****



编译原理实践

——PL/0编译系统

**姓名： 黎昆昌**

**学号：16211022**

**学院：软件学院**

<http://compile.lkc1621.xyz/>

引言

PL/0编译系统，语言：python，HTML界面，IDE：Pycharm2017。编译系统包含词法分析、语法分析、符号表管理、语义分析及目标代码Pcode的生成、错误处理以及Pcode的解释执行。整个编译器以语法分析程序为中心，调用词法分析程序识别单词，生成相应的代码，查询或填写相应的符号表项，在出错时调用错误处理程序，报告错误，同时跳过出错部分继续进行语法分析。整个工程耗时两星期，基本达到课程要求。程序访问网站<http://compile.lkc1621.xyz/>

1. PL/0语言

## 1.1 PL/0描述

PL/0语言是一种类PASCAL语言，是教学用程序设计语言，它比PASCAL语言简单，作了一些限制。它只有整数一种类型，但却具有相当完全的可嵌套型的分程序结构。PL/0可进行常量定义、变量说明和过程调用，并具有通常程序设计语言所必需的最基本的语句，如赋值语句、条件语句、循环语句、过程调用语句和复合语句。考虑到输入/输出的需要，我们添加了简单的读、写语句。PL/0过程没有参数，但可以递归调用，因此，过程所加工的数据只能通过全局变量进行传递。

1.2 PL/0语言文法的扩充BNF表示

<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>

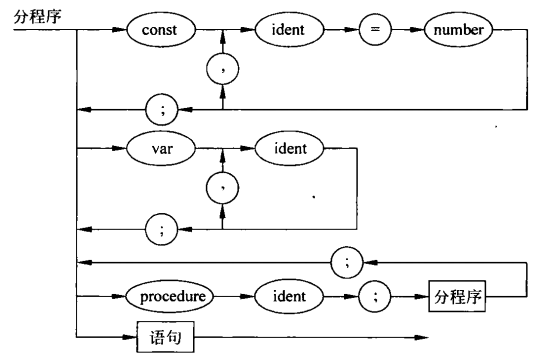
<读语句> ::= read'('<标识符>{,<标识符>}')'

<写语句> ::= write'('<标识符>{,<标识符>}')'

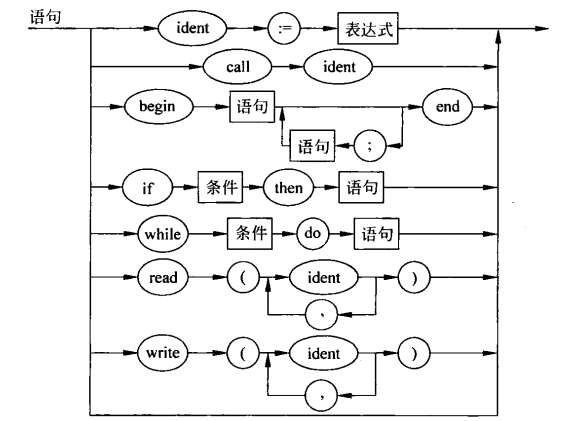
<字母> ::= a|b|...|X|Y|Z

<数字> ::= 0|1|2|...|8|9

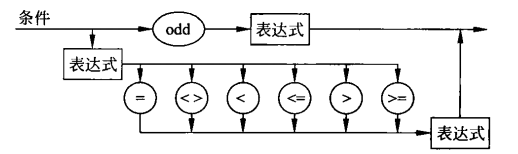
1.3 PL/0的语法图



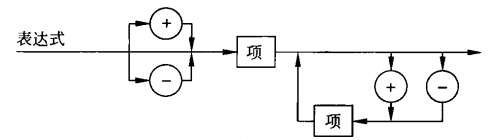
1. 分程序



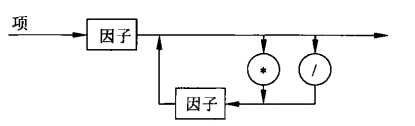
1. 语句



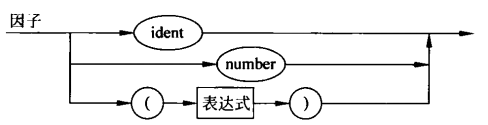
1. 条件



1. 表达式



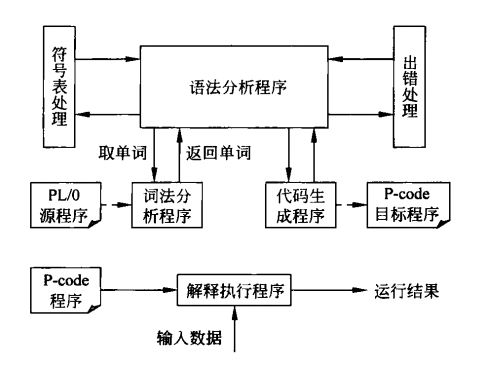
1. 项



1. 因子

## 1.4 PL/0编译系统结构

PL/0编译系统是一个编译-解释执行程序，整个编译过程分两个阶段进行。第一阶段先把PL/0源程序编译成假想计算机的目标程序（P-code指令），第二阶段再对该目标程序进行解释执行，得到运行结果。PL/0编译程序采用一遍扫描，即以语法分析为核心，由它调用词法分析程序取单词，在语法分析过程中同时进行语义分析处理，并生成目标指令。如遇到语法、语义错误，则随时调用出错处理程序，打印出错信息。在编译过程中要利用符号表的登录和查找来进行信息之间的联系。一边扫描的PL/0编译和P-code解释执行框图如下图所示。



**图1 PL/0编译系统结构框图**

二、系统分析

## PL/0的词法分析

PL/0编译系统中所有的字符和字符串的类型可简单分为七类，为如下表格

|  |  |
| --- | --- |
| 保留字 | Begin,end,if,then,else,const,procedure,var,  do,while,odd,call,read,write,repeat,until |
| 算数运算符 | + | - | \* | / |
| 比较运算符 | <> | < | <= | > | >= | = |
| 赋值符 | := | = |
| 标识符 | 变量名，过程名，常数名 |
| 常数 | 整数 |
| 界符 | , | . | ; | ( | ) |

PL/0的词法分析程序作为一个独立的子程序由语法分析程序调用，而词法分析本身是正则表达式的匹配问题，我在设计时采用python语言，定义了相关正则表达式并按照优先级依次匹配，一轮匹配中出现多个符合的字符串则选择长度较长的字符串，定义正则表达式匹配串如下

1. r'(?P<BLANK>\s)',
2. r'(?P<KEYWORD>const|var|procedure|if|then|else|while|do|call|begin|end|repeat|until|read|write|odd)',
3. r'(?P<IDENTIFIER>[A-Za-z][A-Za-z0-9]\*)',
4. r'(?P<NUMBER>\d+(\.\d+)?)',
5. r'(?P<DELIMITER>\(|\)|\.|,|;)',
6. r'(?P<PLUS\_OPERATOR>\+|-)',
7. r'(?P<MULTIPLY\_OPERATOR>\\*|/)',
8. r'(?P<ASSIGN\_OPERATOR>:=)',
9. r'(?P<RELATION\_OPERATOR>=|<>|<=|<|>=|>)',
10. r'(?P<COMMENT>/\\*[\s\S]\*\\*/)',
11. r'(?P<OTHER>[\s\S])'

BLANK用于辅助识别空格等不输出的字符，KEYWORD识别关键词，INDENTIFIER识别标识符，NUMBER识别数字（包括整数和小数，小数用于出错提示），DELIMITER识别界符，PLUS\_OPERATOR识别正负号（同时也是加减操作），MULTIPLY\_OPERATOR识别乘除，ASSIGN\_OPERATOR识别赋值号，RELATION\_OPERATOR识别关系运算符，COMMENT识别注释，OTHER识别其余字符。封装一个nameTuple，命名为Token，里面有两个元素，一个类型，为上述正则表达式中的分组别名，一个为值，为匹配到的字符串。

“按照优先级依次匹配，一轮匹配中出现多个符合的字符串则选择长度较长的字符串”具体含义如下，匹配时会按照匹配串的顺序从上往下匹配，因此需要安排好匹配串的顺序，匹配了多个字符串而长度不同，这种情况只出现在某字符串包含关键词子串或者匹配了双字符运算符包含单字符运算符的时候。如对于字符串constint，正则表达式首先会匹配为关键词const，然后会匹配为标识符constint，根据匹配最长原则，最后识别为标识符。

在匹配串最后添加了OTHER匹配剩余的情况，可以在得到OTHER类型的字符串时进行错误处理，也不至于匹配失败导致程序终止。同时在匹配时定义了两个变量分别记录行列号，逐个字符读取，每次返回一个字符串，这样在出错时可以找出程序出错的位置。

程序设计时，将词法分析器封装为一个类LexerEngine，语法分析器中初始化一个这个类的实体，\_get\_symbol函数中调用这个实体获得token。PL/0的语法分析采用了“单符号先行”技术，即在进入某个语法成分的分析子程序之前，先读取一个单词。

## PL/0的符号表管理

定义符号类Symbol，类的属性包括符号名name，符号类型kind，符号值val，符号所处层次level，符号的地址adr。定义符号表类SymbolTable，负责对符号表进行登录enter和查询get，登录和查询时都从符号表末尾往前遍历，登录时往前遍历确定不存在重名或者同类型，查询时往前遍历找到最近层次的符号。

## PL/0的语法分析

PL/0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。采用不带回溯的递归子程序法，对语言文法有一定的要求：

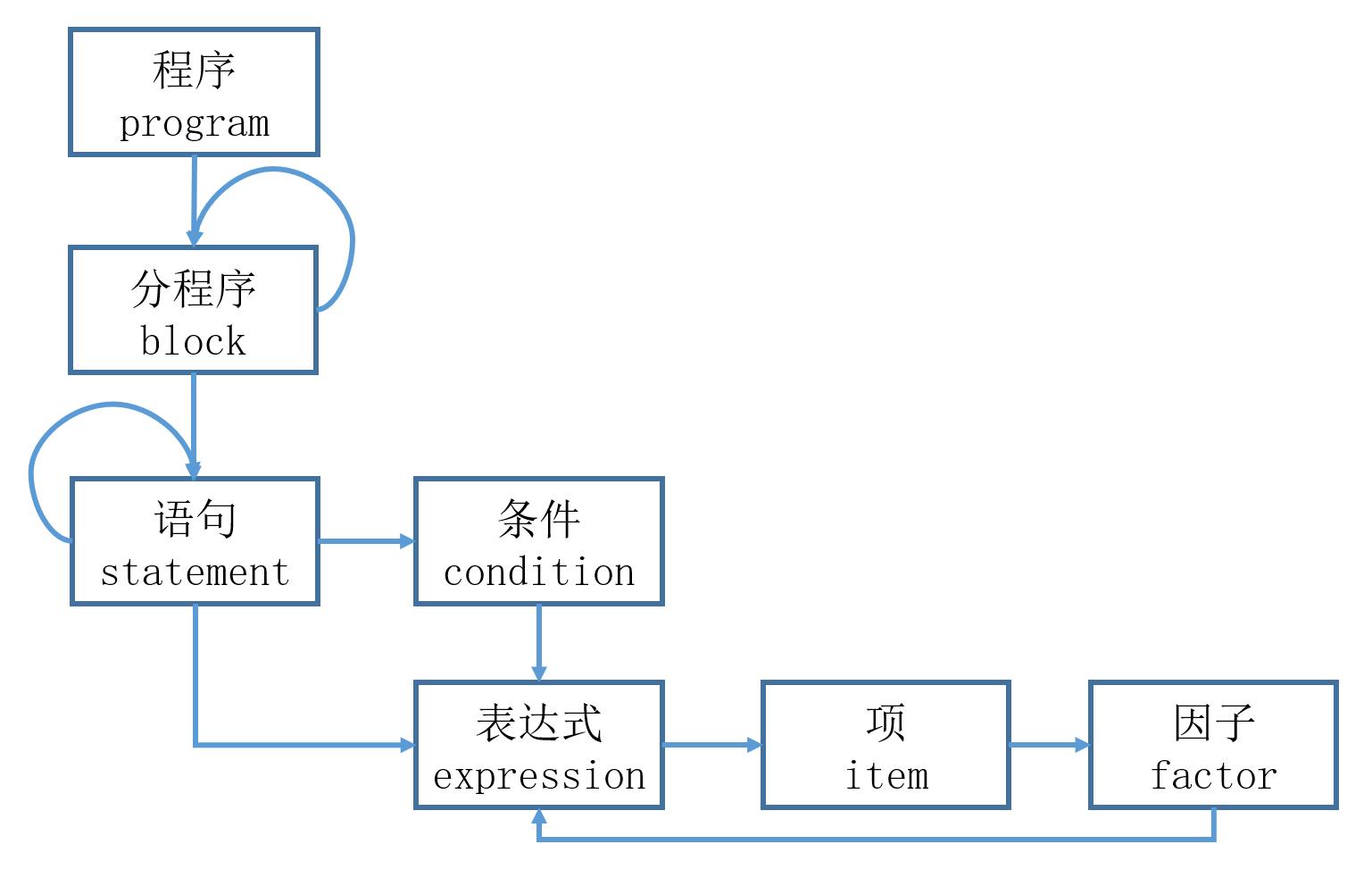
1. 文法必须是非左递归；
2. 文法的任一非终结符，其规则右部多个选择所生成的FIRST集合两两不相交；
3. 若文法具有形如，则。

根据PL/0的文法BNF表达，下表中列出了PL/0有关的语法成分的FIRST集合FOLLOW集，可以判断出满足递归子程序的要求。表说明：R表示关系运算符，id和num分别表示标识符和无符号整数。

如下为PL/0文法非终结符的开始符号集与后继符号集表

|  |  |  |
| --- | --- | --- |
| 非终结符 | FIRST集 | FOLLOW集 |
| 分程序block | const var procedure id if call begin while repeat read write | . ; |
| 语句statement | Id call begin if while read write until | . ; end |
| 条件condition | odd + - ( id num | then do |
| 表达式expression | + - ( id num | , ; ) R end then do |
| 项term | id num ( | , ; ) R + - end then do |
| 因子factor | id num ( | . ; ) R + - \* / end then do |

PL/0的语法关系调用图如下，可以清晰地看出进行直接或间接的递归调用。



语法分析器中定义了Parser类对Pascal源代码进行语法分析，生成P-code指令集，其中定义了多个如上所示的多个子程序。

## 生成目标代码

### 2.4.1 P-code含义

P-code语言：一种栈式机的语言。此类栈式机没有累加器和通用寄存器，有一个栈式存储器，有四个控制寄存器（指令寄存器 I，指令地址寄存器 P，栈顶寄存器 T和基址寄存器 B），算术逻辑运算都在栈顶进行。

指令格式形如F L A，其中F表示操作码，可用枚举变量表示；L表示层次差 （标识符引用层减去定义层）；A不同的指令含义不同。定义指令类PCode，其中包含三个属性f,l,a，对应如上F、L、A。同时定义指令管理类PcodeEngine，负责产生和获得指令。

P-code指令意义如下表所示

|  |  |
| --- | --- |
| 指令 | 具体含义 |
| LIT 0, a | 取常量a放到数据栈栈顶 |
| OPR 0, a | 执行运算，a表示执行何种运算(+ - \* /) |
| LOD l, a | 取变量放到数据栈栈顶(相对地址为a,层次差为l) |
| STO l, a | 将数据栈栈顶内容存入变量(相对地址为a,层次差为l) |
| CAL l, a | 调用过程(入口指令地址为a,层次差为l) |
| INT 0, a | 数据栈栈顶指针增加a |
| JMP 0, a | 无条件转移到指令地址a |
| JPC 0, a | 条件转移到指令地址a |
| OPR 0 0 | 过程调用结束后,返回调用点并退栈 |
| OPR 0 1 | 栈顶元素取反 |
| OPR 0 2 | 次栈顶与栈顶相加，退两个栈元素，结果值进栈 |
| OPR 0 3 | 次栈顶减去栈顶，退两个栈元素，结果值进栈 |
| OPR 0 4 | 次栈顶乘以栈顶，退两个栈元素，结果值进栈 |
| OPR 0 5 | 次栈顶除以栈顶，退两个栈元素，结果值进栈 |
| OPR 0 6 | 栈顶元素的奇偶判断，结果值在栈顶 |
| OPR 0 7 |  |
| OPR 0 8 | 次栈顶与栈顶是否相等，退两个栈元素，结果值进栈 |
| OPR 0 9 | 次栈顶与栈顶是否不等，退两个栈元素，结果值进栈 |
| OPR 0 10 | 次栈顶是否小于栈顶，退两个栈元素，结果值进栈 |
| OPR 0 11 | 次栈顶是否大于等于栈顶，退两个栈元素，结果值进栈 |
| OPR 0 12 | 次栈顶是否大于栈顶，退两个栈元素，结果值进栈 |
| OPR 0 13 | 次栈顶是否小于等于栈顶，退两个栈元素，结果值进栈 |
| OPR 0 14 | 栈顶值输出至屏幕 |
| OPR 0 15 | 屏幕输出换行 |
| OPR 0 16 | 从命令行读入一个输入置于栈顶 |

### 2.4.2地址返填技术

对于if then [else]，while do和repeat until语句，要生成跳转指令，采用地址返填技术。目标代码生成模式如下：

1. （1） **if** <condition> **then** <statement> [**else**]
2. <condition>
3. JPC addr1
4. addr1:
5. <statement>
6. （2） **if** <condition> **then** <statement> [**else**  <statement>]
7. <condition>
8. JPC addr1
9. <statement>
10. JMP addr2
11. addr1:
12. <statement>
13. addr2:
14. （3） **while** <condition> **do** <statement>
15. addr1:
16. <condition>
17. JPC addr2
18. <statement>
19. JMP addr1
20. addr2:
21. （4） **repeat** <statement> **until** <condition>
22. addr1:
23. <statement>
24. <condition>
25. JPC addr1

## 2.5错误处理

### 2.5.1错误处理的原则

1. 任何源程序输入序列都不会导致编译工作的崩溃；
2. 尽可能多地发现源程序中的语法和语义在错误，并尽可能准确指出错误位置和错误属性；
3. 尽可能进行校正，对于出现的错误，予以校正使编译继续进行。

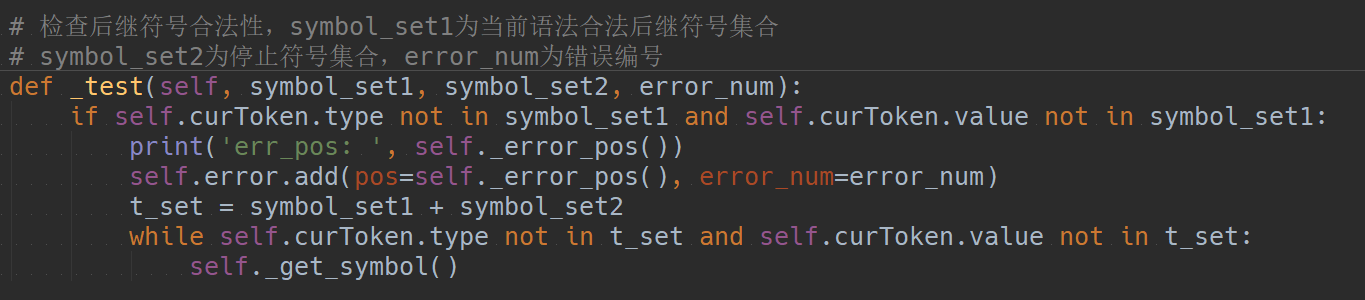
### 2.5.2错误校正

对于简单的错误，如遗忘了分号，赋值符号写错等，除了给出错误信息，可以予以简单校正，使程序尽可能继续进行。

在每个语法分析子程序的入口或出口处，调用\_test函数, 检测下一个取得的单词是否是当前语法成分的合法头符号集或者后继符号，若不是，给出错误信息，并且跳读一段程序，直至取得合法的后继符号。

然而，为了防止跳读的程序段太多，程序中设置了一个停止符号集合，此集合中元素为明显可以使程序开始正常编译的符号，所以只要新取来的符号属于合法后继符号集或者停止符号集，都可以停止跳读。

在block的出口处，statement的出口处，factor的入口和出口处，都调用了合法性检测函数，\_test函数定义如下：



2.5.3错误信息

错误信息应尽可能地检测出源程序中存在的问题，我结合了教材设计了36种错误信息如下表所示

|  |  |
| --- | --- |
| 出错编号 | 出错原因 |
| 1 | 常数说明中应是'='而不是':=' |
| 2 | 常数说明中'='后应为数字 |
| 3 | 常数说明中标识符后应为'=' |
| 4 | const, var, procedure后应为标识符 |
| 5 | 漏掉逗号或分号 |
| 6 | 过程说明后的符号不正确 |
| 7 | 应为语句开始符号 |
| 8 | 程序体内语句部分后的符号不正确 |
| 9 | 程序结尾应为句号 |
| 10 | 语句之间漏了分号 |
| 11 | 标识符未说明 |
| 12 | 不可向常量或过程赋值 |
| 13 | 赋值语句中应为赋值运算符':=' |
| 14 | call后应为标识符 |
| 15 | call后标识符属性应为过程,不可调用常量或变量 |
| 16 | 条件语句中缺失then |
| 17 | 应为分号或end |
| 18 | 当到型循环语句中缺失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 | 缺失右括号 |
| 23 | 因子后不可为此符号 |
| 24 | 表达式不能以此符号开始 |
| 25 | 循环语句中缺失until |
| 26 | 读语句括号内不是标识符 |
| 27 | 写语句括号内不是表达式 |
| 28 | 读语句括号内标识符属性应该为变量 |
| 29 | 标识符类型不正确 |
| 30 | 这个数太大，超过INT32\_MAX |
| 31 | 标识符重名 |
| 32 | 嵌套层数过多 |
| 33 | 句号后程序未结束 |
| 34 | 应为整数而不是浮点数 |
| 40 | 应为左括号 |
| 50 | 无法识别的符 |

三、系统架构

## 3.1 lexer2.py

lexer2.py是一个词法分析器，一开始设计的lexer.py中正则表达式匹配串类型较少，后根据语法分析的需要，设计了2.1节中提到的11种类型，进而设计了lexer2.py，整个程序架构如下：

1. lexicon：匹配串数组
2. Token：命名元组，记录单词类型与值
3. LexerEngine：词法分析器

属性：

1. file：程序代码
2. lexicon：正则表达式匹配串
3. pos：程序指针，指向当前分析位置
4. last\_row：上一次分析到的列号，取名为row是一时错误

方法：

* 1. load\_file\_by\_path：根据文件路径输入程序代码
  2. load\_file\_by\_content：根据文件内容输入程序代码
  3. dec2bin：十进制数转换为二进制数
  4. get\_token：#词法分析关键函数，它将文件内容按行处理，设置cur作内容指针，表明当前读取了几个字符， pos表明当前读取的行列数。每次进行正则表达式匹配，从当前字符位置往后匹配，使用re.match，这会匹配第一个符合条件的字符串，当一轮匹配中出现多个符合条件的字符串时，选择最长的字符串，这种情况只出现在标识符字符串内容开头为关键字或者双字符运算符的时候。若一轮匹配中没有出现符合条件的字符串，说明文件中出现了不规范的单词，程序报错。当识别到换行符时，要设置pos指向下一行的第一个字符，即行数加1，列数设置为1。
  5. print\_token oken：输出程序，当识别出单词类型为NUMBER时要转换为二进制数

## 3.2 paser.py

paser.by是语法分析器的主要部分，包含符号类、符号表类、指令类、指令表类、错误表类和P-code生成器类，程序架构如下：

1. declaration\_head：声明开头字符集
2. factor\_head：因子开头字符集
3. statement\_head：语句开头字符集
4. LEVEL\_MAX：最大嵌套层数
5. INT\_MAX：PL/0整数最大值
6. OPCode：P-code枚举类，表示P-code操作码
7. PCode：指令类

属性：f，l，a对应P-code指令中的FLA

1. Symbol：符号类

属性：

1. name：符号名字
2. kind：符号类型
3. val：符号的值
4. level：符号所处层次
5. adr：符号地址
6. PCodeEngine：指令表类

属性：code指令数组

方法：

* 1. get：返回指令数组
  2. gen：产生指令并存储到指令数组中
  3. clear：清空指令数组

1. SymbolTable：符号表类

属性：table符号表数组

方法：

1. get：查询符号表
2. enter：登录符号表
3. print：输出符号表
4. clear：清空符号表数组
5. ErrorManager：错误控制器

属性：

1. error：错误信息数组
2. errorMsg：错误信息字典，将错误序号与错误信息对应

方法：

1. add：根据错误的位置和错误序号添加错误信息到错误信息数组中
2. direct\_add：直接添加错误信息到错误信息数组中
3. print：打印错误信息
4. clear：清空错误信息数组
5. Parser：P-code生成器类

属性：

1. lexer：词法分析器实体
2. token：单词产生器
3. curToken：当前分析单词
4. curLevel：当前分析层次
5. table：符号表
6. p\_code：指令表
7. error：错误信息表

方法

1. load\_program：加载PL/0程序源代码
2. \_get\_symbol：词法分析读取一个单词
3. \_expect：判断得到单词是否符合文法
4. \_test：检查后继符号合法性，symbol\_set1为当前语法合法后继符号集合，symbol\_set2为停止符号集合，error\_num为错误编号
5. \_error\_pos：获取出错的准确位置
6. get\_result：返回字典格式的结果，用于前端显示
7. analyse：调用\_program()子程序开始语法分析，打印生成的P-code
8. \_program：程序子程序
9. \_block：分程序子程序
10. \_statement：语句子程序
11. \_condition：条件子程序
12. \_expression：表达式子程序
13. \_term：项子程序
14. \_factor：因子子程序

## 3.3 interpreter.py

interpreter.py是一个P-code解释器，将P-code在数据栈上解释执行

Interpreter：解释器类

属性：

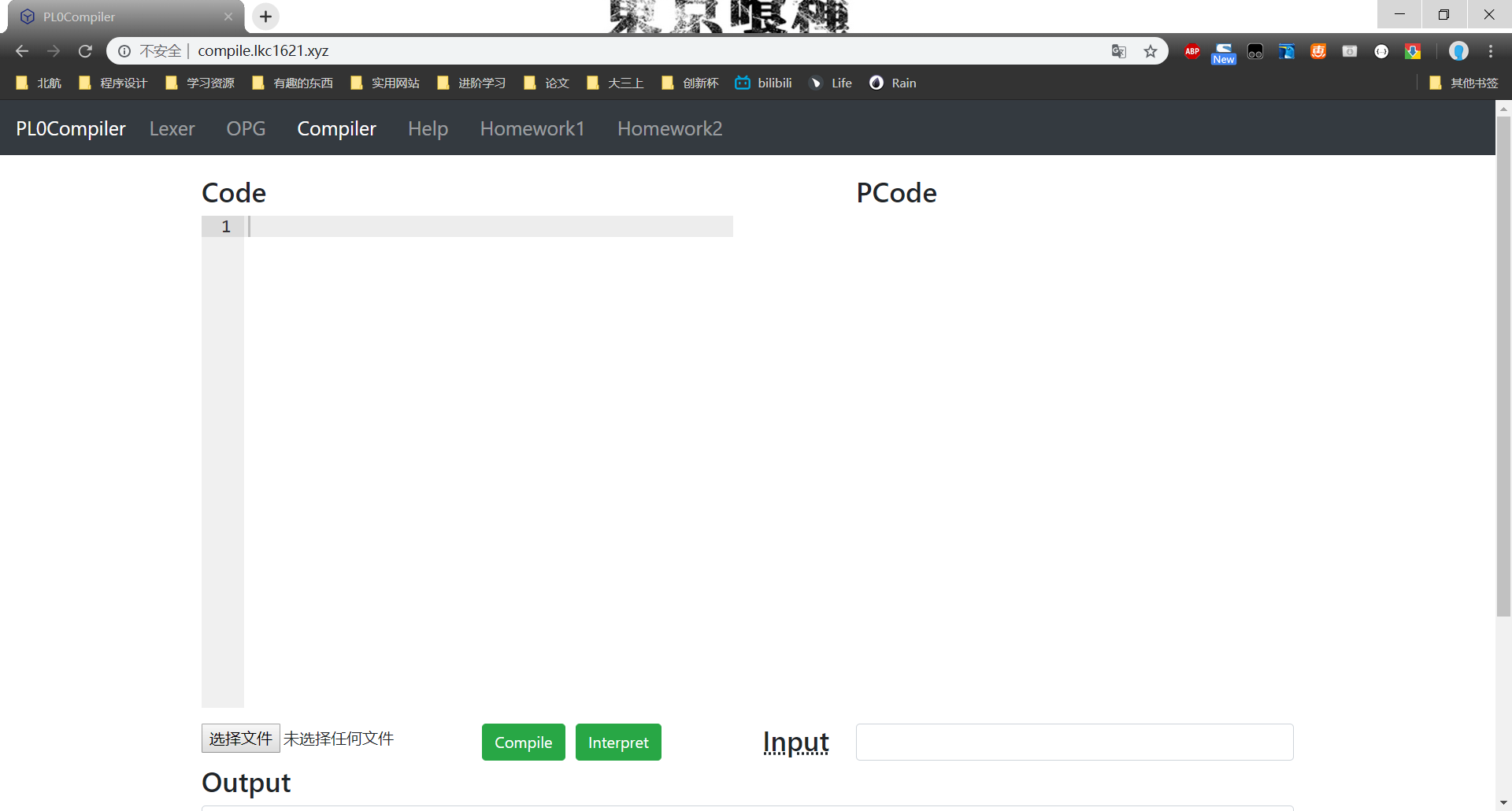
1. input：P-code输入
2. output：结果输出

方法：

1. interpret：调用\_interpret将P-code解释执行
2. \_interpret：解释P-code

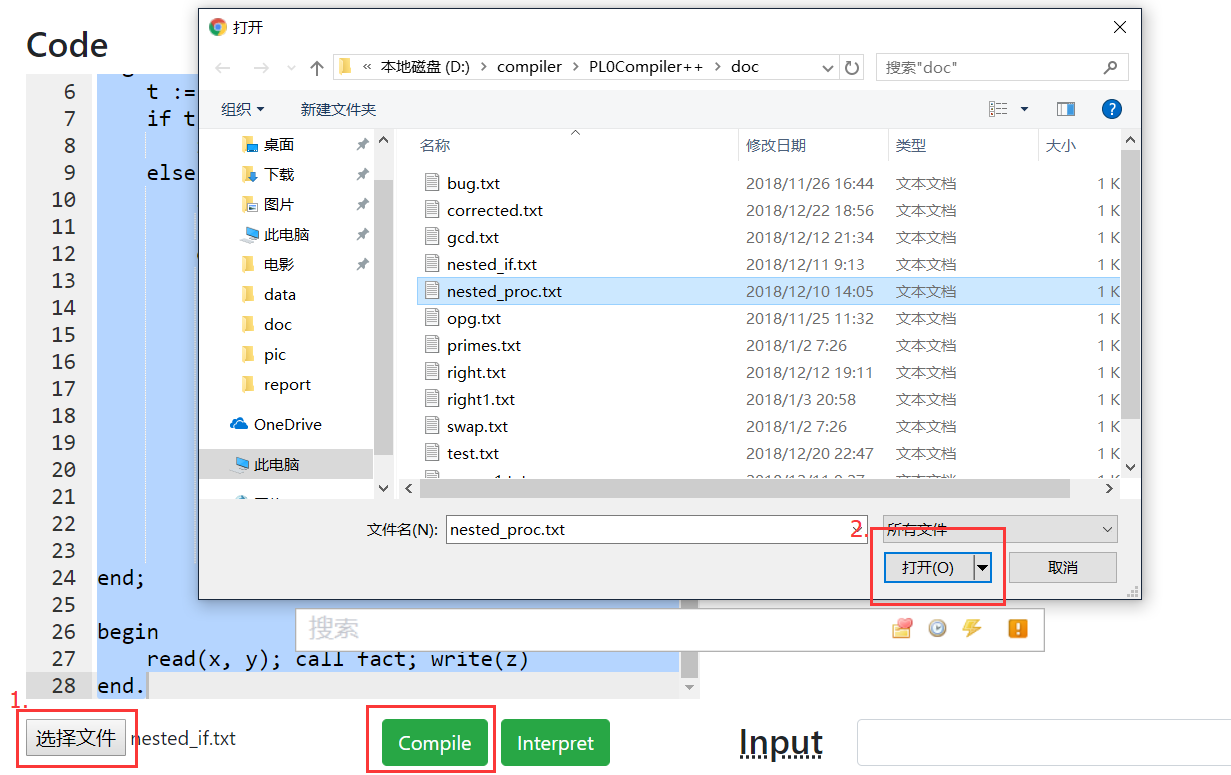
四、界面操作

程序可在网页进行访问<http://compile.lkc1621.xyz/>

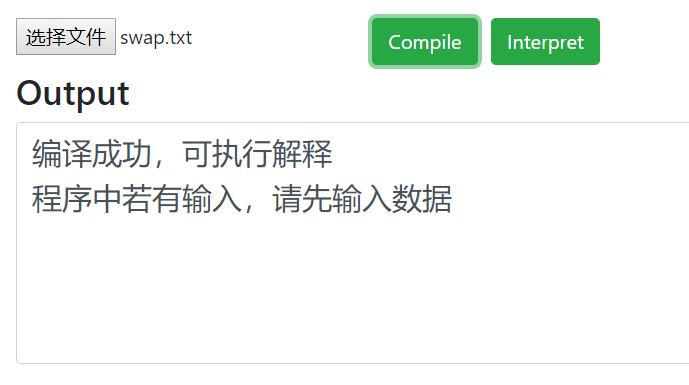


导航栏上有三个功能栏，Lexer是词法分析器，OPG是算符优先程序，Compiler是编译器，而Help为一些帮助信息，Homework1和Homework2是前两次的作业，默认进入的是编译器界面。

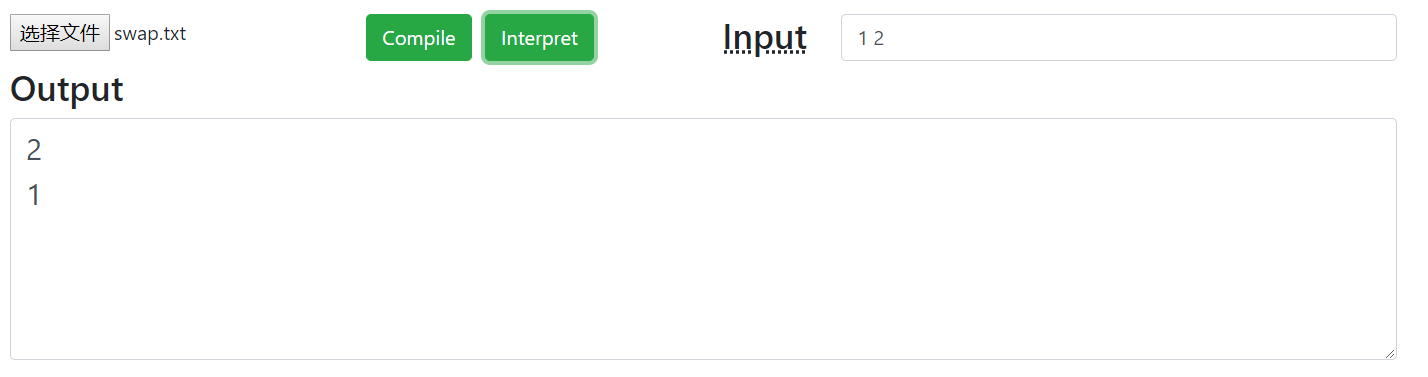
可直接在Code中输入源代码，也可选择文件输入。



点击Compile进行程序编译，若提示编译成功，可进行代码解释。



点击Interpret进行解释，若未输入网页会弹出对话框提示输入，输入完成后重新点金Interpret进行解释。



若提示编译失败，需要根据错误信息进行代码修改，之后重新编译。

