厦門大學

信息学院软件工程系

实验报告

题　　目 PL\0 编译系统实现

班　　级 软件工程 2021 级卓越班

姓　　名 吴桐

学　　号 22920212204458

实验时间 2024 年 6 月 20 日

一. 设计任务

1.1 程序设计要求

PL/0语言可以看成PASCAL语言的子集，它的编译程序是一个编译解释执行系统。PL/0的目标程序为假想栈式计算机的汇编语言，与具体计算机无关。

其编译过程采用一趟扫描方式，以语法分析程序为核心，词法分析和代码生成程序都作为一个独立的过程，当语法分析需要读单词时就调用词法分析程序，而当语法分析正确需要生成相应的目标代码时，则调用代码生成程序。

用表格管理程序建立变量、常量和过程标示符的说明与引用之间的信息联系。

用出错处理程序对词法和语法分析遇到的错误给出在源程序中出错的位置和错误性质。

1.2 PL/0语言的BNF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> ::= <标识符>=<无符号整数>
- <无符号整数> ::= <数字>{<数字>}
- <标识符> ::= <字母>{<字母>|<数字>}
- <变量说明部分>::= var<标识符>{,<标识符>};
- <过程说明部分> ::= <过程首部><分程序>{;<过程说明部分>};
- <过程首部> ::= procedure<标识符>;
- <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>
- <赋值语句> ::= <标识符>:=<表达式>
- <表达式> ::= [+|-]<项>{<加减法运算符><项>}
- <项> ::= <因子>{<乘除法运算符><因子>}
- <因子> ::= <标识符>|<无符号整数>|'('<表达式>')'
- <加减法运算符> ::= +|-
- <乘除法运算符> ::= *|/
- <条件> ::= <表达式><关系运算符><表达式>|odd<表达式>
- <关系运算符> ::= =|<>|<|<=|>|>=
- <条件语句> ::= if<条件>then<语句>[else<语句>]
- <当型循环语句> ::= while<条件>do<语句>
- <过程调用语句> ::= call<标识符>
- <复合语句> ::= begin<语句>{;<语句>}end
- <重复语句> ::= repeat<语句>{;<语句>}until<条件>
- <读语句> ::= read'('<标识符>{,<标识符>}')'
- <写语句> ::= write'('<标识符>{,<标识符>}')'
- <字母> ::= a|b|...|X|Y|Z
- <数字> ::= 0|1|2|...|8|9

1.3 P-code

1. **LIT** （Load Immediate，立即数加载）：将一个常量值加载到栈顶。用法示例：`LIT 0, 5` —— 将5压入栈顶
2. **OPR** （Operation，操作）：执行算术或逻辑运算。用法示例：`OPR 0, 2` —— 栈顶两个数相加。
3. **LOD** （Load，加载）：将变量值加载到栈顶。用法示例：`LOD 1, 3` —— 加载位于距离当前静态链1层距离的数据区偏移为3的变量值。
4. **STO** （Store，存储）：将栈顶的值存储到某个变量中。用法示例：`STO 1, 3` —— 将栈顶值存储到位于距离当前静态链1层距离的数据区偏移为3的变量中。
5. **CAL** （Call，调用）：调用一个过程。用法示例：`CAL 0, 1` —— 调用位于当前静态链同层的地址为1的过程。
6. **INC** （Increment，增加）：增加数据区的分配。用法示例：`INC 0, 4` —— 在数据区增加4个单元的空间。
7. **JMP** （Jump，跳转）：无条件跳转到某个地址。用法示例：`JMP 0, 10` —— 无条件跳转到地址10。
8. **JPC** （Jump on Condition，条件跳转）：如果栈顶值为0，则跳转到某个地址。用法示例：`JPC 0, 20` —— 如果栈顶值为0，则跳转到地址20。
9. **SIO** （Standard Input/Output，标准输入/输出）：进行输入或输出操作。用法示例：`SIO 0, 1` —— 输出栈顶值；`SIO 0, 2` —— 读入一个值到栈顶。

二. 功能结构设计

2.1 词法分析

- 根据所给的PL/0语言的BNF描述，该语言的组成单词包括以下元素：
  - 关键字（程序保留字）{program，const，var，procedure，begin，end，if，else，then，call，while，do，read，write}
  - 运算符：{+，-，*，/，odd}
  - 界符：{“,"，“;”，“(”，“)”，“:=”}
  - 关系运算符：{=，<，<=，>，>=，<>}
  - 数字：只能为整型，且常量不可以是负数
  - 标识符：由用户定义，以字母开头，由数字和字母组成
- 词法分析程序读取源程序文件，过滤掉多余的空格、回车以及换行，接着再识别出文件中的所有关键字、运算符、界符、运算关系、数学、标识符五种元素，供后续程序使用。词法分析器输出的最终结果往往是形如"(单词种类，单词符号的属性值)"的二元式

2.2 语法分析

- 语法分析的主要任务是“组词成句”，将词法分析给出的单词序列按语法规则构成更大的语法单位，如“程序、语句、表达式”等；或者说，语法分析的作用是用来判断给定输入串是否为合乎文法的句子。
- 语法分析结合BNF产生式，利用递归下降的方法实现。具体实现方式是，为每一个非终结符编写一个子程序，子程序中如果遇到非终结符就调用相关非终结符的子程序（这也就是递归下降名字的由来）

```
// 例如，针对产生式 A -> BCD来说，一个可行的递归下降程序是：
void A()
{
    ...     // 其他处理逻辑
    B();    // 非终结符B的处理子程序
    ...     // 其他处理逻辑
    C();    // 非终结符C的处理子程序
    ...     // 其他处理逻辑
    D();    // 非终结符D的处理子程序
}
```

2.3 语义分析

- 语义分析也成为类型检查、上下文相关分析，分析检查程序（语法树）的上下文相关属性，包括：
  - 变量在使用前有进行声明
  - 每个表达式都有合适的类型
  - 函数的调用过程和函数的定义一致
- 由于PL/0语言的特性（仅仅只有整型数，且函数调用不能添加参数），因此在语义分析阶段语义分析器主要集中精力处理，诸如：变量的声明、函数声明等等相关问题。

2.3.1 符号表管理

- 符号表中存储以下数据：定义的变量、定义的常量、定义的过程；
  - 定义的变量需要存储：变量的标识符，相对于该层次基地址的偏移量，变量定义所在层次
  - 定义的常量需要存储：常量的标识符，定义所在层次，常量的值
  - 定义的过程需要存储：过程名，以及其所在层次
- 可以将符号表理解成一个表格，其中储存着名字（name）、种类（kind）、值（val）、层次（level）、和地址（adr）。这样编译程序就可以根据表格中的信息找到用户程序所需要访问的变量以及判断是否有资格访问该变量

2.3.2 中间代码生成

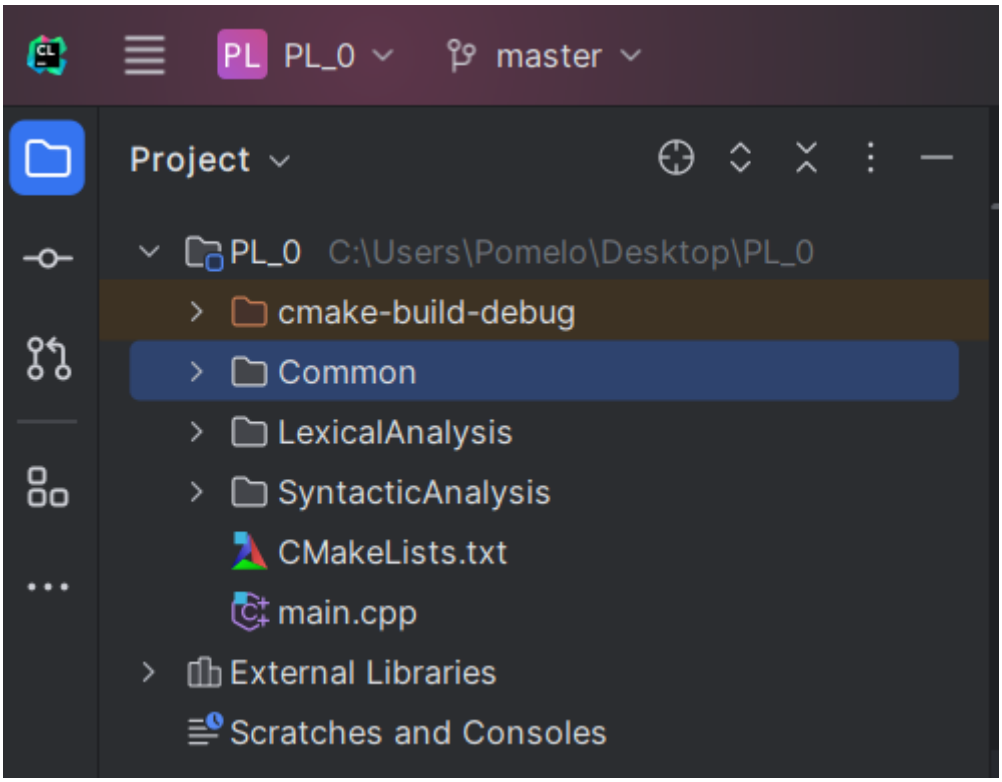
- 中间代码生成是编译过程中的一个关键阶段，它位于语法分析和目标代码生成之间。在这个阶段，编译器将源程序的语法树（或抽象语法树，AST）转换为一种更接近机器语言但仍具有一定抽象级别的代码形式。
- 中间代码生成涉及到翻译模板的相关内容

2.3.3 错误处理

- 出错处理程序对词法和语法分析遇到的错误给出在源程序中出错的位置和错误性质

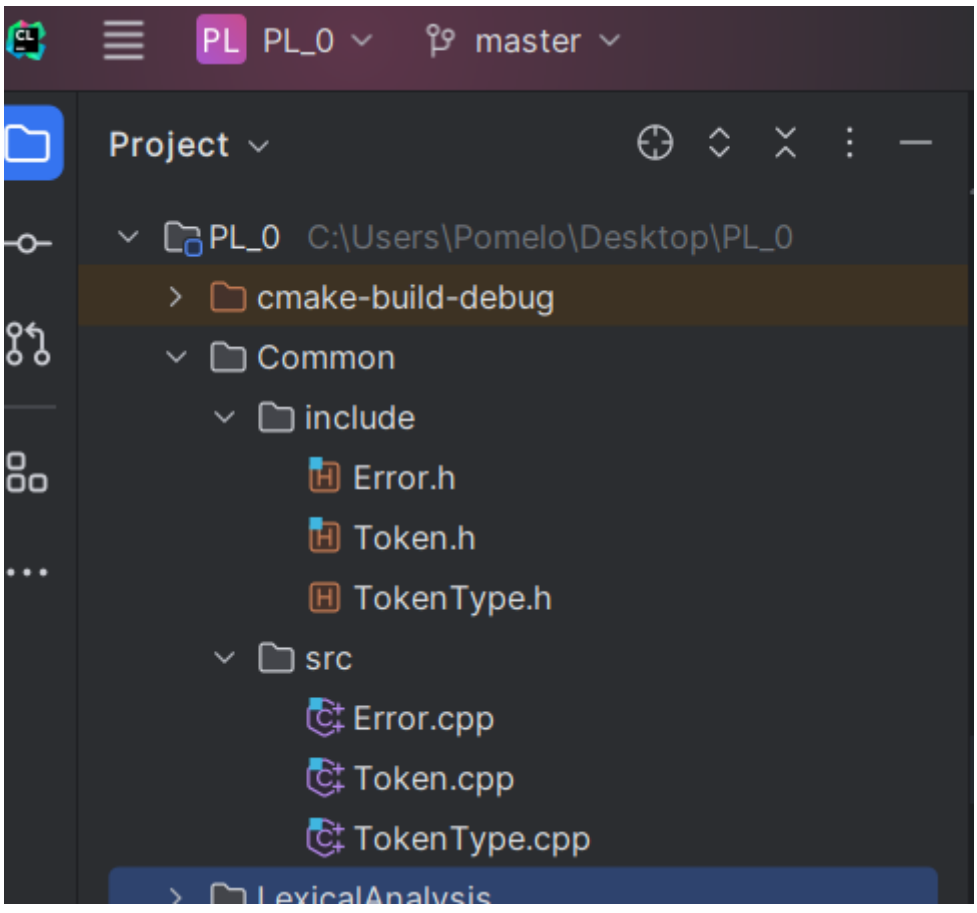
三. 系统实现

3.1 整体代码结构

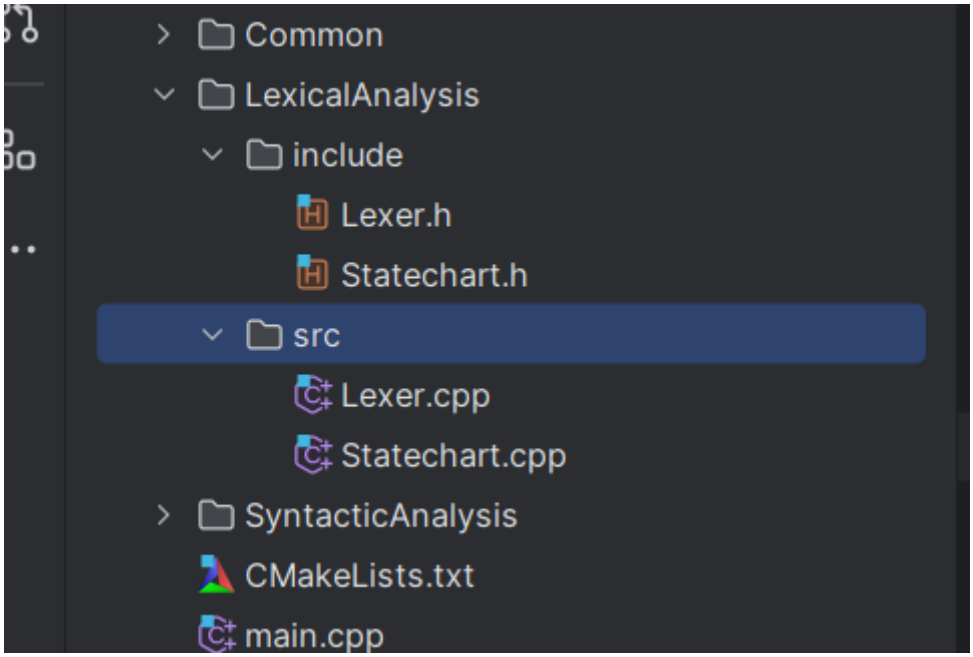


该项目代码主要分为四个模块：

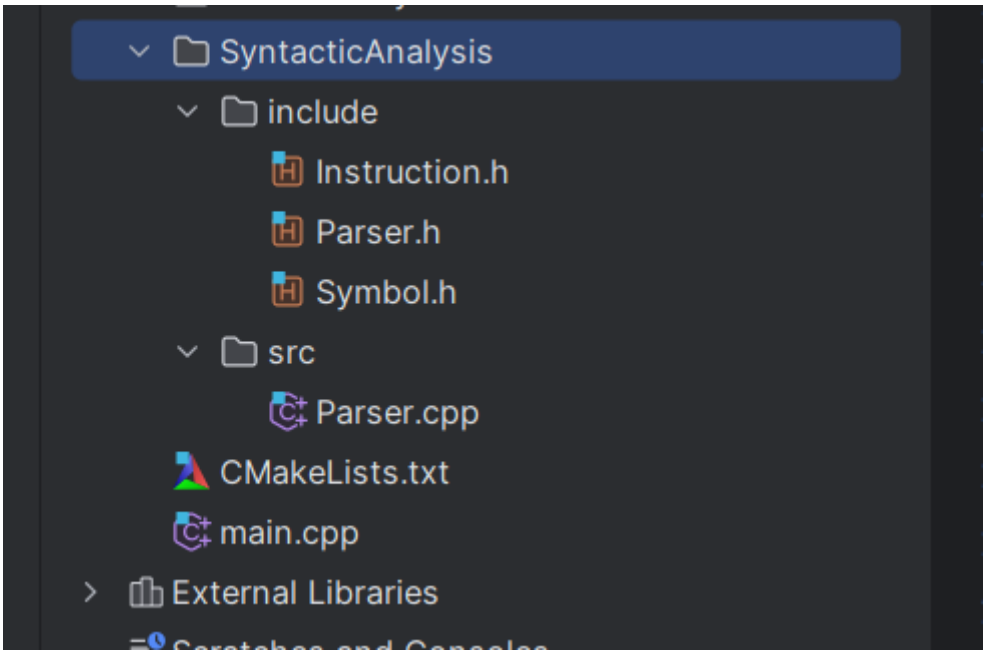
- `Common模块`下定义了整个程序的各个阶段都可能使用到的通用类，其中包括了错误处理类、单词类以及单词类型类



- `LexicalAnalysis模块`下主要包含了和词法分析相关类，如词法分析器类Lexer、状态图类Statechart



- `SyntacticAnalysis模块`下主要包含了语法分析、语义分析相关的类



3.2 具体模块的实现

3.2.1 Common模块

TokenType

```cpp
#include ...

enum TokenType {
    plusSym, minusSym, mulSym, slashSym,
    eqSym, neqSym, leqSym, lesSym, geqSym, gtrSym,
    lparentSym, rparentSym, commaSym, semicolonSym,
    periodSym, becomesSym, oddSym, beginSym, endSym, ifSym,
    thenSym, whileSym, doSym, callSym, constSym, varSym,
    procSym, writeSym, readSym , elseSym, identSym, numberSym,
    tokenTypeCount
};

extern std::unordered_map<TokenType, std::string> type2strMap;
extern std::string type2str(TokenType type);

#endif //PL_0_TOKENTYPE_H
```

在本系统中，主要分为了33个类别，除了**identSym、numberSym和tokenTypeCount**这三个外，其余每个种类都表示了具体的一种单词，例如：**neqSym**表示**<>**。其实现方式是利用枚举类型进行实现，其中每种类型的具体含义如下

类型	说明
plusSym	+（运算符，加法）
minusSym	-（运算符，减法）
mulSym	*（运算符，乘法）
slashSym	/（运算符，除法）
eqSym	=（关系运算符，等于）
neqSym	<>（关系运算符，不等于）
leqSym	<=（关系运算符，小于等于）
lesSym	<（关系运算符，小于）
geqSym	>=（关系运算符，大于等于）
gtrSym	>（关系运算符，大于）
lparentSym	(（界符，左括号）
rparentSym	)（界符，右括号）
commaSym	,（界符，逗号）
semicolonSym	;（界符，分号）
periodSym	.（界符，句号）
becomesSym	:=（运算符，赋值）
oddSym	odd（运算符，判断奇数）
beginSym	begin（关键字）
endSym	end（关键字）
ifSym	if（关键字）
thenSym	then（关键字）
whileSym	while（关键字）
doSym	do（关键字）
callSym	call（关键字）
constSym	const（关键字）
varSym	var（关键字）
procSym	procedure（关键字）
writeSym	write（关键字）
readSym	read（关键字）
elseSym	else（关键字）
identSym	标识符
numberSym	数字

类型	说明
tokenTypeCount	占位符，用来表示有多少个type

实现type2str方法，建立类型和对应描述的连接，便于后续的输出

```
std::unordered_map<TokenType, std::string> type2strMap = {
        { x: TokenType::identSym,  y: "identSym"},
        { x: TokenType::numberSym,  y: "numberSym"},
        { x: TokenType::plusSym,  y: "plusSym"},
        { x: TokenType::minusSym,  y: "minusSym"},
        { x: TokenType::mulSym,  y: "mulSym"},
        { x: TokenType::slashSym,  y: "slashSym"},
        { x: TokenType::oddSym,  y: "oddSym"},
        { x: TokenType::eqSym,  y: "eqSym"},
        { x: TokenType::neqSym,  y: "neqSym"},
        { x: TokenType::lesSym,  y: "lesSym"},
        { x: TokenType::leqSym,  y: "leqSym"},
        { x: TokenType::gtrSym,  y: "gtrSym"},
        { x: TokenType::geqSym,  y: "geqSym"},
        { x: TokenType::lparentSym,  y: "lparentSym"},
        { x: TokenType::rparentSym,  y: "rparentSym"},
        { x: TokenType::commaSym,  y: "commaSym"},
        { x: TokenType::semicolonSym,  y: "semicolonSym"},
        { x: TokenType::periodSym,  y: "periodSym"},
        { x: TokenType::becomesSym,  y: "becomesSym"},
        { x: TokenType::beginSym,  y: "beginSym"},
        { x: TokenType::endSym,  y: "endSym"},
        { x: TokenType::ifSym,  y: "ifSym"},
        { x: TokenType::thenSym,  y: "thenSym"},
        { x: TokenType::whileSym,  y: "whileSym"},
        { x: TokenType::doSym,  y: "doSym"},
        { x: TokenType::callSym,  y: "callSym"},
        { x: TokenType::constSym,  y: "constSym"},
        { x: TokenType::varSym,  y: "varSym"},
        { x: TokenType::procSym,  y: "procSym"},
        { x: TokenType::writeSym,  y: "writeSym"},
        { x: TokenType::readSym,  y: "readSym"},
        { x: TokenType::elseSym,  y: "elseSym"},
};

std::string type2str(TokenType type)
{
    if (type2strMap.count( x: type)) return type2strMap[type];
    else return "NotFound";
}
```

Token

Token类用来表示词法分析解析出来的单词，它有两个成员变量组成，其中type表示Token的类型，val表示Token的具体值（因为只有数字不同于其他单词，这里把数字也作为字符串进行存储）

```
11
12      class Token {
13      private:
14          TokenType type;      // 类型
15          std::string val;     // 值
16      public:
17          Token();
18          Token(TokenType, std::string);
19
20          std::string getValue();
21          TokenType getType();
22          void print();
23      };
```

Error

ErrorType用于表示错误的类型，这里主要分为了5种错误，用来表示在不同的阶段产生的错误；Error类用于将错误类型以及具体错误信息

```
11  enum class ErrorType {
12      LexicalError,
13      SyntaxError,
14      SemanticError,
15      CodeGenError,
16      OtherError,
17  };
18
19
20  class Error {
21  public:
22      static void printErrors(ErrorType type, const std::string &message, bool exitFlag = true, int line = -1);
23
24  private:
25      static std::string getTypeString(ErrorType type);
26  };
```

其中printErrors打印错误信息的详细结果如下图所示，打印的信息有：错误类型，行号以及错误信息描述。并且可以选择是否中断程序

```
17
18  void Error::printErrors(ErrorType type, const std::string &message, bool exitFlag, int line) {
19      std::cout << "Error Type: " << getTypeString(type) << "\n";
20      if (line != -1) std::cout << "Line: " << line << "\n";
21      std::cout << "Message: " << message << "\n\n";
22      if (exitFlag) exit( Code: 1);
23  }
24
```

3.2.2 Lexical Analysis模块

Statechart

Statechart类的作用是从一个不含有空格的字符串中分割出token。其中包含了成员变量diagram（用于表示不同类型的状态图，状态图的表示方式是，(初始状态, 输入字符)->目标状态，如果状态图中不含有相关的转换关系，那么在map中不存在该值对(初始状态, 输入字符)。

另外，这里还使用到了单例模式，保证系统中只有一个Statechart对象；

```
class Statechart {
private:
    Statechart();

    // 对应不同类型的状态图
    std::map<std::pair<int, char>, int> diagram[TokenType::tokenTypeCount];
    // 对应不同类型的状态图的终结态
    std::map<TokenType, int> finalState;

    // 返回对应状态的转移结果，如果返回-1表示转移失败
    int transferTo(std::map<std::pair<int, char>, int> &diagram, int state, char c);
    // 判断对应状态是否终结态
    bool isFinalState(int finalState, int state);

public:
    static Statechart& getStatechart()
    {
        static Statechart statechart;
        return statechart;
    }

    std::vector<Token> getTokens(std::string line);
};


#endif //PL_0_STATECHART_H
```

getTokens函数的实现实现是：循环遍历每一个状态图，判断是否有状态图能接收，如果成功接收，则说明找到一个token符合该类型；因此TokenType的设定顺序很重要，一定要保证关键字的状态图在标识符的状态图之前；标识符的状态图在数字状态图之前

```
std::vector<Token> Statechart::getTokens(std::string line)
{
    std::vector<Token> res;
    for (int i = 0; i < line.size(); ++ i)
    {
        bool bad = true;
        for (int j = 0; j < TokenType::tokenTypeCount; ++ j)
        {
            TokenType type = static_cast<TokenType>(j);
            std::string value = "";
            std::map<std::pair<int, char>, int> &diagram = this->diagram[type];
            int finalState = this->finalState[type];
            int state = 0;
```

3.2.3 SyntacticAnalysis模块

Instruction

Instruction类用来表示pcode代码，其中代码指令用枚举类InstructionType表示

```
enum InstructionType {
    illegal, lit, opr, lod, sto, cal, inc, jmp, jpc, sio
};

class Instruction {
public:
    InstructionType f;  // 代码指令
    int l;              //引用层与声明层的层次差
    int a;              //根据f的不同而不同

    Instruction() {};
};
```

Symbol

Symbol类用来表示符号表管理中涉及到的符号的类型，这里有三种类型：const（常量），var（变量），proc（子程序段）

```
enum SymbolType {
    CONST, VAR, PROC
};

class Symbol {
/*
 * For constants, store type, ident and val.
 * For variables, store type, ident, L and M.
 * For procedures, store type, ident, L and M.
 */
public:
    SymbolType type;    // 类型
    std::string ident;  // 标识符号
    std::string val;    // 值
    int level;          // 定义层
    int addr;           // 地址

    Symbol() {};
};
```

Parser

Parser类是最为关键的类，语法分析和语义分析都在这个类中进行，它的结果如下图所示

```
17
18      class Parser {
19      private:
20          int cx, cur_index;
21          Token cur_token;
22          std::vector<Symbol> symbolTable;
23          std::vector<Instruction> code;
24          std::vector<Token> tokens;
25
26          TokenType cur_type;
27          std::string cur_ident, cur_val;
28
29          void parseInit();
30          void getNextToken();
31          int position(Token, int, int *);
32          void enter(SymbolType, int *, int, int *);
33          void emit(InstructionType, int, int);
34
35          void program();
36          void block(int, int);
37          void constdeclaration(int, int *, int *);
38          void vardeclaration(int, int *, int *);
39          void statement(int, int *);
40          void condition(int, int *);
41          void expression(int, int *);
42          void term(int, int *);
43          void factor(int, int *);
44
45          void output();
46      public:
47          Parser() {};
48
49          void parse(std::vector<Token> tokens);
50      };
51
```

其具体内容的稍微详细的解析如下：

结构	说明
int cx;	pcode下一条语句的数组下标
int cur_index;	表示当前已经处理到的token的下表
std::vector symbolTable;	表示分析过程中的符号表
std::vector code;	表示分析过程中生成的pcode代码
std::vector tokens;	表示待分析的源程序的完整token
TokenType cur_type;	表示最近处理token的类型，便于后续维护符号表
std::string cur_ident;	表示最近处理token的标识符
std::string cur_val;	表示最近处理token的值
void parseInit();	解析过程的预处理
void getNextToken();	获取下一个token
int position(Token, int, int *);	获取制定token在符号表中的位置（可能不存在，不存在返回0）
void enter(SymbolType, int *, int, int *);	更新符号表
void emit(InstructionType, int, int);	生成pcode代码，并存储到code中
void program();	递归下降过程中解析整个源程序
void block(int, int);	递归下降过程中解析分程序
void constdeclaration(int, int *, int *);	递归下降过程中解析常量说明部分
void vardeclaration(int, int *, int *);	递归下降过程中解析变量说明部分
void statement(int, int *);	递归下降过程中解析语句的部分
void condition(int, int *);	递归下降过程中解析条件的部分
void expression(int, int *);	递归下降过程中解析表达式的部分
void term(int, int *);	递归下降过程中解析项的部分
void factor(int, int *);	递归下降过程中解析因子的部分
void output();	进行pcode输出，包括输出到文件中和输出到终端屏幕上
void parse(std::vector tokens);	暴露的接口，用于解析

parseInit函数：初始化cx、cur_index、symbolTable和code

```
void Parser::parseInit()
{
    this->cx = this->cur_index = 0;
    this->symbolTable.clear();
    this->code.clear();
    this->symbolTable.resize( new_size: 1000);
    this->code.resize( new_size: 1000);
}
```

getNextToken函数：获取下一个token并更新cur_type、cur_token以及cur_ident

```
void Parser::getNextToken()
{
    this->cur_token = this->tokens[cur_index ++];
    this->cur_type = cur_token.getType();
    if (this->cur_type == TokenType::identSym) this->cur_ident = this->cur_token.getValue();
    else if (this->cur_type == TokenType::numberSym) this->cur_val = this->cur_token.getValue();
}
```

position函数：在符号管理表中找到与当前层次最近的目标

```
int Parser::position(Token token, int lev, int *ptx)
{
    int s = *ptx;
    int res = 0, count = 0;
    int diff, preDiff;
    while (s)
    {
        if (this->symbolTable[s].ident == token.getValue())
        {
            if (symbolTable[s].level <= lev)
            {
                if (count) preDiff = diff;
                diff = lev - symbolTable[s].level;
                if (!count || diff < preDiff) res = s;
                ++ count;
            }
        }
        -- s;
    }
    return res;
}
```

enter函数：用于将信息保存进符号表中

```
45
46  void Parser::enter(SymbolType symbolType, int *pdx, int lev, int *ptx)
47  {
48      (*ptx) ++;
49      symbolTable[*ptx].ident = this->cur_ident;
50      symbolTable[*ptx].type = symbolType;
51      if (symbolType == SymbolType::CONST) this->symbolTable[*ptx].val = this->cur_val;
52      else if (symbolType == SymbolType::VAR)
53      {
54          symbolTable[*ptx].level = lev;
55          symbolTable[*ptx].addr = *pdx;
56          (*pdx) ++;
57      }
58      else symbolTable[*ptx].level = lev;
59  }
```

emit函数：用于生成pcode代码

```
60
61  void Parser::emit(InstructionType f, int l, int a)
62  {
63      this->code[this->cx].f = f;
64      this->code[this->cx].l = l;
65      this->code[this->cx].a = a;
66      ++ this->cx;
67  }
68
```

剩余的函数根据其产生式的定义书写相关的递归下降子程序即可，这里仅对较为复杂的block进行说明，其他类似不再进行赘述

初始化和跳转设置：

- `int dx = 4, tx0 = tx;` 初始化数据段地址dx和临时符号表索引tx0。
- `this->symbolTable[tx0].addr = this->cx;` 将当前代码索引（cx）存储在符号表的地址部分，用于后续回填。
- `emit(InstructionType::jmp, 0, 0);` 生成一个初始的跳转指令，跳转目的地稍后确定。

```
    void Parser::block(int lev, int tx)
    {
        int dx = 4, tx0 = tx;
        this->symbolTable[tx0].addr = this->cx;
        emit( f: InstructionType::jmp,  l: 0,  a: 0);
```

声明处理：

- 程序通过一个`do-while`循环处理三种声明：常量、变量和过程。
- 对于每种声明类型，使用`getNextToken()`读取下一个词法单元，并根据类型分别调用`constdeclaration`、`vardeclaration`或进行过程相关处理。

错误处理：

- 如果在期待分号（表示声明结束）的位置没有读到分号，将输出语法错误信息。

- 对于过程声明，还额外检查是否有标识符和正确的分号。

过程内的嵌套块处理：

- 当检测到过程声明时，通过递归调用`block`函数处理新的块级别（lev+1），这是实现过程嵌套的关键。

```
114
115     do {
116         if (this->cur_token.getType() == TokenType::constSym)
117         {
118             getNextToken();
119             constdeclaration(lev, pdx: &dx, ptx: &tx);
120             while (this->cur_token.getType() == TokenType::commaSym)
121             {
122                 getNextToken();
123                 constdeclaration(lev, pdx: &dx, ptx: &tx);
124             }
125             if (this->cur_token.getType() == TokenType::semicolonSym) getNextToken();
126             else Error::printErrors( type: ErrorType::SyntaxError, message: "comma missing.");
127         }
128         else if (this->cur_token.getType() == TokenType::varSym)
129         {
130             getNextToken();
131             vardeclaration(lev, pdx: &dx, ptx: &tx);
132             while (this->cur_token.getType() == TokenType::commaSym)
133             {
134                 getNextToken();
135                 vardeclaration(lev, pdx: &dx, ptx: &tx);
136             }
137             if (this->cur_token.getType() == TokenType::semicolonSym) getNextToken();
138             else Error::printErrors( type: ErrorType::SyntaxError, message: "comma missing.");
139         }
140         else if (this->cur_token.getType() == TokenType::procSym)
141         {
142             getNextToken();
143
144             if (this->cur_token.getType() == TokenType::identSym)
145             {
146                 enter( symbolType: SymbolType::PROC, pdx: &dx, lev, ptx: &tx);
147                 getNextToken();
148             }
149             else Error::printErrors( type: ErrorType::SyntaxError, message: "identifier missing.");
150             if (this->cur_token.getType() == TokenType::semicolonSym) getNextToken();
151             else Error::printErrors( type: ErrorType::SyntaxError, message: "comma missing.");
152
153             block( lev: lev + 1, tx);
154             if (this->cur_token.getType() == TokenType::semicolonSym) getNextToken();
155             else Error::printErrors( type: ErrorType::SyntaxError, message: "comma missing.");
156         }
157     } while(this->cur_token.getType() == TokenType::constSym || this->cur_token.getType() == TokenType::varSym
158         || this->cur_token.getType() == TokenType::procSym);
159
```

结束块处理：

- 循环结束后，使用`code[symbolTable[tx0].addr].l = this->cx;`回填之前生成的跳转指令，确保它跳转到正确的地址。
- `emit(InstructionType::inc, 0, dx);`生成增加数据栈指令，为局部变量分配空间。
- `statement(lev, &tx);`处理块内的语句。
- `emit(InstructionType::opr, 0, 0);`生成返回指令。

```
        code[symbolTable[tx0].addr].l = this->cx;
        symbolTable[tx0].addr = this->cx;
        emit( f: InstructionType::inc, l: 0, a: dx);
        statement(lev, ptx: &tx);
        emit( f: InstructionType::opr, l: 0, a: 0);
    }
```

四. 测试结果

测试用例1

```
var a, b;
procedure first;
    var a,b;
    procedure second;
        var a,b;
            begin
                a:=5;
                b:=6;
                write a;
                write b;
            end;
    begin
        a:=3;
        b:=4;
        write a;
        write b;
        call second;
    end;
begin
    a:=1;
    b:=2;
    write a;
    write b;
    call first;
    write a;
    write b;
end.
```

测试结果1

```
JMP 24 0
JMP 13 0
JMP 3 0
INC 0 6
LIT 0 5
STO 0 4
LIT 0 6
STO 0 5
LOD 0 4
SIO 0 1
LOD 0 5
SIO 0 1
OPR 0 0
INC 0 6
LIT 0 3
STO 0 4
LIT 0 4
STO 0 5
LOD 0 4
SIO 0 1
LOD 0 5
SIO 0 1
CAL 0 3
OPR 0 0
INC 0 6
LIT 0 1
```

```
STO 0 4
LIT 0 2
STO 0 5
LOD 0 4
SIO 0 1
LOD 0 5
SIO 0 1
CAL 0 13
LOD 0 4
SIO 0 1
LOD 0 5
SIO 0 1
OPR 0 0
```

测试用例2

```
const m = 7, n = 85;
var  i,x,y,z,q,r;
procedure mult;
    var a, b;
    begin
        a := x;  b := y; z := 0;
        while b > 0 do
        begin
            if odd x then z := z+a;
            a := 2*a;
            b := b/2;
        end
    end;

    begin
        x := m;
        y := n;
        call mult;
    end.
```

测试结果2

```
JMP 30 0
JMP 2 0
INC 0 6
LOD 1 5
STO 0 4
LOD 1 6
STO 0 5
LIT 0 0
STO 1 7
LOD 0 5
LIT 0 0
OPR 0 12
JPC 29 0
LOD 1 5
OPR 0 6
JPC 20 0
LOD 1 7
LOD 0 4
OPR 0 2
STO 1 7
LIT 0 2
LOD 0 4
OPR 0 4
```

```
STO 0 4
LOD 0 5
LIT 0 2
OPR 0 5
STO 0 5
JMP 0 9
OPR 0 0
INC 0 10
LIT 0 7
STO 0 5
LIT 0 85
STO 0 6
CAL 0 2
OPR 0 0
```

测试用例3

```
var f, n;
procedure fact;
    var ans1;
    begin
        ans1:=n;
        n:= n-1;
        if n < 0 then f := -1;
        if n = 0 then f := 1
        else call fact;
        f:=f*ans1;
        write f;
    end;
begin
    read n;
    call fact;
    write f;
end.
```

测试结果3

```
JMP 31 0
JMP 2 0
INC 0 5
LOD 1 5
STO 0 4
LOD 1 5
LIT 0 1
OPR 0 3
STO 1 5
LOD 1 5
LIT 0 0
OPR 0 10
JPC 16 0
LIT 0 1
OPR 0 1
STO 1 4
LOD 1 5
LIT 0 0
OPR 0 8
JPC 23 0
LIT 0 1
STO 1 4
JMP 24 0
CAL 1 2
```

```
LOD 1 4
LOD 0 4
OPR 0 4
STO 1 4
LOD 1 4
SIO 0 1
OPR 0 0
INC 0 6
SIO 0 2
STO 0 5
CAL 0 2
LOD 0 4
SIO 0 1
OPR 0 0
```