# 编译原理实验报告

<table>
<tr><td colspan="2">实验题目：PL/0 词法分析器、语法分析器、编译器</td><td>学号：202000120101</td></tr>
<tr><td>日期：2023.6.10</td><td>班级： 计科 20.2</td><td>姓名： 尹国泰</td></tr>
<tr><td colspan="3">Email：1018693208@qq.com</td></tr>
<tr><td colspan="3">实验目的：<br><br>实验一 PL/0 词法分析器<br>给定一个 PL/0 语言源程序，你需要将其从字符流转换为词语流。具体来说，你需要过滤源程序中的空白符(空格，tab，换行等)，识别关键字、标识符、数字以及运算符。<br><br>实验二 PL/0 语法分析器<br>利用前面实现的词法分析器，向语法分析器提供词语流。需要以合适的方式对词语流进行分析，并生成一颗语法树，或者宣告语法错误。<br><br>实验三 PL/0 编译器以及目标代码解释器<br>将用语法制导翻译的方式，完成给定语法的语义分析以及目标代码生成。你需要修改你的语法生成器，使其在生成语法树的同时，进行语义分析以及目标代码生成。<br><br>完成了目标代码生成部分后，你已经可以将一个 PL/0 语言程序生成为目标代码了。但是，假想计算机的机器指令并不能在现代计算机上直接运行。你还需要为此完成一个解释器来执行对应的机器指令。</td></tr>
<tr><td colspan="3">实验环境介绍：<br>硬件环境：<br>个人计算机<br>OJ 评测机<br>软件环境：<br>Windows10 操作系统<br>VScode 编辑器<br>Mingw64</td></tr>
</table>

解决问题的主要思路：

**实验一 PL/0 词法分析器**

首先我定义了如下结构体用来存储 PL/0 中的一个词语，在将字符流转化为词语流的过程中将每个词语按该结构格式存储

```
struct pl0word{
    string name; // 单词本身
    string sym; // 单词类型
    int line; // 行号
    int col; // 列号
};
```

然后定义如下 LexicalAnalyse 类用来完成词法分析,其中该类中定义了 PL/0 关键字、运算符、分隔符等内容。

同时定义 4 个内部函数 boolisLetter(char ch); bool isDigit(char ch); bool isDelimiter(char ch); bool isOperator(char ch);用来判断一个字符是否为字母、数字、分隔符、运算符

定义内部函数 stringstream readfile(string filename)用来从文件或者标准输入（文件名定义为“stdin”时）读取字符流，返回一个 stringstream

外部调用函数：

bool analyse(string filename); 该函数实现了词法分析的主要过程，可以被其他程序调用。从 filename（文件名定义为“stdin”时为标准输入）中获取字符流，并转化为单词流，每个单词为 pl0word 结构类型

void print(); 依次打印单词流，编写该函数主要用于 OJ 评测

各函数的详细实现见实验步骤部分。

```
class LexicalAnalyse{
public:
    vector<pl0word> words;

    LexicalAnalyse(){};
    ~LexicalAnalyse(){};

    void print();
    bool analyse(string filename);

private:
    //PL0 关键字
    int keylen = 13;
    string keyword[13] =
{ "CONST","VAR","PROCEDURE","BEGIN","END","ODD","IF"
,"THEN","CALL","WHILE","DO","READ","WRITE" };
    string keysym[13] =
{ "CONSTSYM","VARSYM","PROCSYM","BEGINSYM","ENDSYM",
```

```
"ODDSYM","IFSYM","THENSYM"
,"CALLSYM","WHILESYM","DOSYM","READSYM","WRITESYM" };
    //PL0 运算符
    int oplen = 11;
    string op[11] =  {"=",":=", "+", "-", "*", "/", "#", "<", "<=", ">",
">="};
    string opsym[11] =  {"EQL", "BECOMES", "PLUS", "MINUS", "TIMES",
"SLASH",
"NEQ", "LSS", "LEQ", "GTR", "GEQ"};
    //PL0 分隔符
    int dellen = 5;
    string delimiter[5] = { ";", ",", ".", "(", ")" };
    string delsym[5] = {"SEMICOLON", "COMMA", "PERIOD", "LPAREN",
"RPAREN" };
    //标识符 IDENTIFIER
    //数 NUMBER

    bool isLetter(char ch); //判断是否为字母
    bool isDigit(char ch);  //判断是否为数字
    bool isDelimiter(char ch); //判断是否为分隔符
    bool isOperator(char ch); //判断是否为运算符
    stringstream readfile(string filename); //读取文件
};
```

**实验二 PL/0 语法分析器**

目的是按以下 PL/0 文法，以合适的方式对词语流进行分析，并生成一颗语法树，或者宣告语法错误。

```
<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>}
<变量说明部分> → VAR<标识符>{ ,<标识符>};
<标识符> → <字母>{<字母>|<数字>}
<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>|<空语句>
<赋值语句> → <标识符>:=<表达式>
<复合语句> → BEGIN<语句>{ ;<语句>} END
<条件> → <表达式><关系运算符><表达式>|ODD<表达式>
<表达式> → [+|-]<项>{<加减运算符><项>}
<项> → <因子>{<乘除运算符><因子>}
```

<因子> → <标识符>|<无符号整数>|(<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<标识符>
<当型循环语句> → WHILE<条件>DO<语句>
<读语句> → READ(<标识符>{ ,<标识符>})
<写语句> → WRITE(<标识符>{,<标识符>})
<字母> → A|B|C…X|Y|Z
<数字> → 0|1|2…7|8|9
<空语句> → epsilon

并且有如下对各节点描述的规定

上下文无关文法中的节点 对应的替换词语
程序 PROGRAM
分程序 SUBPROG
常量说明部分 CONSTANTDECLARE
常量定义 CONSTANTDEFINE
无符号整数 < 这是一个叶子节点，用其本身替代 >
变量说明部分 VARIABLEDECLARE
标识符 < 这是一个叶子节点，用其本身替代 >
过程说明部分 PROCEDUREDECLARE
过程首部 PROCEDUREHEAD
语句 SENTENCE
赋值语句 ASSIGNMENT
复合语句 COMBINED
条件 CONDITION
表达式 EXPRESSION
项 ITEM
因子 FACTOR
加减运算符 < 这是一个叶子节点，用其本身替代 >
乘除运算符 < 这是一个叶子节点，用其本身替代 >
关系运算符 < 这是一个叶子节点，用其本身替代 >
条件语句 IFSENTENCE
过程调用语句 CALLSENTENCE
当型循环语句 WHILESENTENCE
读语句 READSENTENCE
写语句 WRITESENTENCE
空语句 EMPTY

如果该节点是(，即左括号，该节点应当输出为 LP
如果该节点是)，即右括号，该节点应当输出为 RP
如果该节点是,，即逗号，该节点应当输出为 COMMA
如果该节点是其他的叶子节点（对应词法分析器中的某一个词语），那么其名称为其本

身
如果该节点不是一个叶子节点，则遵循下表进行替换输出

为建立语法树，我首先创建如下语法树节点。element 为按前面的规则对节点的命名，child 为所有子节点指针

```
struct treeNode
{
    string element;
    vector<treeNode*> child;

    treeNode(string _element)
    {
        element = _element;
    }
    ~treeNode()
    {
        for (int i = 0; i < child.size(); i++)
        {
            if(child[i] != NULL) delete child[i];
        }
    }
};
```

然后我定义了 SyntaxAnalyse 类用于语法分析。

Public:

void analyse(vector<pl0word> words); 调用接口，words 为 LexicalAnalyse 产生的单词流，调用后根据单词流 words 生成以 root 为根的语法树

void printTree(); 按树的括号表示形式，打印语法树，主要用于 OJ 测试

Private:

vector<pl0word> words; 记录了 LexicalAnalyse 产生的单词流，是要进行分析的主体

vector<pl0word>::iterator it; 当前分析到的单词的迭代器

treeNode *root; 语法树的根

int procedure_depth; 过程嵌套层数，记录了迭代器 it 所在位置的过程深度

void _printTree(treeNode *t); 打印语法树

剩余的函数：根据 PL/0 的上下文无关文法，逐一进行分析，生成语法树节点

代码具体实现见实验步骤部分

```
class SyntaxAnalyse{
public:
    SyntaxAnalyse();
```

```
    ~SyntaxAnalyse();
    void analyse(vector<pl0word> words);
    void printTree();

private:
    vector<pl0word> words;
    vector<pl0word>::iterator it;
    treeNode *root;
    int procedure_depth; //嵌套层数

    void _printTree(treeNode *t);
    void analyseProgram(treeNode *t); //程序
    void analyseSubprog(treeNode *t); //分程序
    void analyseConstDeclare(treeNode *t); //常量说明部分
    void analyseConstDefine(treeNode *t); //常量定义
    void analyseUnsignedInteger(treeNode *t); //无符号整数
    void analyseVariableDeclare(treeNode *t);  //变量说明部分
    void analyseIdentifier(treeNode *t); //标识符
    void analyseProcedureDeclare(treeNode *t); //过程说明部分
    void analyseProcedureHead(treeNode *t); //过程首部
    void analyseSentence(treeNode *t); //语句
    void analyseAssignment(treeNode *t); //赋值语句
    void analyseCombined(treeNode *t); //复合语句
    void analyseCondition(treeNode *t); //条件
    void analyseExpression(treeNode *t); //表达式
    void analyseItem(treeNode *t); //项
    void analyseFactor(treeNode *t); //因子
    void analyseAddSubOperator(treeNode *t); //加减运算符
    void analyseMulDivOperator(treeNode *t); //乘除运算符
    void analyseRelationOperator(treeNode *t); //关系运算符
    void analyseIfSentence(treeNode *t); //条件语句
    void analyseCallSentence(treeNode *t); //过程调用语句
    void analyseWhileSentence(treeNode *t); //当型循环语句
    void analyseReadSentence(treeNode *t); //读语句
    void analyseWriteSentence(treeNode *t); //写语句
    void analyseEmpty(treeNode *t); //空语句
};
```

**实验三 PL/0 编译器以及目标代码解释器**

**PL/0 编译器部分：**将用语法制导翻译的方式，完成给定语法的语义分析以及目标代码生成。你需要修改你的语法生成器，使其在生成语法树的同时，进行语义分析以及目标代码生成。

我首先制作名字表，填写所在层次、属性，并分配相对地址。

```
// NAME KIND    PARAMETER1  PARAMETER2
// a    CONSTANT    VAL:35  --
// b    CONSTANT    VAL:49  --
// c    VARIABLE    LEVEL: LEV  ADR: DX
// d    VARIABLE    LEVEL: LEV  ADR: DX+1
// e    VARIABLE    LEVEL: LEV  ADR: DX+2
// p    PROCEDURE   LEVEL: LEV  ADR: <UNKNOWN>
// g    VARIABLE    LEVEL: LEV+1    ADR: DX
// 其中，LEVEL 给出的是层次，DX 是每一层局部量的相对地址。
// 对于过程名的地址，需要等待目标代码生成后再填写。
// 考虑到需要存储调用信息、返回信息等维护函数运行的数据，在本实验中，DX 取 3.
struct NameTable{
    string name;
    string kind;
    int parameter1; // VAL/LEVEL
    int parameter2; // ADR
};
```

然后创建 TargetCode 用来存储每条目标代码

```
// 生成的目标代码是一种假想栈式计算机的汇编语言，其格式如下：

// f l a
// 其中 f 为功能码，l 代表层次差，a 代表位移量。

// f 功能码
// LIT: l 域无效，将 a 放到栈顶
// LOD: 将于当前层层差为 l 的层，变量相对位置为 a 的变量复制到栈顶
// STO: 将栈顶内容复制到于当前层层差为 l 的层，变量相对位置为 a 的变量
// CAL: 调用过程。l 标明层差，a 表明目标程序地址
// INT: l 域无效，在栈顶分配 a 个空间
// JMP: l 域无效，无条件跳转到地址 a 执行
// JPC: l 域无效，若栈顶对应的布尔值为假（即 0）则跳转到地址 a 处执行，否则顺序执行
// OPR: l 域无效，对栈顶和栈次顶执行运算，结果存放在次顶，a=0 时为调用返回
struct TargetCode{
    string f;
    int l;
    int a;
};
```

然后就是对 SyntaxAnalyse 类进行修改，在语法分析过程中，采用语法制导翻译的方式，完成给定语法的语义分析以及目标代码生成

在前面实现的 SyntaxAnalyse 类中，我又添加了以下部分

```
public:
    void printCode(); 依次打印目标代码
    void printNameTable(); 依次打印名字表
private:
    string OPR_Table[]  OPR 指令中，OPR_Table[]中为为实际执行的功能，主要用于生成指令时，对 OPR 指令中的 a 赋值
    vector<NameTable> table; 名字表
    vector<TargetCode> code; 生成的目标代码
    void addTable(string name, string kind, int parameter1, int parameter2); 向名字表中添加一个名字
    void addCode(string f, int l, int a);添加一行目标代码

    int getIndexbyName(int level, string name); 根据 name，获得过程深度比 level 低的名字在名字表中的索引
    int getIndexbyOPR(string opr); 获得 OPR_Table[opr]，主要用于对 OPR 指令中的 a 赋值
```

具体对 SyntaxAnalyse 类中识别语法单元各函数的修改，详见实验步骤

```
class SyntaxAnalyse{
public:
    void printCode();
    void printNameTable();
private:
//------------------目标代码生成------------------start
    int OPR_cnt = 14;
    string OPR_Table[20] = {"RETURN","ODD","READ","WRITE", "+", "-", "*", "/","=", "#", "<", "<=", ">", ">="};
    vector<NameTable> table;
    vector<TargetCode> code;
    void addTable(string name, string kind, int parameter1, int parameter2);
    void addCode(string f, int l, int a);

    int getIndexbyName(int level, string name);
    int getIndexbyOPR(string opr);
//-----------------------------------------------end
};
```

**目标代码解释器：**完成了目标代码生成部分后，你已经可以将一个 PL/0 语言程序生成为目标代码了。但是，假想计算机的机器指令并不能在现代计算机上直接运行。你还需要为此完成一个解释器来执行对应的机器指令。

我实现了一个 Interpreter 类用于目标代码的解释，该类中实现 readCode(string

filename)读取目标代码，然后调用 interpret()解释目标代码
此外在该类内部，模拟下面四个寄存器

IR，指令寄存器，存放正在执行的指令。
IP，指令地址寄存器，存放下一条指令的地址。
SP，栈顶寄存器，指向运行栈的顶端。
BP，基址寄存器，指向当前过程调用所分配的空间在栈中的起始地址。

创建了 ir，ip，sp，bp
创建 stack 为程序栈进行运算
创建 sp_stack 记录每个过程的栈顶指针
创建 ret_addr = 0，dynamic_link = 1，static_link = 2 为栈中返回地址，动态链，静态链的相对地址。比如每个过程的 bp+ret_addr 就是该过程的返回地址在栈中的地址

详细的 interpret 函数实现见实验步骤部分

```
class Interpreter
{
    public:
        Interpreter();
        ~Interpreter();
        void interpret(); //主要函数，完成对目标代码的解释
        void readCode(string filename); //读取目标代码
    private:
        vector<TargetCode> code; //目标代码
        const static int ret_addr = 0, dynamic_link = 1, static_link = 2;
        //栈中返回地址，动态链，静态链的相对位置
        TargetCode ir; //当前指令
        int ip = 0, sp = 0, bp = 0; //指令指针，栈顶指针，基址指针
        int stack[100000] = { 0 }; //栈
        int sp_stack[1000]; //栈顶指针栈，用于存放每个过程的栈顶指针
        int sp_top = 0; //栈顶指针栈的栈顶指针
};
```

实验步骤：

**一、 PL/0 词法分析器**

首先完成读取源程序的函数 readfile，当 filename 为“stdin”时从标准输入读，注意要将所有字母转化为大写字母，统一后续处理的判断

```
stringstream LexicalAnalysis::readfile(string filename){
    ifstream fin;
    stringstream buffer;
    if(filename == "stdin"){
        //从标准输入所有字符后赋给 buffer
        char ch;
        while(cin.get(ch)){
            //转换成大写字母
            if(ch >= 'a' && ch <= 'z'){
                ch -= 32;
            }
            buffer << ch;
        }
    }else{
        fin.open(filename);
        //转换成大写字母
        char ch;
        while(fin.get(ch)){
            if(ch >= 'a' && ch <= 'z'){
                ch -= 32;
            }
            buffer << ch;
        }
        fin.close();
    }
    return buffer;
}
```

然后是对字符类型的判断函数

```
bool LexicalAnalysis::isLetter(char ch){
    if ((ch >= 'a' && ch <= 'z') || (ch >= 'A' && ch <= 'Z'))
    {
        return true;
    }
    else
    {
        return false;
    }
}
bool LexicalAnalysis::isDigit(char ch){
```

```
    if (ch >= '0' && ch <= '9')
    {
        return true;
    }
    else
    {
        return false;
    }
}
bool LexicalAnalysis::isDelimiter(char ch){
    for (int i = 0; i < dellen; i++)
    {
        if (ch == delimiter[i][0])
        {
            return true;
        }
    }
    return false;
}
bool LexicalAnalysis::isOperator(char ch){
    for (int i = 0; i < oplen; i++)
    {
        if (ch == op[i][0])
        {
            return true;
        }
    }
    return false;
}
```

最后是对词法分析过程的实现，将字符流转化为单词流，当出现词法错误时返回 0，否则返回 1

```
bool LexicalAnalyse::analyse(string filename){

    int i;
    char ch;
    stringstream buffer = readfile(filename);
    while (!buffer.eof())
    {
        pl0word wd;
        buffer.get(ch);
        if(buffer.eof()){
            break;
```

```
        }
        if (ch == ' ' || ch == '\t' || ch == '\n' || ch == '\r')
        {
            continue;
        }
        else if (isLetter(ch))
        {
            while (!buffer.eof() && (isLetter(ch) || isDigit(ch)))
            {
                wd.name += ch;
                buffer.get(ch);
            }
            if( !(isLetter(ch) || isDigit(ch)) ) buffer.putback(ch);
            if(wd.name.length() > 10){
                return false;
            }
            for (i = 0; i < keylen; i++)
            {
                if (wd.name == keyword[i])
                {
                    break;
                }
            }
            if (i < keylen)
                wd.sym = keysym[i];
            else
                wd.sym = "IDENTIFIER";
        }
        else if (isDigit(ch))
        {
            while (!buffer.eof() && isDigit(ch))
            {
                wd.name += ch;
                buffer.get(ch);
            }
            // 数字后面不能跟字母
            if(isLetter(ch)){
                return false;
            }
            if(!isDigit(ch)) buffer.putback(ch);
            //去除前导 0
            while(wd.name[0] == '0'&& wd.name.length() > 1){
                wd.name.erase(0,1);
            }
```

```
            wd.sym = "NUMBER";
        }
        else if (isOperator(ch))
        {
            wd.name = ch;
            buffer.get(ch);
            if (ch == '=')
            {
                wd.name += ch;
            }
            else
            {
                buffer.putback(ch);
            }
            for (i = 0; i < oplen; i++)
            {
                if (wd.name == op[i])
                {
                    wd.sym = opsym[i];
                    break;
                }
            }
            if (i == oplen)
                return false;
        }
        else if (isDelimiter(ch))
        {
            wd.name = ch;
            for (i = 0; i < dellen; i++)
            {
                if (wd.name == delimiter[i])
                {
                    wd.sym = delsym[i];
                    break;
                }
            }
        }
        else
            return false;
        words.push_back(wd);
    }
    return true;
}
```

## 二、PL/0 语法分析器

SyntaxAnalyse::analyse 为程序调用总入口

```
void SyntaxAnalyse::analyse(vector<pl0word> _words)
{
    for(int i = 0; i < _words.size(); i++)
    {
        words.push_back(_words[i]);
    }
    it = words.begin();
    root = new treeNode("PROGRAM");
    analyseProgram(root);
}
```

SyntaxAnalyse::_printTree 用来打印生成的语法树的括号表示形式

```
void SyntaxAnalyse::_printTree(treeNode *t)
{
    if(t == NULL)
        return;
    if (t->child.size() == 0)
    {
        cout << t->element;
        return;
    }
    cout << t->element;
    cout << "(";
    for (int i = 0; i < t->child.size(); i++)
    {
        _printTree(t->child[i]);
        if(i != t->child.size()-1) cout<<",";
    }
    cout << ")";
}
```

接下来是根据 PL/0 上下文无关文法逐条语法分析的函数

〈程序〉→〈分程序〉.

```
//程序
void SyntaxAnalyse::analyseProgram(treeNode *t){
    if(words.size() == 0)
        throw "Syntax Error";

    treeNode *tSubprog = new treeNode("SUBPROG");
    analyseSubprog(tSubprog);
    if (it->name == ".")
```

```
    {
        t->child.push_back(tSubprog);
        t->child.push_back(new treeNode("."));
    }
    else
        throw "Syntax Error";

    if(it != words.end() - 1) //还有未分析的单词
        throw "Syntax Error";
}
```

<分程序>→ [<常量说明部分>][<变量说明部分>][<过程说明部分>]<语句>

```
//分程序
void SyntaxAnalyse::analyseSubprog(treeNode *t){

    if (it->name == "CONST")
    {
        treeNode *tConstDeclare = new treeNode("CONSTANTDECLARE");
        analyseConstDeclare(tConstDeclare);
        t->child.push_back(tConstDeclare);
    }
    if (it->name == "VAR")
    {
        treeNode *tVariableDeclare = new treeNode("VARIABLEDECLARE");
        analyseVariableDeclare(tVariableDeclare);
        t->child.push_back(tVariableDeclare);
    }
    if (it->name == "PROCEDURE")
    {
        treeNode *tProcedureDeclare = new treeNode("PROCEDUREDECLARE");
        analyseProcedureDeclare(tProcedureDeclare);
        t->child.push_back(tProcedureDeclare);
    }
    treeNode *tSentence = new treeNode("SENTENCE");
    analyseSentence(tSentence);
    t->child.push_back(tSentence);
}
```

<常量说明部分> → CONST<常量定义>{ ,<常量定义>};

```
//常量说明部分
void SyntaxAnalyse::analyseConstDeclare(treeNode *t){
    if (it->name == "CONST")
    {
        it++;
```

```
        treeNode *tConstDefine = new treeNode("CONSTANTDEFINE");
        analyseConstDefine(tConstDefine);
        t->child.push_back(new treeNode("CONST"));
        t->child.push_back(tConstDefine);
        while (it->name == ",")
        {
            it++;
            tConstDefine = new treeNode("CONSTANTDEFINE");
            analyseConstDefine(tConstDefine);
            t->child.push_back(new treeNode("COMMA"));
            t->child.push_back(tConstDefine);
        }
        if (it->name == ";")
        {
            it++;
            t->child.push_back(new treeNode(";"));
        }
        else
            throw "Syntax Error";
    }
    else
        throw "Syntax Error";
}
```

<常量定义> → <标识符>=<无符号整数>

```
//常量定义
void SyntaxAnalyse::analyseConstDefine(treeNode *t){
    treeNode *tIdentifier = new treeNode("IDENTIFIER?");
    analyseIdentifier(tIdentifier);
    t->child.push_back(tIdentifier);
    if (it->name == "=")
    {
        it++;
        treeNode *tUnsignedInteger = new treeNode("UNSIGNEDINTEGER?");
        analyseUnsignedInteger(tUnsignedInteger);
        t->child.push_back(new treeNode("="));
        t->child.push_back(tUnsignedInteger);
    }
    else
        throw "Syntax Error";
}
```

<无符号整数> → <数字>{<数字>}

```
//无符号整数
void SyntaxAnalyse::analyseUnsignedInteger(treeNode *t){
    if (it->sym == "NUMBER")
```

```
    {
        t->element = it->name;
        it++;
    }
    else
        throw "Syntax Error";
}
```

<变量说明部分> ➞ VAR<标识符>{ ,<标识符>};

```
//变量说明部分
void SyntaxAnalyse::analyseVariableDeclare(treeNode *t){
    if (it->name == "VAR")
    {
        it++;
        treeNode *tIdentifier = new treeNode("IDENTIFIER?");
        analyseIdentifier(tIdentifier);
        t->child.push_back(new treeNode("VAR"));
        t->child.push_back(tIdentifier);
        while (it->name == ",")
        {
            it++;
            tIdentifier = new treeNode("IDENTIFIER?");
            analyseIdentifier(tIdentifier);
            t->child.push_back(new treeNode("COMMA"));
            t->child.push_back(tIdentifier);
        }
        if (it->name == ";")
        {
            it++;
            t->child.push_back(new treeNode(";"));
        }
        else
            throw "Syntax Error";
    }
    else
        throw "Syntax Error";
}
```

<标识符> ➞ <字母>{<字母>|<数字>}

```
//标识符
void SyntaxAnalyse::analyseIdentifier(treeNode *t){
    if (it->sym == "IDENTIFIER")
    {
        t->element = it->name;
```

```
        it++;
    }
    else
        throw "Syntax Error";
}
```

<过程说明部分> → <过程首部><分程序>;{<过程说明部分>}

```
//过程说明部分
void SyntaxAnalyse::analyseProcedureDeclare(treeNode *t){
    treeNode *tProcedureHead = new treeNode("PROCEDUREHEAD");
    analyseProcedureHead(tProcedureHead);
    t->child.push_back(tProcedureHead);
    treeNode *tSubprog = new treeNode("SUBPROG");

    procedure_depth++;
    if(procedure_depth > 3) //过程嵌套深度不能超过 3
        throw "Syntax Error";
    analyseSubprog(tSubprog);
    procedure_depth--;

    t->child.push_back(tSubprog);
    if(it->name == ";")
    {
        it++;
        t->child.push_back(new treeNode(";"));
    }
    else
        throw "Syntax Error";

    if(it->name == "PROCEDURE")
    {
        treeNode *tProcedureDeclare = new treeNode("PROCEDUREDECLARE");
        analyseProcedureDeclare(tProcedureDeclare);
        t->child.push_back(tProcedureDeclare);
    }
}
```

<过程首部> → PROCEDURE <标识符>;

```
//过程首部
void SyntaxAnalyse::analyseProcedureHead(treeNode *t){
    if (it->name == "PROCEDURE")
    {
        it++;
        treeNode *tIdentifier = new treeNode("IDENTIFIER?");
        analyseIdentifier(tIdentifier);
        t->child.push_back(new treeNode("PROCEDURE"));
```

```
        t->child.push_back(tIdentifier);
        if (it->name == ";")
        {
            it++;
            t->child.push_back(new treeNode(";"));
        }
        else
            throw "Syntax Error";
    }
    else
        throw "Syntax Error";
}
```

<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>|<空语句>

```
//语句
void SyntaxAnalyse::analyseSentence(treeNode *t){
    if (it->sym == "IDENTIFIER")
    {
        treeNode *tAssignment = new treeNode("ASSIGNMENT");
        analyseAssignment(tAssignment);
        t->child.push_back(tAssignment);
    }
    else if (it->name == "IF")
    {
        treeNode *tIfSentence = new treeNode("IFSENTENCE");
        analyseIfSentence(tIfSentence);
        t->child.push_back(tIfSentence);
    }
    else if (it->name == "WHILE")
    {
        treeNode *tWhileSentence = new treeNode("WHILESENTENCE");
        analyseWhileSentence(tWhileSentence);
        t->child.push_back(tWhileSentence);
    }
    else if (it->name == "CALL")
    {
        treeNode *tCallSentence = new treeNode("CALLSENTENCE");
        analyseCallSentence(tCallSentence);
        t->child.push_back(tCallSentence);
    }
    else if (it->name == "READ")
    {
```

```
        treeNode *tReadSentence = new treeNode("READSENTENCE");
        analyseReadSentence(tReadSentence);
        t->child.push_back(tReadSentence);
    }
    else if (it->name == "WRITE")
    {
        treeNode *tWriteSentence = new treeNode("WRITESENTENCE");
        analyseWriteSentence(tWriteSentence);
        t->child.push_back(tWriteSentence);
    }
    else if (it->name == "BEGIN")
    {
        treeNode *tCombined = new treeNode("COMBINED");
        analyseCombined(tCombined);
        t->child.push_back(tCombined);
    }
    else if (it->name == "END" || it->name == "." || it->name == ";"){
        treeNode *tEmpty = new treeNode("EMPTY");
        analyseEmpty(tEmpty);
        t->child.push_back(tEmpty);
    }else
        throw "Syntax Error";
}
```

〈赋值语句〉➡〈标识符〉:=〈表达式〉

```
//赋值语句
void SyntaxAnalyse::analyseAssignment(treeNode *t){
    if (it->sym == "IDENTIFIER")
    {
        t->child.push_back(new treeNode(it->name));
        it++;
        if (it->name == ":=")
        {
            t->child.push_back(new treeNode(":="));
            it++;
            treeNode *tExpression = new treeNode("EXPRESSION");
            analyseExpression(tExpression);
            t->child.push_back(tExpression);
        }
        else
            throw "Syntax Error";
    }
    else
```

```
        throw "Syntax Error";
}
```

<复合语句> → BEGIN<语句>{ ;<语句>} END

```
//复合语句
void SyntaxAnalyse::analyseCombined(treeNode *t){
    if(it->name == "BEGIN"){
        it++;
        t->child.push_back(new treeNode("BEGIN"));
        treeNode *tSentence = new treeNode("SENTENCE");
        analyseSentence(tSentence);
        t->child.push_back(tSentence);
        while(it->name == ";"){
            t->child.push_back(new treeNode(";"));
            it++;
            tSentence = new treeNode("SENTENCE");
            analyseSentence(tSentence);
            t->child.push_back(tSentence);
        }
        if(it->name == "END"){
            it++;
            t->child.push_back(new treeNode("END"));
        }
        else
            throw "Syntax Error";
    }
    else
        throw "Syntax Error";
}
```

<条件> → <表达式><关系运算符><表达式>|ODD<表达式>

```
//条件
void SyntaxAnalyse::analyseCondition(treeNode *t){
    if (it->name == "ODD")
    {
        it++;
        treeNode *tExpression = new treeNode("EXPRESSION");
        analyseExpression(tExpression);
        t->child.push_back(new treeNode("ODD"));
        t->child.push_back(tExpression);
    }
    else
    {
```

```
        treeNode *tExpression1 = new treeNode("EXPRESSION");
        analyseExpression(tExpression1);
        t->child.push_back(tExpression1);
        treeNode *tRelationOperator = new treeNode("RELATIONOPERATOR?");
        analyseRelationOperator(tRelationOperator);
        t->child.push_back(tRelationOperator);
        treeNode *tExpression2 = new treeNode("EXPRESSION");
        analyseExpression(tExpression2);
        t->child.push_back(tExpression2);
    }
}
```

<表达式> → [+|-]<项>{<加减运算符><项>}

```
//表达式
void SyntaxAnalyse::analyseExpression(treeNode *t){
    if (it->name == "+" || it->name == "-")
    {
        t->child.push_back(new treeNode(it->name));
        it++;
    }
    treeNode *tItem = new treeNode("ITEM");
    analyseItem(tItem);
    t->child.push_back(tItem);
    while (it->name == "+" || it->name == "-")
    {
        treeNode *tAddSubOperator = new treeNode("ADD_SUB_OPERATOR?");
        analyseAddSubOperator(tAddSubOperator);
        t->child.push_back(tAddSubOperator);

        tItem = new treeNode("ITEM");
        analyseItem(tItem);
        t->child.push_back(tItem);
    }
}
```

<项> → <因子>{<乘除运算符><因子>}

```
//项
void SyntaxAnalyse::analyseItem(treeNode *t){
    treeNode *tFactor = new treeNode("FACTOR");
    analyseFactor(tFactor);
    t->child.push_back(tFactor);
    while (it->name == "*" || it->name == "/")
```

```
    {
        treeNode *tMulDivOperator = new treeNode("MUL_DIV_OPERATOR?");
        analyseMulDivOperator(tMulDivOperator);
        t->child.push_back(tMulDivOperator);

        tFactor = new treeNode("FACTOR");
        analyseFactor(tFactor);
        t->child.push_back(tFactor);
    }
}
```

〈因子〉 → 〈标识符〉|〈无符号整数〉|(〈表达式〉)

```
//因子
void SyntaxAnalyse::analyseFactor(treeNode *t){
    if (it->sym == "IDENTIFIER")
    {
        t->child.push_back(new treeNode(it->name));
        it++;
    }
    else if (it->sym == "NUMBER")
    {
        t->child.push_back(new treeNode(it->name));
        it++;
    }
    else if (it->name == "(")
    {
        it++;
        t->child.push_back(new treeNode("LP"));
        treeNode *tExpression = new treeNode("EXPRESSION");
        analyseExpression(tExpression);
        t->child.push_back(tExpression);
        if (it->name == ")")
        {
            it++;
            t->child.push_back(new treeNode("RP"));
        }
        else
            throw "Syntax Error";
    }
    else
        throw "Syntax Error";
}
```

〈加减运算符〉 → +|−

```
//加减运算符
void SyntaxAnalyse::analyseAddSubOperator(treeNode *t){
    if (it->name == "+" || it->name == "-")
    {
        t->element = it->name;
        it++;
    }
    else
        throw "Syntax Error";
}
```

<乘除运算符> → *|/

```
//乘除运算符
void SyntaxAnalyse::analyseMulDivOperator(treeNode *t){
    if (it->name == "*" || it->name == "/")
    {
        t->element = it->name;
        it++;
    }
    else
        throw "Syntax Error";
}
```

<关系运算符> → =|#|<|<=|>|>=

```
//关系运算符
void SyntaxAnalyse::analyseRelationOperator(treeNode *t){
    if (it->name == "=" || it->name == "#" || it->name == "<" || it->name
== "<=" || it->name == ">" || it->name == ">=")
    {
        t->element = it->name;
        it++;
    }
    else
        throw "Syntax Error";
}
```

<条件语句> → IF<条件>THEN<语句>

```
//条件语句
void SyntaxAnalyse::analyseIfSentence(treeNode *t){
    if (it->name == "IF")
    {
```

```
        it++;
        treeNode *tCondition = new treeNode("CONDITION");
        analyseCondition(tCondition);
        t->child.push_back(new treeNode("IF"));
        t->child.push_back(tCondition);
        if (it->name == "THEN")
        {
            it++;
            treeNode *tSentence = new treeNode("SENTENCE");
            analyseSentence(tSentence);
            t->child.push_back(new treeNode("THEN"));
            t->child.push_back(tSentence);
        }
        else
            throw "Syntax Error";
    }
    else
        throw "Syntax Error";
}
```

<过程调用语句> → CALL<标识符>

```
//过程调用语句
void SyntaxAnalyse::analyseCallSentence(treeNode *t){
    if (it->name == "CALL")
    {
        it++;
        treeNode *tIdentifier= new treeNode("IDENTIFIER?");
        analyseIdentifier(tIdentifier);
        t->child.push_back(new treeNode("CALL"));
        t->child.push_back(tIdentifier);
    }
    else
        throw "Syntax Error";
}
```

<当型循环语句> → WHILE<条件>DO<语句>

```
/当型循环语句
void SyntaxAnalyse::analyseWhileSentence(treeNode *t){
    if (it->name == "WHILE")
    {
        it++;
        treeNode *tCondition = new treeNode("CONDITION");
        analyseCondition(tCondition);
        t->child.push_back(new treeNode("WHILE"));
```

```
        t->child.push_back(tCondition);
        if (it->name == "DO")
        {
            it++;
            treeNode *tSentence = new treeNode("SENTENCE");
            analyseSentence(tSentence);
            t->child.push_back(new treeNode("DO"));
            t->child.push_back(tSentence);
        }
        else
            throw "Syntax Error";
    }
    else
        throw "Syntax Error";
}
```

<读语句> → READ(<标识符>{ ,<标识符>})

```
//读语句
void SyntaxAnalyse::analyseReadSentence(treeNode *t){
    if (it->name == "READ")
    {
        it++;
        t->child.push_back(new treeNode("READ"));
        if (it->name == "(")
        {
            it++;
            t->child.push_back(new treeNode("LP"));

            treeNode *tIdentifier = new treeNode("IDENTIFIER?");
            analyseIdentifier(tIdentifier);
            t->child.push_back(tIdentifier);
            while (it->name == ",")
            {
                it++;
                tIdentifier = new treeNode("IDENTIFIER?");
                analyseIdentifier(tIdentifier);
                t->child.push_back(new treeNode("COMMA"));
                t->child.push_back(tIdentifier);
            }

            if (it->name == ")")
            {
                it++;
                t->child.push_back(new treeNode("RP"));
```

```
            }
            else
                throw "Syntax Error";
        }
        else
            throw "Syntax Error";
    }
    else
        throw "Syntax Error";
}
```

<写语句> → WRITE(<标识符>{,<标识符>})

```
//写语句
void SyntaxAnalyse::analyseWriteSentence(treeNode *t){
    if (it->name == "WRITE")
    {
        it++;
        t->child.push_back(new treeNode("WRITE"));
        if (it->name == "(")
        {
            it++;
            t->child.push_back(new treeNode("LP"));

            treeNode *tIdentifier = new treeNode("IDENTIFIER?");
            analyseIdentifier(tIdentifier);
            t->child.push_back(tIdentifier);
            while (it->name == ",")
            {
                it++;
                tIdentifier = new treeNode("IDENTIFIER?");
                analyseIdentifier(tIdentifier);
                t->child.push_back(new treeNode("COMMA"));
                t->child.push_back(tIdentifier);
            }

            if (it->name == ")")
            {
                it++;
                t->child.push_back(new treeNode("RP"));
            }
            else
                throw "Syntax Error";
        }
        else
```

```
            throw "Syntax Error";
    }
    else
        throw "Syntax Error";
}
```

〈空语句〉→ epsilon

```
//空语句
void SyntaxAnalyse::analyseEmpty(treeNode *t){
    t->element = "EMPTY";
}
```

## 三、PL/0 编译器

首先实现

int getIndexbyName(int level,string name)根据 name，获得过程深度比 level 低的名字在名字表中的索引

int getIndexbyOPR(string opr); 获得 OPR_Table[opr]，主要用于对 OPR 指令中的 a 赋值

```
int SyntaxAnalyse::getIndexbyName(int level,string name)
{
    for(int i = table.size() - 1; i >= 0; i--)
    {
        if(table[i].name == name ){
            if(table[i].kind == "CONSTANT") return i;
            if(table[i].kind == "VARIABLE" && table[i].parameter1 <=
level) return i;
        }
    }
    return -1;
}
int SyntaxAnalyse::getIndexbyOPR(string opr)
{
    for(int i = 0; i < OPR_cnt; i++)
    {
        if(OPR_Table[i] == opr) return i;
    }
    return -1;
}
```

然后是进行语法制导翻译的过程，需要对 SyntaxAnalyse 类中分析得到语法单元的函数进行修改，使其同时生成目标代码。

比如在分程序部分，添加了 JMP，INT，OPR 三条指令

JMP 0 a 跳转到分程序入口地址 a，其中 a 要在子程序声明结束后的目标代码位置
INT 0 a 申请 a 个栈空间，为该分程序定义的变量个数+3，其中+3 是因为栈中前三个地址依次存放了程序返回地址、动态链、静态链
OPR 0 0 调用 OPR 中的 0 号功能，对应为 RETURN 功能

```
//分程序
void SyntaxAnalyse::analyseSubprog(treeNode *t){
//-------------------------------------start
    addCode("JMP", 0, 0);
    int tmpidx = code.size() - 1;
    bool flag = false;
//-------------------------------------end
    if (it->name == "CONST")
    {
        treeNode *tConstDeclare = new treeNode("CONSTANTDECLARE");
        analyseConstDeclare(tConstDeclare);
        t->child.push_back(tConstDeclare);
    }
    if (it->name == "VAR")
    {
        treeNode *tVariableDeclare = new treeNode("VARIABLEDECLARE");
        analyseVariableDeclare(tVariableDeclare);
        t->child.push_back(tVariableDeclare);
    }
    if (it->name == "PROCEDURE")
    {
        flag = true;
        treeNode *tProcedureDeclare = new treeNode("PROCEDUREDECLARE");
        analyseProcedureDeclare(tProcedureDeclare);
        t->child.push_back(tProcedureDeclare);
    }
//-------------------------------------start
    if(!flag)
        code.erase(code.end()-1);
    else
        code[tmpidx].a = code.size();
    int varCount = 0;
    for(int i = 0; i < table.size(); i++)
    {
        if(table[i].kind == "VARIABLE" && table[i].parameter1 ==
procedure_depth)
            varCount++;
    }
    addCode("INT", 0, varCount+3);
```

```
//-----------------------------------------------end
    treeNode *tSentence = new treeNode("SENTENCE");
    analyseSentence(tSentence);
    t->child.push_back(tSentence);
//-------------------------------------start
    addCode("OPR", 0, 0);
//-------------------------------------end
}
```

在赋值语句分析函数中添加一条 STO 指令
Index 为被赋值变量在名字表中的索引
abs(table[index].parameter1 - procedure_depth)为变量所在过程的深度与当前单词流迭代器 it 所在过程深度的差值
table[index].parameter2 为被赋值变量地址（相对于定义该变量的分过程的基地址）

```
//赋值语句
        int index = getIndexbyName(procedure_depth,idname);
        if(index == -1)
            throw "GenCode Error";
        if(table[index].kind != "VARIABLE")
            throw "GenCode Error";
        addCode("STO", abs(table[index].parameter1 - procedure_depth),
table[index].parameter2);
```

在下面这几个函数中添加了若干 OPR 指令，来执行对应算数运算以及关系运算

```
//条件
void SyntaxAnalyse::analyseCondition(treeNode *t)
//表达式
void SyntaxAnalyse::analyseExpression(treeNode *t)
//项
void SyntaxAnalyse::analyseItem(treeNode *t)
```

在下述函数中添加 LIT 和 LOD 指令用来读取常量或变量，作为表达式的项的因子

```
//因子
void SyntaxAnalyse::analyseFactor(treeNode *t)
```

在下述函数中添加了 CAL 指令用来调用过程

```
//过程调用语句
void SyntaxAnalyse::analyseCallSentence(treeNode *t)
```

在下面两函数中添加 JPC 指令用于条件跳转

```
//条件语句
void SyntaxAnalyse::analyseIfSentence(treeNode *t)
//当型循环语句
void SyntaxAnalyse::analyseWhileSentence(treeNode *t)
```

在以下函数中添加 READ 和 STO 指令用于读取标准输入，并保存到栈中变量

```
//读语句
void SyntaxAnalyse::analyseReadSentence(treeNode *t)
```

在以下函数中添加 LIT 或 LOD 指令，加载要输出的常量或变量到栈顶，然后添加 WRITE 指令输出栈顶值

```
//写语句
void SyntaxAnalyse::analyseWriteSentence(treeNode *t)
```

此外，在下述三个函数中，还添加了将常量名、变量名、过程名添加到名字表中的代码

```
//常量定义
void SyntaxAnalyse::analyseConstDefine(treeNode *t)
//变量说明部分
void SyntaxAnalyse::analyseVariableDeclare(treeNode *t)
//过程首部
void SyntaxAnalyse::analyseProcedureHead(treeNode *t)
```

在附录的 SyntaxAnalyse.cpp 部分可以看到以上的修改

**四、目标代码解释器**

Interpreter::interpret()函数中实现了目标代码的解释

stack 为运行中的程序栈

ret_addr = 0, dynamic_link = 1, static_link = 2 为栈中返回地址，动态链，静态链的相对地址。比如每个过程的 bp+ret_addr 就是该过程的返回地址在栈中的地址

sp_stack 记录运行中的每个过程的栈顶指针

变量 ir，ip，sp，bp 模拟下面四个寄存器

```
IR，指令寄存器，存放正在执行的指令。
IP，指令地址寄存器，存放下一条指令的地址。
SP，栈顶寄存器，指向运行栈的顶端。
BP，基址寄存器，指向当前过程调用所分配的空间在栈中的起始地址。
```

要根据如下功能码对目标代码进行解释

LIT：l 域无效，将 a 放到栈顶
LOD：将于当前层层差为 l 的层，变量相对位置为 a 的变量复制到栈顶
STO：将栈顶内容复制到于当前层层差为 l 的层，变量相对位置为 a 的变量
CAL：调用过程。l 标明层差，a 表明目标程序地址
INT：l 域无效，在栈顶分配 a 个空间
JMP：l 域无效，无条件跳转到地址 a 执行
JPC：l 域无效，若栈顶对应的布尔值为假（即 0）则跳转到地址 a 处执行，否则顺序执行
OPR：l 域无效，对栈顶和栈次顶执行运算，结果存放在次顶，a=0 时为调用返回

其中 OPR 中的 a 对应如下 OPR_Table 中的实际功能

```
string OPR_Table[20] = {"RETURN","ODD","READ","WRITE", "+", "-", "*", "/","=", "#", "<", "<=", ">", ">="};
```

```
void Interpreter::interpret()
{

    while (ip<code.size())
    {
        ir = code[ip++];
        if(ir.f == "LIT"){
            stack[sp++] = ir.a;
        }else if(ir.f == "LOD"){
            if (ir.l == 0)
                stack[sp++] = stack[bp + ir.a];
            else
            {
                int outer_bp = stack[bp + static_link];
                while (--ir.l)
                    outer_bp = stack[outer_bp + static_link];
                stack[sp++] = stack[outer_bp + ir.a];
            }
        }else if(ir.f == "STO"){
            if (ir.l == 0)
                stack[bp + ir.a] = stack[sp - 1];
            else
            {
                int outer_bp = stack[bp + static_link];
                while (--ir.l)
                    outer_bp = stack[outer_bp + static_link];
                stack[outer_bp + ir.a] = stack[sp - 1];
            }
        }else if(ir.f == "CAL"){
```

```
            stack[sp + ret_addr] = ip;
            stack[sp + dynamic_link] = bp;
            if(ir.l == 0)
                stack[sp + static_link] = bp;
            else
            {
                int outer_bp = stack[bp + static_link];
                while (--ir.l)
                    outer_bp = stack[outer_bp + static_link];
                stack[sp + static_link] = outer_bp;
            }
            ip = ir.a;
            bp = sp;
        }else if(ir.f == "INT"){
            sp_stack[sp_top++] = sp;
            sp += ir.a;
        }else if(ir.f == "JMP"){
            ip = ir.a;
        }else if (ir.f == "JPC"){
            if (stack[sp - 1] == 0)
                ip = ir.a;
            sp--;
        }else if(ir.f == "OPR"){
            //{"RETURN","ODD","READ","WRITE", "+", "-", "*", "/",
            //"=", "#", "<", "<=", ">", ">="};
            switch (ir.a)
            {
                case 0: //OPR_RET
                {
                    ip = stack[bp + ret_addr];
                    bp = stack[bp + dynamic_link];
                    sp = sp_stack[--sp_top];
                    if (sp_top <= 0)
                    {
                        return;
                    }
                    break;
                }
                case 1: //OPR_ODD
                {
                    stack[sp - 1] = stack[sp - 1] % 2;
                    break;
                }
                case 2: //OPR_READ
```

```
                {
                    scanf("%d", &stack[sp++]);
                    break;
                }
                case 3: //OPR_WRITE
                {
                    printf("%d\n", stack[sp - 1]);
                    sp--;
                    break;
                }
                case 4: //OPR_+
                {
                    stack[sp - 2] = stack[sp - 1] + stack[sp - 2];
                    sp--;
                    break;
                }
                case 5: //OPR_-
                {
                    stack[sp - 2] = stack[sp - 2] - stack[sp - 1];
                    sp--;
                    break;
                }
                case 6: //OPR_*
                {
                    stack[sp - 2] = stack[sp - 1] * stack[sp - 2];
                    sp--;
                    break;
                }
                case 7: //OPR_/
                {
                    stack[sp - 2] = stack[sp - 2] / stack[sp - 1];
                    sp--;
                    break;
                }
                case 8: //OPR_=
                {
                    stack[sp - 2] = (stack[sp - 2] == stack[sp - 1]) ;
                    sp--;
                    break;
                }
                case 9: //OPR_#
                {
                    stack[sp - 2] = (stack[sp - 2] != stack[sp - 1]) ;
                    sp--;
```

```
                    break;
                }
                case 10: //OPR_<
                {
                    stack[sp - 2] = (stack[sp - 2]<stack[sp - 1]);
                    sp--;
                    break;
                }
                case 11: //OPR_<=
                {
                    stack[sp - 2] = (stack[sp - 2] <= stack[sp - 1]);
                    sp--;
                    break;
                }
                case 12: //OPR_>
                {
                    stack[sp - 2] = (stack[sp - 2]>stack[sp - 1]);
                    sp--;
                    break;
                }
                case 13: //OPR_>=
                {
                    stack[sp - 2] = (stack[sp - 2] >= stack[sp - 1]);
                    sp--;
                    break;
                }
                default:{
                   return;
                }
            }
        }
    }
}
```

实验一的 main 函数

```
int main()
{
    //freopen("lexical4.txt", "w", stdout);
    LexicalAnalysis la;
    if(la.analysis("stdin")){
        la.print();
    }else{
        cout << "Lexical Error" << endl;
```

```
    }
    return 0;
}
```

实验二的 main 函数

```
int main()
{
    string filename="stdin";
    //string filename="in.txt";

    LexicalAnalyse la;
    SyntaxAnalyse sa;
    if(!la.analyse(filename)){
        cout<<"Lexical Error"<<endl;
        return 0;
    }else{
        try{
            sa.analyse(la.words);
        }catch(...){
            cout<<"Syntax Error"<<endl;
            return 0;
        }
        sa.printTree();
    }
    return 0;
}
```

实验三的 main 函数
编译器

```
int main()
{
    string filename="stdin";

    LexicalAnalyse la;
    SyntaxAnalyse sa;
    if(!la.analyse(filename)){
//        cout<<"Lexical Error"<<endl;
        return -1;
    }else{
        try{
            sa.analyse(la.words);
        }catch(...){
//            cout<<"Syntax Error"<<endl;
            return -1;
```

```
        }
        sa.printCode();
    }
    return 0;
}
```

解释器

```
int main(){
    Interpreter interpreter;
    interpreter.readCode("program.code");
    interpreter.interpret();
    return 0;
}
```

实验结果展示及分析：

**实验一 PL/0 词法分析器**

对样例 1 的测试

```
VAR A,B;
CONST C=0;
BEGIN
    READ(B,A);
    A:=B+C;
    WRITE(A);
END.
```

```
PS C:\Users\Sunstrider\Desktop\dasanxia\bianyi\Codes\E1> cd
E1.cpp -o E1 } ; if ($?) { .\E1 }
VAR
IDENTIFIER A
=
NUMBER 0
;
BEGIN
READ
(
IDENTIFIER B
,
IDENTIFIER A
)
;
IDENTIFIER A
:=
IDENTIFIER B
+
IDENTIFIER C
;
WRITE
(
IDENTIFIER A
)
;
END
.
```

在 OJ 中的测试

| ID | 用户名 | 题目编号 | 题目名 | 结果 | 得分 | 评测模板 | 内存使用 | 时间使用 | 提交时间 |
|---|---|---|---|---|---|---|---|---|---|
| 61a4c4b3b00e2fc | 202000120101 | 1 | 词法分析器 | 测试通过 | | C++14 | 13028 KB | 108 ms | 2 个月前 |
| 61a4bab0cc0e2fb | 202000120101 | 1 | 词法分析器 | 时间超限 | | C++14 | 249036 KB | 2006 ms | 2 个月前 |
| 61a4716ba00e2ee | 202000120101 | 1 | 词法分析器 | 时间超限 | | C++14 | 249148 KB | 2005 ms | 2 个月前 |
| 61a45efb6c0e2eb | 202000120101 | 1 | 词法分析器 | 时间超限 | | C++14 | 249024 KB | 2007 ms | 2 个月前 |
| 61a4251f940e2df | 202000120101 | 1 | 词法分析器 | 时间超限 | | C++14 | 249024 KB | 2009 ms | 2 个月前 |
| 61a41745d00e2da | 202000120101 | 1 | 词法分析器 | 答案错误 | | C++14 | 3688 KB | 18 ms | 2 个月前 |
| 61a4102dac0e2d6 | 202000120101 | 1 | 词法分析器 | 答案错误 | | C++14 | 3500 KB | 1 ms | 2 个月前 |
| 61a40830f00e2d2 | 202000120101 | 1 | 词法分析器 | 时间超限 | | C++14 | 249080 KB | 2003 ms | 2 个月前 |
| 61a3baa64c0e2bc | 202000120101 | 1 | 词法分析器 | 答案错误 | | C++14 | 249136 KB | 2004 ms | 2 个月前 |
| 61a39cb6780e2b6 | 202000120101 | 1 | 词法分析器 | 答案错误 | | C++14 | 249144 KB | 2005 ms | 2 个月前 |
| 61a3910a500e2b2 | 202000120101 | 1 | 词法分析器 | 答案错误 | | C++14 | 249032 KB | 2005 ms | 2 个月前 |

在 OJ 的测试中，我出现了以下错误

（1）测试点 5,14 有前导零，我没有进行前导 0 的删除，同时要注意数字本身为 0 时不要删除最后一个 0

（2）测试点 17 存在运算符为双字符，但是该双字符不在 op 中，此时应输出 Lexical Error，我判断为了运算符

（3）测试点 7 存在数字后面跟字母，此时应输出 Lexical Error，我原本认为这不属于词法错误

## 实验二 PL/0 语法分析器

对样例 1 的测试

```
VAR A,B;
CONST C=0;
BEGIN
    READ(B,A);
    A:=B+C;
    WRITE(A);
END.
```

```
PS C:\Users\Sunstrider\Desktop\dasanxia\bianyi\Codes\E1> cd "c:\Users\Sunstrider\Desktop\dasanxia\bianyi\Codes\E2\" ; if ($?) { g++
main.cpp -o main } ; if ($?) { .\main }
PROGRAM(SUBPROG(CONSTANTDECLARE(CONST,CONSTANTDEFINE(C,=,0),;),VARIABLEDECLARE(VAR,A,COMMA,B,;),SENTENCE(COMBINED(BEGIN,SENTENCE(REA
DSENTENCE(READ,LP,B,COMMA,A,RP)),;,SENTENCE(ASSIGNMENT(A,:=,EXPRESSION(ITEM(FACTOR(B)),+,ITEM(FACTOR(C))))),;,SENTENCE(WRITESENTENCE
(WRITE,LP,A,RP)),;,SENTENCE(EMPTY),END))),.)
PS C:\Users\Sunstrider\Desktop\dasanxia\bianyi\Codes\E2>
```

```
pretty.cpp -o pretty } ; if ($?) { .\pretty }
PROGRAM(
        SUBPROG(
                CONSTANTDECLARE(
                        CONST,
                        CONSTANTDEFINE(
                                C,
                                =,
                                0
                        ),
                        ;
                ),
                VARIABLEDECLARE(
                        VAR,
                        A,
                        COMMA,
                        B,
                        ;
                ),
                SENTENCE(
                        COMBINED(
                                BEGIN,
                                SENTENCE(
                                        READSENTENCE(
                                                READ,
                                                LP,
                                                B,
                                                COMMA,
                                                A,
                                                RP
                                        )
                                ),
                                ;,
                                SENTENCE(
                                        ASSIGNMENT(
                                                A,
                                                :=,
                                                EXPRESSION(
                                                        ITEM(
                                        )
                                )
                        )
                )
        ),
        .
)
```

## 在 OJ 中的测试

| ID | 用户名 | 题目编号 | 题目名 | 结果 | 得分 | 评测模板 | 内存使用 | 时间使用 | 提交时间 |
|---|---|---|---|---|---|---|---|---|---|
| 65047658ac06530 | 202000120101 | 2 | 语法分析器 | 测试通过 | | PL/0 语法分析器 C/C++ | 98176 KB | 940 ms | 14 天前 |
| 65046fa2e80652e | 202000120101 | 2 | 语法分析器 | 答案错误 | | PL/0 语法分析器 C/C++ | 98208 KB | 1028 ms | 14 天前 |
| 65046d19840652d | 202000120101 | 2 | 语法分析器 | 答案错误 | | PL/0 语法分析器 C/C++ | 98304 KB | 1008 ms | 14 天前 |
| 65046af2fc0652c | 202000120101 | 2 | 语法分析器 | 答案错误 | | PL/0 语法分析器 C/C++ | 98236 KB | 959 ms | 14 天前 |
| 650466a45c0652a | 202000120101 | 2 | 语法分析器 | 答案错误 | | PL/0 语法分析器 C/C++ | 98376 KB | 938 ms | 14 天前 |
| 64768bf2d0060c8 | 202000120101 | 2 | 语法分析器 | 答案错误 | | PL/0 语法分析器 C/C++ | 98300 KB | 948 ms | 21 天前 |
| 64767ce648060c4 | 202000120101 | 2 | 语法分析器 | 答案错误 | | PL/0 语法分析器 C/C++ | 94516 KB | 812 ms | 21 天前 |
| 647624abc8060af | 202000120101 | 2 | 语法分析器 | 答案错误 | | PL/0 语法分析器 C/C++ | 98356 KB | 959 ms | 21 天前 |
| 6475c3a45006090 | 202000120101 | 2 | 语法分析器 | 答案错误 | | PL/0 语法分析器 C/C++ | 98324 KB | 949 ms | 21 天前 |
| 6475bfb7700608d | 202000120101 | 2 | 语法分析器 | 编译错误 | | PL/0 语法分析器 C/C++ | 21812 KB | 38 ms | 21 天前 |
| 6475b6ef7406088 | 202000120101 | 2 | 语法分析器 | 编译错误 | | PL/0 语法分析器 C/C++ | 21900 KB | 27 ms | 21 天前 |
| 6475a48c3406083 | 202000120101 | 2 | 语法分析器 | 编译错误 | | PL/0 语法分析器 C/C++ | 21940 KB | 31 ms | 21 天前 |
| 64759b3e0c06082 | 202000120101 | 2 | 语法分析器 | 编译错误 | | PL/0 语法分析器 C/C++ | 22040 KB | 68 ms | 21 天前 |
| 647579ea4c0607b | 202000120101 | 2 | 语法分析器 | 编译错误 | | PL/0 语法分析器 C/C++ | 22020 KB | 67 ms | 21 天前 |
| 6475706ac806079 | 202000120101 | 2 | 语法分析器 | 编译错误 | | PL/0 语法分析器 C/C++ | 21956 KB | 79 ms | 21 天前 |

## 主要出现下述两个错误

```
>>>>Running Testcase constant_declare_err_4
[FAIL] Wrong Answer
```

第一个错误是因为我在之前的词法分析部分写错了一部分，把一些不应当 Lexical Error 的情况错判成了 Lexical Error

错误部分代码如下，获得一个运算符的时候，只有第二个字符为'='的情况下，该运算符才可能是双字符，然而我原本的判断是第二个字符只要是单运算符的其中一种，就将该单词判断为双运算符。

这样对 a=-b 这样的语句，会将'=-'判断为一个双字符运算符，但是并不存在该双字符运算符，就会在后续判断中报 Lexical Error，然而实际上该语句是合法的

修改前

```
if (isOperator(ch))
{
    wd.name += ch;
}
else
{
    buffer.putback(ch);
}
```

修改后

```
if (ch == '=')
{
    wd.name += ch;
}
else
{
    buffer.putback(ch);
}
```

```
>>>>Running Testcase empty
/workspace/452431692909535376/jt.sh: line 34: 129
/dev/null
[ERROR] Runtime Error
```

第二个错误是整个源程序为空（不含任何字符）的时候，没有特判，在程序分析函数中添加如下代码即可

```
if(words.size() == 0)
    throw "Syntax Error";
```

**实验三 PL/0 编译器以及目标代码解释器**

对样例 1 的测试

```
VAR A,B;
CONST C=0;
```

```
BEGIN
    READ(B,A);
    A:=B+C;
    WRITE(A);
END.
```

编译器

```
PS C:\Users\Sunstrider\Desktop\dasanxia\bianyi\Codes\E3> cd "c:\Us
CompilerMain.cpp -o CompilerMain } ; if ($?) { .\CompilerMain }
INT 0 5
OPR 0 2
STO 0 3
OPR 0 2
STO 0 4
LOD 0 3
LOD 0 4
OPR 0 4
STO 0 3
LOD 0 3
OPR 0 3
OPR 0 0
PS C:\Users\Sunstrider\Desktop\dasanxia\bianyi\Codes\E3>
```

解释器

```
PS C:\Users\Sunstrider\Desktop\dasanxia\bianyi\Codes\E3> cd
InterpreterMain.cpp -o InterpreterMain } ; if ($?) { .\Inter
233
666
899
PS C:\Users\Sunstrider\Desktop\dasanxia\bianyi\Codes\E3>
```

在 OJ 中的测试

| | | | | | | | | |
|---|---|---|---|---|---|---|---|---|
| 65c2fca28c06bab | 202000120101 | 3 | PL/0 编译器 | 测试通过 | PL/0 编译器(Compiler)模式(C/C++) | 115504 KB | 2840 ms | 4 天前 |
| 65c2b7420806b92 | 202000120101 | 3 | PL/0 编译器 | 测试通过 | PL/0 编译器(Compiler)模式(C/C++) | 115432 KB | 2790 ms | 4 天前 |
| 65c21f9e7406b56 | 202000120101 | 3 | PL/0 编译器 | 答案错误 | PL/0 编译器(Compiler)模式(C/C++) | 114796 KB | 2822 ms | 4 天前 |
| 65c1fdf11006b4e | 202000120101 | 3 | PL/0 编译器 | 答案错误 | PL/0 编译器(Compiler)模式(C/C++) | 114872 KB | 2828 ms | 4 天前 |
| 65c0c2f19406b2c | 202000120101 | 3 | PL/0 编译器 | 答案错误 | PL/0 编译器(Compiler)模式(C/C++) | 114872 KB | 2708 ms | 5 天前 |
| 65bfae75c806b11 | 202000120101 | 3 | PL/0 编译器 | 答案错误 | PL/0 编译器(Compiler)模式(C/C++) | 114980 KB | 2722 ms | 5 天前 |
| 65beb9e2c006ae4 | 202000120101 | 3 | PL/0 编译器 | 答案错误 | PL/0 编译器(Compiler)模式(C/C++) | 113468 KB | 2062 ms | 5 天前 |
| 65be9ad0b006ae1 | 202000120101 | 3 | PL/0 编译器 | 答案错误 | PL/0 编译器(Compiler)模式(C/C++) | 113404 KB | 2067 ms | 5 天前 |
| 65b973ce9006aba | 202000120101 | 3 | PL/0 编译器 | 答案错误 | PL/0 编译器(Compiler)模式(C/C++) | 113424 KB | 2092 ms | 5 天前 |
| 65b95d25b006ab9 | 202000120101 | 3 | PL/0 编译器 | 时间超限 | PL/0 编译器(Compiler)模式(C/C++) | 1632 KB | 41001 ms | 5 天前 |
| 65b939a9c406ab8 | 202000120101 | 3 | PL/0 编译器 | 答案错误 | PL/0 编译器(Compiler)模式(C/C++) | 113224 KB | 1680 ms | 5 天前 |
| 65b8c7038006ab7 | 202000120101 | 3 | PL/0 编译器 | 答案错误 | PL/0 编译器(Compiler)模式(C/C++) | 113292 KB | 2088 ms | 5 天前 |
| 65b8441d4806ab6 | 202000120101 | 3 | PL/0 编译器 | 时间超限 | PL/0 编译器(Compiler)模式(C/C++) | 113456 KB | 23484 ms | 5 天前 |
| 65b7e04e2c06ab5 | 202000120101 | 3 | PL/0 编译器 | 时间超限 | PL/0 编译器(Compiler)模式(C/C++) | 113316 KB | 23553 ms | 5 天前 |
| 65b7ce98ec06ab4 | 202000120101 | 3 | PL/0 编译器 | 答案错误 | PL/0 编译器(Compiler)模式(C/C++) | 113536 KB | 3196 ms | 5 天前 |
| 65b72cbec806ab3 | 202000120101 | 3 | PL/0 编译器 | 时间超限 | PL/0 编译器(Compiler)模式(C/C++) | 1612 KB | 41000 ms | 5 天前 |

出现的错误主要有以下几点

（1）

```
>>>>Running Testcase if_statement
Compile Successful. Running testcases.

Files answer.tmp and greater.out differ
Wrong Answer on if_statement - greater
[!]  Something went wrong on if_statement
```

这个报错实际上是 WIRTE 指令生成时的错误（和 if 语句没关系）
WRITE 的可能是一个常量名，但是我全都默认为变量名了，当为常量名时应当用 LIT 而不是 LOD

```
if(table[tmpidx].kind == "CONSTANT")
                    addCode("LIT", 0, table[tmpidx].parameter1);
```

（2）

```
>>>>Running Testcase complicate_program
[!]  Expected Compiled Program But Compile Error Reported
```

对于同层次过程中，定义的重名变量，是合法的，但是我判断成了报错。
（3）

```
>>>>Running Testcase recursion_fibonacci
Compile Successful. Running testcases.

Accepted recursion_fibonacci - test1
Accepted recursion_fibonacci - test2
Files answer.tmp and test3.out differ
Wrong Answer on recursion_fibonacci - test3
[!]  Something went wrong on recursion_fibonacci
```

```
>>>>Running Testcase recursion_static_link
Compile Successful. Running testcases.

Accepted recursion_static_link - test1
/workspace/458254658511596332/jt.sh: line 75: 421796 Segmentation fault
Interpreter Runtime Error
[!]  Something went wrong on recursion_static_link
```

```
>>>>Running Testcase recursion_indirect
Compile Successful. Running testcases.

Accepted recursion_indirect - test1
Files answer.tmp and test2.out differ
Wrong Answer on recursion_indirect - test2
[!]  Something went wrong on recursion_indirect
```

出错原因是解释器中 CAL 指令中静态链设置错误

修改前

```
stack[sp + static_link] = bp;
```

修改后

```
if(ir.l == 0)
    stack[sp + static_link] = bp;
else
{
    int outer_bp = stack[bp + static_link];
    while (--ir.l)
        outer_bp = stack[outer_bp + static_link];
    stack[sp + static_link] = outer_bp;
}
```

（4）

```
>>>>Running Testcase error_variable_redeclare
[!]  Expected Compile Error, But You Didn't Report Error
```

变量的命名是已有常量时，没有报错

（5）

```
>>>>Running Testcase error_assign_procedure
[!]  Expected Compile Error, But You Didn't Report Error
```

被赋值的是过程名，没有报错

（6）

```
>>>>Running Testcase error_calling_variable
[!]  Expected Compile Error, But You Didn't Report Error
```

CALL 的是变量名时，没有报错

**附件：关键程序代码**

**struct.hpp**

```
#include <string>
#include <vector>
using namespace std;

struct pl0word{
    string name; // 单词本身
    string sym; // 单词类型
    int line; // 行号
    int col; // 列号
    pl0word(){
        name = "";
        sym = "";
        line = 0;
        col = 0;
    }
    pl0word(const pl0word& wd){
        name = wd.name;
        sym = wd.sym;
        line = wd.line;
        col = wd.col;
    }
    //重载赋值运算符
    pl0word& operator=(const pl0word& wd){
        name = wd.name;
        sym = wd.sym;
        line = wd.line;
        col = wd.col;
        return *this;
    }
};

struct treeNode
{
    string element;
    vector<treeNode*> child;

    treeNode(string _element)
    {
        element = _element;
    }
    ~treeNode()
```

```
    {
        for (int i = 0; i < child.size(); i++)
        {
            if(child[i] != NULL) delete child[i];
        }
    }
};

// 生成的目标代码是一种假想栈式计算机的汇编语言，其格式如下：

// f l a
// 其中 f 为功能码，l 代表层次差，a 代表位移量。

// 这种假想栈式计算机有一个无限大的栈，以及四个寄存器 IR,IP,SP,BP。

// IR，指令寄存器，存放正在执行的指令。

// IP，指令地址寄存器，存放下一条指令的地址。

// SP，栈顶寄存器，指向运行栈的顶端。

// BP，基址寄存器，指向当前过程调用所分配的空间在栈中的起始地址。
struct TargetCode{
    string f;
    int l;
    int a;
};

// NAME KIND    PARAMETER1  PARAMETER2
// a    CONSTANT    VAL:35  --
// b    CONSTANT    VAL:49  --
// c    VARIABLE    LEVEL: LEV  ADR: DX
// d    VARIABLE    LEVEL: LEV  ADR: DX+1
// e    VARIABLE    LEVEL: LEV  ADR: DX+2
// p    PROCEDURE   LEVEL: LEV  ADR: <UNKNOWN>
// g    VARIABLE    LEVEL: LEV+1    ADR: DX
// 其中，LEVEL 给出的是层次，DX 是每一层局部量的相对地址。
// 对于过程名的地址，需要等待目标代码生成后再填写。
// 考虑到需要存储调用信息、返回信息等维护函数运行的数据，在本实验中，DX 取 3.
struct NameTable{
    string name;
    string kind;
    int parameter1; // VAL/LEVEL
    int parameter2; // ADR
```

```
};
```

**LexicalAnalyse.cpp**

```
// 把关键字、算符、界符称为语言固有的单词，标识符、常量称为用户自定义的单词。为此设置两个全程量：NAME,SYM 。
//  SYM：存放每个单词的类别。
//  NAME：存放单词值。
//  要完成的任务：
// 1.  滤掉单词间的空格。
// 2.  识别关键字，用查关键字表的方法识别。当单词是关键字时，将对应的类别放在 SYM 中。如 IF 的类别为 IFSYM，THEN 的类别为 THENSYM。
// 3.  识别标识符，标识符的类别为 IDENT，IDENT 放在 SYM 中，标识符本身的值放在 NAME 中。
// 4.  拼数，将数的类别 NUMBER 放在 SYM 中，数本身的值放在 NAME 中。
// 5.  拼由两个字符组成的运算符，如：>=、<=等等，识别后将类别存放在 SYM 中。
// 6.  打印源程序，边读入字符边打印。
// 注意：由于一个单词是由一个或多个字符组成的，所以需实现一个读字符过程。

// 在 PL/0 语言中，标识符不会超过 10 字符长。如果超过了 10 个字符，你应该认为这是一个词法错误。
// 如果程序包含词法错误，你的程序应当仅输出一行"Lexical Error"，不含引号。
// 输出时，除 Lexical Error 外，输出字母应当均为大写。

//关键词
// CONST,VAR,PROCEDURE,BEGIN,END,ODD,IF,THEN,CALL,WHILE,DO,READ,WRITE

//分隔符
// ; , . ( )

#include <iostream>
#include <string>
#include <fstream>
#include <sstream>
#include <vector>
#include "struct.hpp"

class LexicalAnalyse{
public:
    vector<pl0word> words;

    LexicalAnalyse(){};
    ~LexicalAnalyse(){};

    void print();
```

```
    bool analyse(string filename);

private:
    //PL0 关键字
    int keylen = 13;
    string keyword[13] = { "CONST","VAR","PROCEDURE","BEGIN",
    "END","ODD","IF","THEN","CALL","WHILE","DO","READ","WRITE" };
    string keysym[13] = { "CONSTSYM","VARSYM","PROCSYM","BEGINSYM",
    "ENDSYM","ODDSYM","IFSYM","THENSYM","CALLSYM","WHILESYM","DOSYM",
    "READSYM","WRITESYM" };
    //PL0 运算符
    int oplen = 11;
    string op[11] =  {"=",":=", "+", "-", "*", "/", "#", "<", "<=", ">",
">="};
    string opsym[11] =  {"EQL", "BECOMES", "PLUS", "MINUS", "TIMES",
"SLASH",
     "NEQ", "LSS", "LEQ", "GTR", "GEQ"};
    //PL0 分隔符
    int dellen = 5;
    string delimiter[5] = { ";", ",", ".", "(", ")" };
    string delsym[5] = {"SEMICOLON", "COMMA", "PERIOD", "LPAREN",
"RPAREN" };
    //标识符 IDENTIFIER
    //数 NUMBER

    bool isLetter(char ch); //判断是否为字母
    bool isDigit(char ch);  //判断是否为数字
    bool isDelimiter(char ch); //判断是否为分隔符
    bool isOperator(char ch); //判断是否为运算符
    stringstream readfile(string filename); //读取文件
};


stringstream LexicalAnalyse::readfile(string filename){
    ifstream fin;
    stringstream buffer;
    if(filename == "stdin"){
        //从标准输入所有字符后赋给 buffer
        char ch;
        while(cin.get(ch)){
            //转换成大写字母
            if(ch >= 'a' && ch <= 'z'){
                ch -= 32;
            }
```

```
            buffer << ch;
        }
    }else{
        fin.open(filename);
        //转换成大写字母
        char ch;
        while(fin.get(ch)){
            if(ch >= 'a' && ch <= 'z'){
                ch -= 32;
            }
            buffer << ch;
        }
        fin.close();
    }
    return buffer;
}
bool LexicalAnalyse::isLetter(char ch){
    if ((ch >= 'a' && ch <= 'z') || (ch >= 'A' && ch <= 'Z'))
    {
        return true;
    }
    else
    {
        return false;
    }
}
bool LexicalAnalyse::isDigit(char ch){
    if (ch >= '0' && ch <= '9')
    {
        return true;
    }
    else
    {
        return false;
    }
}
bool LexicalAnalyse::isDelimiter(char ch){
    for (int i = 0; i < dellen; i++)
    {
        if (ch == delimiter[i][0])
        {
            return true;
        }
    }
```

```
    return false;
}
bool LexicalAnalyse::isOperator(char ch){
    for (int i = 0; i < oplen; i++)
    {
        if (ch == op[i][0])
        {
            return true;
        }
    }
    return false;
}
bool LexicalAnalyse::analyse(string filename){

    int i;
    char ch;
    stringstream buffer = readfile(filename);
    while (!buffer.eof())
    {
        pl0word wd;
        buffer.get(ch);
        if(buffer.eof()){
            break;
        }
        if (ch == ' ' || ch == '\t' || ch == '\n' || ch == '\r')
        {
            continue;
        }
        else if (isLetter(ch))
        {
            while (!buffer.eof() && (isLetter(ch) || isDigit(ch)))
            {
                wd.name += ch;
                buffer.get(ch);
            }
            if( !(isLetter(ch) || isDigit(ch)) ) buffer.putback(ch);
            if(wd.name.length() > 10){
                return false;
            }
            for (i = 0; i < keylen; i++)
            {
                if (wd.name == keyword[i])
                {
                    break;
```

```
                }
            }
            if (i < keylen)
                wd.sym = keysym[i];
            else
                wd.sym = "IDENTIFIER";
        }
        else if (isDigit(ch))
        {
            while (!buffer.eof() && isDigit(ch))
            {
                wd.name += ch;
                buffer.get(ch);
            }
            // 数字后面不能跟字母
            if(isLetter(ch)){
                return false;
            }
            if(!isDigit(ch)) buffer.putback(ch);
            //去除前导 0
            while(wd.name[0] == '0'&& wd.name.length() > 1){
                wd.name.erase(0,1);
            }
            wd.sym = "NUMBER";
        }
        else if (isOperator(ch))
        {
            wd.name = ch;
            buffer.get(ch);
            if (ch == '=')
            {
                wd.name += ch;
            }
            else
            {
                buffer.putback(ch);
            }
            for (i = 0; i < oplen; i++)
            {
                if (wd.name == op[i])
                {
                    wd.sym = opsym[i];
                    break;
                }
```

```
            }
            if (i == oplen)
                return false;
        }
        else if (isDelimiter(ch))
        {
            wd.name = ch;
            for (i = 0; i < dellen; i++)
            {
                if (wd.name == delimiter[i])
                {
                    wd.sym = delsym[i];
                    break;
                }
            }
        }
        else
            return false;
        words.push_back(wd);
    }
    return true;
}
void LexicalAnalyse::print(){
    for(int i = 0; i < words.size(); i++){
        if(words[i].sym == "NUMBER"||words[i].sym == "IDENTIFIER")
            cout << words[i].sym << " " << words[i].name << endl;
        else
            cout << words[i].name << endl;
    }
}

//测试点 5,14 有前导零
//测试点 17 存在运算符为双字符，但是不在 op 中，此时应输出 Lexical Error
//测试点 7 存在数字后面跟字母，此时应输出 Lexical Error
```

**SyntaxAnalyse.cpp**

```
// 完成 PL/0 语言的语法分析器
// 词法分析器会向语法分析器提供词语流。你需要以合适的方式对词语流进行分析，并生成一颗语法树，或者宣告语法错误。
// 如果输入的程序包含语法错误，仅输出一行"Syntax Error"（不含引号）
// 评测程序只会读取你的程序输出到标准输出的内容。你可以向标准错误流打印若干信息以方便调试，美观输出或是出于其他任何原因。

// 如果该节点是(，即左括号，该节点应当输出为 LP
```

```
// 如果该节点是)，即右括号，该节点应当输出为 RP
// 如果该节点是,，即逗号，该节点应当输出为 COMMA
// 如果该节点是其他的叶子节点（对应词法分析器中的某一个词语），那么其名称为其本身
// 如果该节点不是一个叶子节点，则遵循下表进行替换输出

// 上下文无关文法中的节点  对应的替换词语
// 程序   PROGRAM
// 分程序  SUBPROG
// 常量说明部分   CONSTANTDECLARE
// 常量定义 CONSTANTDEFINE
// 无符号整数    < 这是一个叶子节点，用其本身替代 >
// 变量说明部分   VARIABLEDECLARE
// 标识符  < 这是一个叶子节点，用其本身替代 >
// 过程说明部分   PROCEDUREDECLARE
// 过程首部 PROCEDUREHEAD
// 语句   SENTENCE
// 赋值语句 ASSIGNMENT
// 复合语句 COMBINED
// 条件   CONDITION
// 表达式  EXPRESSION
// 项    ITEM
// 因子   FACTOR
// 加减运算符    < 这是一个叶子节点，用其本身替代 >
// 乘除运算符    < 这是一个叶子节点，用其本身替代 >
// 关系运算符    < 这是一个叶子节点，用其本身替代 >
// 条件语句 IFSENTENCE
// 过程调用语句   CALLSENTENCE
// 当型循环语句   WHILESENTENCE
// 读语句  READSENTENCE
// 写语句  WRITESENTENCE
// 空语句  EMPTY

// <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>}
// <变量说明部分> → VAR<标识符>{ ,<标识符>};
// <标识符> → <字母>{<字母>|<数字>}
// <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>|<空语句>
```

```
// <赋值语句> → <标识符>:=<表达式>
// <复合语句> → BEGIN<语句>{ ;<语句>} END
// <条件> → <表达式><关系运算符><表达式>|ODD<表达式>
// <表达式> → [+|-]<项>{<加减运算符><项>}
// <项> → <因子>{<乘除运算符><因子>}
// <因子> → <标识符>|<无符号整数>|(<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<标识符>
// <当型循环语句> → WHILE<条件>DO<语句>
// <读语句> → READ(<标识符>{ ,<标识符>})
// <写语句> → WRITE(<标识符>{,<标识符>})
// <字母> → A|B|C…X|Y|Z
// <数字> → 0|1|2…7|8|9
// <空语句> → epsilon
#include <iostream>
#include <vector>
#include <string>
#include <stack>
#include "LexicalAnalyse.cpp"


class SyntaxAnalyse{
public:
    SyntaxAnalyse();
    ~SyntaxAnalyse();
    void analyse(vector<pl0word> words);
    void printTree();
    void printCode();

    void printNameTable();
private:
    vector<pl0word> words;
    vector<pl0word>::iterator it;
    treeNode *root;
    int procedure_depth; //过程嵌套层数

    void _printTree(treeNode *t);
    void analyseProgram(treeNode *t); //程序
    void analyseSubprog(treeNode *t); //分程序
    void analyseConstDeclare(treeNode *t); //常量说明部分
    void analyseConstDefine(treeNode *t); //常量定义
```

```
    void analyseUnsignedInteger(treeNode *t); //无符号整数
    void analyseVariableDeclare(treeNode *t);  //变量说明部分
    void analyseIdentifier(treeNode *t); //标识符
    void analyseProcedureDeclare(treeNode *t); //过程说明部分
    void analyseProcedureHead(treeNode *t); //过程首部
    void analyseSentence(treeNode *t); //语句
    void analyseAssignment(treeNode *t); //赋值语句
    void analyseCombined(treeNode *t); //复合语句
    void analyseCondition(treeNode *t); //条件
    void analyseExpression(treeNode *t); //表达式
    void analyseItem(treeNode *t); //项
    void analyseFactor(treeNode *t); //因子
    void analyseAddSubOperator(treeNode *t); //加减运算符
    void analyseMulDivOperator(treeNode *t); //乘除运算符
    void analyseRelationOperator(treeNode *t); //关系运算符
    void analyseIfSentence(treeNode *t); //条件语句
    void analyseCallSentence(treeNode *t); //过程调用语句
    void analyseWhileSentence(treeNode *t); //当型循环语句
    void analyseReadSentence(treeNode *t); //读语句
    void analyseWriteSentence(treeNode *t); //写语句
    void analyseEmpty(treeNode *t); //空语句

//------------------目标代码生成------------------start
    int OPR_cnt = 14;
    string OPR_Table[20] = {"RETURN","ODD","READ","WRITE", "+", "-", "*",
"/","=", "#", "<", "<=", ">", ">="};
    vector<NameTable> table;
    vector<TargetCode> code;
    void addTable(string name, string kind, int parameter1, int parameter2);
    void addCode(string f, int l, int a);

    int getIndexbyName(int level, string name);
    int getIndexbyOPR(string opr);
//-----------------------------------------------end
};
void SyntaxAnalyse::printNameTable(){
    cout << "NameTable:" << endl;
    cout << "name\tkind\tparameter1\tparameter2" << endl;
    for(int i = 0; i < table.size(); i++){
        cout << table[i].name << "\t" << table[i].kind << "\t" <<
table[i].parameter1 << "\t" << table[i].parameter2 << endl;
    }
}
```

```
//------------------目标代码生成------------------start
void SyntaxAnalyse::printCode()
{
    for(int i = 0; i < code.size(); i++)
    {
        cout<< code[i].f << " " << code[i].l << " " << code[i].a << endl;
    }
}
void SyntaxAnalyse::addTable(string name, string kind, int parameter1, int
parameter2)
{
    NameTable t;
    t.name = name;
    t.kind = kind;
    t.parameter1 = parameter1;
    t.parameter2 = parameter2;
    table.push_back(t);
}
void SyntaxAnalyse::addCode(string f, int l, int a)
{
    TargetCode t;
    t.f = f;
    t.l = l;
    t.a = a;
    code.push_back(t);
}
int SyntaxAnalyse::getIndexbyName(int level,string name)
{
    for(int i = table.size() - 1; i >= 0; i--)
    {
        if(table[i].name == name ){
            if(table[i].kind == "CONSTANT") return i;
            if(table[i].kind == "VARIABLE" && table[i].parameter1 <= level)
return i;
        }
    }
    return -1;
}
int SyntaxAnalyse::getIndexbyOPR(string opr)
{
    for(int i = 0; i < OPR_cnt; i++)
    {
        if(OPR_Table[i] == opr) return i;
    }
```

```
    return -1;
}

//-----------------------------------------------end
SyntaxAnalyse::SyntaxAnalyse()
{
    root = NULL;
    procedure_depth = 0;
}

SyntaxAnalyse::~SyntaxAnalyse()
{
    if(root != NULL) delete root;
}

void SyntaxAnalyse::printTree()
{
    _printTree(root);
}
void SyntaxAnalyse::_printTree(treeNode *t)
{
    if(t == NULL)
        return;
    if (t->child.size() == 0)
    {
        cout << t->element;
        return;
    }
    cout << t->element;
    cout << "(";
    for (int i = 0; i < t->child.size(); i++)
    {
        _printTree(t->child[i]);
        if(i != t->child.size()-1) cout<<",";
    }
    cout << ")";
}

void SyntaxAnalyse::analyse(vector<pl0word> _words)
{
    for(int i = 0; i < _words.size(); i++)
    {
        words.push_back(_words[i]);
    }
```

```
    it = words.begin();
    root = new treeNode("PROGRAM");
    analyseProgram(root);
}


//程序
void SyntaxAnalyse::analyseProgram(treeNode *t){
    if(words.size() == 0)
        throw "Syntax Error";
    treeNode *tSubprog = new treeNode("SUBPROG");
    analyseSubprog(tSubprog);
    if (it->name == ".")
    {
        t->child.push_back(tSubprog);
        t->child.push_back(new treeNode("."));
    }
    else
        throw "Syntax Error";

    if(it != words.end() - 1) //还有未分析的单词
        throw "Syntax Error";
}

//分程序
void SyntaxAnalyse::analyseSubprog(treeNode *t){
//-------------------------------------start
    addCode("JMP", 0, 0);
    int tmpidx = code.size() - 1;
    bool flag = false;
//-------------------------------------end
    if (it->name == "CONST")
    {
        treeNode *tConstDeclare = new treeNode("CONSTANTDECLARE");
        analyseConstDeclare(tConstDeclare);
        t->child.push_back(tConstDeclare);
    }
    if (it->name == "VAR")
    {
        treeNode *tVariableDeclare = new treeNode("VARIABLEDECLARE");
        analyseVariableDeclare(tVariableDeclare);
        t->child.push_back(tVariableDeclare);
    }
    if (it->name == "PROCEDURE")
```

```
    {
        flag = true;
        treeNode *tProcedureDeclare = new treeNode("PROCEDUREDECLARE");
        analyseProcedureDeclare(tProcedureDeclare);
        t->child.push_back(tProcedureDeclare);
    }
//---------------------------------------start
    if(!flag)
        code.erase(code.end()-1);
    else
        code[tmpidx].a = code.size();
    int varCount = 0;
    for(int i = 0; i < table.size(); i++)
    {
        if(table[i].kind == "VARIABLE" && table[i].parameter1 ==
procedure_depth)
            varCount++;
    }
    addCode("INT", 0, varCount+3);
//-----------------------------------------------end
    treeNode *tSentence = new treeNode("SENTENCE");
    analyseSentence(tSentence);
    t->child.push_back(tSentence);
//---------------------------------------start
    addCode("OPR", 0, 0);
//---------------------------------------end
}

//常量说明部分
void SyntaxAnalyse::analyseConstDeclare(treeNode *t){
    if (it->name == "CONST")
    {
        it++;
        treeNode *tConstDefine = new treeNode("CONSTANTDEFINE");
        analyseConstDefine(tConstDefine);
        t->child.push_back(new treeNode("CONST"));
        t->child.push_back(tConstDefine);
        while (it->name == ",")
        {
            it++;
            tConstDefine = new treeNode("CONSTANTDEFINE");
            analyseConstDefine(tConstDefine);
            t->child.push_back(new treeNode("COMMA"));
            t->child.push_back(tConstDefine);
```

```
        }
        if (it->name == ";")
        {
            it++;
            t->child.push_back(new treeNode(";"));
        }
        else
            throw "Syntax Error";
    }
    else
        throw "Syntax Error";
}

//常量定义
void SyntaxAnalyse::analyseConstDefine(treeNode *t){
//-------------------------------------------------start
    for(int i = 0; i < table.size(); i++)
    {
        if(table[i].name == it->name)
            throw "GenCode Error";
    }
    addTable(it->name, "CONSTANT", 0, 0);
//-------------------------------------------------end
    treeNode *tIdentifier = new treeNode("IDENTIFIER?");
    analyseIdentifier(tIdentifier);
    t->child.push_back(tIdentifier);
    if (it->name == "=")
    {
        it++;
//-------------------------------------------------start
        table[table.size()-1].parameter1 = atoi(it->name.c_str());
//-------------------------------------------------end
        treeNode *tUnsignedInteger = new treeNode("UNSIGNEDINTEGER?");
        analyseUnsignedInteger(tUnsignedInteger);
        t->child.push_back(new treeNode("="));
        t->child.push_back(tUnsignedInteger);
    }
    else
        throw "Syntax Error";
}

//无符号整数
void SyntaxAnalyse::analyseUnsignedInteger(treeNode *t){
    if (it->sym == "NUMBER")
```

```
    {
        t->element = it->name;
        it++;
    }
    else
        throw "Syntax Error";
}

//变量说明部分
void SyntaxAnalyse::analyseVariableDeclare(treeNode *t){
    int dx = 3;
    if (it->name == "VAR")
    {
        it++;
//-------------------------------------------------start
        int idx = getIndexbyName(procedure_depth,it->name);
        if(idx != -1 && table[idx].kind == "CONSTANT")
            throw "GenCode Error";
        addTable(it->name, "VARIABLE", procedure_depth, dx++);
//-------------------------------------------------end
        treeNode *tIdentifier = new treeNode("IDENTIFIER?");
        analyseIdentifier(tIdentifier);
        t->child.push_back(new treeNode("VAR"));
        t->child.push_back(tIdentifier);
        while (it->name == ",")
        {
            it++;
//-------------------------------------------------start
            int idx = getIndexbyName(procedure_depth,it->name);
            if(idx != -1 && table[idx].kind == "CONSTANT")
                throw "GenCode Error";
            addTable(it->name, "VARIABLE", procedure_depth, dx++);
//-------------------------------------------------end
            tIdentifier = new treeNode("IDENTIFIER?");
            analyseIdentifier(tIdentifier);
            t->child.push_back(new treeNode("COMMA"));
            t->child.push_back(tIdentifier);
        }
        if (it->name == ";")
        {
            it++;
            t->child.push_back(new treeNode(";"));
        }
        else
```

```
            throw "Syntax Error";
    }
    else
        throw "Syntax Error";
}

//标识符
void SyntaxAnalyse::analyseIdentifier(treeNode *t){
    if (it->sym == "IDENTIFIER")
    {
        t->element = it->name;
        it++;
    }
    else
        throw "Syntax Error";
}

//过程说明部分
void SyntaxAnalyse::analyseProcedureDeclare(treeNode *t){
    procedure_depth++;
    treeNode *tProcedureHead = new treeNode("PROCEDUREHEAD");
    analyseProcedureHead(tProcedureHead);
    t->child.push_back(tProcedureHead);
    treeNode *tSubprog = new treeNode("SUBPROG");


    if(procedure_depth > 3) //过程嵌套深度不能超过 3
        throw "Syntax Error";
    analyseSubprog(tSubprog);
    procedure_depth--;

    t->child.push_back(tSubprog);
    if(it->name == ";")
    {
        it++;
        t->child.push_back(new treeNode(";"));
    }
    else
        throw "Syntax Error";

    if(it->name == "PROCEDURE")
    {
        treeNode *tProcedureDeclare = new treeNode("PROCEDUREDECLARE");
        analyseProcedureDeclare(tProcedureDeclare);
```

```
        t->child.push_back(tProcedureDeclare);
    }
}

//过程首部
void SyntaxAnalyse::analyseProcedureHead(treeNode *t){
    if (it->name == "PROCEDURE")
    {
        it++;
//-------------------------------------------------start
        addTable(it->name, "PROCEDURE", procedure_depth-1, code.size());
//-------------------------------------------------end
        treeNode *tIdentifier = new treeNode("IDENTIFIER?");
        analyseIdentifier(tIdentifier);
        t->child.push_back(new treeNode("PROCEDURE"));
        t->child.push_back(tIdentifier);
        if (it->name == ";")
        {
            it++;
            t->child.push_back(new treeNode(";"));
        }
        else
            throw "Syntax Error";
    }
    else
        throw "Syntax Error";
}

//语句
void SyntaxAnalyse::analyseSentence(treeNode *t){
    if (it->sym == "IDENTIFIER")
    {
        treeNode *tAssignment = new treeNode("ASSIGNMENT");
        analyseAssignment(tAssignment);
        t->child.push_back(tAssignment);
    }
    else if (it->name == "IF")
    {
        treeNode *tIfSentence = new treeNode("IFSENTENCE");
        analyseIfSentence(tIfSentence);
        t->child.push_back(tIfSentence);
    }
    else if (it->name == "WHILE")
    {
```

```
        treeNode *tWhileSentence = new treeNode("WHILESENTENCE");
        analyseWhileSentence(tWhileSentence);
        t->child.push_back(tWhileSentence);
    }
    else if (it->name == "CALL")
    {
        treeNode *tCallSentence = new treeNode("CALLSENTENCE");
        analyseCallSentence(tCallSentence);
        t->child.push_back(tCallSentence);
    }
    else if (it->name == "READ")
    {
        treeNode *tReadSentence = new treeNode("READSENTENCE");
        analyseReadSentence(tReadSentence);
        t->child.push_back(tReadSentence);
    }
    else if (it->name == "WRITE")
    {
        treeNode *tWriteSentence = new treeNode("WRITESENTENCE");
        analyseWriteSentence(tWriteSentence);
        t->child.push_back(tWriteSentence);
    }
    else if (it->name == "BEGIN")
    {
        treeNode *tCombined = new treeNode("COMBINED");
        analyseCombined(tCombined);
        t->child.push_back(tCombined);
    }
    else if (it->name == "END" || it->name == "." || it->name == ";"){
        treeNode *tEmpty = new treeNode("EMPTY");
        analyseEmpty(tEmpty);
        t->child.push_back(tEmpty);
    }else
        throw "Syntax Error";
}

//赋值语句
void SyntaxAnalyse::analyseAssignment(treeNode *t){
    if (it->sym == "IDENTIFIER")
    {
        string idname = it->name;
        t->child.push_back(new treeNode(it->name));
        it++;
        if (it->name == ":=")
```

```
        {
            t->child.push_back(new treeNode(":="));
            it++;
            treeNode *tExpression = new treeNode("EXPRESSION");
            analyseExpression(tExpression);
            t->child.push_back(tExpression);
//-----------------------------------
        int index = getIndexbyName(procedure_depth,idname);
        if(index == -1)
            throw "GenCode Error";
        if(table[index].kind != "VARIABLE")
            throw "GenCode Error";
        addCode("STO", abs(table[index].parameter1 - procedure_depth),
table[index].parameter2);
//-----------------------------------
        }
        else
            throw "Syntax Error";
    }
    else
        throw "Syntax Error";
}

//复合语句
void SyntaxAnalyse::analyseCombined(treeNode *t){
    if(it->name == "BEGIN"){
        it++;
        t->child.push_back(new treeNode("BEGIN"));
        treeNode *tSentence = new treeNode("SENTENCE");
        analyseSentence(tSentence);
        t->child.push_back(tSentence);
        while(it->name == ";"){
            t->child.push_back(new treeNode(";"));
            it++;
            tSentence = new treeNode("SENTENCE");
            analyseSentence(tSentence);
            t->child.push_back(tSentence);
        }
        if(it->name == "END"){
            it++;
            t->child.push_back(new treeNode("END"));
        }
        else
            throw "Syntax Error";
```

```
    }
    else
        throw "Syntax Error";
}

//条件
void SyntaxAnalyse::analyseCondition(treeNode *t){
    if (it->name == "ODD")
    {
        it++;
        treeNode *tExpression = new treeNode("EXPRESSION");
        analyseExpression(tExpression);
        t->child.push_back(new treeNode("ODD"));
        t->child.push_back(tExpression);
//-----------------------------------
        addCode("OPR", 0, getIndexbyOPR("ODD"));
//-----------------------------------
    }
    else
    {
        treeNode *tExpression1 = new treeNode("EXPRESSION");
        analyseExpression(tExpression1);
        t->child.push_back(tExpression1);
//-----------------------------------
        string opr = it->name;
//-----------------------------------
        treeNode *tRelationOperator = new treeNode("RELATIONOPERATOR?");
        analyseRelationOperator(tRelationOperator);
        t->child.push_back(tRelationOperator);
        treeNode *tExpression2 = new treeNode("EXPRESSION");
        analyseExpression(tExpression2);
        t->child.push_back(tExpression2);
//-----------------------------------
        addCode("OPR", 0, getIndexbyOPR(opr));
//-----------------------------------
    }
}

//表达式
void SyntaxAnalyse::analyseExpression(treeNode *t){
    string opr = it->name;
    if (it->name == "+" || it->name == "-")
    {
        addCode("LIT", 0, 0);
```

```
        t->child.push_back(new treeNode(it->name));
        it++;
    }
    treeNode *tItem = new treeNode("ITEM");
    analyseItem(tItem);
    t->child.push_back(tItem);
    //-----------------------------------
    if (opr == "-" || opr == "+"){
        addCode("OPR", 0, getIndexbyOPR(opr));
    }
    //-----------------------------------
    while (it->name == "+" || it->name == "-")
    {
//-----------------------------------
        string opr = it->name;
//-----------------------------------
        treeNode *tAddSubOperator = new treeNode("ADD_SUB_OPERATOR?");
        analyseAddSubOperator(tAddSubOperator);
        t->child.push_back(tAddSubOperator);

        tItem = new treeNode("ITEM");
        analyseItem(tItem);
        t->child.push_back(tItem);
//-----------------------------------
        addCode("OPR", 0, getIndexbyOPR(opr));
//-----------------------------------
    }
}

//项
void SyntaxAnalyse::analyseItem(treeNode *t){
    treeNode *tFactor = new treeNode("FACTOR");
    analyseFactor(tFactor);
    t->child.push_back(tFactor);
    while (it->name == "*" || it->name == "/")
    {
//-----------------------------------
        string opr = it->name;
//-----------------------------------
        treeNode *tMulDivOperator = new treeNode("MUL_DIV_OPERATOR?");
        analyseMulDivOperator(tMulDivOperator);
        t->child.push_back(tMulDivOperator);

        tFactor = new treeNode("FACTOR");
```

```
        analyseFactor(tFactor);
        t->child.push_back(tFactor);
//-----------------------------------
        addCode("OPR", 0, getIndexbyOPR(opr));
//-----------------------------------
    }
}

//因子
void SyntaxAnalyse::analyseFactor(treeNode *t){
    if (it->sym == "IDENTIFIER")
    {
//-----------------------------------
        int index = getIndexbyName(procedure_depth,it->name);
        if(index == -1)
            throw "GenCode Error";
        if(table[index].kind == "CONSTANT")
            addCode("LIT", 0, table[index].parameter1);
        else if(table[index].kind == "VARIABLE")
            addCode("LOD", abs(table[index].parameter1 - procedure_depth),
table[index].parameter2);
        else
            throw "GenCode Error";
//-----------------------------------
        t->child.push_back(new treeNode(it->name));
        it++;
    }
    else if (it->sym == "NUMBER")
    {
//-----------------------------------
        addCode("LIT", 0, stoi(it->name));
//-----------------------------------
        t->child.push_back(new treeNode(it->name));
        it++;
    }
    else if (it->name == "(")
    {
        it++;
        t->child.push_back(new treeNode("LP"));
        treeNode *tExpression = new treeNode("EXPRESSION");
        analyseExpression(tExpression);
        t->child.push_back(tExpression);
        if (it->name == ")")
        {
```

```
            it++;
            t->child.push_back(new treeNode("RP"));
        }
        else
            throw "Syntax Error";
    }
    else
        throw "Syntax Error";
}

//加减运算符
void SyntaxAnalyse::analyseAddSubOperator(treeNode *t){
    if (it->name == "+" || it->name == "-")
    {
        t->element = it->name;
        it++;
    }
    else
        throw "Syntax Error";
}
//乘除运算符
void SyntaxAnalyse::analyseMulDivOperator(treeNode *t){
    if (it->name == "*" || it->name == "/")
    {
        t->element = it->name;
        it++;
    }
    else
        throw "Syntax Error";
}
//关系运算符
void SyntaxAnalyse::analyseRelationOperator(treeNode *t){
    if (it->name == "=" || it->name == "#" || it->name == "<" || it->name
== "<=" || it->name == ">" || it->name == ">=")
    {
        t->element = it->name;
        it++;
    }
    else
        throw "Syntax Error";
}

//-------------------------------------------------------
//条件语句
```

```
void SyntaxAnalyse::analyseIfSentence(treeNode *t){
    if (it->name == "IF")
    {
        it++;
        treeNode *tCondition = new treeNode("CONDITION");
        analyseCondition(tCondition);
        t->child.push_back(new treeNode("IF"));
        t->child.push_back(tCondition);
//-----------------------------------
        addCode("JPC", 0, 0);
        int index = code.size() - 1;
//-----------------------------------
        if (it->name == "THEN")
        {
            it++;
            treeNode *tSentence = new treeNode("SENTENCE");
            analyseSentence(tSentence);
            t->child.push_back(new treeNode("THEN"));
            t->child.push_back(tSentence);
//-----------------------------------start
            code[index].a = code.size();
//-----------------------------------end
        }
        else
            throw "Syntax Error";
    }
    else
        throw "Syntax Error";
}

//过程调用语句
void SyntaxAnalyse::analyseCallSentence(treeNode *t){
    if (it->name == "CALL")
    {
        it++;
//-----------------------------------start
        int tmpidx = -1;
        for(int i = table.size() - 1; i >= 0; i--)
        {
            if(table[i].name == it->name && table[i].parameter1 <=
procedure_depth &&table[i].kind=="PROCEDURE")
            {
                tmpidx = i;
                break;
```

```
            }
        }
        if(tmpidx == -1)
            throw "GenCode Error";
        int tmpaddr = table[tmpidx].parameter2;
        addCode("CAL", abs(table[tmpidx].parameter1 - procedure_depth),
tmpaddr);
//-----------------------------------end
        treeNode *tIdentifier= new treeNode("IDENTIFIER?");
        analyseIdentifier(tIdentifier);
        t->child.push_back(new treeNode("CALL"));
        t->child.push_back(tIdentifier);
    }
    else
        throw "Syntax Error";
}

//当型循环语句
void SyntaxAnalyse::analyseWhileSentence(treeNode *t){
    if (it->name == "WHILE")
    {
        it++;
//-----------------------------------start
        int tmpaddr1 = code.size();
//-----------------------------------end
        treeNode *tCondition = new treeNode("CONDITION");
        analyseCondition(tCondition);
        t->child.push_back(new treeNode("WHILE"));
        t->child.push_back(tCondition);
//-----------------------------------start
        addCode("JPC", 0, 0);
        int tmpaddr2 = code.size() - 1;
//-----------------------------------end
        if (it->name == "DO")
        {
            it++;
            treeNode *tSentence = new treeNode("SENTENCE");
            analyseSentence(tSentence);
            t->child.push_back(new treeNode("DO"));
            t->child.push_back(tSentence);
//-----------------------------------start
            addCode("JMP", 0, tmpaddr1);
            code[tmpaddr2].a = code.size();
//-----------------------------------end
```

```
        }
        else
            throw "Syntax Error";
    }
    else
        throw "Syntax Error";
}

//读语句
void SyntaxAnalyse::analyseReadSentence(treeNode *t){
    if (it->name == "READ")
    {
        it++;
        t->child.push_back(new treeNode("READ"));
        if (it->name == "(")
        {
            it++;
//-----------------------------------start
            addCode("OPR", 0, getIndexbyOPR("READ"));
            int tmpidx = getIndexbyName(procedure_depth,it->name);
            if(tmpidx == -1)
                throw "GenCode Error";
            if(table[tmpidx].kind == "CONSTANT")
                throw "GenCode Error";
            addCode("STO", abs(table[tmpidx].parameter1 -
procedure_depth), table[tmpidx].parameter2);
//-----------------------------------end
            t->child.push_back(new treeNode("LP"));
            treeNode *tIdentifier = new treeNode("IDENTIFIER?");
            analyseIdentifier(tIdentifier);
            t->child.push_back(tIdentifier);
            while (it->name == ",")
            {
                it++;
//-----------------------------------start
                addCode("OPR", 0, getIndexbyOPR("READ"));
                int tmpidx = getIndexbyName(procedure_depth,it->name);
                if(tmpidx == -1)
                    throw "GenCode Error";
                if(table[tmpidx].kind == "CONSTANT")
                    throw "GenCode Error";
                addCode("STO", abs(table[tmpidx].parameter1 -
procedure_depth), table[tmpidx].parameter2);
//-----------------------------------end
```

```
                tIdentifier = new treeNode("IDENTIFIER?");
                analyseIdentifier(tIdentifier);
                t->child.push_back(new treeNode("COMMA"));
                t->child.push_back(tIdentifier);
            }

            if (it->name == ")")
            {
                it++;
                t->child.push_back(new treeNode("RP"));
            }
            else
                throw "Syntax Error";
        }
        else
            throw "Syntax Error";
    }
    else
        throw "Syntax Error";
}

//写语句
void SyntaxAnalyse::analyseWriteSentence(treeNode *t){
    if (it->name == "WRITE")
    {
        it++;
        t->child.push_back(new treeNode("WRITE"));
        if (it->name == "(")
        {
            it++;
//-----------------------------------start
            int tmpidx = getIndexbyName(procedure_depth,it->name);
            if(tmpidx == -1)
                throw "GenCode Error";
            if(table[tmpidx].kind == "CONSTANT")
                addCode("LIT", 0, table[tmpidx].parameter1);
            else
                addCode("LOD", abs(table[tmpidx].parameter1 -
procedure_depth), table[tmpidx].parameter2);
            addCode("OPR", 0, getIndexbyOPR("WRITE"));
//-----------------------------------end
            t->child.push_back(new treeNode("LP"));

            treeNode *tIdentifier = new treeNode("IDENTIFIER?");
```

```
            analyseIdentifier(tIdentifier);
            t->child.push_back(tIdentifier);

            while (it->name == ",")
            {
                it++;
//-----------------------------------start
                tmpidx = getIndexbyName(procedure_depth,it->name);
                if(tmpidx == -1)
                    throw "GenCode Error";
                if(table[tmpidx].kind == "CONSTANT")
                    addCode("LIT", 0, table[tmpidx].parameter1);
                else
                    addCode("LOD", abs(table[tmpidx].parameter1 -
procedure_depth), table[tmpidx].parameter2);
                addCode("OPR", 0, getIndexbyOPR("WRITE"));
//-----------------------------------end
                tIdentifier = new treeNode("IDENTIFIER?");
                analyseIdentifier(tIdentifier);
                t->child.push_back(new treeNode("COMMA"));
                t->child.push_back(tIdentifier);
            }
            if (it->name == ")")
            {
                it++;
                t->child.push_back(new treeNode("RP"));
            }
            else
                throw "Syntax Error";
        }
        else
            throw "Syntax Error";
    }
    else
        throw "Syntax Error";
}

//空语句
void SyntaxAnalyse::analyseEmpty(treeNode *t){
    t->element = "EMPTY";
}
```

Interpreter.cpp

```cpp
#include <stack>
#include <vector>
#include <string>
#include <iostream>
#include <fstream>
#include "struct.hpp"
using namespace std;
class Interpreter
{
    public:
        Interpreter();
        ~Interpreter();
        void interpret(); //主要函数，完成对目标代码的解释
        void readCode(string filename); //读取目标代码
    private:
        vector<TargetCode> code; //目标代码
        const static int ret_addr = 0, dynamic_link = 1, static_link = 2;
        //栈中返回地址，动态链，静态链的相对位置
        TargetCode ir; //当前指令
        int ip = 0, sp = 0, bp = 0; //指令指针，栈顶指针，基址指针
        int stack[100000] = { 0 }; //栈
        int sp_stack[1000]; //栈顶指针栈，用于存放每个过程的栈顶指针
        int sp_top = 0; //栈顶指针栈的栈顶指针
};
Interpreter::Interpreter()
{
    ip = 0, sp = 0, bp = 0;
    sp_top = 0;
}
Interpreter::~Interpreter()
{
}
void Interpreter::readCode(string filename)
{
    ifstream fin;
    fin.open(filename);
    TargetCode temp;
    while(fin>>temp.f>>temp.l>>temp.a)
    {
        code.push_back(temp);
    }
    fin.close();
}
```

```
// LIT: l 域无效，将 a 放到栈顶
// LOD: 将于当前层层差为 l 的层，变量相对位置为 a 的变量复制到栈顶
// STO: 将栈顶内容复制到于当前层层差为 l 的层，变量相对位置为 a 的变量
// CAL: 调用过程。l 标明层差，a 表明目标程序地址
// INT: l 域无效，在栈顶分配 a 个空间
// JMP: l 域无效，无条件跳转到地址 a 执行
// JPC: l 域无效，若栈顶对应的布尔值为假（即 0）则跳转到地址 a 处执行，否则顺序执行
// OPR: l 域无效，对栈顶和栈次顶执行运算，结果存放在次顶，a=0 时为调用返回
void Interpreter::interpret()
{

    while (ip<code.size())
    {
        ir = code[ip++];
        if(ir.f == "LIT"){
            stack[sp++] = ir.a;
        }else if(ir.f == "LOD"){
            if (ir.l == 0)
                stack[sp++] = stack[bp + ir.a];
            else
            {
                int outer_bp = stack[bp + static_link];
                while (--ir.l)
                    outer_bp = stack[outer_bp + static_link];
                stack[sp++] = stack[outer_bp + ir.a];
            }
        }else if(ir.f == "STO"){
            if (ir.l == 0)
                stack[bp + ir.a] = stack[sp - 1];
            else
            {
                int outer_bp = stack[bp + static_link];
                while (--ir.l)
                    outer_bp = stack[outer_bp + static_link];
                stack[outer_bp + ir.a] = stack[sp - 1];
            }
        }else if(ir.f == "CAL"){
            stack[sp + ret_addr] = ip;
            stack[sp + dynamic_link] = bp;
            if(ir.l == 0)
                stack[sp + static_link] = bp;
            else
            {
```

```
                int outer_bp = stack[bp + static_link];
                while (--ir.l)
                    outer_bp = stack[outer_bp + static_link];
                stack[sp + static_link] = outer_bp;
            }
            ip = ir.a;
            bp = sp;
        }else if(ir.f == "INT"){
            sp_stack[sp_top++] = sp;
            sp += ir.a;
        }else if(ir.f == "JMP"){
            ip = ir.a;
        }else if (ir.f == "JPC"){
            if (stack[sp - 1] == 0)
                ip = ir.a;
            sp--;
        }else if(ir.f == "OPR"){
            //{"RETURN","ODD","READ","WRITE", "+", "-", "*", "/",
            //"=", "#", "<", "<=", ">", ">="};
            switch (ir.a)
            {
                case 0: //OPR_RET
                {
                    ip = stack[bp + ret_addr];
                    bp = stack[bp + dynamic_link];
                    sp = sp_stack[--sp_top];
                    if (sp_top <= 0)
                    {
                        return;
                    }
                    break;
                }
                case 1: //OPR_ODD
                {
                    stack[sp - 1] = stack[sp - 1] % 2;
                    break;
                }
                case 2: //OPR_READ
                {
                    scanf("%d", &stack[sp++]);
                    break;
                }
                case 3: //OPR_WRITE
                {
```

```
                printf("%d\n", stack[sp - 1]);
                sp--;
                break;
            }
            case 4: //OPR_+
            {
                stack[sp - 2] = stack[sp - 1] + stack[sp - 2];
                sp--;
                break;
            }
            case 5: //OPR_-
            {
                stack[sp - 2] = stack[sp - 2] - stack[sp - 1];
                sp--;
                break;
            }
            case 6: //OPR_*
            {
                stack[sp - 2] = stack[sp - 1] * stack[sp - 2];
                sp--;
                break;
            }
            case 7: //OPR_/
            {
                stack[sp - 2] = stack[sp - 2] / stack[sp - 1];
                sp--;
                break;
            }
            case 8: //OPR_=
            {
                stack[sp - 2] = (stack[sp - 2] == stack[sp - 1]) ;
                sp--;
                break;
            }
            case 9: //OPR_#
            {
                stack[sp - 2] = (stack[sp - 2] != stack[sp - 1]) ;
                sp--;
                break;
            }
            case 10: //OPR_<
            {
                stack[sp - 2] = (stack[sp - 2]<stack[sp - 1]);
                sp--;
```

```
                    break;
                }
                case 11: //OPR_<=
                {
                    stack[sp - 2] = (stack[sp - 2] <= stack[sp - 1]);
                    sp--;
                    break;
                }
                case 12: //OPR_>
                {
                    stack[sp - 2] = (stack[sp - 2]>stack[sp - 1]);
                    sp--;
                    break;
                }
                case 13: //OPR_>=
                {
                    stack[sp - 2] = (stack[sp - 2] >= stack[sp - 1]);
                    sp--;
                    break;
                }
                default:{
                   return;
                }
            }
        }
    }
}
```

CompilerMain.cpp

```
#include "SyntaxAnalyse.cpp"
#include <iostream>
#include <string>
using namespace std;
int main()
{
    string filename="in.txt";

    LexicalAnalyse la;
    SyntaxAnalyse sa;
    if(!la.analyse(filename)){
//        cout<<"Lexical Error"<<endl;
        return -1;
    }else{
        try{
```

```
            sa.analyse(la.words);
        }catch(...){
//            cout<<"Syntax Error"<<endl;
            return -1;
        }
        sa.printCode();
    }
    return 0;
}
```

InterpreterMain.cpp

```
#include "Interpreter.cpp"
int main(){
    Interpreter interpreter;
    interpreter.readCode("program.code");
    interpreter.interpret();
    return 0;
}
```