PL/0编译器

一、实验要求

要求：设计并实现一个PL/0语言的编译器，能够将PL/0语言翻译成P-code语言（具体语言描述见《编译原理》（第3版），清华大学出版社，王生原等编著）。

完成时间：2020.6.1-6.20

作业提交形式：实验报告+源代码，提交到学院ftp服务器上。

提交截止时间：6.20

二、PL/0语言描述

PL/0型语言是Pascal语言的一个子集。作为一门教学用程序设计语言，它比PASCAL语言简单，并作了一些限制。PL0的程序结构比较完全，相应的选择，不但有常量，变量及过程声明，而且分支和循环结构也是一应俱全。

> PL/0文法的EBNF所示：
>
> <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

三、程序功能及介绍

本工程实现了对PL/0源代码的词法分析，语法分析，语义分析及生成Pcode代码，还对Pcode的代码进行了解释，使其能在Java虚拟机上运行。

3.1 运行方法

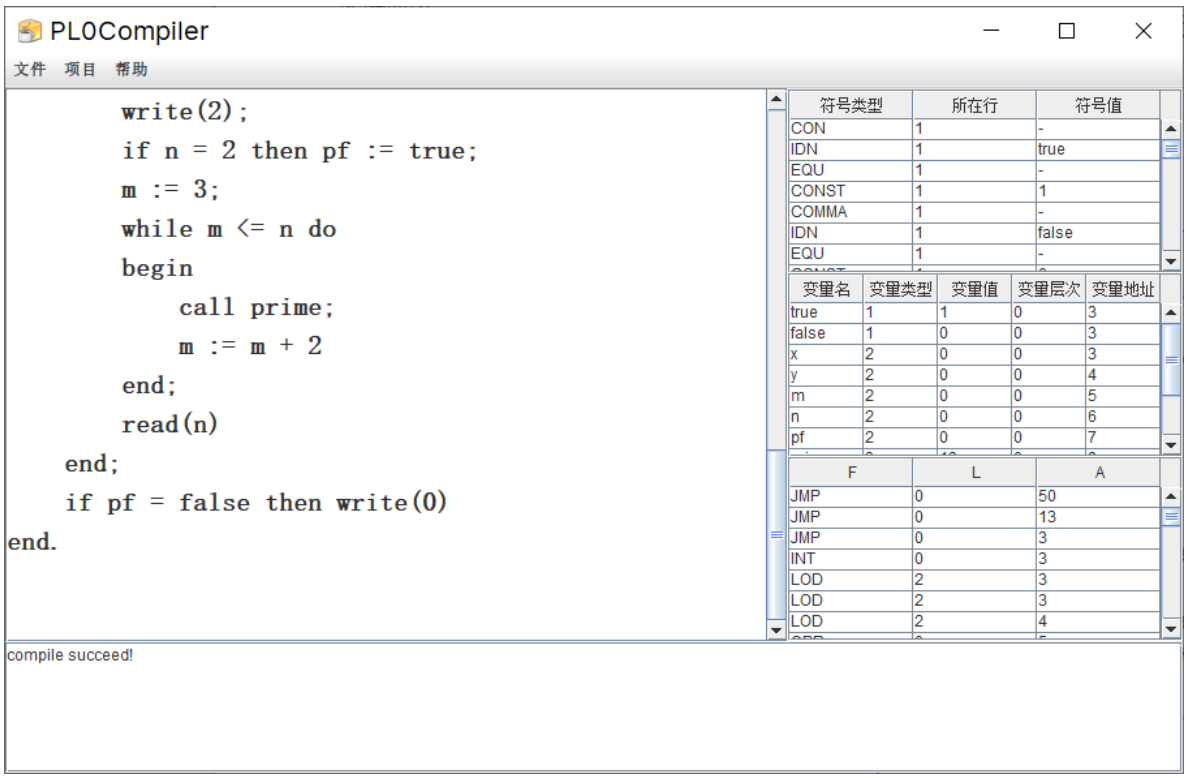
项目已经编译好了，只需要切换到文件目录下在终端运行：

```
$ java Main
```

即可，若想要自行重新编译，可以运行：

```
$ javac *.java
```

3.2 运行截图



3.3 功能介绍

项目整个界面分为导航栏，代码区，token表区，符号表区，Pcode区以及控制台。

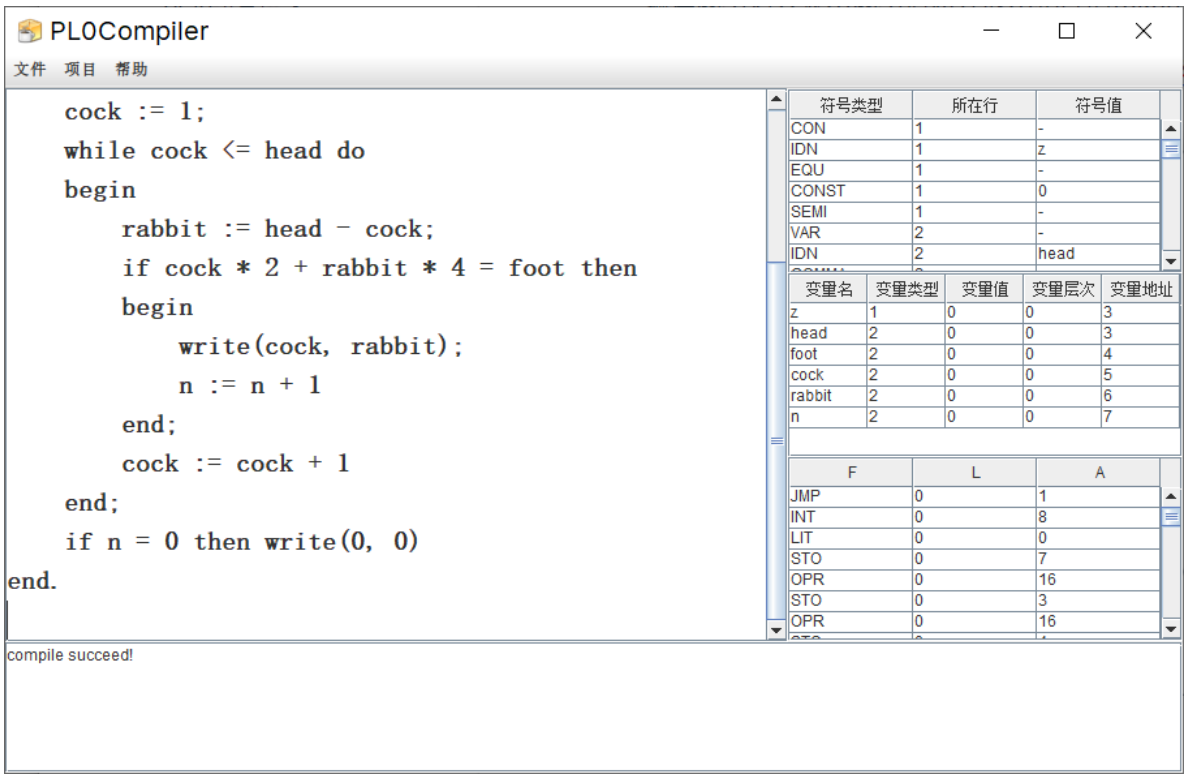
项目各区主要功能介绍:

导航栏		**打开，关闭或保存文件；编译和执行代码**
代码区		展示并编辑代码
Token表区		展示代码中的token
Symbol表区		展示代码编译过程中生成的symbol表
Pcode表区		展示代码生成的所有Pcode
控制台		输出编译信息

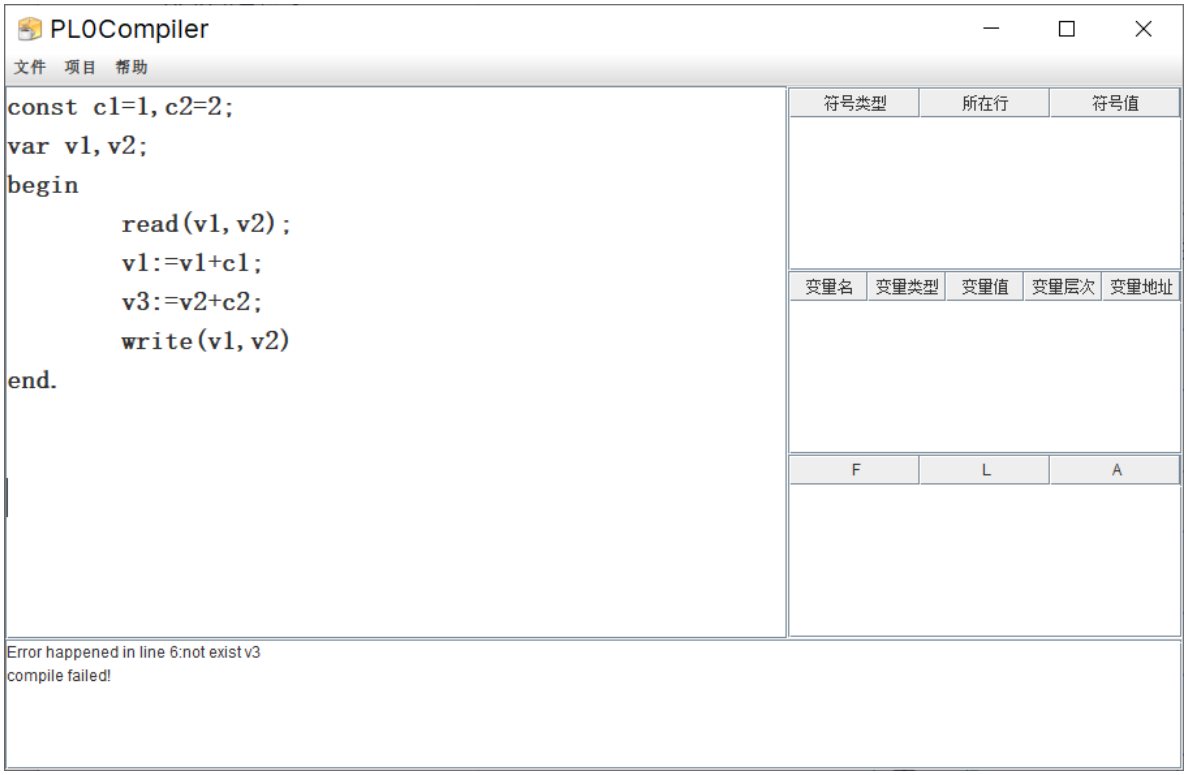
3.3.1 功能演示

打开一段代码到代码区，选择 项目→编译。

编译成功则会显示成功信息并展示相应的token表，symbol表及Pcode表。



编译失败则会给出错误信息。



四、项目架构

一个经典的编译程序一般包括7个部分：词法分析，语法分析，语义分析及代码生成，代码优化（可省略），代码执行，符号表管理，出错管理。这7个部分之间的关联关系如下图所示：

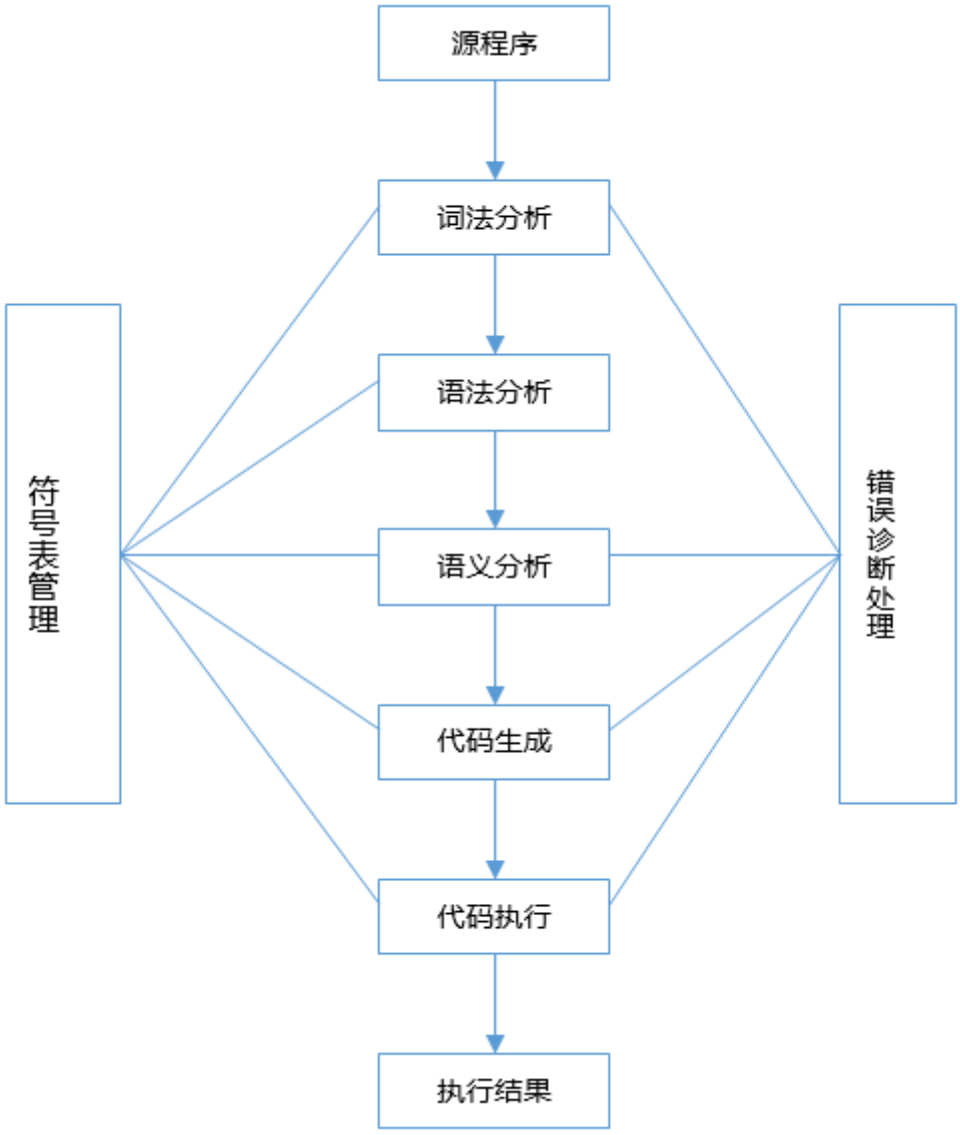
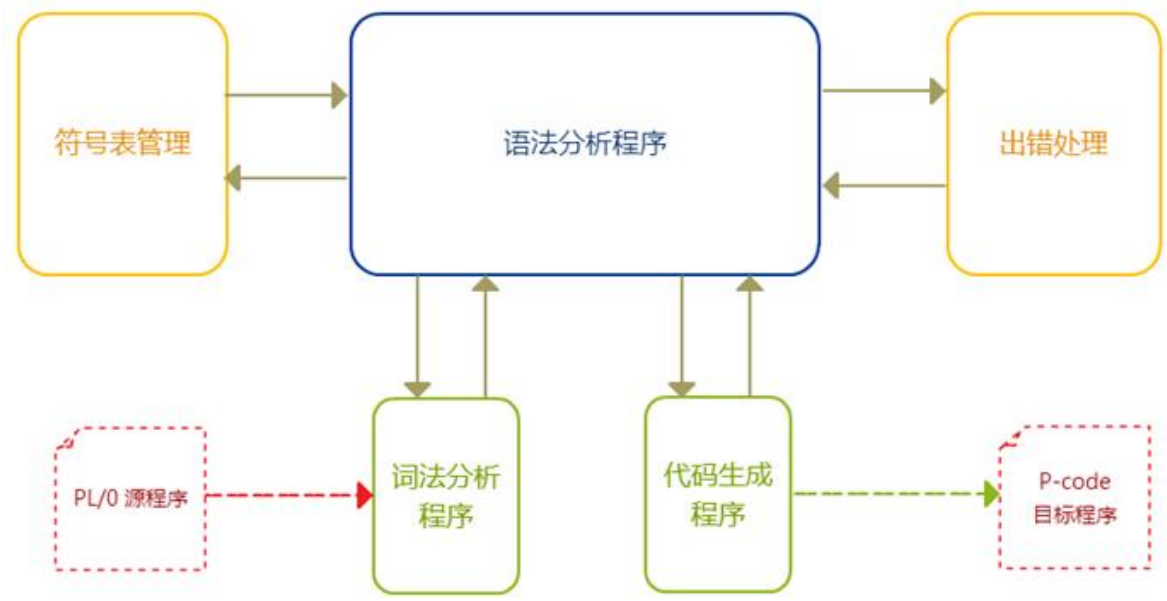


图4 编译程序各部分之间的关联关系

本项目将整个编译过程分为了3个部分：

1. 词法分析将PI/0源代码分为一个个token
2. 语法分析和语义分析同时进行，生成Pcode（包括符号表管理和出错管理）
3. 对Pcode进行解释执行，使其在java虚拟机上能够运行

整个编译过程可以如下图所示：



由上图我们可以看到，整个编译程序分为四个小部分和一个综合部分。

下面，对本工程中的各个部分一一做介绍：

4.1 词法分析

PL/0编译系统中所有的字符，字符串的类型为，如下表格：

表2 PL/0编译系统中所有的字符，字符串的类型

保留字	**begin, end, if,then, else, const, procedure, var,do,while, 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	10,25等整数
界符	‘,’ , ‘.’ , ‘;’ , ‘(’ , ‘)’
其他符号	： , EOF

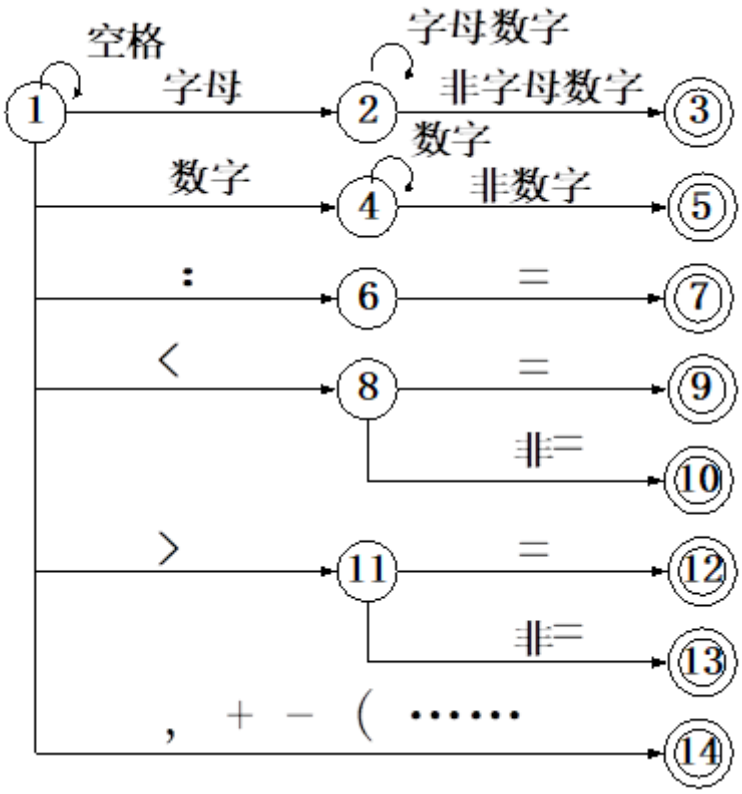
每个token的结构定义:

```
public class Token {
    /**
     * token的类别
     */
    private final SymType st;
    /**
     * token所在行，错误处理使用
     */
    private final int line;
```

```
    /**
     * token的值，只有标识符和常量有值
     */
    private final String value;
}
```

LexAnalysis.java是本项目的词法分析类。其中，doAnalysis()函数对源代码进行了一遍遍历，将源代码中分析成为了一个token数组。下面重点介绍每一个Token的分析程序analysis()。

状态转换图如下：



具体代码实现如下：

```
private Token analysis() {
    //当前正在进行词法分析的字符串
    StringBuilder strToken = new StringBuilder();
    getChar();
    while ((ch == ' ' || ch == '\n' || ch == '\t' || ch == '\0') && searchPtr <
buffer.length) {
        if (ch == '\n') {
            line++;
        }
        getChar();
    }
    //到达文件末尾
    if (ch == '$' && searchPtr ≥ buffer.length) {
        return new Token(SymType.EOF, line, "-1");
    }
    //首位为字母，可能为保留字或者变量名
    if (isLetter()) {
        while (isLetter() || isDigit()) {
            strToken.append(ch);
            getChar();
        }
        retract();
        for (int i = 0; i < keyWords.length; i++) {
```

```
            //说明是保留字
            if (strToken.toString().equals(keyWords[i])) {
                return new Token(SymType.values()[i], line, "-");
            }
        }
        //不是保留字，则为标识符，需要保存值
        return new Token(SymType.IDN, line, strToken.toString());
    } else if (isDigit()) {
        //首位为数字，即为整数
        while (isDigit()) {
            strToken.append(ch);
            getChar();
        }
        retract();
        return new Token(SymType.CONST, line, strToken.toString());
    } else if (ch == '=') {
        return new Token(SymType.EQU, line, "-");
    } else if (ch == '+') {
        return new Token(SymType.ADD, line, "-");
    } else if (ch == '-') {
        return new Token(SymType.SUB, line, "-");
    } else if (ch == '*') {
        return new Token(SymType.MUL, line, "-");
    } else if (ch == '/') {
        return new Token(SymType.DIV, line, "-");
    } else if (ch == '<') {
        getChar();
        if (ch == '=') {
            return new Token(SymType.LESE, line, "-");
        } else if (ch == '>') {
            return new Token(SymType.NEQE, line, "-");
        } else {
            retract();
            return new Token(SymType.LES, line, "-");
        }
    } else if (ch == '>') {
        getChar();
        if (ch == '=') {
            return new Token(SymType.LARE, line, "-");
        } else {
            retract();
            return new Token(SymType.LAR, line, "-");
        }
    } else if (ch == ',') {
        return new Token(SymType.COMMA, line, "-");
    } else if (ch == ';') {
        return new Token(SymType.SEMI, line, "-");
    } else if (ch == '.') {
        return new Token(SymType.POI, line, "-");
    } else if (ch == '(') {
        return new Token(SymType.LBR, line, "-");
    } else if (ch == ')') {
        return new Token(SymType.RBR, line, "-");
    } else if (ch == ':') {
        getChar();
        if (ch == '=') {
            return new Token(SymType.CEQU, line, "-");
        } else {
```

```
                retract();
                return new Token(SymType.COL, line, "-");
            }
        }
        return new Token(SymType.EOF, line, "-");
    }
```

4.2 符号表管理

4.2.1 符号表结构

每个符号定义的形式:

```
public class PerSymbol {

    /**
     * 表示常量、变量或过程
     */
    private final int type;
    /**
     * 表示常量或变量的值
     */
    private int value;
    /**
     * 嵌套层次
     */
    private final int level;
    /**
     * 相对于所在嵌套过程基地址的地址
     */
    private final int address;
    /**
     * 表示常量，变量，过程所占的大小(这一项其实默认为0，  并没有用到)
     */
    private final int size;
    /**
     * 变量、常量或过程名
     */
    private final String name;
}
```

我们可以看到，每个symbol明显要比token复杂的多，相关变量也复杂的多。其中level和address在运行时起到非常大的作用。

4.2.2 符号表管理

各函数名功能表示图:

函数名	功能
enterConst	向符号表中插入常量
enterVar	向符号表中插入变量
enterProc	向符号表中插入过程
isNowExists	在符号表当前层查找变量是否存在
isPreExists	在符号表之前层查找符号是否存在
getSymbol	按名称查找变量
getLevelProc	查找当前层所在的过程

4.2.3 语法分析和语义分析

本项目使用了递归下降子程序法，对每一个PL/0中的语法成分都进行了分析，并单独编写为一个过程。

各函数名功能表示图:

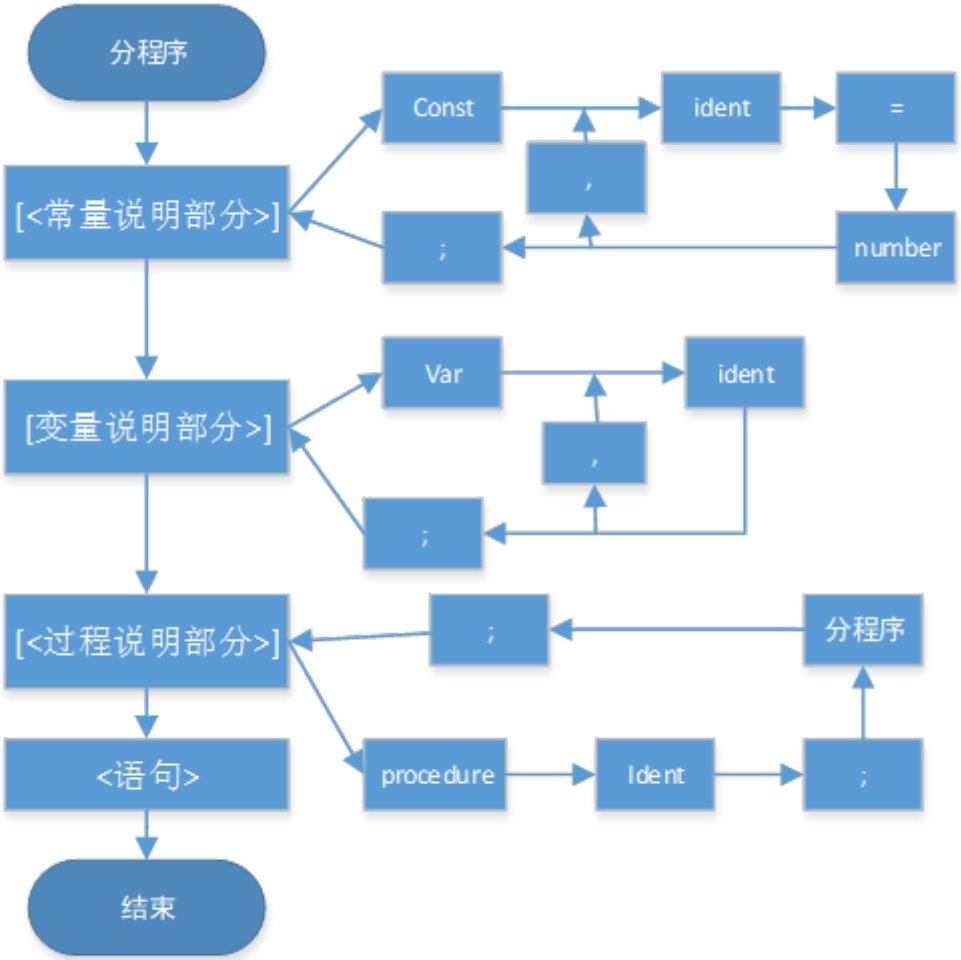
函数名	功能
program()	<主程序>::=<分程序>.
block()	<分程序>::=[<常量说明部分>][<变量说明部分>][<过程说明部分>]<语句>
conDeclare()	<常量说明部分>::=const <常量定义>{,<常量定义>}
conHandle()	<常量定义>::=<标识符>=<无符号整数>
varDeclare()	<变量说明部分>::=var<标识符>{,<标识符>}
proc()	<过程说明部分>::=<过程首部><分程序>{;<过程说明部分>}; <过程首部>::=procedure<标识符>;
body()	<复合语句>::=begin<语句>{;<语句>}end
statement()	<语句>::=<赋值语句> | <条件语句> | <当循环语句> | <过程调用语句> | < 复合语句> | <读语句> | <写语句> | <空>
condition()	<条件>::=<表达式><关系运算符><表达式> | odd<表达式>
expression()	<表达式>::=[+|-]<项>{<加法运算符><项>} <加法运算符>::=+|-
term()	<项>::=<因子>{<乘法运算符><因子>} <乘法运算符>::=*|/
factor()	<因子>::=<标识符> | <无符号整数> | '('<表达式>')'

其中每个语法成分的分析按照语法图（参考书本P308）进行，并同时进行符号表管理及Pcode代码生成。

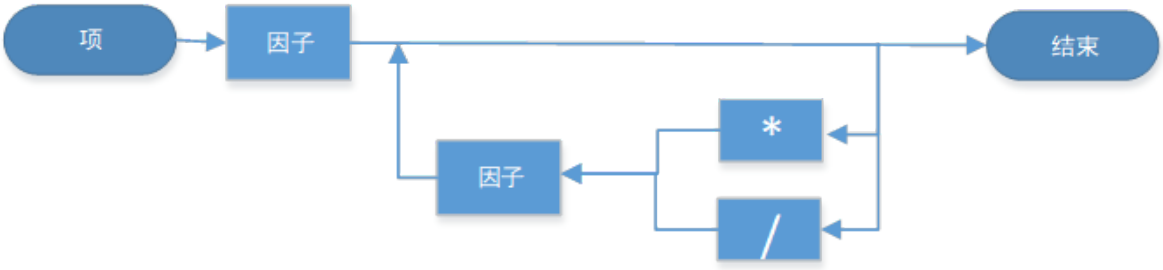
各个语法的描述图如下：

程序语法描述图：

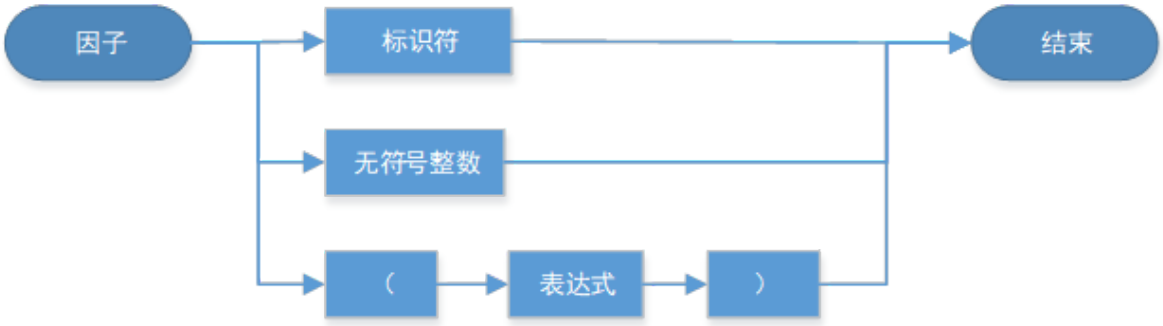
分程序语法描述图：



项语法描述图：



因子语法描述图



4.2.4 Pcode生成

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。

F	L	A
操作码	层次差(标识符引用层减去定义层)	不同的指令含义不同

P-code 指令的含义：

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

Pcode结构:

```
public class PerPcode {

    private final Operator f;
    private final int l;
    private int a;
}
```

地址回填

对于可能出现的跳转语句，需要采用地址回填。

if-then语句的目标代码生成模式：

```
if <condition> then <statement>
        <condition>
        JPC addr1
        <statement>
addr1:
```

If-then-else语句的目标代码生成模式：

```
if <condition> then <statement>[else]
        <condition>
        JPC addr1
        <statement>
        JMP addr2
addr1:  [else]
        <statement>
addr2
```

while-do语句的目标代码生成模式：

```
while <condition> do <statement>
addr2:  <condition>
        JPC addr3
        <statement>
        JPC addr2
addr3:
```

repeat-until语句的目标代码生成模式：

```
repeat <statement> until <condition>
addr4:  <statement>
        <condition>
        JPC addr4
```

4.2.5 出错管理

本项目将错误分为24类，其中包含了语法错误和语义错误。以下是出错信息表：

出错编号	出错原因
-1	常量定义不是const开头，变量定义不是var开头
0	缺少分号
1	标识符不合法
2	不合法的比较符
3	常量赋值没用 =
4	缺少 (
5	缺少)
6	缺少 begin
7	缺少 end’.
8	缺少 then.
9	缺少 do
10	call, write, read语句中，不存在标识符
11	该标识符不是proc类型
12	read, write语句中，该标识符不是var类型
13	赋值语句中，该标识符不是var类型
14	赋值语句中，该标识符不存在
15	该标识符已存在
16	调用函数参数错误
17	缺少 .
18	多余代码
19	缺少 until
20	赋值符应为 :=
21	until前多了 ；
22	缺少 ,

五、测试代码说明

我编写了5个测试用例。ChickenAndRabbit.txt, GCD&LCM.txt, Prime.txt分别是没有问题的鸡兔同笼问题，求最大公约数和最小公倍数以及找素数功能。两个Error的名字很清楚，分别是含有语义错误和语法错误。经测试，以上代码在本项目中均能正确运行。