (LOD)                 1                       3
4 (STO)                 0                       3
3 (LOD)                 1                       4
4 (STO)                 1                       3
3 (LOD)                 0                       3
4 (STO)                 1                       4
7 (OPR)                 0                       0
5 (JMP)                 0                       11
0 (INT)                 0                       3
3 (LOD)                 1                       3
3 (LOD)                 1                       3
3 (LOD)                 1                       4
7 (OPR)                 0                       5
3 (LOD)                 1                       4
7 (OPR)                 0                       4
7 (OPR)                 0                       3
4 (STO)                 1                       3
7 (OPR)                 0                       0
0 (INT)                 0                       9
2 (LIT)                 0                       45
4 (STO)                 0                       3
2 (LIT)                 0                       27
4 (STO)                 0                       4
1 (CAL)                 0                       11
3 (LOD)                 0                       3
2 (LIT)                 0                       0
7 (OPR)                 0                       9
6 (JPC)                 0                       34
1 (CAL)                 0                       2
1 (CAL)                 0                       11
5 (JMP)                 0                       27
3 (LOD)                 0                       4
4 (STO)                 0                       5
2 (LIT)                 0                       45
2 (LIT)                 0                       27
7 (OPR)                 0                       4
3 (LOD)                 0                       5
7 (OPR)                 0                       5
4 (STO)                 0                       6
3 (LOD)                 0                       5
7 (OPR)                 0                       14
3 (LOD)                 0                       6
7 (OPR)                 0                       14
7 (OPR)                 0                       15
7 (OPR)                 0                       0
```

用例 2 输入：

test3.txt | test2.txt | test1.txt | PL0PcodeManager.h | PL0SyntaxAna

```
1  const true = 1, false = 0;
2  var x, y, m, n, pf;
3  procedure prime;
4      var i, f;
5      procedure mod;
6          x := x - x / y * y;
7      begin
8          f := true;
9          i := 3;
10         while i < m do
```

```
        begin
            x := m;
            y := i;
            call mod;
            if x = 0 then f := false;
            i := i + 2
        end;
        if f = true then
        begin
            write(m);
            pf := true
        end
    end;
begin
    pf := false;
    read(n);
    while n >= 2 do
    begin
        write(2);
        if n = 2 then pf := true;
        m := 3;
        while m <= n do
        begin
            call prime;
            m := m + 2
        end;
        read(n)
    end;
    if pf = false then write(0)
end.
```

用例 2 输出：

```
Pcode Table:
Opcode (Operation)      Level Difference (L)    Address/Value (A)
5 (JMP)                 0                       50
5 (JMP)                 0                       13
5 (JMP)                 0                       3
0 (INT)                 0                       3
3 (LOD)                 2                       3
3 (LOD)                 2                       3
3 (LOD)                 2                       4
7 (OPR)                 0                       5
3 (LOD)                 2                       4
7 (OPR)                 0                       4
7 (OPR)                 0                       3
4 (STO)                 2                       3
7 (OPR)                 0                       0
0 (INT)                 0                       6
2 (LIT)                 0                       1
4 (STO)                 0                       4
2 (LIT)                 0                       3
4 (STO)                 0                       3
3 (LOD)                 0                       3
3 (LOD)                 1                       5
7 (OPR)                 0                       10
6 (JPC)                 0                       39
3 (LOD)                 1                       5
4 (STO)                 1                       3
3 (LOD)                 0                       3
4 (STO)                 1                       4
1 (CAL)                 0                       3
3 (LOD)                 1                       3
2 (LIT)                 0                       0
7 (OPR)                 0                       8
6 (JPC)                 0                       34
2 (LIT)                 0                       0
4 (STO)                 0                       4
5 (JMP)                 0                       34
3 (LOD)                 0                       3
2 (LIT)                 0                       2
7 (OPR)                 0                       2
4 (STO)                 0                       3
5 (JMP)                 0                       18
3 (LOD)                 0                       4
```

```
2 (LIT)                 0                       1
7 (OPR)                 0                       8
6 (JPC)                 0                       49
3 (LOD)                 1                       5
7 (OPR)                 0                       14
7 (OPR)                 0                       15
2 (LIT)                 0                       1
4 (STO)                 1                       7
5 (JMP)                 0                       49
7 (OPR)                 0                       0
0 (INT)                 0                       9
2 (LIT)                 0                       0
4 (STO)                 0                       7
7 (OPR)                 0                       16
4 (STO)                 0                       6
3 (LOD)                 0                       6
2 (LIT)                 0                       2
7 (OPR)                 0                       11
6 (JPC)                 0                       84
2 (LIT)                 0                       2
7 (OPR)                 0                       14
7 (OPR)                 0                       15
3 (LOD)                 0                       6
2 (LIT)                 0                       2
7 (OPR)                 0                       8
6 (JPC)                 0                       69
2 (LIT)                 0                       1
4 (STO)                 0                       7
5 (JMP)                 0                       69
2 (LIT)                 0                       3
4 (STO)                 0                       5
3 (LOD)                 0                       5
3 (LOD)                 0                       6
7 (OPR)                 0                       13
6 (JPC)                 0                       81
1 (CAL)                 0                       13
3 (LOD)                 0                       5
2 (LIT)                 0                       2
7 (OPR)                 0                       2
4 (STO)                 0                       5
5 (JMP)                 0                       71
7 (OPR)                 0                       16
4 (STO)                 0                       6
5 (JMP)                 0                       55
3 (LOD)                 0                       7
2 (LIT)                 0                       0
7 (OPR)                 0                       8
6 (JPC)                 0                       92
2 (LIT)                 0                       0
7 (OPR)                 0                       14
7 (OPR)                 0                       15
5 (JMP)                 0                       92
7 (OPR)                 0                       0

compile succeed!
```

用例 3 输入：

test3.txt | test2.txt | test1.txt | PL0PcodeManager.h | PL0Syntax

```
const z = 0;
var head, foot, cock, rabbit, n;
begin
    n := z;
    read(head, foot);
    cock := 1;
    while cock <= head do
    begin
        rabbit := head - cock;
        if cock * 2 + rabbit * 4 = foot then
        begin
            write(cock, rabbit);
            n := n + 1
        end;
        cock := cock + 1
    end;
    if n = 0 then write(0, 0)
end.
```

用例 3 输出：

```
Pcode Table:
Opcode (Operation)      Level Difference (L)    Address/Value (A)
5 (JMP)                 0                       1
0 (INT)                 0                       8
2 (LIT)                 0                       0
4 (STO)                 0                       7
7 (OPR)                 0                       16
4 (STO)                 0                       3
7 (OPR)                 0                       16
4 (STO)                 0                       4
2 (LIT)                 0                       1
4 (STO)                 0                       5
3 (LOD)                 0                       5
3 (LOD)                 0                       3
7 (OPR)                 0                       13
6 (JPC)                 0                       43
3 (LOD)                 0                       3
3 (LOD)                 0                       5
7 (OPR)                 0                       3
4 (STO)                 0                       6
3 (LOD)                 0                       5
2 (LIT)                 0                       2
7 (OPR)                 0                       4
3 (LOD)                 0                       6
2 (LIT)                 0                       4
7 (OPR)                 0                       4
7 (OPR)                 0                       2
3 (LOD)                 0                       4
7 (OPR)                 0                       8
6 (JPC)                 0                       38
3 (LOD)                 0                       5
7 (OPR)                 0                       14
3 (LOD)                 0                       6
7 (OPR)                 0                       14
7 (OPR)                 0                       15
3 (LOD)                 0                       7
2 (LIT)                 0                       1
7 (OPR)                 0                       2
4 (STO)                 0                       7
5 (JMP)                 0                       38
3 (LOD)                 0                       5
2 (LIT)                 0                       1
7 (OPR)                 0                       2
4 (STO)                 0                       5
5 (JMP)                 0                       10
3 (LOD)                 0                       7
2 (LIT)                 0                       0
7 (OPR)                 0                       8
6 (JPC)                 0                       53
2 (LIT)                 0                       0
7 (OPR)                 0                       14
2 (LIT)                 0                       0
7 (OPR)                 0                       14
7 (OPR)                 0                       15
5 (JMP)                 0                       53
7 (OPR)                 0                       0
```

五、 实验心得

通过本次实验，我深入理解了编译器的设计和实现过程，特别是词法分析和语法分析的关键步骤。在编写 PL/0 编译器的过程中，我学习了如何将源代码转化为 Token 序列，并利用语法分析生成中间代码（P-code）。通过对 PL/0 语言的编译和解释执行，我不仅巩固了对编译原理课程的理论知识，还提升了实际编程能力，尤其是在 C++语言的使用和面向对象编程的实践中受益匪浅。此外，调试和测试编译器的过程也让我意识到处理错误和异常的重要性，以及如何通过完善的错误报告机制提升编译器的健壮性。