编译原理I

课程设计报告

班级：\_\_\_\_\_1616104\_\_\_\_\_\_

学号：\_\_\_\_161610116\_\_\_\_

姓 名：\_\_严令垚\_\_\_\_\_

2019-01

1. 设计任务

1.1**任务分析**

一个PASCAL语言子集（PL/0）编译器的设计与实现（具体要求见“编译原理课程设计与要求”文档）

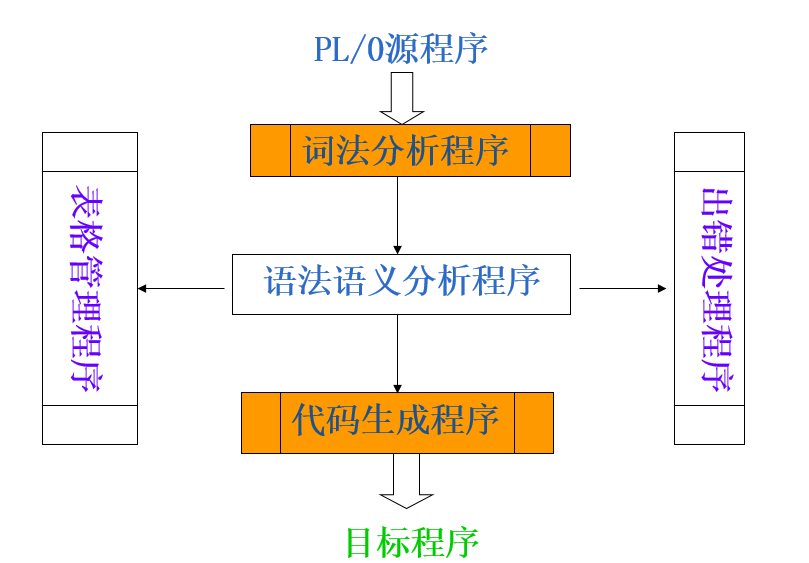


采用第三方编程语言完成编译器，通过运行该程序完成对PASCAL语言子集的编译任务，具体分为以下部分：  
**词法分析，语法分析，语义分析，代码生成，解释执行**

具体设计实现将在报告后续内容中给出。

1.2 **总体结构**

编译程序的总体结构如下图



通过词法分析将输入的源程序转化成独立的词法单元（lexical），以流传输的方式传输给语法分析程序，语法分析程序中继承了予以分析，代码生成程序，可以在一遍扫描结束后完成目标代码的生成。在语法分析处理过程中同时动态管理符号表，以及对错误产生一定的提示信息

1.3 开发工具

采用python语言开发编译器

编译器平台为Pycharm

1. 功能设计
   1. 词法分析器

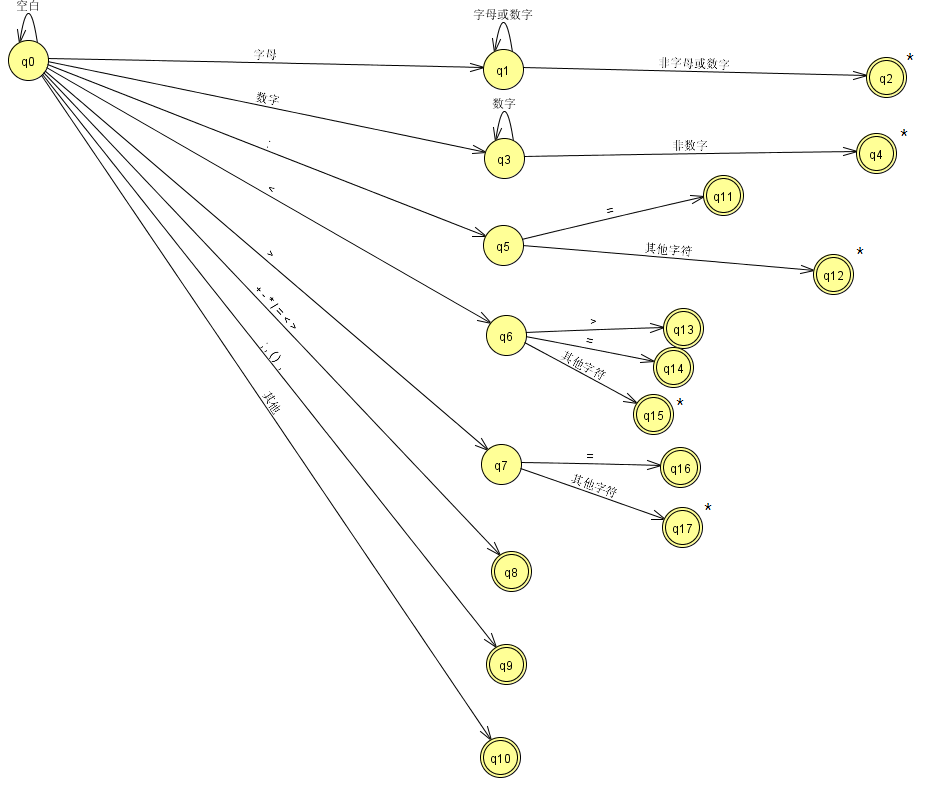
词法分析器的输入为源程序文本文件，输出为词法单元

具体设计如下：

设计过程参照课本P45

采用NFA来识别词法单元

状态转换图设计如下：



上图中：

q1,3,5,6,7,8,9,10分别代表输入为字母，数字，’:’，’<’，’>’,其他算符，界符，其他的情况

其中q1情况下，再次读取其后所有字母或数字，之后与关键字比较，若不是关键字，则标记为标识符

在q3情况下，读取其后所有数字，存作整数

在q5情况下，往后再读一个字符，若为=，则构成 := 算符，否则 ：是非法字符

在q6情况下，往后再读一个字符，若为>,则构成<>算符，若为=，则构成<=算符，否则，是<算符

在q7情况下，往后在读一个字符，若为=，则构成>=算符，否则，是>算符

在q8情况下，为对应的算符

在q9情况下，为对应的界符

在q10情况下，为非法字符，报错

程序设计思想：

类lexical(词法)设计如下：

lexical类属性表

|  |  |  |
| --- | --- | --- |
| 中文名 | 英文名 | 数据类型 |
| 保留字 | reservedWord | List |
| 算符 | calsign | List |
| 界符 | rimsign | List |
| 关键字 | id | List |
| 常量 | const | List |
| 源程序内容 | content | String |
| 结束符 | end | Char |
| 行号 | line | int |
| 列号 | col | int |
| 读指针 | i | int |
| 当前字符 | ch | Char |
| 当前串 | StrToken | String |

lexical类方法表

|  |  |  |
| --- | --- | --- |
| 中文名 | 英文名 | 方法描述 |
| 初始化 | \_\_init\_\_( self) | 打开源文件，生成源文件中没有的单个字符作为结束符，将源代码读入content属性中 |
| 读取一个字符 | GetChar(self) | 根据i指针位置，从content中读取一个字符，i指针下移，列号col+1 |
| 跳过空格 | GetBC(self) | 遇到空格跳过。但是遇到换行时行号line+1，列号col归零 |
| 连接字符 | Concat(self) | 将ch字符拼接到strToken尾部 |
| 是否保留字 | Reserve(self) | 若strToken在reservedWord中，返回索引，否则返回-1 |
| 回跳 | Retract(self) | i指针-1，列col-1 |
| 加入关键字 | InsertId(self,strToken) | 将stroToken加入id属性中 |
| 加入常数 | InsertConst(self,strToken) | 将strToken加入const属性中 |
| 解析 | analys(self) | 采用上述自动机模型进行词法单元的识别，当识别出一个词法单元时，返回该单元信息，反之继续识别 |

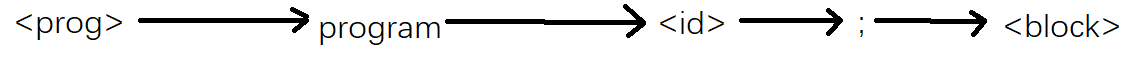
报错定位：通过line和col确定错误位置

* 1. 语法分析器

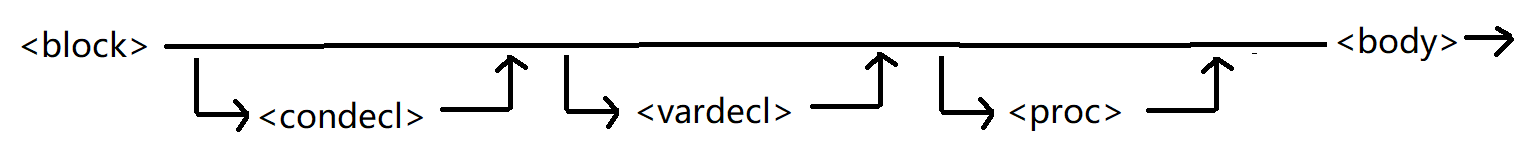
根据该文法可构造出每个产生式的语法分析图：

语法分析图：

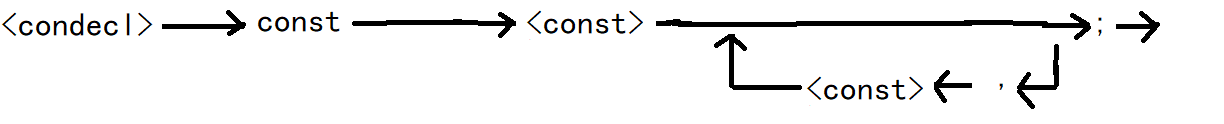
<prog> → program <id>；<block>



<block> → [<condecl>][<vardecl>][<proc>]<body>



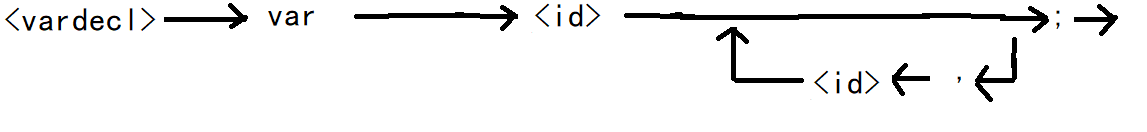
<condecl> → const <const>{,<const>};



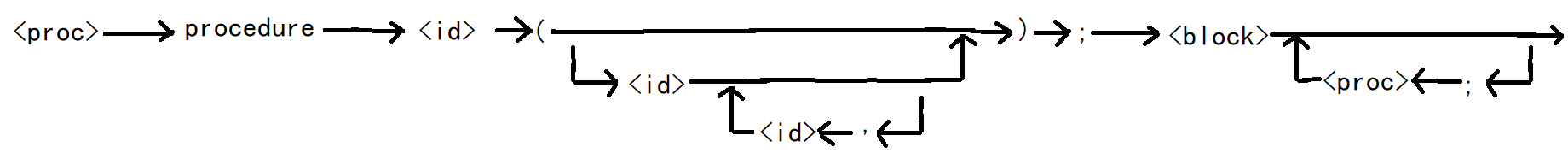
<const> → <id>:=<integer>



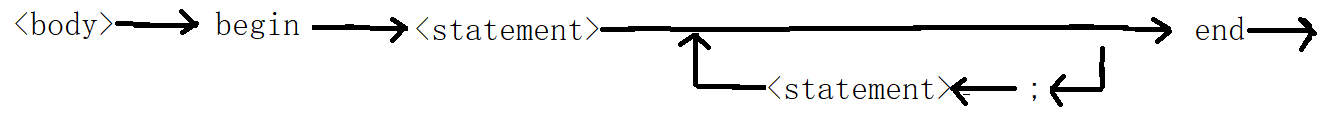
vardecl <vardecl> → var <id>{,<id>};



<proc> → procedure <id>（[<id>{,<id>]}）;<block>{;<proc>}



<body> → begin <statement>{;<statement>}end



<statement> → <id> := <exp>

|if <lexp> then <statement>[else <statement>]

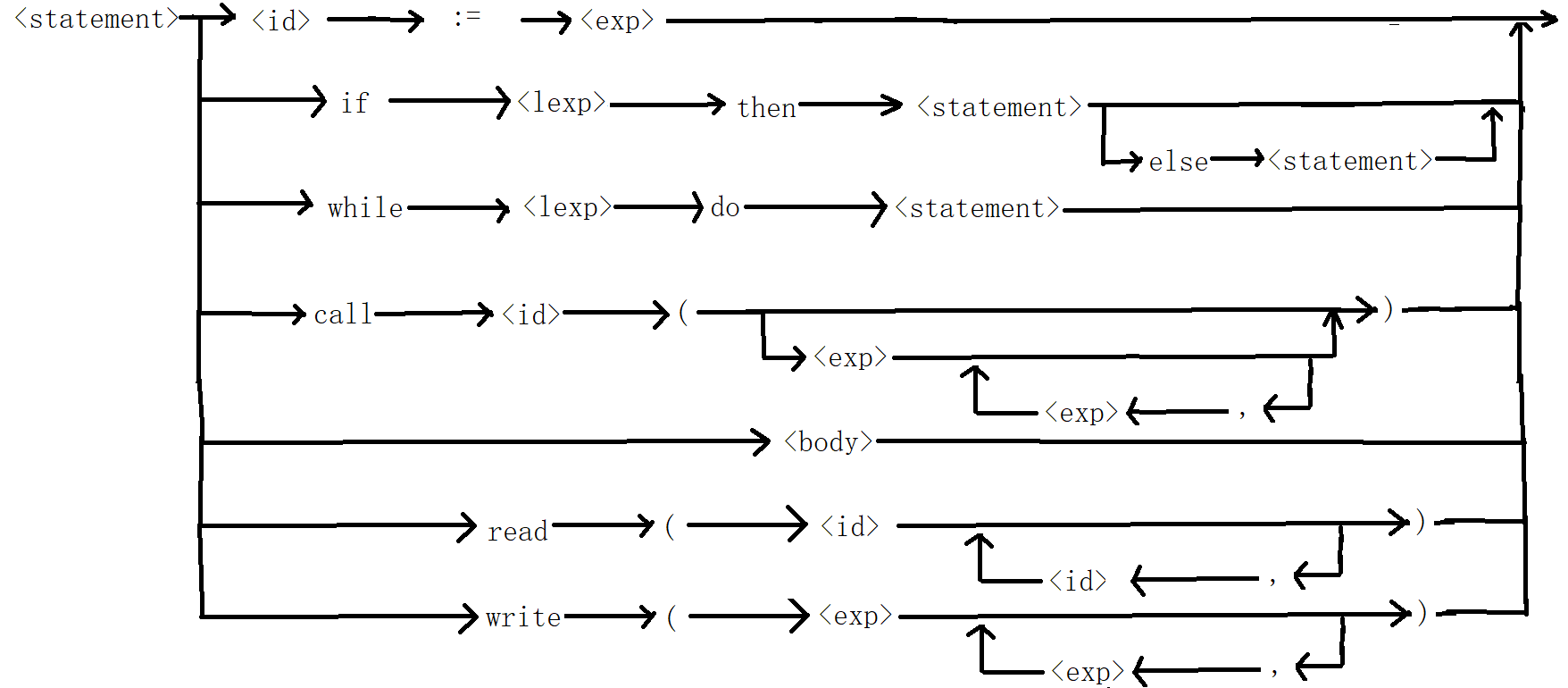
|while <lexp> do <statement>

|call <id>（[<exp>{,<exp>}]）

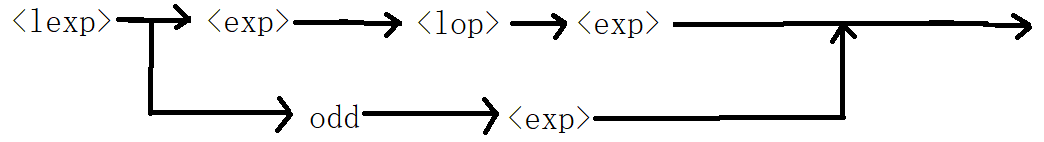
|<body>

|read (<id>{，<id>})

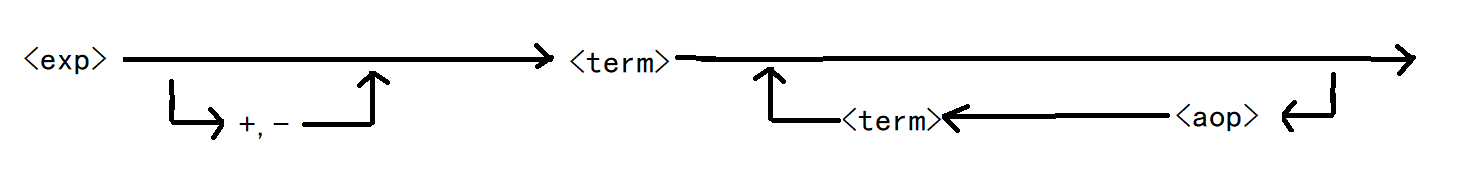
|write (<exp>{,<exp>})

****

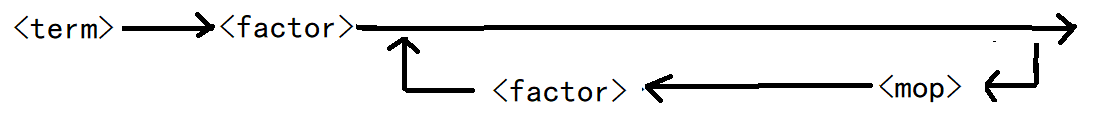
<lexp> → <exp> <lop> <exp>|odd <exp>



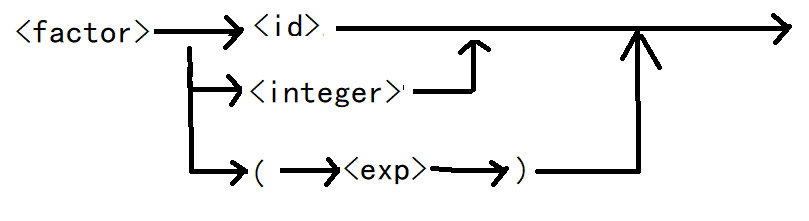
<exp> → [+|-]<term>{<aop><term>}

****

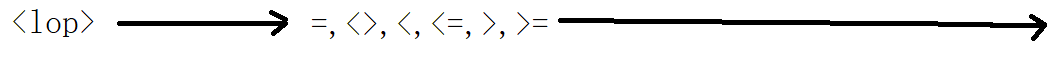
<term> → <factor>{<mop><factor>}

****

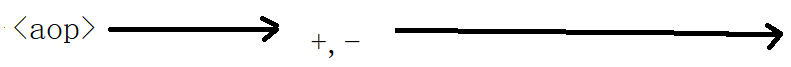
<factor>→<id>|<integer>|(<exp>)

****

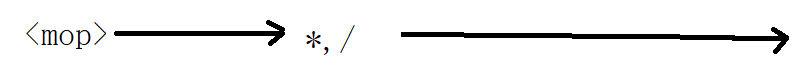
<lop> → =|<>|<|<=|>|>=



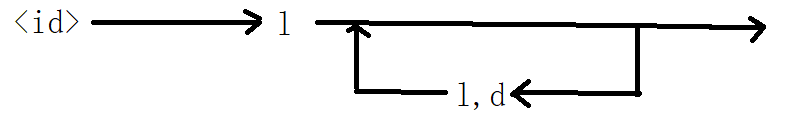
<aop> → +|-



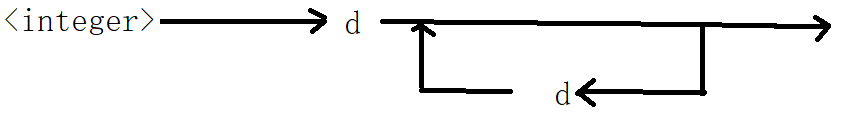
<mop> → \*|/



<id> → l{l|d} （注：l表示字母）



<integer> → d{d}

****

采用了递归下降的方法实现上述文法：  
 对于给定的产生式，以其左部命名一个函数，处理其有部产生式的内容

承接词法分析器的实现类（lexicalimpl）的设计：

lexicalimpl类属性表

|  |  |  |
| --- | --- | --- |
| 中文名 | 英文名 | 数据类型 |
| 词法单元索引 | i | int |
| 词法对象 | lex | lexical |
| 前一个词法单元(用于缺失时，找当前词法单元的前一个单元的位置) | previous | String（格式如下：“符号.符号类型.行号”） |
| 当前词法单元 | current | String |
| 后续词法单元 | next | String |

lexicalimpl类方法表

|  |  |  |
| --- | --- | --- |
| 中文名 | 英文名 | 方法说明 |
| 取词法单元 | sym(self) | 返回current的符号 |
| 取得类别 | gettype(self) | 返回current的类别 |
| 取得行号 | getline(self) | 返回current的行号 |
| 取得前一个单元的行号 | getpreline(self) | 返回previous的行号 |
| 后移 | advance(self) | i属性+1，将current赋值给previous，将next赋值给current，调用lexical的analyse方法赋值给next （平行右移） |

语法分析（parser类）的设计：

parser类属性表

|  |  |  |
| --- | --- | --- |
| 中文名 | 英文名 | 数据类型 |
| 最内层的符号表 | topTable | Stable |
| 过程信息（记录过程的代码起始位置，参数个数） | procs | List |
| 代码段 | Codes | PcodeTable |
| 嵌套层次 | level | int |
| 是否出错 | iserror | int |
| 变量在活动记录中的相对位置 | dx | int |
| 词法实现 | lex | lexicalimpl |
| 起始集 | first | Dictionary |
| 错误信息 | errorinfo | Dictionary |
| 已报告错误的词法单元下标 | reportedi | Dictionary |

parser类方法表

|  |  |  |
| --- | --- | --- |
| 中文名 | 英文名 | 方法说明 |
| 初始化 | \_\_init\_\_(self) | 初始化属性值 |
| 程序 | prog(self) | 按照语法分析图对表达式右部进行分析 |
| 块 | block(self) | 按照语法分析图对表达式右部进行分析 |
| … | 对应文法左部 | 按照语法分析图对表达式右部进行分析 |
| 出错处理 | error(self) | 根据错误信息报错 |
| 开始集判断 | infirst(self) | 返回当前词法单元是否在给定非终结符的first集合中，返回布尔值类型 |
| 寻找 | find(self) | 给定非终结符（或终结符，即目前需要的词法单元）及其之后的非终结符，搜索直到当前词法单元为其first集合或为给定终结符，或拼写出错(字符串相似度>60%)，或者是读到之后非终结符的first集合中的词法单元（处理为缺失当前所需的first集合的字符），否则产生对应的错误处理 |

* 1. 语义分析
     1. 符号表项（Symbol类）的设计

Symbol类属性表

|  |  |  |
| --- | --- | --- |
| 中文名 | 英文名 | 数据类型 |
| 名称 | name | String |
| 类型 | kind | String |
| 值（常量值和过程的形参个数） | val | int |
| 层次 | level | int |
| 偏移 | adr | int |
| 是否形参 | form | bool |
| 最大表项个数 | maxnum | int |

Symbol类方法表

|  |  |  |
| --- | --- | --- |
| 中文名 | 英文名 | 方法描述 |
| 初始化 | \_\_init\_\_(self) | 初始化属性初值 |
| 设置表项 | set(self,name,kind,level,adr,val,form) | 给定名称，类型，层次，偏移，值，是否形参等参数。根据其类型产生不同的赋值策略 |
| 打印方法 | \_\_str\_\_(self) | 返回格式化的字符串，便于print()方法输出 |

* + 1. 符号表(Stable类)的设计

STable类属性表

|  |  |  |
| --- | --- | --- |
| 中文名 | 英文名 | 数据类型 |
| 符号表 | table | STable |
| 外层表 | outer | STable |

STable类方法表

|  |  |  |
| --- | --- | --- |
| 中文名 | 英文名 | 方法描述 |
| 初始化 | \_\_init\_\_(self) | 初始化属性初值 |
| 创建表链 | MakeTable(self,outer) | 给定外层符号表和当前符号表，将外层符号表赋值给当前符号表的outer参数 |
| 增加表项 | Enter(self,name,kind,level=0,adr=0,val=0,form=False) | 给定表项的名称，类型，层次，偏移，值，是否形参等参数，赋值给新表项并加入table属性中 |
| 取表项 | GetSymbol(self,s) | 又内层表向外层表查找名字与给定字符串相同的表项，将其返回；若未找到返回None |
| 返回当前函数的形参 | GetFormalId(self) | 外层表的倒数第一项就是当前函数（因为进入block建表，而proc中的函数名是在上层表中最后加入符号表的），根据其参数值val，逆序查找val个表项加入列表中，这个列表就是形参表 |
| 返回最后一项 | GetLast(self) | 返回当前列表的最后一项 |
| 返回当前表中多余的形参所占空间个数 | GetExtraFormalParamNum(self) | 查找当前表中的函数的形参个数（形参在函数名之前如表），累加到num后返回 |
| 打印当前符号表 | show(self) | 输出当前表的所有表项 |
| 打印当前表链 | showall(self) | 由内到外输出每个表 |

符号表采用链表的形式实现，每次进入<block>块时创建一个符号表，同时将其outer属性写为当前符号表的指针，完成链接。

* + 1. 符号表的管理

对每个产生式设计相应的语义动作来完成符号表的管理，具体如下：（蓝色大括号内为语义动作）(之前未提及的属性变量为函数中的临时变量)

<prog> → program <id>；<block>

<block>→***{* saved=topTable;topTable=STable();topTable.MakeTable(saved) *}***[<condecl>][<vardecl>][<proc>]<body>***{* topTable=saved *}***

<condecl> → const <const>{,<const>};

<const>→***{* id=id.lex *}***<id>:=***{* value=integer.num *}***<integer>***{* topTable.Enter(id,“const”,value) *}***

<vardecl>→var ***{* id=id.lex *}***<id>***{* topTable.Enter(id,“var”,level,dx) *}***{,***{* id=id.lex *}***<id>***{* topTable.Enter(id,“var”,level,dx) *}***};

<proc>→procedure***{*** ***pro*id=id.lex *}***<id>（[***{* id=id.lex *}***<id>***{* topTable.Enter(id,“var”,level+1,4,0,True);paramnum++ *}***{,***{* id=id.lex *}***<id>***{* topTable.Enter(id,“var”,level+1,4,0,True) ;paramnum++ *}***}]）; ***{* topTable.Enter(proid,“procedure”,level,Codes.i,paramnum) *}***<block>{;<proc>}

* 1. 代码生成
     1. 伪代码项（Pcode类）的设计：

Pcode类属性表

|  |  |  |
| --- | --- | --- |
| 中文名 | 英文名 | 数据类型 |
| 操作符 | F | String |
| 层差 | L | int |
| 偏移 | A | int |

Pcode类方法表

|  |  |  |
| --- | --- | --- |
| 中文名 | 英文名 | 方法描述 |
| 初始化 | \_\_init\_\_() | 初始化属性初值 |
| 设置代码项 | set() | 根据给定的三个参数对应赋值 |
| 打印转换 | \_\_str\_\_() | 返回便于print()输出的格式化字符串 |

* + 1. 代码表（PcodeTable类）的设计：

PcodeTable类属性表

|  |  |  |
| --- | --- | --- |
| 中文名 | 英文名 | 数据类型 |
| 代码项指针 | i | int |
| 代码表 | Codes | List |

PcodeTable类方法表

|  |  |  |
| --- | --- | --- |
| 中文名 | 英文名 | 方法描述 |
| 初始化 | \_\_init\_\_(self) | 初始化属性初值 |
| 生成代码 | gen(self,f,l,a) | 新建Pcode对象，根据给定的参数赋值后加入Codes列表中，指针i+1 |
| 显示 | show(self) | 打印所有的代码 |
| 回填 | backPatch(self, i, adr) | 根据给定的指令指针和偏移设定对应代码的偏移值 |
| 解释执行 | interpreter(self, procs,debug=False) | 又内层表向外层表查找名字与给定字符串相同的表项，将其返回；若未找到返回None |

* + 1. 生成动作

对每个产生式设计相应的语义动作来完成代码的生成，具体如下：（蓝色大括号内为语义动作）(之前未提及的属性变量为函数中的临时变量)

<prog> → program <id>；<block>

<block>→***{* jmpPoint=Codes.i;gen(JMP,0,0) *}***[<condecl>][<vardecl>][<proc>]***{* backPatch(jmpPoint,Codes.i);gen(INT, 0, dx) *}***<body>***{* gen(OPR, 0, 0) *}***

<condecl> → const <const>{,<const>};

<const> → <id>:=<integer>

<vardecl> → var <id>{,<id>};

<proc>→procedure<id>（[<id>{,<id>}]）;<block>{;<proc>}

<body> → begin <statement>{;<statement>}end

<statement> → ***{ id = id.lex adr = id.adr}***<id> := <exp> ***{* gen(STO, leveldiff, adr) *}***

|if <lexp>***{* jmpPoint=Codes.i;gen(JPC,0,0) *}*** then <statement>***{ backPatch(jmpPoint,Codes.i) }***[else ***{* backPatch(+1);tempPoint=Codes.i;gen(JMP,0,0) *}***<statement>***{* backPatch(tempPoint,Codes.i) *}***]

|while ***{* conditionPoint=Codes.i *}***<lexp> do ***{* jmpPoint=Codes.i;gen(JPC,0,0) *}***<statement>***{* gen(JMP,0,conditionPoint);backPatch(jmpPoint,Codes.i) *}***

|call <id>（[<exp>{,<exp>}]）***{* gen(CAL,leveldiff,adr) *}***

|<body>

|read (<id>***{* gen(RED,leveldiff,adr) *}***{，<id>***{* gen(RED,leveldiff,adr) *}*** })

|write (<exp>***{* gen(WRT,0,0) *}***{,<exp>}***{* gen(WRT,0,0) *}***)

<lexp> → <exp> <lop> <exp>***{* gen(OPR,0, lop) *}***|odd <exp>***{* gen(OPR,0,6) *}***

<exp> → [+|-]<term>***{* gen(OPR,0,+/-) *}***{<aop><term>***{* gen(OPR,0,aop) *}***}

<term> → <factor>{<mop><factor>***{* gen(OPR,0,mop) *}***}

<factor>→<id>***{* if id.kind = const gen(LIT,0,id.val) else gen(LOD,leveldiff,adr) *}***|<integer>***{* gen(LIT,0,integer.val) *}***|(<exp>)

<lop> → =|<>|<|<=|>|>=

<aop> → +|-

<mop> → \*|/

<id> → l{l|d} （注：l表示字母）

<integer> → d{d}

* 1. 解释执行
     1. 运行时栈的设计

Stack, 栈，变量类型为List

maxStackSize, 栈最大空间，变量类型为int

code，当前代码，变量类型为String

sp，栈顶指针，变量类型为int

bp，当前活动记录基地址，变量类型为int

* + 1. 活动记录的设计

|  |  |
| --- | --- |
| 栈高地址 | 变量存储空间 |
|  | 参数个数个空间存放形参 |
|  | 参数个数 |
|  | display表 |
|  | 返回地址 |
| 栈低地址 | 老bp |

* + 1. 解释执行

解释执行具体在PcodeTable类中的interpreter实现，根据不同的指令产生相应的动作，在运行时栈上进行操作

1. 功能实现

具体实现参见代码：

代码组织如下：

测试代码：sourcecode.txt

词法分析器：lexical.py

语法语义分析器(主程序): parser.py

符号表及表项：symbols.py

代码表及表项：Pcodes.py

1. 运行测试
   1. 测试用PL/0语言源码：

program abc;  
const a:=10;  
var b,c;  
procedure p(x,y);  
begin  
c:=b+x;  
write(x);  
if c < 20 then  
call p(x,y)  
end  
begin  
read(b);  
while b<>0 do  
begin  
call p(a,b);  
write(2\*c);  
read(b)  
end  
end

上述代码描述了一个名为abc的程序，其中含有过程p（x,y）；主程序中读取变量b的值并判断当其不为0是调用p函数，参数x，y分别为a，b。过程p中当c小于阈值时递归调用自己

* 1. 测试结果

(0) JMP 0 18

(1) JMP 0 2

(2) INT 0 6

(3) LOD 1 4

(4) LOD 0 4

(5) OPR 0 2

(6) STO 1 5

(7) LOD 0 4

(8) WRT 0 0

(9) OPR 0 15

(10) LOD 1 5

(11) LIT 0 20

(12) OPR 0 10

(13) JPC 0 17

(14) LOD 0 4

(15) LOD 0 5

(16) CAL 1 1

(17) OPR 0 0

(18) INT 0 6

(19) RED 0 4

(20) LOD 0 4

(21) LIT 0 0

(22) OPR 0 9

(23) JPC 0 34

(24) LIT 0 10

(25) LOD 0 4

(26) CAL 0 1

(27) LIT 0 2

(28) LOD 0 5

(29) OPR 0 4

(30) WRT 0 0

(31) OPR 0 15

(32) RED 0 4

(33) JMP 0 20

(34) OPR 0 0

input:1

output:10

output:10

output:22

input:0

Process finished with exit code 0

第一次输入b为1，递归调用两次p，输出两次x的值（a）为10，调用结束后输出2\*c,其值为22；第二次输入b为0，结束while循环，程序运行结束，测试结果正确

1. 试验体会

通过这次编译原理的课程设计，我成功的将课本上的理论付诸实践，更加深刻的体会了其含义，理论不是凭空产生的，都是因为在生产，生活中遇到了问题，提出了解决方案，后来才被总结为理论知识。通过亲身实践，实现每一个模块的功能，我体会到了编译器结构中词法分析，语法分析，语义分析和代码生成及执行之间环环相扣，紧密相连的关系，也对编译，程序执行有了更加底层的认识。虽然我们本次课设没有中间代码生成以及优化等过程，但是直接生成伪代码（目标代码）并解释执行降低了我们实现起来的难度，让我们能在规定时间内完成。同时，结合本学期软件工程相关知识，我对编程，尤其是工程系统开发有了完全不同的认识，在软件工程课中我们学习了软件开发中的各种方法，例如面向对象，自顶向下等。在本次开发过程中我都深切的感受到了，从一开始的模块划分，到各种表的设计，再到每个类的方法的设计实现。每一次实现设计后发现问题，单步调试并找到问题的所在后都能给予我巨大的成就感。最后，非常感谢老师给我们提供了这样的实践机会，也感谢老师这几天在实验室孜孜不倦的陪伴！

1. 参考资料

|  |  |  |
| --- | --- | --- |
| 编号 | 参考资料 | 作者 |
| 1 | 《编译原理》（第二版），机械工业出版社 | Alfred V.Aho等著,赵建宇等译 |
| 2 | 《程序设计语言 编译原理》（第三版），国防工业出版社 | 陈火旺等 |