PL0_Compiler说明文档

一、项目说明

本项目编译的源语言是PL/0语言，按照PL/0文法，对源PL0程序进行了词法分析，语法分析（递归子程序法实现），语义分析，生成了P-code指令目标代码，最后提供了解释执行P-code指令的功能。在编译过程中，本程序有错误处理模块，针对输入代码可能出现的一些常见错误进行了提醒。(一些常见错误详见下文)

此外，项目进行了可视化处理，模仿codeblocks做了UI，方便用户操作，可以进行代码的编辑，编译，运行。

二、PL0文法说明

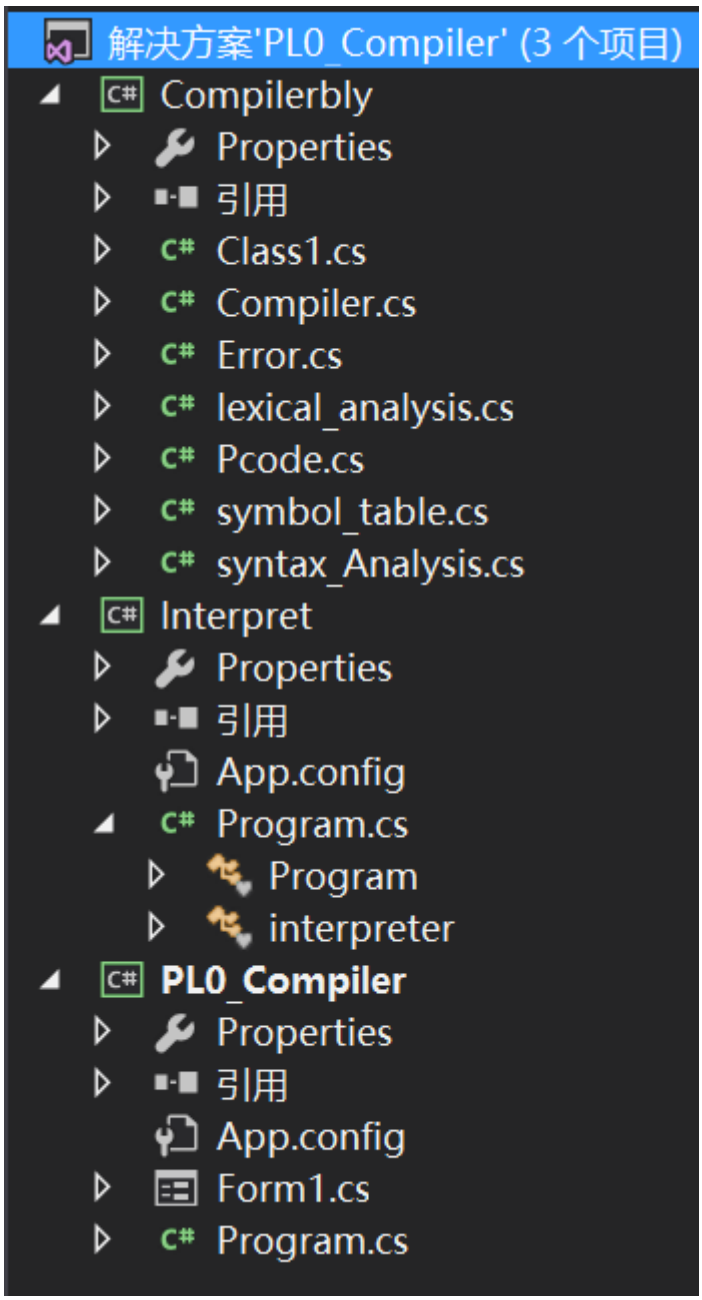
```
<程序> ::= <分程序>.
<分程序> ::= [<常量说明部分>][变量说明部分>][<过程说明部分>]<语句>
<常量说明部分> ::= const<常量定义>{,<常量定义>};
<常量定义> ::= <标识符>=<无符号整数>
<无符号整数> ::= <数字>{<数字>}
<标识符> ::= <字母>{<字母>|<数字>}
<变量说明部分>::= var<标识符>{,<标识符>};
<过程说明部分> ::= <过程首部><分程序>;{<过程说明部分>}
<过程首部> ::= procedure<标识符>;
<语句> ::= <赋值语句>|<条件语句>|<当型循环语句>|<过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

另外还有一些要求：

程序层次嵌套层数最大为3

数值的最大值为INT_MAX

三、项目结构

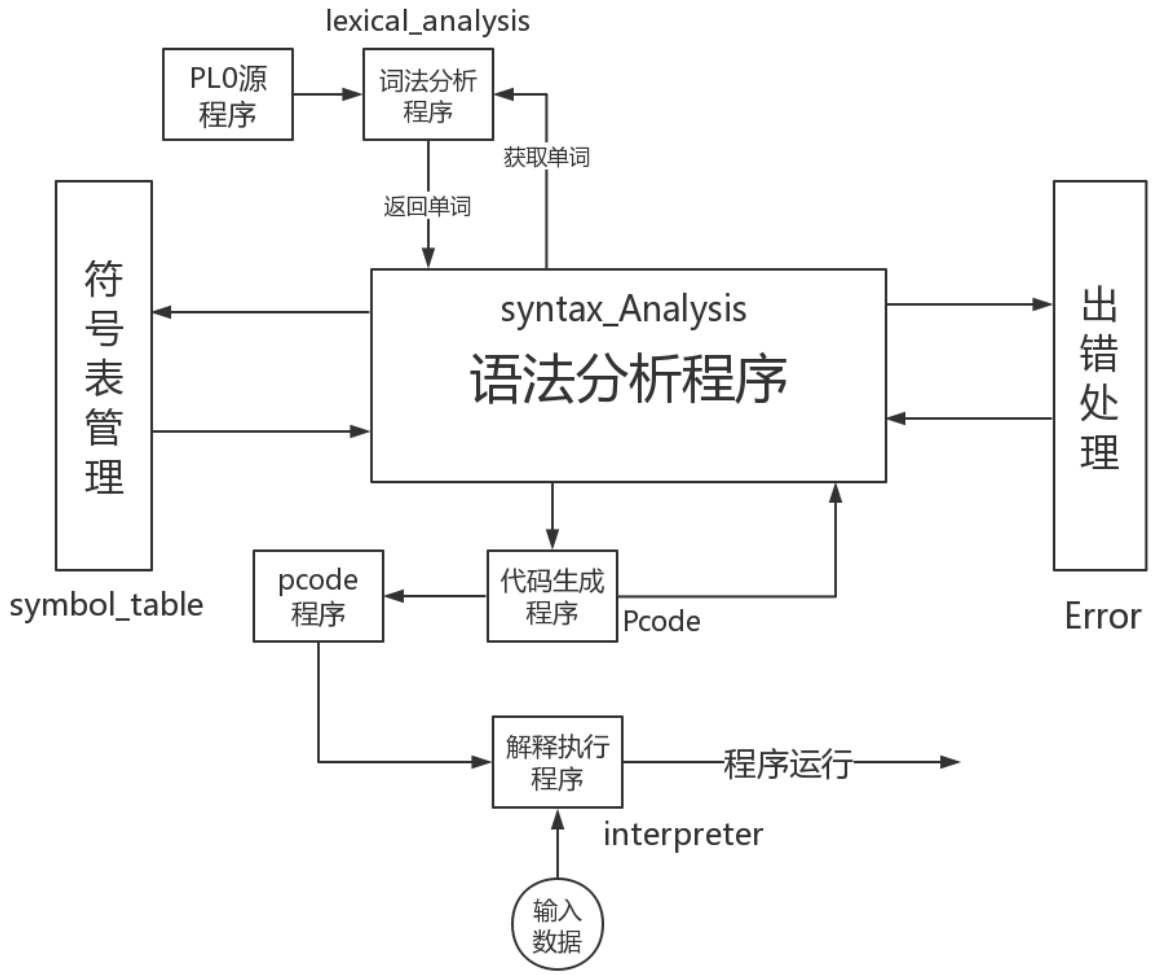


整个项目有三个小项目：

1. **PL0_Compiler** 这是UI模块，用于和用户的交互
2. **Compilerbly** 这是编译器类库模块，包含词法分析类lexical_analysis、语法分析类syntax_Analysis、符号表类symbol_table、pcode代码生成类Pcode、错误处理类Error
3. **Interpret** 这是解释执行pcode代码模块

四、PL0编译系统

这是PL0_Compiler的系统结构图：



1.词法分析类lexical_analysis

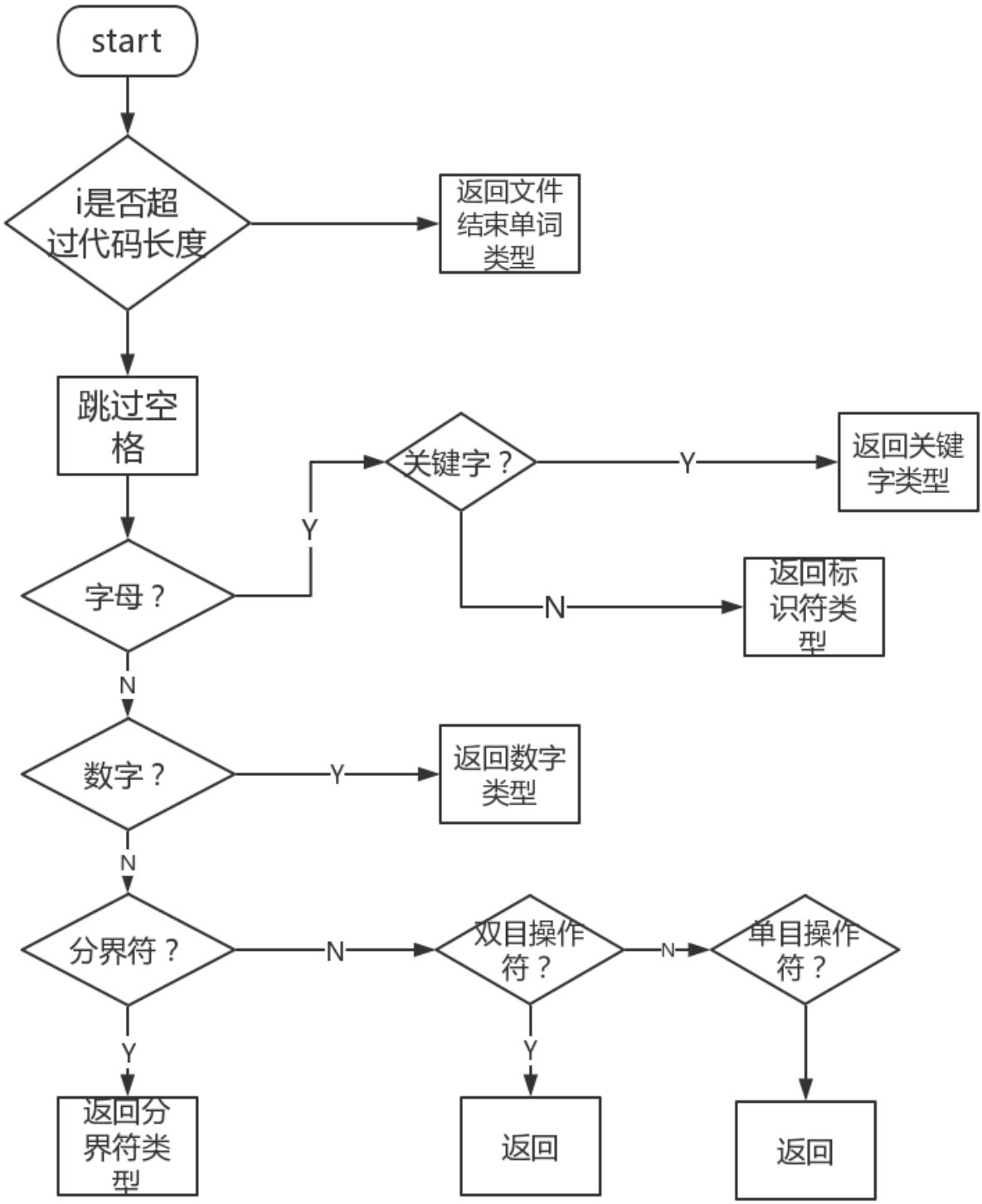
词法分析模块主要向语法分析提供的接口是getsym()，返回读取的单词类型。如果是非数值类型的单词，用word变量存储，若是数值类型，则用num存储所代表的数字,并会进行检查数是否溢出。

我把单词类型分为了35类，用枚举类型symlist

```
public enum symlist
{
    //关键字16
    beginsym =1, endsym, constsym,
    varsym, proceduresym, oddsym,
    ifsym, thensym, elsesym,
    callsym, whilesym, dosym,
    repeatsym, untilsym, readsym, writesym,
    //分隔符5
    LParenthesis, RParenthesis, comma, semicolon, period,//, ; .
    //运算符11
    plus, minus, times, division, equality, LessThanE, MoreThanE,
    becomes,LessThan, MoreThan,inequality,
    //数值1
```

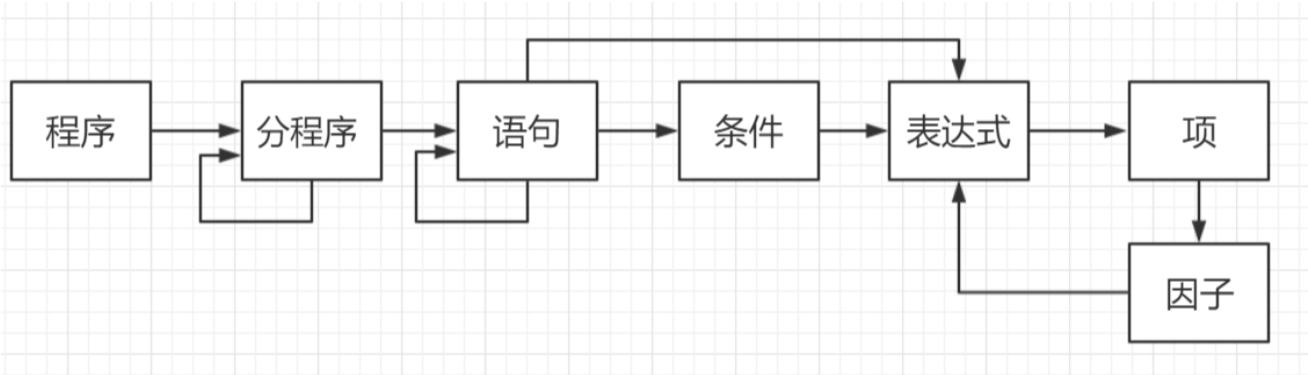
```
    number,
    //标识符变量1
    iden,

    end=-1
}
```

这是词法分析的流程



2.语法分析类syntax_Analysis

语法分析模块按照要求采用了递归子程序法进行语法分析，即为每一个语法成分编写一个分析子程序，根据当前读取的符号，可以选择相应的子程序进行语法分析。语法分析的结构图如下：



这是我语法分析结构的相应的函数，每一个函数在上面注释都有对应的文法规则。

```
//语法分析入口
public void analysis()...
//<程序> ::= <分程序>.
public void program(List<symlist> lst)...
//<分程序>::=[<常量说明部分>][变量说明部分>][<过程说明部分>]<语句>
public void subprogram(List<symlist> followlst)...
//<常量定义> ::= <标识符>=<无符号整数>
public void decconst()...
//<变量说明部分>::= var<标识符>{,<标识符>};
public void decvar()...
//<语句> ::= <赋值语句>|<条件语句>|<当型循环语句>|<过程调用语句>|<读语句>|<写语句>|<复合语句>|<重复语句>|<空>
public void statement(List<symlist> followlst)...
//<表达式> ::= [+|-]<项>{<加法运算符><项>}
public void expression(List<symlist> followlst)...
//<条件> ::= <表达式><关系运算符><表达式>|odd<表达式>
public void ifstatement(List<symlist> followlst)...
//<项> ::= <因子>{<乘法运算符><因子>}
public void item(List<symlist> followlst)...
//<因子> ::= <标识符>|<无符号整数>|'('<表达式>')'
public void factor(List<symlist> followlst)...
```

具体的代码可以看源程序

3.符号表类symbol_tabel

符号表主要记录常量、变量、过程，对合法的常量名、变量名、过程名加入到符号表中。

符号表的每一个表项record结构为这样：

```
public class record
{
    //constant variable procedure
    public string name { get; set; }
    public string kind { get; set; }
    public int val { get; set; }
    public int level { get; set; }
    public int adr { get; set; }
    public int size { get; set; }
}
```

符号表我命名为了stable。

关于符号表有两个接口：

position 根据标识符的名称获得其在符号表的索引。查表操作需要从当前层的最后一个元素倒序查找，直到最开始的那一层。如果找不到返回-1.找到了就返回索引

```
    public int position(string s)
    {
        for(int i=dx[point];i>0;i--)
        {
            if (stable[i].name.Equals(s))
                return i;
        }
        return -1;
    }
```

add 将符号加入到符号表中。加入符号表需要注意符号的作用域，也就是声明他所在的分程序的层次。如果不在同一个层次，要将原来层次的符号删除

dx是是一个数组，记录每一层的在符号表最后一个元素的索引，point和语法分析的level决定了当前是否进入了一个新的程序层

```
    public void add(symlist sym)
    {
        syntax_Analysis  sa = compiler.sa;
        if (sa.level > point)
        {
            point++;
            dx[point] = dx[point - 1];
        }
        else if (sa.level < point)
        {
            int delnum = dx[point] - dx[point - 1];
            for (int i = 0; i < delnum; i++)
                stable.RemoveAt(stable.Count - 1);
            point--;
        }
        dx[point]++;
        record re = new record();
        re.name = compiler.la.word;
```

```
        if(sym==symlist.constsym)
        {
            re.kind = "constant";
            re.val = compiler.la.num;
            stable.Add(re);
        }
        else if(sym==symlist.varsym)
        {
            re.kind = "variable";
            re.level = sa.level;
            //获取地址还没写re.adr =
            re.adr = compiler.sa.getadd();
            stable.Add(re);
        }
        else if(sym==symlist.proceduresym)
        {
            re.kind = "procedure";
            re.level = sa.level;
            stable.Add(re);
        }
    }
```

4.错误类 Error

对于每种错误，都有对应的错误码，可以用一个Dictionary来表示：

`public Dictionary<int, string> errordict = new Dictionary<int, string>();//错误结构`

对于要编译的程序中的每一个错误我用了一个类echerror来表示，他记录每个错误的错误码和错误所在的行号。

将所有的错误都存储在一个`List<echerror>`中

```
public class echerror
{
    public int errnum { get; set; }
    public int line { get; set; }
    public echerror(int num ,int line)
    {
        errnum = num;
        this.line = line;
    }
}
public List<echerror> errlst = new List<echerror>();
```

对于错误类型，我结合课本自己总结了如下：

(1, "常数说明赋值符号应为"="")

(2, "常数说明"="后应是数字"); (3, "常数说明中标识符后应是"=""); (4, "const, var, procedure 后应是标识符"); (5, "缺少了',' 或';'"); (6, "过程说明后的符号应是语句开始符或过程定义符"); (7, "应是语句开始符"); (8, "程序体内的语句部分的后继符不正确"); (9, "程序结尾应为'.'"); (10, "语句之间缺少';'"); (11, "标识符未说明"); (12, "不可向常量或过程赋值，赋值语句左值标识符属性应是变量"); (13, "赋值语句赋值符号应是':='"); (14, "call 后应为标识符"); (15, "call

不可调用常量或变量，应为过程"); (16, "条件语句中缺少'then'"); (17, "语句结束缺少'end' 或';'"); (18, "while 语句中缺少'do'"); (19, "语句后的符号不正确"); (20, "应为关系运算符"); (21, "表达式内不可有过程标识符"); (22, "表达式中缺少')'"); (23, "因子后为非法符号"); (24, "表达式不能以此符号开始"); (25, "repeat 语句中缺少until"); (26, "程序层次嵌套层数过多，最多为3"); (31, "数过大"); (32, "read语句括号中的标识符不是变量"); (33, "语句缺少'('"); (34, "语句缺少')'"); (35, "read 语句缺少变量"); (36, "程序之外出现字母");

adderror这个接口就是将发现的错误加入到errlst中

```
    public void adderror(int errnum)
    {
        echerror e = new echerror(errnum,compiler.la.line);
        errlst.Add(e);
    }
```

5.生成pcode代码 Pcode

对于Pcode指令，我按照书上给出的pcode指令，形如f l，a 其中f是操作码，l为变量或者过程被引用的分程序与声明该变量或过程的分程序的层次差，a对于不同的指令有不同的含义

1. LIT 0,a 取常量a置于栈顶
2. OPR 0,a 执行运算，a的值表示执行何种运算
3. LOD l,a 取变量（层次差为l，相对地址为a）置于栈顶
4. STO l,a 取变量（层次差为l，相对地址为a）置于栈顶
5. CAL l,a 调用过程（层次差为l，入口指令地址为a）
6. INT l,a 分配空间，栈指针top增加a
7. JMP 0,a 无条件跳转至地址a
8. JPC 0,a 条件跳转至地址a
9. RED l,a 读数据，存入变量（层次差为l，相对地址为a）
10. WRT 0,0 将栈顶值输出

用CODE存储每个pcode 指令，每个pcode 指令就如上面所说：

```
public class CODE
{
    public string op{get;set;}
    public int l { get; set; }
    public int a { get; set; }
}
```

将所有的pcode指令保存在 `List<CODE> pcdeolst` 中

接口gen 用来语法分析调用生成每个pcode 存储在pcdeolst中提供给解释程序执行。

```
public void gen(string op,int l,int a)
{
    CODE instru = new CODE();
    instru.op = op;
    instru.l = l;
    instru.a = a;
    pcdeolst.Add(instru);
    cx++;
}
```

6.解释程序Interpret

对于解释程序，我是这样处理：

我模仿codeblocks 对于每一个编译需要运行的代码，我都创建一个新的进程，这个进程中运行解释之前生成的Pcode指令。这就相当于 解释执行了。而要创建新的进程，我选择重新创建一个项目，就是创建这个项目的一个进程就可以了。这部分代码如下：

```
string pCode = "";
List<CODE> codelst = compiler.pc.pcdeolst;
for (int i = 0; i < codelst.Count; i++)
{
    this.richTextBox2.Text += (codelst[i].op + " " + codelst[i].l.ToString() + " " +
codelst[i].a.ToString() + "\n");
    pCode += (codelst[i].op + '*' + codelst[i].l.ToString() + '*'+
codelst[i].a.ToString() + "\n");
}
if (pCode != "")
{
   this.richTextBox3.Text += "start to run ";
   string curPath = Directory.GetCurrentDirectory();
   Process pInte = new Process();
   pInte.StartInfo.FileName = curPath + "\\Interpret.exe";
   pInte.StartInfo.Arguments = pCode;
   pInte.Start();
}
```

这样就可以创建一个新的进程 运行解释程序了。

对解释程序我还是封装了成一个类 interpreter。

接口interpreter 就是解释程序的主体，他接受一个string，就是前面生成所有的pcode代码的

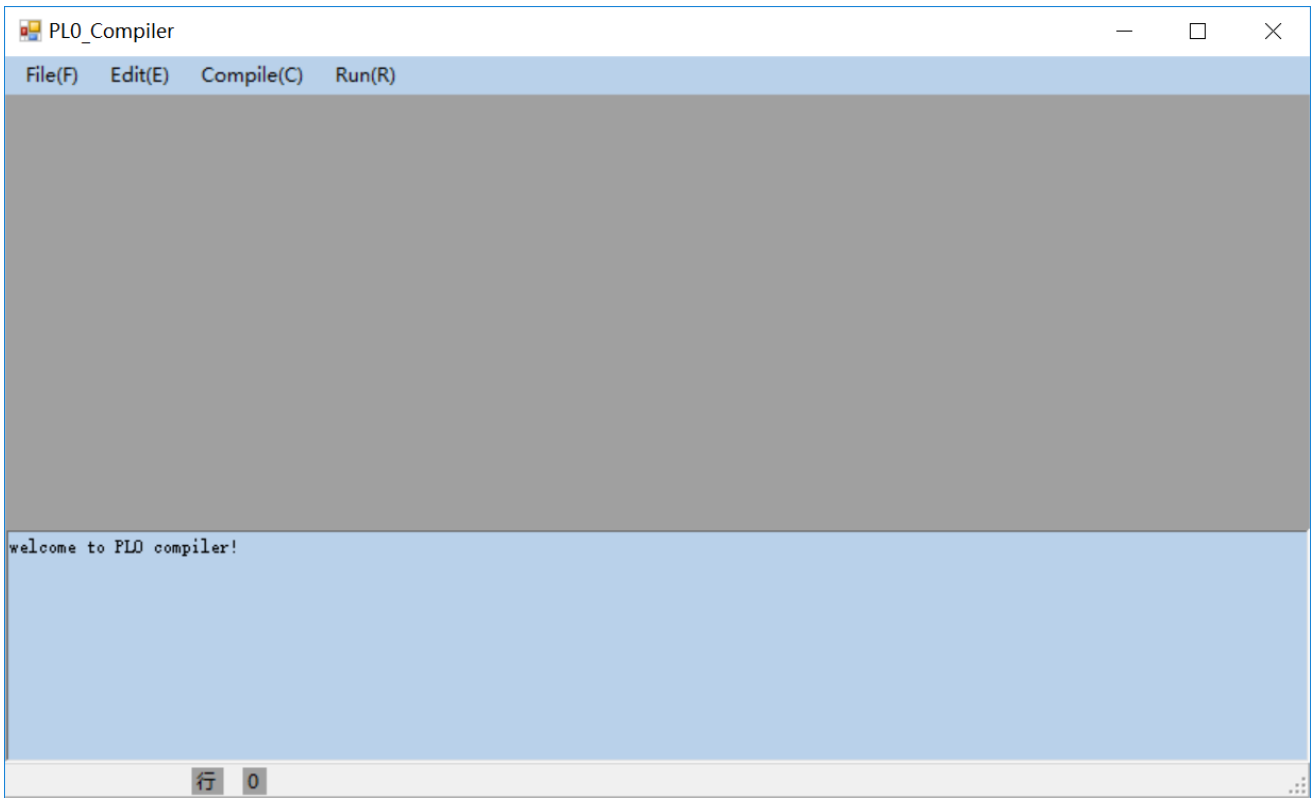
解释器有两个存储器、一个指令寄存器和三个地址寄存器组成。存储器code存放程序pcode指令，存储器stack作为运行栈， 实现栈式动态存储分配。指令寄存器i存放当前执行的pcode指令；栈顶地址寄存器t作为栈顶指针，总是指向运行栈stack的栈顶；程序地址寄存器i存放下一条要执行的指令地址（数组索引）；基地址寄存器badd，存放当前运行的分程序数据区在数据栈S中的起始地址，即badd总是指向动态链的链头。

getadd 这个接口获得 那个 层数差为d的那一层的栈中的起始位置

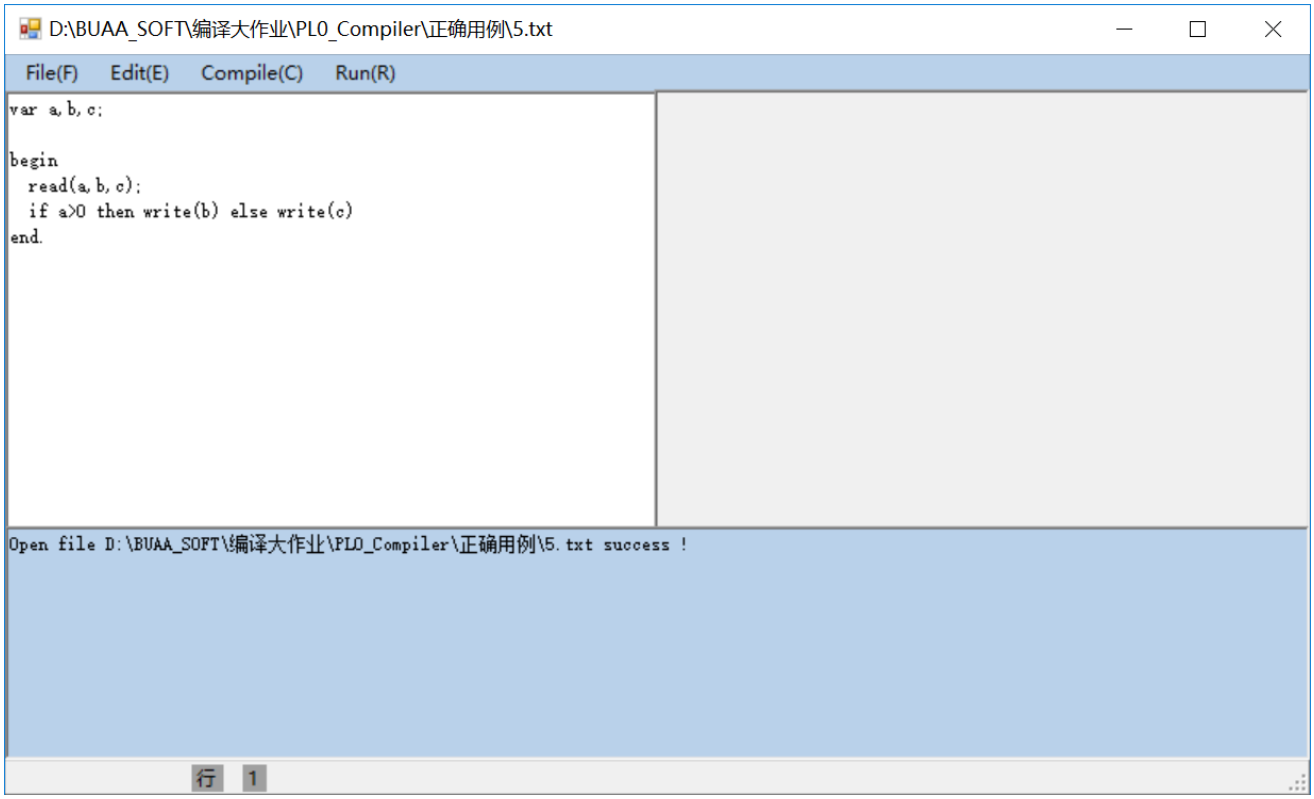
```
    public int getadd(int d)
    {
        int b = badd;
        while (d > 0)
        {
            b = stack[b];
            d--;
        }
        return b;
    }
```

7.用户界面

这是程序的初始界面

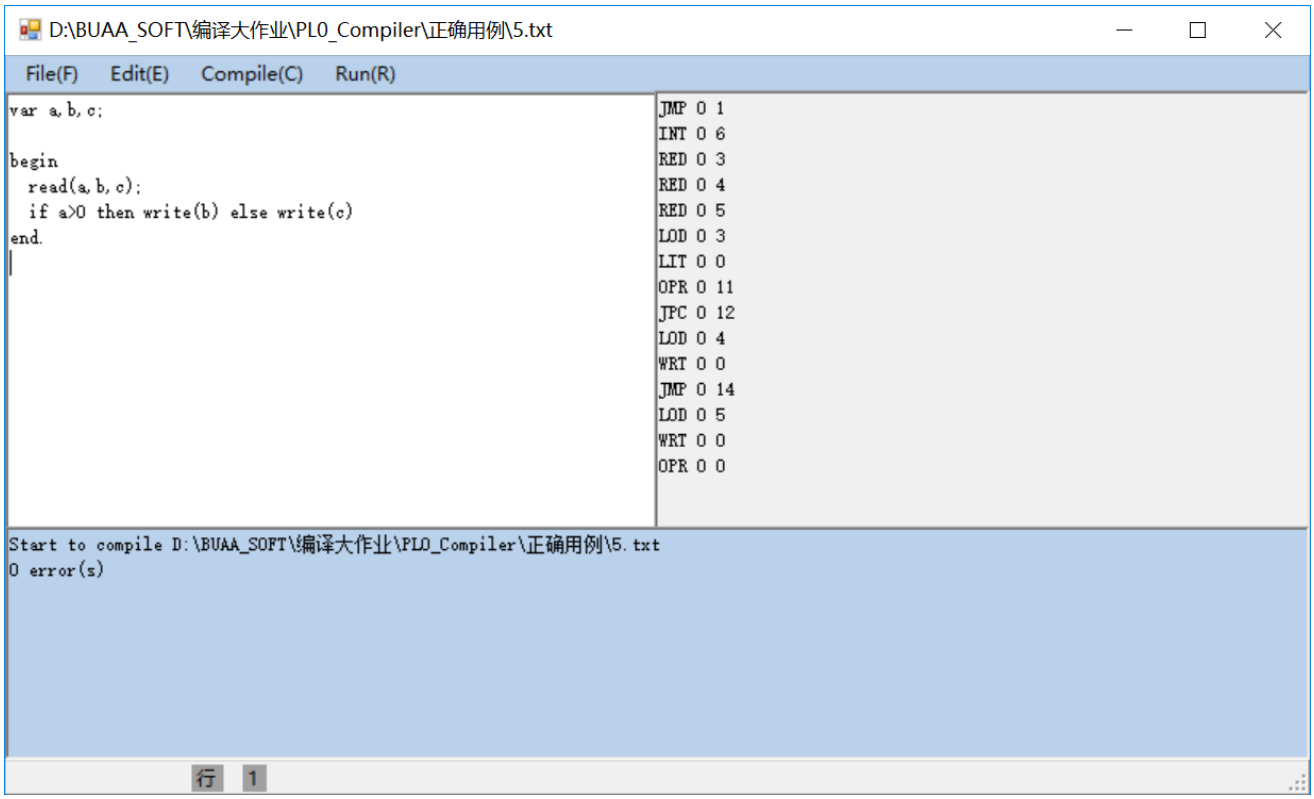


可以选择打开某个txt源程序文件

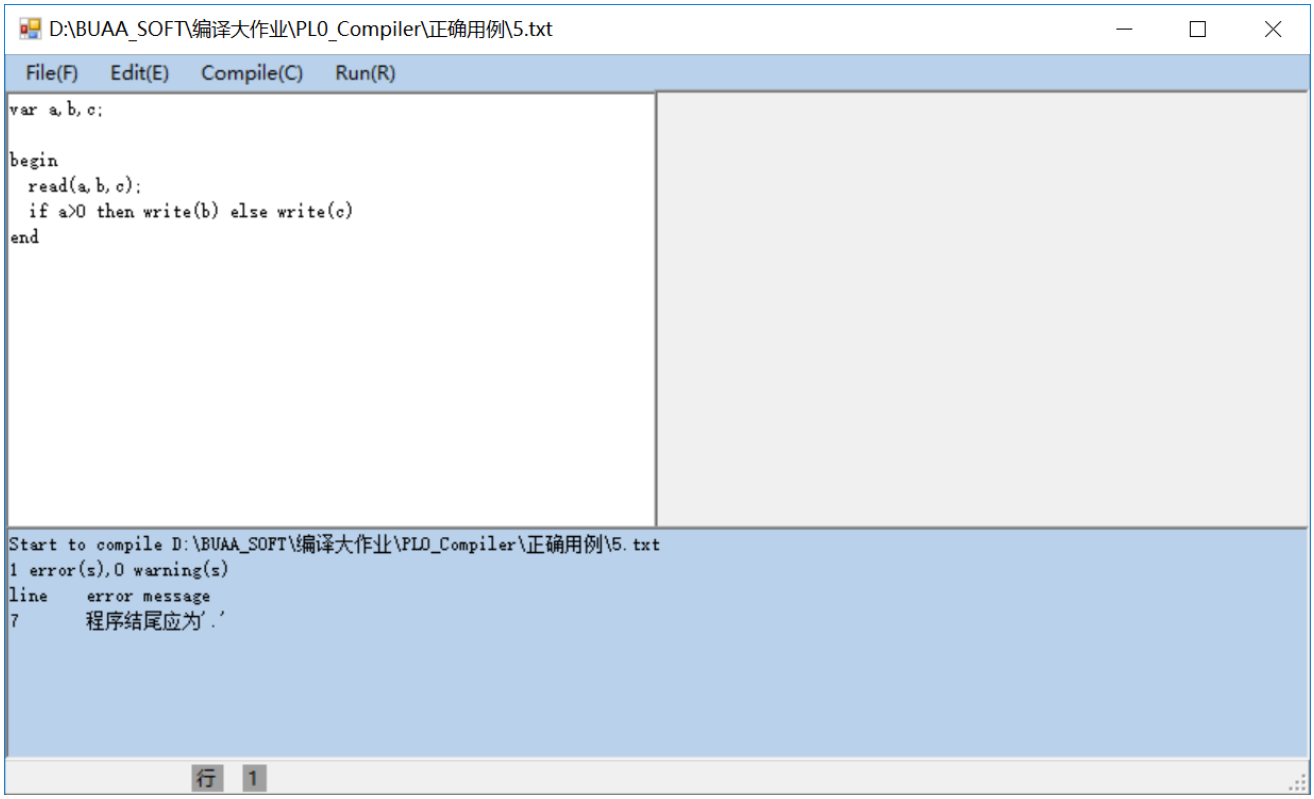


选择 选项栏中Compile 在右边的 文本框中 可以生成pcode 代码，在下面有编译的信息，是否成功

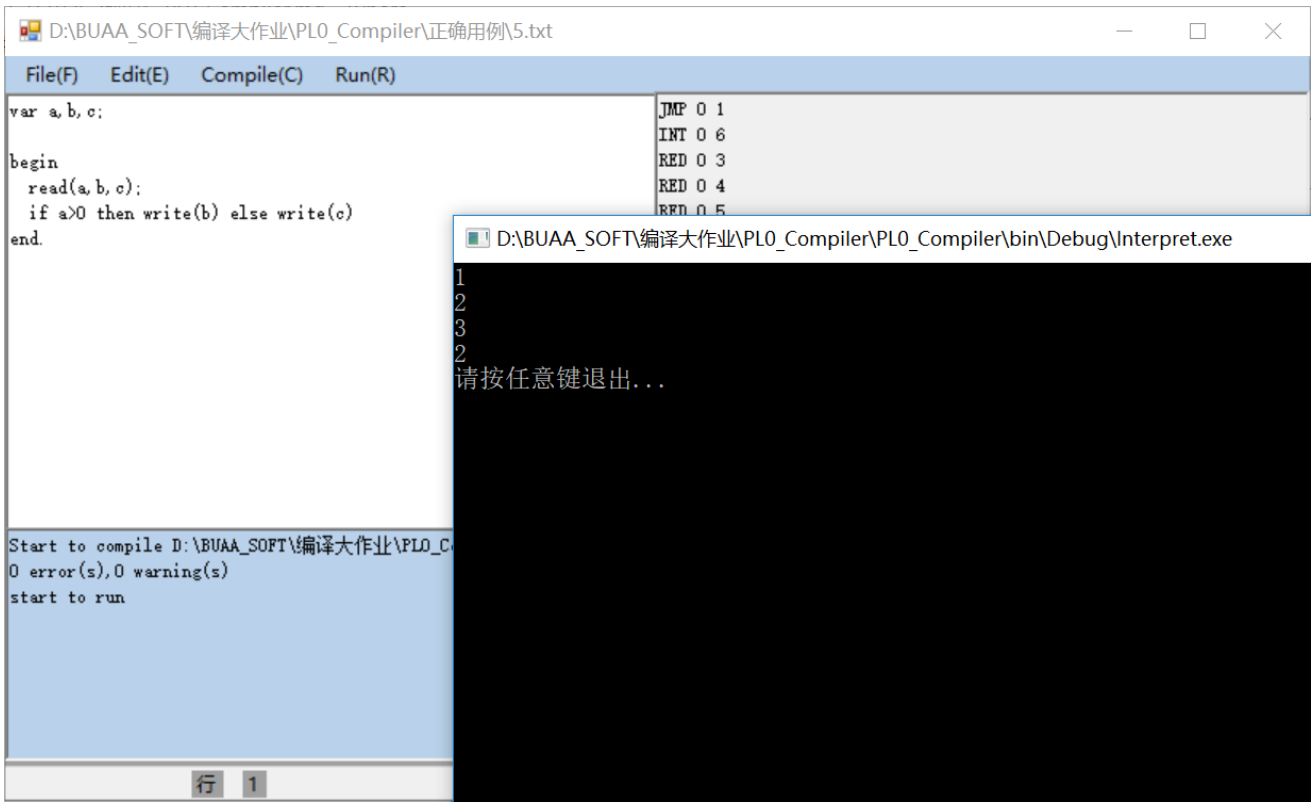
这是编译没有出错的截图：



如果编译报错：



如果需要运行这个程序，可以选择选项栏里的 按钮Run



五、实验感想

这次的实验和以往的两次实验差别很大，这次的任务最重要的一点事语法分析的结构，难点也是在语法分析。因为语法分析里需要接受词法分析来的单词种类，根据单词种类进行语法分析，然后还要对符号进行符号表的管理，又会牵扯到程序嵌套层数的问题，另外还要涉及到错误处理和pcode代码生成。总的来说就是语法分析里就参合了所有的编译器的类库。

经过查阅大量的资料，我参考了课本附录的PL0编译系统的源代码，语法分析的结构和解释程序的结构基本和书上提供的源代码类似。递归子程序虽说比较简单，程序框图看起来也比较容易实现，但真正动手敲代码会发现很多的问题。

另外我认为对实际问题的抽象我觉得也非常重要，在这个系统中，我采用的是面向对象的编程思维，将所有的模块都类化，将他们通过类封装。另外对于解释程序如何嵌入到系统中，我认为也是一个难点，我是通过重新创建一个进程，而要创建一个进程，就需要在创建一个新的解释程序的项目，在主项目中还要引用到这个项目，引用时又会出现一个问题：如果只是有两个项目是无法完成的，因为会造成循环引用，所以，我就将整个项目分成了三个小的项目，UI作为单独的一个项目。

经过这次课程设计，我对面向对象编程、C#编程、WINFORM编程得到了进一步的提高，对编译原理也有了更深的理解。