编译原理-PL0 语言编译器

一、 实验目的

完成一个 PL/0 语言的编译器，包括词法分析、语法分析、语义分析+目标代码生成部分，以及假想栈式计算机的解释器，以更好的掌握编译的几个阶段。

二、 实验环境

- **硬件环境：**
  - Windows10
  - 内存8G
  - 硬盘1T

- **软件环境：**
  - visual studio2017

三、 实验内容

1. 词法分析

1.1 概述

给定一个 PL/0 语言源程序，需要将其从字符流转换为词语流，为语法分析做好准备。具体来说，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。

1. **PL/0 的标识符上下文无关文法**

   <标识符> → <字母> {<字母>|<数字>}

   <字母> → A|B|C...X|Y|Z

   <数字> → 0|1|2...7|8|9

   - 标识符不会超过 10 字符长
   - 关键字不能是标识符
   - PL/0 语言不区分大小写

2. **PL/0 的关键字表**

   CONST

   VAR

   PROCEDURE

   BEGIN

END
ODD
IF
THEN
CALL
WHILE
DO
READ
WRITE

3. PL/0 的数字上下文无关文法

<无符号整数> → <数字>{<数字>}

<数字> → 0|1|2...7|8|9

- PL/0 仅支持无符号整数。因此，对于-123，你的词法应当将其分为两个词语-和 123
- 保证输入的整数的数值不会超出 32 位带符号整数能表示的范围

4. PL/0 的运算符表

=
:=
+
-
*
/
#
<
<=
>
>=

- 拼接两个字符组成的运算符中间不会存在空白符

5. PL/0 的分隔符表

;
,
.
(
)

1.2 设计思路

(一) 设计一个词法分析器类：

- 词法分析器结果结构体及词语类型：

```
enum Type { SYM, ID, NUM, OPT, SEP, END };
struct LxclInfo {
    Type type;
    string val;
    LxclInfo() {}
    LxclInfo(Type t, string v) :type(t), val(v) {}
};
```

- 方法：

  几个 DFA：

  1. 标识符　Identifier(char&)
  2. 数字　Digit(char&)
  3. 运算符　checkOpts(char&)

  几种检查函数：

  1. 关键字检查　checkKeys()
  2. 分隔符检查　checkSeps(char&)
  3. 终止状态正确性检查　checkState()

  输出词语　output()

  初始化当前各种变量　clear()

  启动词法分析：run()

  变量：

  1. 当前词语字符串：idntf, dgt, opt
  2. 当前各自动机的状态：id_state, dgt_state, opt_state
  3. 当前各自动机的正确性：crct_id, crct_dgt, crct_opt
  4. 词法分析错误：lxcError
  5. 词语流结果：vector<LxclInfo> outputs
  6. 几种表：关键字、运算符（仅单个字符，及包括多个字符两张）、分隔符

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

    void run();                     // begin(main)
    void restart();

    bool checkKeys();               // sym
    bool Identifier(char&);         // id
    bool Digit(char&);              // num
    bool checkOpts(char&);          // opt
    bool checkSeps(char&);          // sep

    void checkState();
    void output();
    void clear();

    vector<LxclInfo> getRes(bool &err) { err = lxcError; return outputs; }

private:
    string idntf, dgt, opt;
    int id_state, dgt_state, opt_state;     // state
    bool crct_id, crct_dgt, crct_opt;       // correctness
    bool lxcError;                          // lexical error
    vector<LxclInfo> outputs;               // results

    const vector<string> keywords = { "CONST", "VAR", "PROCEDURE", "BEGIN", "END", "ODD", "IF", "THEN", "CALL", "WHILE", "DO", "READ", "WRITE" };
    const vector<string> operators = { "=", ":=", "+", "-", "*", "/", "#", "<", "<=", ">", ">=" };
    const vector<char> separators = { ',', ';', '.', '(', ')' };
    const vector<char> opts_stl = { '=', ':', '+', '-', '*', '/', '#', '<', '>' };
};
```

(二)部分实现细节

- 启动词法分析　run()

1. 以单个字符形式读入（getchar），并将每个字符转换为对应大写字符 c；
2. 当字符 c 为空白符（\n, \t, ‘ ’）或分隔符时，输出；
3. 否则，同时将该字符送入 3 个自动机（Identifier, Digit, checkOpts），分别获取正确性以判断当前可能词语类别（标识符、数字、分隔符第一个字符互斥）；
4. 下一步根据词语类型，不断处理字符，直至自动机错误，之后判断终态正误，并输出。

```
void LexicalAls::run() {
    char c;
    c = getchar();
    while (c != EOF) {
        c = toupper(c);
        if (c == ' ' || c == '\n' || c == '\t' || checkSeps(c)) {  // refresh: output&clear or nothing
            checkState();
            output();
            clear();
            if (checkSeps(c)) outputs.emplace_back(SEP, string(1, c));
            c = getchar();
            continue;
        }

        if (crct_opt) {  //opt=1
            crct_opt = checkOpts(c);
        }
        if (crct_id) { //id=2
            crct_id = Identifier(c);
        }
        if (crct_dgt) { //dgt=3
            crct_dgt = Digit(c);
        }

        int opt = 0;
        if (crct_opt) opt = 1;
        if (crct_id) opt = 2;
        if (crct_dgt) opt = 3;

        switch (opt) {
        case 0:
            lxcError = true;
            c = getchar();
            break;
        case 1:
            while (crct_opt) {
                c = getchar();
                c = toupper(c);
                crct_opt = checkOpts(c);
            }
            crct_opt = true;
            break;
        case 2:
            while (crct_id) {
                c = getchar();
                c = toupper(c);
                crct_id = Identifier(c);
            }
            crct_id = true;
            break;
        case 3:
            while (crct_dgt) {
                c = getchar();
                c = toupper(c);
                crct_dgt = Digit(c);
            }
            if (!isalpha(c)) { crct_dgt = true; }
            else { c = getchar(); }
            break;
        }

        //output
        checkState();
        output();
        clear();
    }

    //printout();
}
```

- 几个 DFA（手动化简）

```
bool LexicalAls::Identifier(char& c) {
    bool crct = true;
    if (!isalnum(c)) { crct = false; }
    else {
        switch (id_state) {
        case 0:
            if (isalpha(c)) {
                idntf += c;
                id_state += 1;
            }
            else {
                crct = false;
            }
            break;
        case 1:
            idntf += c;
            id_state += 1;
            break;
        case 2:
            idntf += c;
            break;
        }
    }

    return crct;
}
```

```
bool LexicalAls::Digit(char& c) {
    bool crct = true;
    if (!isdigit(c)) { crct = false; }
    else {
        switch (dgt_state) {
        case 0:
            dgt += c;
            dgt_state += 1;
            break;
        case 1:
            dgt += c;
            break;
        }
    }

    //if(state != 1) { crct = false; }
    return crct;
}
```

```cpp
bool LexicalAls::checkOpts(char& c) {
    bool crct = false;
    switch (opt_state) {
    case 0:
        for (auto opt1 : opts_st1) {
            if (opt1 == c) {
                crct = true;
                opt += c;
                opt_state += 1;
                if (c == '<' || c == '>' || c == ':') {
                    opt_state += 1;
                }
                break;
            }
        }
        break;
    case 1:
        crct = false; // wrong
        break;
    case 2:
        if (c == '=') {
            opt += c;
            crct = true;
        }
        break;
    }
    return crct;
}
```

2. 语法分析

2.1 概述

使用递归子程序下降法，对词语流进行分析，并生成一颗语法树，或者宣告语法错误。

- PLO 语言的上下文无关文法

<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
```

- 细节

  允许嵌套定义函数，但嵌套不得超过三层；

  语法分析后，如果语法正确，应当生成以<程序>为根节点的抽象语法树；

2.2 设计思路

(一) 设计一个语法分析类

- 语法树节点结构体

```
/* Syntax Tree */
struct Node {
    string val;
    vector<Node*> children;

    Node() {}
    Node(string v) :val(v) {}
    void InsertChild(Node* child) {
        children.push_back(child);
    }
};
```

- 方法

  1. 针对上下文文法左方的部分各设计一个方法，并且返回值均为当前方法已构造好的语法树子树的根节点，若为 NULL 则表示构建失败、此方法没有构建成功节点：

```
Node* PROGRAM();                // 程序
Node* SUBPROG(int px);          // 分程序
Node* CONSTANTDECLARE();        // 常量说明部分
Node* CONSTANTDEFINE();         // 常量定义
Node* VARIABLEDECLARE();        // 变量说明部分
Node* PROCEDUREDECLARE();       // 过程说明部分
Node* PROCEDUREHEAD();          // 过程首部

Node* SENTENCE();               // 语句
Node* ASSIGNMENT();             // 赋值语句
Node* COMBINED();               // 复合语句
Node* IFSENTENCE();             // 条件语句
Node* CALLSENTENCE();           // 过程调用语句
Node* WHILESENTENCE();          // 当型循环语句
Node* READSENTENCE();           // 读语句
Node* WRITESENTENCE();          // 写语句
Node* EMPTY();                  // 空语句

Node* CONDITION();              // 条件
Node* EXPRESSION();             // 表达式
Node* ITEM();                   // 项
Node* FACTOR();                 // 因子
```

2. 启动语法分析器并输出

```
void SyntaxAls::run() {
    Node* root = PROGRAM();
    if (!synErr) {
        layout();
    }
}
```

- 变量
  1. 词法分析结果：vector<LxclInfo> _wordseq;
  2. 索引当前词：int ptr;
  3. 语法分析错误：bool synErr;
  4. 语法分析树（程序的根节点）：Node *ASTroot;
  5. procedure 嵌套计数：int cntPro;

(二) 程序思路及实现细节

在获取词语流之后，在末尾加 $ 符标识 END 类型。递归子程序下降即根据上下文无关文法直接 **处理词语（判断类型+构造节点） + 调用对应函数执行**，若能成功分析完对应的文法，即返回节点，否则返回 NULL，语法分析错误。

被调用方法返回的节点插入到调用自己的孩子中，符合递归结构并能成功构造语法分析树。程序框图即上下文无关文法结构。

递归下降实现细节

■ **程序+声明部分：**

```
/**
* <程序>→<分程序>.
*/
Node* SyntaxAls::PROGRAM() {
    ASTroot = new Node("PROGRAM");
    ASTroot->InsertChild(SUBPROG());
    if (_wordseq[ptr].type == SEP && _wordseq[ptr].val == ".") {
        ASTroot->InsertChild(new Node("."));
        ++ptr;

        //
        if (_wordseq[ptr].type == END && _wordseq[ptr].val == "$")
            return ASTroot;
    }
    synErr = true;
    return NULL;
}
/**
* <分程序>→ [<常量说明部分>][<变量说明部分>][<过程说明部分>]<语句>
*/
Node* SyntaxAls::SUBPROG() {
    Node* node = new Node("SUBPROG");
    if (_wordseq[ptr].type == SYM && _wordseq[ptr].val == "CONST") {
        node->InsertChild(CONSTANTDECLARE());
    }
    if (_wordseq[ptr].type == SYM && _wordseq[ptr].val == "VAR") {
        node->InsertChild(VARIABLEDECLARE());
    }
    if (_wordseq[ptr].type == SYM && _wordseq[ptr].val == "PROCEDURE") {
        node->InsertChild(PROCEDUREDECLARE());
    }
    node->InsertChild(SENTENCE());
    return node;
}
```

```cpp
/**
* <常量说明部分> → CONST<常量定义>{ ,<常量定义>};
*/
Node* SyntaxAls::CONSTANTDECLARE() {
    Node* node = new Node("CONSTANTDECLARE");
    if (_wordseq[ptr].type == SYM && _wordseq[ptr].val == "CONST") {
        node->InsertChild(new Node("CONST"));
        ++ptr;
        node->InsertChild(CONSTANTDEFINE());
        while (_wordseq[ptr].type == SEP && _wordseq[ptr].val == ",") {
            node->InsertChild(new Node("COMMA"));
            ++ptr;
            node->InsertChild(CONSTANTDEFINE());
        }
        if (_wordseq[ptr].type == SEP && _wordseq[ptr].val == ";") {
            node->InsertChild(new Node(";"));
            ++ptr;
            return node;
        }
    }
    synErr = true;
    return NULL;
}
/**
* <常量定义> → <标识符>=<无符号整数>
*/
Node* SyntaxAls::CONSTANTDEFINE() {
    if (_wordseq[ptr].type == ID) {
        Node* node = new Node("CONSTANTDEFINE");
        node->InsertChild(new Node(_wordseq[ptr].val));
        ++ptr;
        if (_wordseq[ptr].type == OPT && _wordseq[ptr].val == "=") {
            node->InsertChild(new Node("="));
            ++ptr;
            if (_wordseq[ptr].type == NUM) {
                node->InsertChild(new Node(_wordseq[ptr].val));
                ++ptr;
                return node;
            }
        }
    }
    synErr = true;
    return NULL;
}
/**
```

```
/**
 * <变量说明部分> → VAR<标识符>{ ,<标识符>};
 */
Node* SyntaxAls::VARIABLEDECLARE() {
    Node* node = new Node("VARIABLEDECLARE");
    if (_wordseq[ptr].type == SYM && _wordseq[ptr].val == "VAR") {
        node->InsertChild(new Node("VAR"));
        ++ptr;
        if (_wordseq[ptr].type == ID) {
            node->InsertChild(new Node(_wordseq[ptr].val));
            ++ptr;
            while (_wordseq[ptr].type == SEP && _wordseq[ptr].val == ",") {
                node->InsertChild(new Node("COMMA"));
                ++ptr;
                if (_wordseq[ptr].type = ID) {
                    node->InsertChild(new Node(_wordseq[ptr].val));
                    ++ptr;
                }
                else {
                    synErr = true;
                    return NULL;
                }
            }
            if (_wordseq[ptr].type == SEP && _wordseq[ptr].val == ";") {
                node->InsertChild(new Node(";"));
                ++ptr;
                return node;
            }
        }
    }
    synErr = true;
    return NULL;
}
```

```
/**
 * <过程说明部分> → <过程首部><分程序>;{<过程说明部分>}
 */
Node* SyntaxAls::PROCEDUREDECLARE() {
    if (_wordseq[ptr].type == SYM && _wordseq[ptr].val == "PROCEDURE") {
        cntPro++;  // 创建一层
        if (cntPro > 3) {  // 不大于3层procedure
            synErr = true;
            return NULL;
        }
        Node* node = new Node("PROCEDUREDECLARE");
        node->InsertChild(PROCEDUREHEAD());
        node->InsertChild(SUBPROG());
        if (_wordseq[ptr].type == SEP && _wordseq[ptr].val == ";") {
            node->InsertChild(new Node(";"));
            cntPro--;  // 分号;完成一层
            ++ptr;
            while (_wordseq[ptr].type == SYM && _wordseq[ptr].val == "PROCEDURE") {
                node->InsertChild(PROCEDUREDECLARE());
            }
            return node;
        }
    }
    synErr = true;
    return NULL;
}
```

```
/**
* <过程首部> → PROCEDURE <标识符>;
*/
Node* SyntaxAls::PROCEDUREHEAD() {
    Node* node = new Node("PROCEDUREHEAD");
    if (_wordseq[ptr].type == SYM && _wordseq[ptr].val == "PROCEDURE") {
        node->InsertChild(new Node("PROCEDURE"));
        ++ptr;
        if (_wordseq[ptr].type == ID) {
            node->InsertChild(new Node(_wordseq[ptr].val));
            ++ptr;
            if (_wordseq[ptr].type == SEP && _wordseq[ptr].val == ";") {
                node->InsertChild(new Node(";"));
                ++ptr;
                return node;
            }
        }
    }
    synErr = true;
    return NULL;
}
```

■ 语句部分：

```
/**
* <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>
*/
Node* SyntaxAls::SENTENCE() {
    Node* node = new Node("SENTENCE");
    if (_wordseq[ptr].type == ID) {
        node->InsertChild(ASSIGNMENT());
    }
    else if (_wordseq[ptr].type == SYM && _wordseq[ptr].val == "IF") {
        node->InsertChild(IFSENTENCE());
    }
    else if (_wordseq[ptr].type == SYM && _wordseq[ptr].val == "WHILE") {
        node->InsertChild(WHILESENTENCE());
    }
    else if (_wordseq[ptr].type == SYM && _wordseq[ptr].val == "CALL") {
        node->InsertChild(CALLSENTENCE());
    }
    else if (_wordseq[ptr].type == SYM && _wordseq[ptr].val == "READ") {
        node->InsertChild(READSENTENCE());
    }
    else if (_wordseq[ptr].type == SYM && _wordseq[ptr].val == "WRITE") {
        node->InsertChild(WRITESENTENCE());
    }
    else if (_wordseq[ptr].type == SYM && _wordseq[ptr].val == "BEGIN") {
        node->InsertChild(COMBINED());
    }
    else {
        node->InsertChild(EMPTY());
    }
    return node;
}
```

```
/**
 * <赋值语句> → <标识符>:=<表达式>
 */
Node* SyntaxAls::ASSIGNMENT() {
    Node* node = new Node("ASSIGNMENT");
    if (_wordseq[ptr].type == ID) {
        node->InsertChild(new Node(_wordseq[ptr].val));
        ++ptr;
        if (_wordseq[ptr].type == OPT && _wordseq[ptr].val == ":=") {
            node->InsertChild(new Node(_wordseq[ptr].val));
            ++ptr;
            node->InsertChild(EXPRESSION());
            return node;
        }
    }
    synErr = true;
    return NULL;
}
/**
 * <条件语句> → IF<条件>THEN<语句>
 */
Node* SyntaxAls::IFSENTENCE() {
    Node* node = new Node("IFSENTENCE");
    if (_wordseq[ptr].type == SYM && _wordseq[ptr].val == "IF") {
        node->InsertChild(new Node("IF"));
        ++ptr;
        node->InsertChild(CONDITION());
        if (_wordseq[ptr].type == SYM && _wordseq[ptr].val == "THEN") {
            node->InsertChild(new Node("THEN"));
            ++ptr;
            node->InsertChild(SENTENCE());
            return node;
        }
    }
    synErr = true;
    return NULL;
}
```

```
/**
 * <当型循环语句> → WHILE<条件>DO<语句>
 */
Node* SyntaxAls::WHILESENTENCE() {
    Node* node = new Node("WHILESENTENCE");
    if (_wordseq[ptr].type == SYM && _wordseq[ptr].val == "WHILE") {
        node->InsertChild(new Node("WHILE"));
        ++ptr;
        node->InsertChild(CONDITION());
        if (_wordseq[ptr].type == SYM && _wordseq[ptr].val == "DO") {
            node->InsertChild(new Node("DO"));
            ++ptr;
            node->InsertChild(SENTENCE());
            return node;
        }
    }
    synErr = true;
    return NULL;
}
/**
 * <过程调用语句> → CALL<标识符>
 */
Node* SyntaxAls::CALLSENTENCE() {
    Node* node = new Node("CALLSENTENCE");
    if (_wordseq[ptr].type == SYM && _wordseq[ptr].val == "CALL") {
        node->InsertChild(new Node("CALL"));
        ++ptr;
        if (_wordseq[ptr].type == ID) {
            node->InsertChild(new Node(_wordseq[ptr].val));
            ++ptr;
            return node;
        }
    }
    synErr = true;
    return NULL;
}
```

```
/**
* <读语句> → READ(<标识符>{ ,<标识符>})
*/
Node* SyntaxAls::READSENTENCE() {
    Node* node = new Node("READSENTENCE");
    if (_wordseq[ptr].type == SYM && _wordseq[ptr].val == "READ") {
        node->InsertChild(new Node("READ"));
        ++ptr;
        if (_wordseq[ptr].type == SEP && _wordseq[ptr].val == "(") {
            node->InsertChild(new Node("LP"));
            ++ptr;
            if (_wordseq[ptr].type == ID) {
                node->InsertChild(new Node(_wordseq[ptr].val));
                ++ptr;
                while (_wordseq[ptr].type == SEP && _wordseq[ptr].val == ",") {
                    node->InsertChild(new Node("COMMA"));
                    ++ptr;
                    if (_wordseq[ptr].type == ID) {
                        node->InsertChild(new Node(_wordseq[ptr].val));
                        ++ptr;
                    }
                    else {
                        synErr = true;
                        return NULL;
                    }
                }
                if (_wordseq[ptr].type == SEP && _wordseq[ptr].val == ")") {
                    node->InsertChild(new Node("RP"));
                    ++ptr;
                    return node;
                }
            }
        }

    }
    synErr = true;
    return NULL;
}
```

```
/**
 * <写语句> → WRITE(<标识符>{,<标识符>})
 */
Node* SyntaxAls::WRITESENTENCE() {
    Node* node = new Node("WRITESENTENCE");
    if (_wordseq[ptr].type == SYM && _wordseq[ptr].val == "WRITE") {
        node->InsertChild(new Node("WRITE"));
        ++ptr;
        if (_wordseq[ptr].type == SEP && _wordseq[ptr].val == "(") {
            node->InsertChild(new Node("LP"));
            ++ptr;
            if (_wordseq[ptr].type == ID) {
                node->InsertChild(new Node(_wordseq[ptr].val));
                ++ptr;
                while (_wordseq[ptr].type == SEP && _wordseq[ptr].val == ",") {
                    node->InsertChild(new Node("COMMA"));
                    ++ptr;
                    if (_wordseq[ptr].type == ID) {
                        node->InsertChild(new Node(_wordseq[ptr].val));
                        ++ptr;
                    }
                    else {
                        synErr = true;
                        return NULL;
                    }
                }
                if (_wordseq[ptr].type == SEP && _wordseq[ptr].val == ")") {
                    node->InsertChild(new Node("RP"));
                    ++ptr;
                    return node;
                }
            }
        }

    }
    synErr = true;
    return NULL;
}
```

```
/**
 * <复合语句> → BEGIN<语句>{ ;<语句>} END
 */
Node* SyntaxAls::COMBINED() {
    Node* node = new Node("COMBINED");
    if (_wordseq[ptr].type == SYM && _wordseq[ptr].val == "BEGIN") {
        node->InsertChild(new Node("BEGIN"));
        ++ptr;
        node->InsertChild(SENTENCE());
        while (_wordseq[ptr].type == SEP && _wordseq[ptr].val == ";") {
            node->InsertChild(new Node(";"));
            ++ptr;
            node->InsertChild(SENTENCE());
        }
        if (_wordseq[ptr].type == SYM && _wordseq[ptr].val == "END") {
            node->InsertChild(new Node("END"));
            ++ptr;
            return node;
        }
    }
    synErr = true;
    return NULL;
}
/**
 * <空语句> → epsilon
 */
Node* SyntaxAls::EMPTY() {
    Node* node = new Node("EMPTY");
    return node;
}
```

■ 其他

```
/**
 * <条件> → <表达式><关系运算符><表达式>|ODD<表达式>
 */
Node* SyntaxAls::CONDITION() {
    Node* node = new Node("CONDITION");
    if (_wordseq[ptr].type == SYM && _wordseq[ptr].val == "ODD") {
        node->InsertChild(new Node(_wordseq[ptr].val));
        ++ptr;
        node->InsertChild(EXPRESSION());
        return node;
    }
    else {
        node->InsertChild(EXPRESSION());
        if (_wordseq[ptr].type == OPT && (_wordseq[ptr].val == "=" || _wordseq[ptr].val == "#" || _wordseq[ptr].val == "<" || _wordseq[ptr].val == ">" || _wordseq[ptr].val == "<=" || _wordseq[ptr].val == ">=")) {
            node->InsertChild(new Node(_wordseq[ptr].val));
            ++ptr;
            node->InsertChild(EXPRESSION());
            return node;
        }
        else {
            synErr = true;
            return NULL;
        }
    }
}
/**
 * <表达式> → [+|-]<项>{<加减运算符><项>}
 */
Node* SyntaxAls::EXPRESSION() {
    Node* node = new Node("EXPRESSION");
    if (_wordseq[ptr].type == OPT && (_wordseq[ptr].val == "+" || _wordseq[ptr].val == "-")) {
        node->InsertChild(new Node(_wordseq[ptr].val));
        ++ptr;
    }
    node->InsertChild(ITEM());
    while (_wordseq[ptr].type == OPT && (_wordseq[ptr].val == "+" || _wordseq[ptr].val == "-")) {
        node->InsertChild(new Node(_wordseq[ptr].val));
        ++ptr;
        node->InsertChild(ITEM());
    }
    return node;
}
```

```
/**
 * <项> → <因子>{<乘除运算符><因子>}
 */
Node* SyntaxAls::ITEM() {
    Node* node = new Node("ITEM");
    node->InsertChild(FACTOR());
    while (_wordseq[ptr].type == OPT && (_wordseq[ptr].val == "*" || _wordseq[ptr].val == "/")) {
        node->InsertChild(new Node(_wordseq[ptr].val));
        ++ptr;
        node->InsertChild(FACTOR());
    }
    return node;
}
/**
 * <因子> → <标识符>|<无符号整数>|(<表达式>)
 */
Node* SyntaxAls::FACTOR() {
    Node* node = new Node("FACTOR");
    if (_wordseq[ptr].type == ID || _wordseq[ptr].type == NUM) {
        node->InsertChild(new Node(_wordseq[ptr].val));
        ++ptr;
        return node;
    }
    else if (_wordseq[ptr].type == SEP || _wordseq[ptr].val == "(") {
        node->InsertChild(new Node("LP"));
        ++ptr;
        node->InsertChild(EXPRESSION());
        if (_wordseq[ptr].type == SEP || _wordseq[ptr].val == ")") {
            node->InsertChild(new Node("RP"));
            ++ptr;
            return node;
        }
        synErr = true;
        return NULL;
    }
    else {
        synErr = true;
        return NULL;
    }
}
```

3.语义分析与目标代码产生

3.1 概述

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。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。整体可划分为两个部分：处理说明部分，处理语句以及代码生成。

- **处理说明部分**：构造符号表（CONSTANT, VARIABLE, PROCEDURE）

  符号表表项及其含义：

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

- **处理语句以及代码生成：**使用 PL0 八种功能码实现不同语句，并编写目标代码（f l a）
  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
- 语义错误类型
  1. 符号表插入失败（同名类型冲突）
  2. 符号表查找失败（使用未声明的变量）
     - 只给变量标识符赋值、输入（查找时带目标类型）
     - 只调用函数标识符（查找时带目标类型）
     - 不能使用函数标识符作为表达式的项或输出语句的标识符（查找时带目标类型）

3.2 设计思路

3.2.1 处理说明部分

(一) 设计一个符号表类

- 根据符号表，设计表行结构体：

```
/* Symbol Table info */
struct SymbolNode {
    string name, kind;
    int para1, para2;
    SymbolNode() {
        para1 = 0;
        para2 = 0;
    }
    SymbolNode(string n, string k, int p1, int p2) :name(n), kind(k), para1(p1), para2(p2) {}
    bool operator ==(SymbolNode &b) {
        if (this->name == b.name && this->kind == b.kind) return true;
        return false;
    }
};
```

■ 符号表类需包含：

方法：

插入表项、查找某项位置、获取表项、获取当前符号表大小、设置某项的 para2（用于回填 procedure 地址信息）

变量：

符号表向量、声明类型的 map 映射（便于代码编写）

```
class SymbolTable {
public:
    SymbolTable(){
        mp = {
            {1, "CONSTANT"},
            {2, "VARIABLE"},
            {3, "PROCEDURE"}
        };
        valid = true;
    }
    bool insert(SymbolNode n, int lev);
    int checkPos(string name, vector<int> kind, int lev);
    SymbolNode getNode(int i) { return symbolTbl[i]; }
    int getSize() { return symbolTbl.size(); }
    bool setPara2(int pos, int val) { ... }

    // for test;
    void PrintSymbolTable();

private:
    map<int, string> mp;
    vector<SymbolNode> symbolTbl;    // 符号表
    bool valid;

};
```

- 实现细节：插入表项

  需要保证没有声明类型冲突，即相对于当前声明层的高层次，没有同名、不同类型的声明情况出现，若有则插入失败，否则成功。

```
bool SymbolTable::insert(SymbolNode n, int lev) {  // check高层次是否 同名 不同类型
    vector<int> checktype;
    if (n.kind == "CONSTANT") {
        checktype = { 2,3 };
    }
    if (n.kind == "VARIABLE") {
        checktype = { 1,3 };
    }
    if (n.kind == "PROCEDURE") {
        checktype = { 1,2 };
    }
    int pos = checkPos(n.name, checktype, lev);
    if (pos >= 0) return false;
    symbolTbl.push_back(n);
    return true;
}
```

- 实现细节：查找某项位置

  根据访问细节，符号表需要从后往前查找；

  1. **查找成功：**仅当目标与表项的名字、类型相同，且当前查找的目标层次 大于等于表项所在层次。若遍历完表项没有成功，即失败，返回-1.
  2. **查找过程中的目标层次：**需要有“升层”的操作。因为只能访问自己及包含自己所在层次的各个声明项。随着表项号减小，声明越靠前，当遇到层次比当前层数还小的量时，说明已到达某个包含自己的外层，需要更新目标层次。

```
int SymbolTable::checkPos(string name, vector<int> kind, int lev) {
    int levSr = lev;
    int size = symbolTbl.size();
    for (int i = size - 1; i >= 0; i--) {
        bool chk = false;
        for (auto kk:kind) {
            if (mp[kk] == symbolTbl[i].kind)
                chk = true;
        }

        int laim;
        if (symbolTbl[i].kind == "CONSTANT") laim = symbolTbl[i].para2;
        else laim = symbolTbl[i].para1;

        if (name == symbolTbl[i].name && chk) {

            if (levSr >= laim) {
                return i;
            }
        }
        else {
            if (levSr > laim) {
                levSr = laim;
            }
        }
    }
    return -1;
}
```

(二) 处理 CONST、VAR 说明

根据符号表项含义，在对应语法分析的对应位置填充（在常量定义函数部分 CONSTANTDEFINE、变量说明 VARIABLEDECLARE 函数中），最后在返回语法节点之前插入表项，若不成功（返回值-1）则语义错误。

(三) 处理 PROCEDURE 说明

根据符号表项含义，procedure 的 para2 应该存储该函数第一行代码的下标位置，因此需要等待反填。

根据文法：<过程说明部分> → <过程首部><分程序>;{<过程说明部分>}，在过程首部 PROCEDUREHEAD 中填充表项并插入后，向分程序函数传递参数，描述该符号在符号表项中的下标位置。

在子程序最后（声明之后、语句之前），反填该表项位置的 para2 为当前代码条数，并且此时也需要反填等待此地址信息的 call 语句 a 域（内部调用外部的情况，生成 call 时 para2 还未反填）。

3.2.2 处理语句以及代码生成

对每种语句都需要生成目标代码，并且需要在程序开始时生成一条跳转指令（第一句），使程序从 main 的第一条代码开始执行，跳转位置等待反填。

方便起见，预先规定整数与代码对应值如下：

1- LIT
2- LOD
3- STO
4- CAL
5- INT
6- JMP
7- JPC
8- OPR

整数与 opr 的 a 域对应值如下：

\+ - * / = # < <= > >=
1 2 3 4 5 6 7 8 9 10
11(单目运算-)
12(条件语句 odd 判奇运算)
13(读语句)
14(写语句)

目标代码结构体：

```
struct TgtLan {
    int func, lyr, amt;
    TgtLan() {
        func = 0, lyr = 0, amt = 0;
    }
    TgtLan(int f, int l, int a):func(f), lyr(l), amt(a) { }
};
```

补充完善语义分析类成员：

```
SymbolTable symbolTbl;          // 符号表
int dx;                         // 在procedure和构造函数中 = 3;  and +1 when var declared
bool smnErr;                    // 语义错误

vector<TgtLan> code;            // 目标代码
int codeLine;                   // 跟踪目标代码行号（==code.size()-1）
void emit(int f, int l, int a) { code.emplace_back(f, l, a); }      // 写入目标代码

map<int, string> mp_ins;
vector<pair<int, SymbolNode> > incmpltCode;
```

(一) 在 SUBPROGRAM 分程序时候需要的操作

1. 开辟空间（数据区）
   INT（DX=3），有 x 个 VAR，DX+x：`INT 0 DX+x`
2. 反填第一句 jmp 的 a 域；procedure 声明符号表项的 para2

(二) 赋值语句 - 目标代码

1. **<赋值语句> → <标识符>:=<表达式>**
   将表达式的值（在栈顶）赋给标识符
   - 在符号表中查找标识符 name, para1(lev), para2(a)
   - 目标代码 `STO lev-para1 para2`
2. **<表达式> → [+|-]<项>{<加减运算符><项>}**
   - [+|-]<项>：仅当有-才 `OPR 0 11`（单目 - 运算，+不需要）
   - <加减运算符><项>：`OPR 0 1` 或 `OPR 0 2`
3. **<项> → <因子>{<乘除运算符><因子>}**
   - 乘除：`OPR 0 3` 或 `OPR 0 4`
4. **<因子> → <标识符>|<无符号整数>|(<表达式>)**
   - 无符号整数：`LIT 0 整数`
   - 标识符：
     1. 在符号表中查找标识符 name, kind, para1(lev/val), para2(a)
     2. 分类 kind
     + const：`LIT 0 para1`
     + var：`LOD lev-para1 para2`

(三) 条件语句 - 目标代码

1. **<条件语句> → if<条件>then<语句>**
   - `JPC 0 a` 条件非真跳转到代码 a

   注：此处 JPC 的 a 需要在生成所有<语句>之后反填代码行号
2. **<条件> → <表达式><关系运算符><表达式>|odd<表达式>**
   - <表达式><关系运算符><表达式>
     在两个表达式之后：`OPR 0 关系运算符[种类见上]`
   - odd<表达式>：`OPR 0 12`

(四) 过程调用语句 - 目标代码

- **<过程调用语句> → call<标识符>**
  1. 在符号表中查找标识符 name, kind, para1(lev/val), para2(a)
  2. `CAL lev-para1 para2`

(五) 当型循环语句 - 目标代码

- **<当型循环语句> → while<条件>do<语句>**
  1. <条件>之后：`JPC 0 a` 条件非真跳转语句 a（while 之外的位置），否则顺序执行。此处 JPC 的 a 需要在生成所有<语句>之后反填 a 的代码行号
  2. 语句之后循环回去：`JMP 0 a` 无条件跳转 到<条件>代码位置 a
  -

(六) 读语句 - 目标代码

- **<读语句> → read(<标识符>{ ，<标识符>})**
  1. 在符号表中查找标识符 name, kind, para1(lev/val), para2(a)
  2. `OPR 0 13`：读指令 cin
  3. `(STO lev-para1 para2)`：存值

(七) 写语句 - 目标代码

- **<写语句> → write(<标识符>{，<标识符>})**
  1. 在符号表中查找标识符 name, kind, para1(lev/val), para2(a)
  2. `LIT lev-para1 para2` 或 `LOD lev-para1 para2`：读值
  3. `OPR 0 14`：写指令 cout

3.2.3 样例代码及其生成符号表、目标代码（含解释说明）

样例代码 1：

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

符号表 1

```
=====================================================
Name      Kind      Para1     Para2
=====================================================
A     CONSTANT      10   0
B     VARIABLE      0     3
```

```
C     VARIABLE      0     4
P     PROCEDURE     0     1
```

目标代码 1

```
==================================
odr f     l     a
==================================
0     JMP 0     7     //  跳转到 main 第一条    code-7

1     INT 0     3     // p 第一条  dx = 3
2     LOD 1     3     // b
3     LIT 0     10    // a
4     OPR 0     1     // +
5     STO 1     4     // c:=
6     OPR 0     0     // end p  返回

7     INT 0     5     // main 第一条  dx+b,c = 5
8     OPR 0     13    // read cin
9     STO 0     3     // b:= (cin)
10    LOD 0     3     // b
11    LIT 0     0     // 0
12    OPR 0     6     // b#0
13    JPC 0     20    // while 条件非真跳转  code-20
14    CAL 0     1     // call p    code-1
15    LOD 0     4     // c
16    OPR 0     14    // write cout c
17    OPR 0     13    // read cin
18    STO 0     3     // b:= (cin)
19    JMP 0     10    // while  无条件返回条件句  code-10
20    OPR 0     0     //end main  返回
```

样例代码 2：

```
const a=10;
var b,c;
begin
      b := 2;
      c := 3;
      if b=3 then b:=a;
      b:=c;
end.
```

符号表 2

```
Name       Kind       Para1     Para2
```

```
================================================
A     CONSTANT       10    0
B     VARIABLE       0     3
C     VARIABLE       0     4
```

目标代码 2

```
odr f     l     a
==================================
0     JMP 0     1     // main: code-1
1     INT 0     5     // 3+2
:=
2     LIT 0     2     // 2
3     STO 0     3     // b:=
4     LIT 0     3     // 3
5     STO 0     4     // c:=
condition
6     LOD 0     3     // b(==2)
7     LIT 0     3     // 3
8     OPR 0     5     // =
9     JPC 0     12    //  条件不成立，不执行 then，跳转至 code-12
then
10    LIT 0     10    // a
11    STO 0     3     // b:=
===then 之后===
12    LOD 0     4     // c
13    STO 0     3     // b:=
14    OPR 0     0     // main-end
```

4.代码解释执行

4.1 概述

对无词法、语法、语义错误，成功生成的目标代码解释执行。
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。

4.2 设计思路

(一) 设计一个解释器类

方法：

对于每种功能码设计一个函数，并添加一个运行解释器函数 run()

成员变量：

目标代码

栈

4 个寄存器

沿静态链查找基址的函数

```
class Interpreter {
public:
    Interpreter();
    Interpreter(vector<TgtLan> &c);
    void run();

    void LIT();
    void LOD();
    void STO();
    void CAL();
    void INT();
    void JMP();
    void JPC();
    void OPR();

    // for test
    void printStack();

private:
    vector<TgtLan> code;        // 目标代码
    //vector<int> stk;          // 解释执行栈数据区
    int *stk;
    int stksize;                // 栈大小
    int findPos(int lyr);       // 静态链 查找层差为lyr的基址

    map<int, string> mp_ins;

    // registers
    TgtLan regI;                // 指令寄存器
    int regAddr;                // 程序地址寄存器，指向下一条要执行的目标指令（相当于CODE数组的下标）
    int regT;                   // 栈顶寄存器T，指出了当前栈中最新分配的单元（T也是数组S的下标）
    int regB;                   // 基地址寄存器，指出每个过程被调用时，在数据区S中给出它分配的数据段起始地址，也称为基地址
```

(二) 实现细节

- 运行解释器 run：

  循环取指，更新程序地址寄存器、指令寄存器当前值，并根据当前指令功能码调用对应函数解释执行。

```
void Interpreter::run() {
    do {
        regI = code[regAddr++];
        //printStack();
        switch (regI.func)
        {
        case 1:
            LIT();
            break;
        case 2:
            LOD();
            break;
        case 3:
            STO();
            break;
        case 4:
            CAL();
            break;
        case 5:
            INT();
            break;
        case 6:
            JMP();
            break;
        case 7:
            JPC();
            break;
        case 8:
            OPR();
            break;
        }
    } while (regAddr != 0);
}
```

- 各功能码对应函数

```
// 将常数放到运栈顶，a域为常数
void Interpreter::LIT() {
    stk[regT] = regI.amt;
    regT++;
}
// 将变量放到栈顶。a域为变量在所说明层中的相对位置，l为调用层与说明层的层差值
void Interpreter::LOD() {
    int var = stk[findPos(regI.lyr) + regI.amt];
    stk[regT] = var;
    regT++;
}
// 将栈顶的内容送到某变量单元中。a,l域的含义与LOD的相同
void Interpreter::STO() {
    --regT;
    stk[findPos(regI.lyr) + regI.amt] = stk[regT];
}
// 调用过程的指令。a为被调用过程的目标程序的入中地址，l为层差
```

```
// 调用过程的指令。a为被调用过程的目标程序的入中地址，l为层差
void Interpreter::CAL() {
    /*
        SL：静态链，它是指向定义该过程的直接外过程运行时数据段的基地址。
        DL：动态链，它是指向调用该过程前正在运行过程的数据段的基地址。
        RA：返回地址，记录调用该过程时目标程序的断点，即当时的程序地址寄存器P的值。
    */
    stk[regT] = findPos(regI.lyr);
    stk[regT + 1] = regB;
    stk[regT + 2] = regAddr;
    regB = regT;
    regAddr = regI.amt;

}
// 为被调用的过程（或主程序）在运行栈中开辟数据区。a域为开辟的个数
void Interpreter::INT() {
    int size = regI.amt;
    regT += size;
}
// 无条件转移指令，a为转向地址
void Interpreter::JMP() {
    regAddr = regI.amt;
}
// 条件转移指令，当栈顶的布尔值为非真时，转向a域的地址，否则顺序执行
void Interpreter::JPC() {
    if (stk[--regT] == 0)
        regAddr = regI.amt;
}
```

```
// 关系和算术运算。具体操作由a域给出。运算对象为栈顶和次顶的内容进行运算，结果存放在次顶。a域为0时是退出数据区
void Interpreter::OPR() {
    int a, b;
    switch (regI.amt) {
    case 0:  // 退出
        regT = regB;                // 栈顶位置指向现在过程的基址，作为退栈操作
        regAddr = stk[regT + 2];            // 下一条指令指针p 指向当前过程的 返回地址
        regB = stk[regT + 1];           // 基址 指向当前过程的 动态链所标识的位置
        break;
    case 1:  // +
        regT--;
        a = stk[regT - 1], b = stk[regT];
        stk[regT - 1] = a + b;
        break;
    case 2:  // -
        regT--;
        a = stk[regT - 1], b = stk[regT];
        stk[regT - 1] = a - b;
        break;
    case 3:  // *
        regT--;
        a = stk[regT - 1], b = stk[regT];
        stk[regT - 1] = a * b;
        break;
    case 4:  // /
        regT--;
        a = stk[regT - 1], b = stk[regT];
        stk[regT - 1] = a / b;
        break;
    case 5:  // ==
        regT--;
        a = stk[regT - 1], b = stk[regT];
        stk[regT - 1] = (a == b);
        break;
    case 6:  // !=
        regT--;
        a = stk[regT - 1], b = stk[regT];
        stk[regT - 1] = (a != b);
        break;
    case 7:  // <
        regT--;
        a = stk[regT - 1], b = stk[regT];
        stk[regT - 1] = (a < b);
        break;
    case 8:  // <=
        regT--;
        a = stk[regT - 1], b = stk[regT];
        stk[regT - 1] = (a <= b);
        break;
    case 9:  // >
        regT--;
        a = stk[regT - 1], b = stk[regT];
        stk[regT - 1] = (a > b);
        break;
    case 10:  // >=
        regT--;
        a = stk[regT - 1], b = stk[regT];
        stk[regT - 1] = (a >= b);
        break;
    case 11:  // -单目
        stk[regT - 1] = -stk[regT - 1];
        break;
    case 12:  // odd
        stk[regT - 1] = (stk[regT - 1] % 2);
        break;
    case 13:  // read  cin
        cin >> stk[regT];
        regT++;
        break;
    case 14:  // write cout
        cout << stk[--regT] << endl;
        break;
    }
}
```

- 根据静态链及层差查找数据段基址

```
// 静态链 查找层差为lyr的基址
int Interpreter::findPos(int lyr) {
    int blink = regB;
    while (lyr--)
        blink = stk[blink];
    return blink;
}
```

四、 实验结论分析与体会

1. 实验结果（OJ）

#	Title	AC / Submits	Status
A	词法分析器	1 / 1	✓Accepted
B	语法分析器	1 / 1	✓Accepted
C	PL/0 编译器	1 / 1	✓Accepted

2. 结论分析与体会

通过本实验，我对编译器的词法分析、语法分析、语义分析及中间代码生成（目标代码生成）以及解释执行的含义有了更深刻的理解。

词法分析将字符流转为词语流，交给语法分析构建语法树。语义分析和中间代码生成在语法分析的过程中，一次扫描，同时生成。最终在解释阶段，使用的假想栈式结构，也让我更明白栈进栈出的过程，以及运行栈的整体结构。

五、 附录