

PL/0编译器

原理与实现

PL/0编译器——原理与实现

目录

1. PL/0语言	5
1.1. 概述	5
1.2. PL/0语言的扩充BNF表示	5
2. PL/0编译程序的结构	6
2.1. 代码结构	6
3. PL/0词法分析	10
3.1. PL/0编译系统中的终结符号	10
3.2. PL/0	10
3.2.1. 标识符	10
3.2.2. 无符号整数	10
3.2.3. 分界符和运算符	11
3.2.4. 关系运算符 (:=,<>,<=,>=, >,<)	11
3.3. PL/0词法分析器总程序框图（摘自编译第1次实验报告）	11
3.4. 实现	12
3.4.1. 概述	12
3.4.2. Tokenizer的主要功能	12
4. PL/0的语法分析	13
4.1. 递归子程序法	13
4.2. PL/0语法分析方法	13
4.2.1. PL/0头符号与后继符号	13
4.2.2. PL/0语法关系调用图	14
4.2.3. PL/0非终结符的语法图	14
4.3. PL/0语法分析实例	16
5. PL/0符号表	18

5.1. 符号表	18
5.1.1. 符号表的作用以及内容	18
5.1.2. 符号表的基本操作	18
5.2. PL/0符号表的组织	18
5.2.1. PL/0的符号表表项结构	18
5.2.2. PL/0中对符号表操作的实现	19
6. PL/0错误处理	20
6.1. 错误处理概述	20
6.2. PL/0编译程序对语法错误的处理	20
6.3. 错误列表	21
7. 目标代码的生成和解释执行	22
7.1. 目标代码的生成	22
7.1.1. PL/0的目标代码	22
7.1.2. 目标代码的生成	22
7.2. 目标代码的解释与执行	23
7.2.1. PL/0计算机	23
7.2.2. 特殊指令	23
7.2.3. 解释执行的流程图	24
7.2.4. 实现	25
8. 总结语	26
9. 附录 样例测试以及运行结果	27
9.1. 求两个数的最大公约数和最小公倍数	27
9.1.1. 源码	27
9.1.2. 运行结果	27
9.2. 鸡兔同笼	28
9.2.1. 源码	28
9.2.2. 运行结果	28
9.3. repeat测试	29
9.3.1. 源代码	29

9.3.2. 运行结果	29
9.4. else测试	30
9.4.1. 源代码	30
9.4.2. 运行结果	30
9.5. 错误样例	31
9.5.1. 源码	31
9.5.2. 运行结果	31

1. PL/0语言

1.1. 概述

PL/0语言是一种非常简单的高级程序设计语言，可以看做是Pascal语言的一个子集，通常用作教学使用。它只有整数一种类型，但却具有相当完全的可嵌套的分程序结构。PL/0可进行常量定义、变量说明以及过程调用，并具有通常程序设计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、复合语句以及读和写语句等。PL/0过程没有参数，但是可以递归调用。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> ::= =|<|<=|>|>=

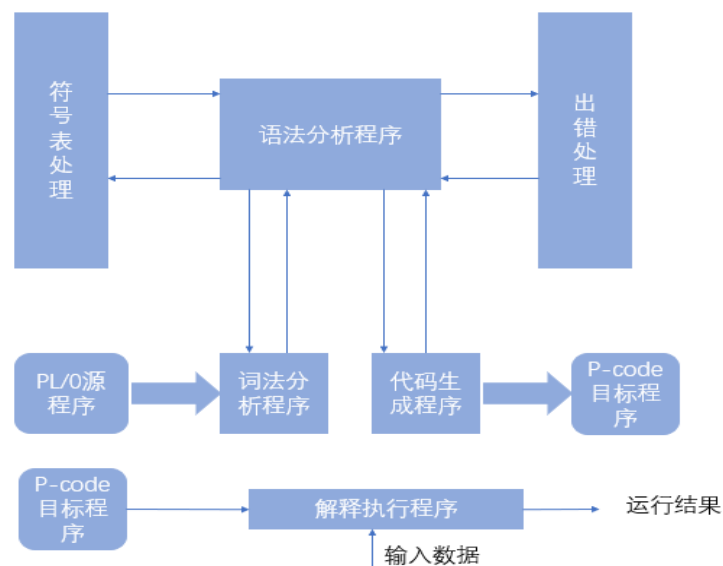
<条件语句> ::= if<条件>then<语句>[else<语句>]

<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

2. PL/0编译程序的结构

PL/0编译系统是一个编译-解释执行程序，整个编译过程分成两大阶段。第一阶段先把PL/0源程序编译成假想计算机的目标代码（P-code）程序，第二阶段在对生成的目标代码程序进行解释执行，得到运行结果。PL/0编译程序采用一趟扫描，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。如果遇到语法、语义上的错误，则调用错误处理程序，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。程序框图如所示。



图表 31 编译系统结构

2.1. 代码结构

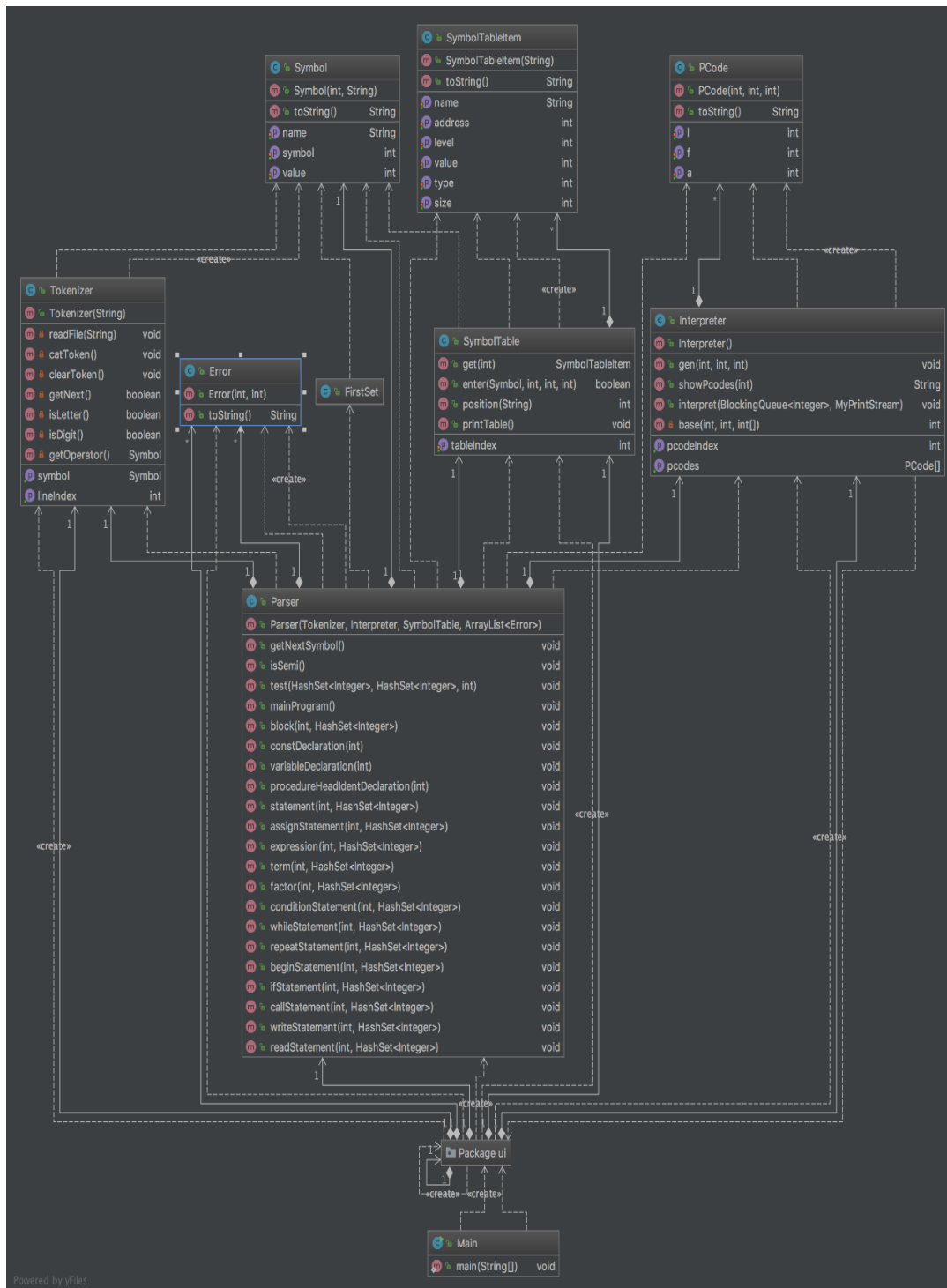
该PL/0编译系统是用Java语言编写，整个程序由10个类组成，每个类具有各自的功能。在这些类中，最重要的是SymbolTable, Tokenizer, Parser以及Interpreter。

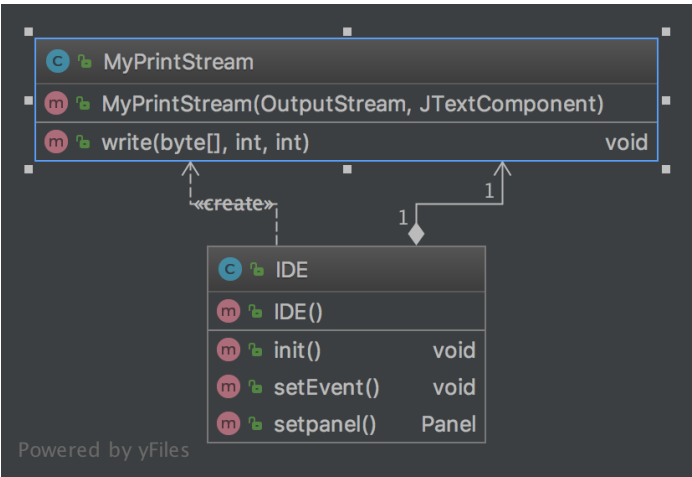
表格 31 类名及其作用

类名	作用
Main	入口方法main所在的类
Symbol	定义PL/0语言中的终结符号，如关键字，运算符等
SymbolTable	符号表有关的操作
SymbolTableItem	符号表中一个条目的结构体定义
PCode	P-code指令的结构体定义
Tokenizer	词法分析器
Parser	语法分析器
Interpreter	代码生成与解释执行
Error	与错误处理有关
FirstSet	常用的头符号集合

表格 32 重要方法

所在类	方法名	作用
SymbolTable	enter	将标识符登录到符号表中
	position	根据标识符名查找它在符号表中的位置
Tokenizer	getSymbol	从源文件中获取一个符号
Interpreter	gen	生成一条P-code指令并放入存放P-code的数组中
	interpret	解释执行目标代码程序
	showPcodes	展示虚拟机代码
Parser	mainProgram	解析非终结符<程序>
	block	解析<分程序>
	statement	解析<语句>
	assignStatement	解析<赋值语句>
	expression	解析<表达式>
	term	解析<项>
	factor	解析<因子>
	conditionStatement	解析<条件>
	whileStatement	<当型循环语句>
	repeatStatement	<重复语句>
	beginStatement	<复合语句>
	ifStatement	<条件语句>
	callStatement	<过程调用语句>
	readStatement	<读语句>
	writeStatement	<写语句>





图表 32 程序代码框架图

3. PL/0词法分析

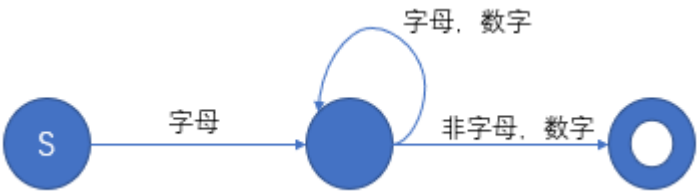
3.1. PL/0编译系统中的终结符号

表格 41 终结符号

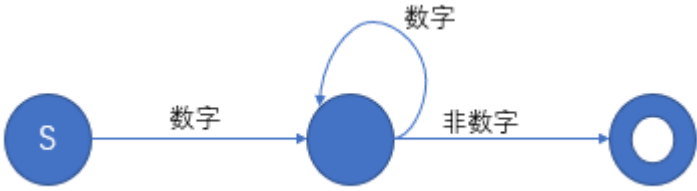
关键字或保留字	begin,end,const,var,procedure,odd,if,then,else,while,do,call,repeat,until,read,write
运算符	+ - * / = < > <= >= :=
分界符	() , . ;

3.2. PL/0 各类符号状态图

3.2.1. 标识符



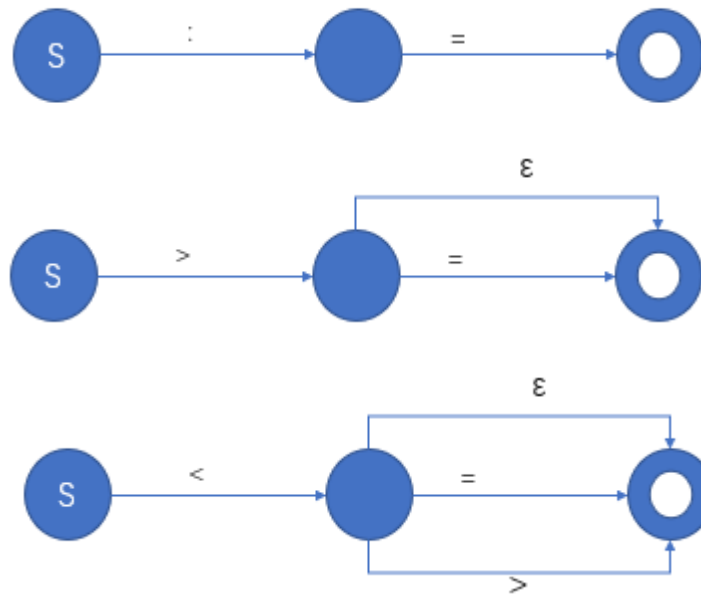
3.2.2. 无符号整数



3.2.3. 分界符和运算符



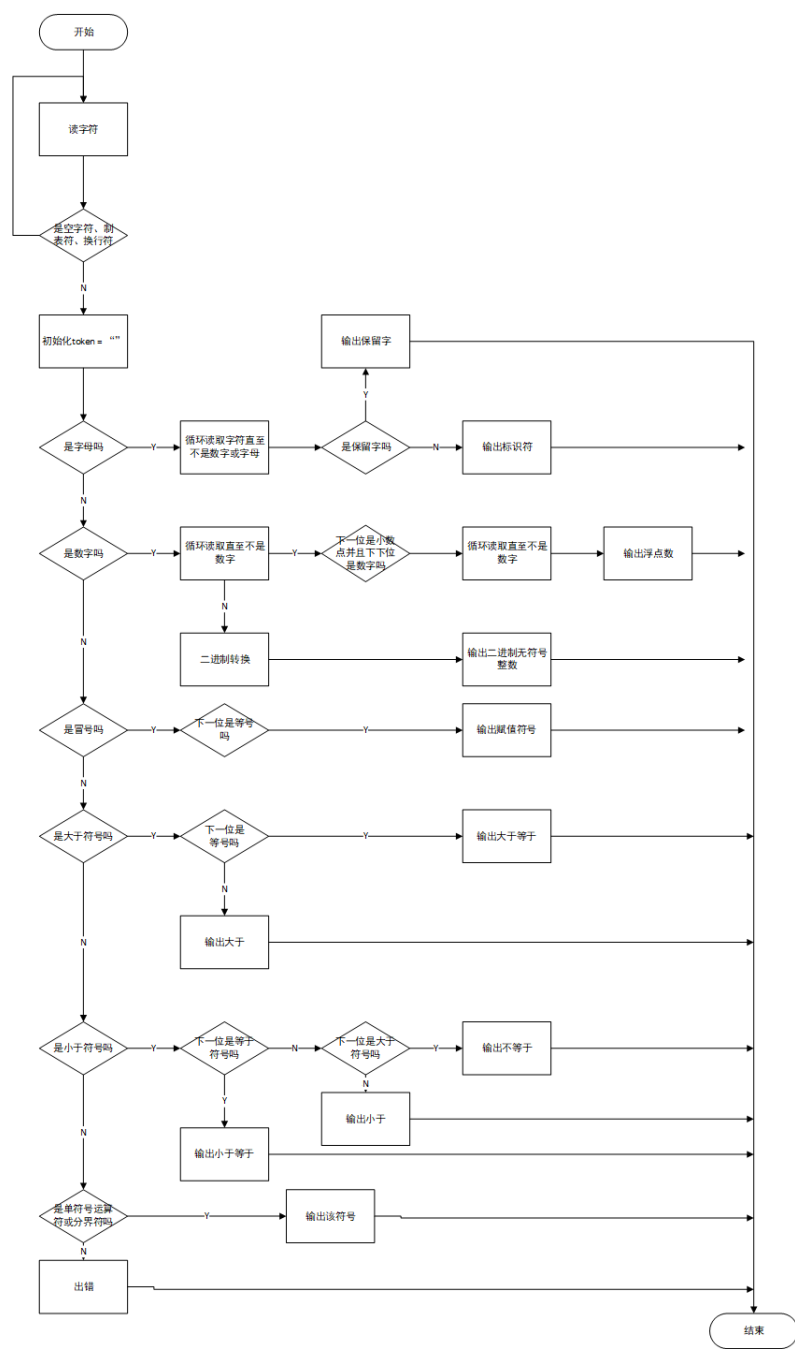
3.2.4. 关系运算符 (:=, <, <=, >=, >, <)



注: ϵ 表示空, 即符号为 < 或者 > 时

3.3. PL/0词法分析器总程序框图 (摘自编译第1次实验报告)

注:跟本次略有一定区别, 比如在本次实验中没有识别浮点数等, 但大体一致



图表 41 词法分析流程图

3.4. 实现

3.4.1. 概述

在本PL/0编译系统中，类Tokenizer负责对源程序进行词法分析，而getSymbol是该类中负责每次获取一个符号的方法，也是最重要的一个方法。构造词法分析对象时需要传入源程序代码文件的路径。

3.4.2. Tokenizer的主要功能

1. 跳过源程序中的空白，换行以及制表符。其中换行时还需统计当前所在的行号，以便代码出错时能够定位。

- 2. 从源程序正文字符序列中识别出单词符号，并把该单词符号的类别以相应枚举值的形式送入语法分析程序中。
- 3. 用变量token存放识别出的标识符，查找保留字表，确定token的种类
- 4. 如果token是无符号整数，则将该字符串转换为整数存放在Symbol的value字段中。
- 5. 由于文本文件不是很大，为提高效率，将源文件一次性读入到内存中，保存在Tokenizer的sourceFile字段中。

4. PL/0的语法分析

4.1. 递归子程序法

递归子程序法是一种自顶向下的语法分析方法，它要求文法需要具备以下几种特征：

- 1. 该文法必须是非左递归的。
- 2. 文法中的任一非终结符，如果在其规则右部有多个选择时，则各选择所生产的终结符号串的头符号集合要两两不想交
- 3. 如果文法具有形如 $A \rightarrow \epsilon$ 的规则，则A的头符号集合不得与由他产生的任何符号的后继符号集合相交。

4.2. PL/0语法分析方法

PL/0采用了递归子程序方法进行语法分析，即为每个语法成分都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。

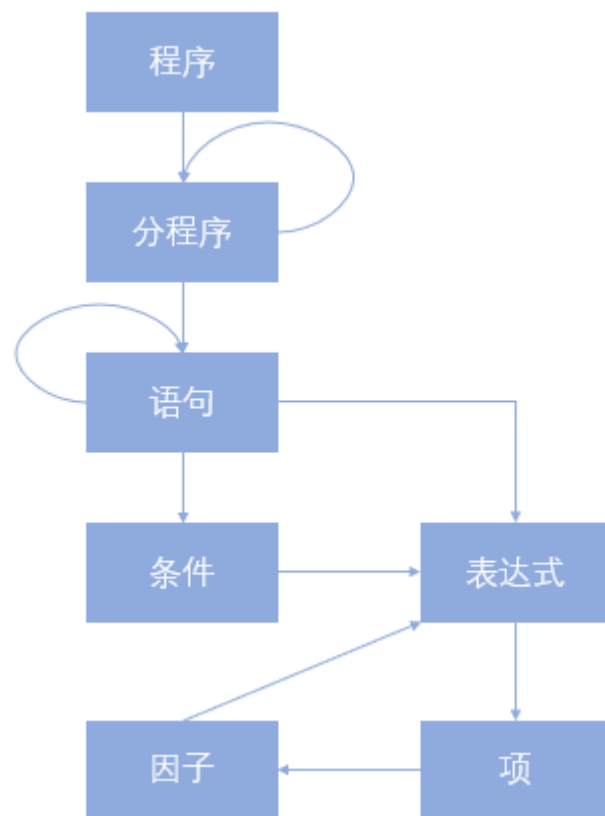
4.2.1. PL/0头符号与后继符号

表格 51 头符号与后继符号

非终结符号	FIRST集合	FOLLOW集合
分程序	const var procedure ident if call begin while read write repeat	. ;
语句	ident call begin if while read write repeat	. ; end
条件	odd + - (ident number	then do
表达式	= + - (ident number	. ; R then do)
项	ident number (. ; R + - end then do
因子	ident number (. ; R + - end then do * /

注：表中R表示关系运算符，ident和number分别为标识符和无符号整数

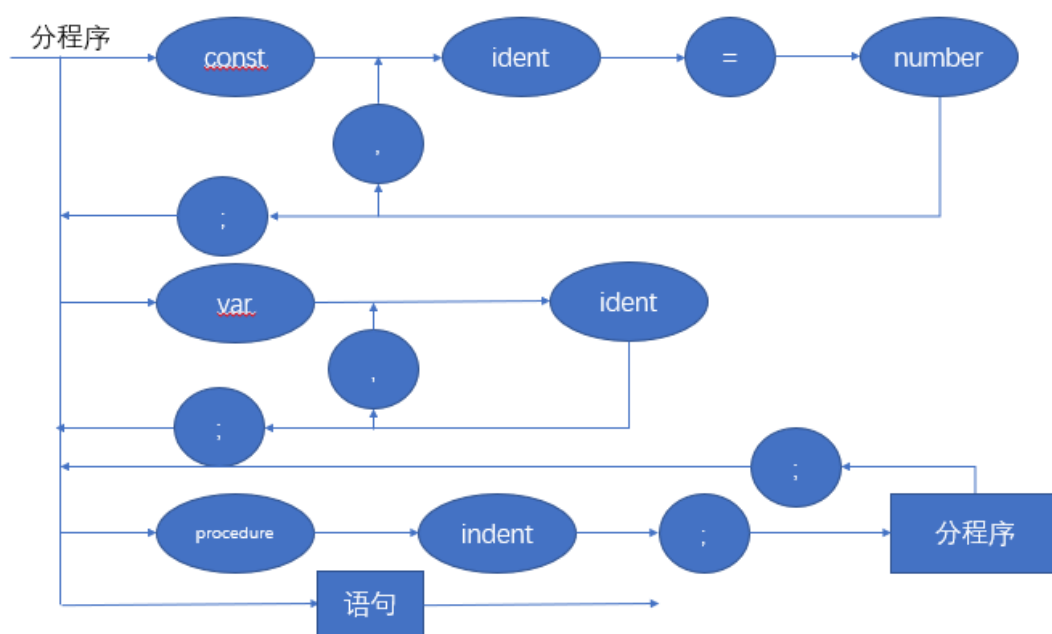
4.2.2. PL/0语法关系调用图



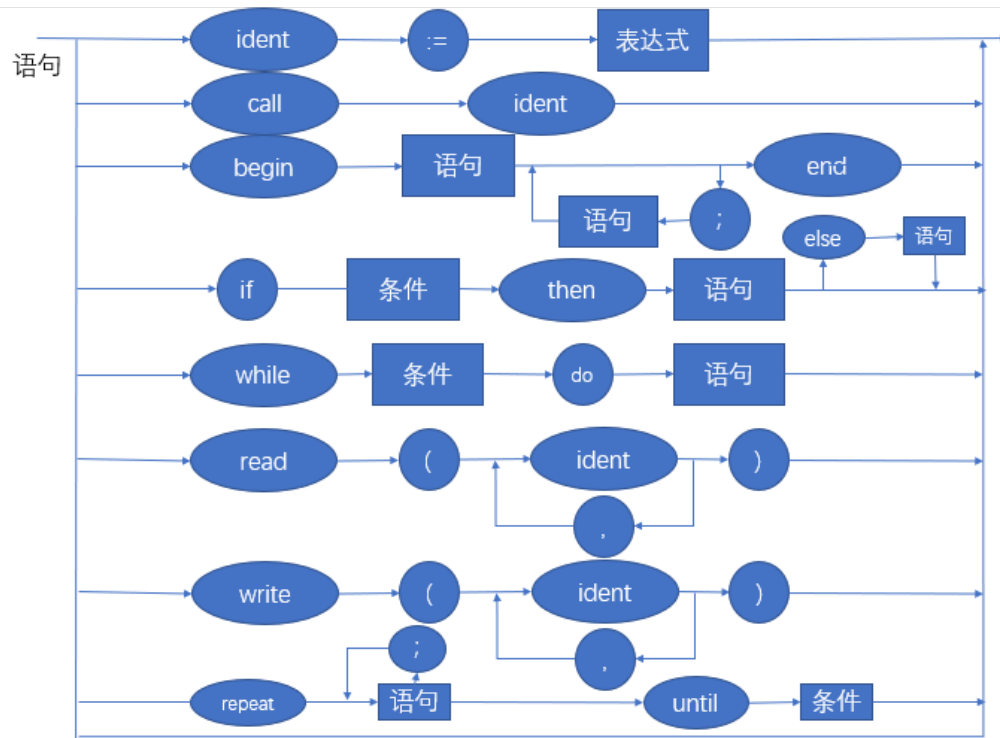
图表 51 关系调用图

4.2.3. PL/0非终结符的语法图

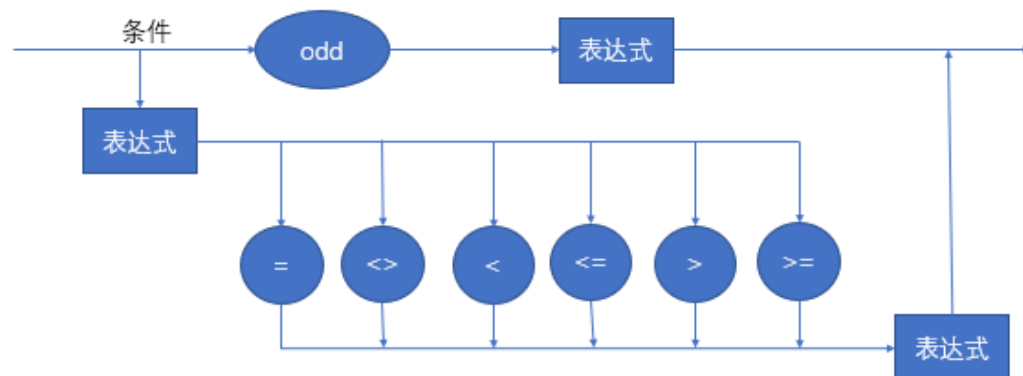
4.2.3.1. 分程序



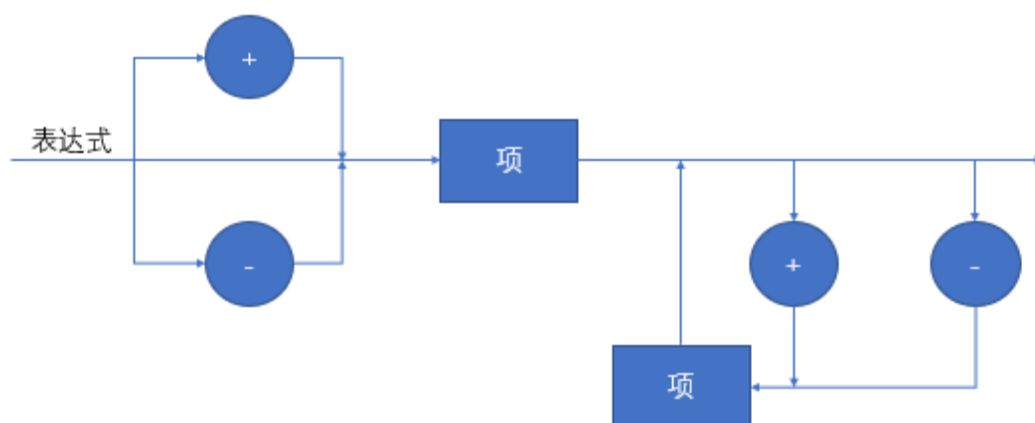
4.2.3.2. 语句



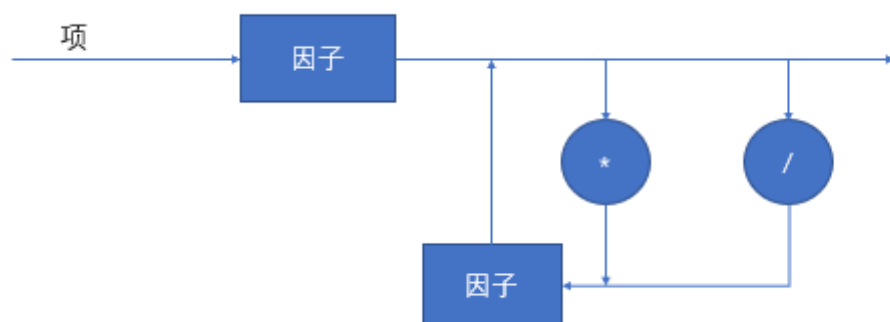
4.2.3.3. 条件



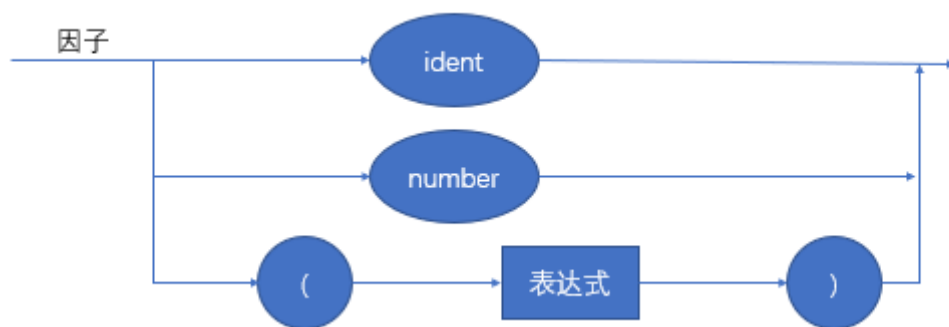
4.2.3.4. 表达式



4.2.3.5. 项



4.2.3.6. 因子



4.3. PL/0语法分析实例

以产生式 $\langle \text{分程序} \rangle ::= [\langle \text{常量说明部分} \rangle][\langle \text{变量说明部分} \rangle][\langle \text{过程说明部分} \rangle]\langle \text{语句} \rangle$ 为例，在实际的语法分析工作，在进入分程序以后，可分成两部分进行：首先对说明部分进行分析处理，再对语句部分进行分析处理。对于说明部分，如下图，分别对常量说明部分（蓝色块），变量说明部（绿色块）分以及过程说明部分（黄色部分）进行分析。

```

/**
 * 分析分程序
 * <分程序> ::= [<常量说明部分>][<变量说明部分>][<过程说明部分>]<语句>;
 * @param level 分程序在第几层
 * @param follow 分程序的FOLLOW集合
 */
public void block(int level, HashSet<Integer> follow){
    { ...
    }

    // 开始分析[说明部分]
    while(symbol!=null&&FirstSet.DECLARE_FIRST.contains(symbol.getSymbol())){ ...
    }
    { ...
    }

    // 分析<语句>, 将分号和end符号添加到FOLLOW集中
    next = new HashSet<>(follow);
    next.add(Symbol.SEMI_SY);next.add(Symbol.END_SY);
    statement(level,next);
    { ...
    }
}

```

图表 52 说明部分和语句

```

// 开始分析[说明部分]
while(symbol!=null&&FirstSet.DECLARE_FIRST.contains(symbol.getSymbol())){
    /**
     * 常量说明部分
     * <常量说明部分> ::= const<常量定义>{,<常量定义>;}
     */
    if(symbol.getSymbol() == Symbol.CONST_SY){ ...
    }

    /**
     * 变量说明部分
     * <变量说明部分> ::= var<标识符>{,<标识符>;}
     */
    if(symbol!=null&&symbol.getSymbol() == Symbol.VAR_SY){ ...
    }

    /**
     * 过程说明部分
     * <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>;
     */
    while(symbol!= null&&symbol.getSymbol() == Symbol.PROCEDURE_SY){ ...
    }
}

```

图表 53 说明部分详解

5. PL/0符号表

5.1. 符号表

5.1.1. 符号表的作用以及内容

符号表是在编译过程中，编译程序用来记录源程序中各种标识符的特性信息的表格。它的作用有以下几点：

1. 是编译过程中语义检查的依据。比如在运算过程中出现未说明的标识符，也即是说符号表中没有该标识符，则编译程序应该报错。
2. 生成目标代码的时候需要根据符号表来做地址分配；
3. 在符号表中，包括名字，种类，类型，层次，相对地址，存储类别等名字的属性信息，对于更复杂的程序设计语言，还涉及到动态数组的内情向量、函数和过程的形参等结构信息。

5.1.2. 符号表的基本操作

1. 创建符号表：在编译开始时或者进入一个分程序；
2. 插入表项：将一个符号记录插入符号表中；
3. 查询表项：在遇到执行语句时，需要在符号表中查找相关名字。

5.2. PL/0符号表的组织

5.2.1. PL/0的符号表表项结构

PL/0的符号表的一个表项结构主要由标识符名字、种类（常量，变量以及过程）、值（当类型为常量时）、所在层次（类型为变量和过程时），地址（变量和常量）以及数据空间大小（类型为过程时）。

```

3 public class SymbolTableItem {
4     /**
5      * name: 变量名
6      * value: 常量的值
7      * type: 类型 (变量, 常量, 过程)
8      * level: 所在层次 (变量, 过程)
9      * address: 地址 (变量, 过程)
10     * size: 大小 (过程)
11     */
12     private String name;
13     private int value;
14     private int type;
15     private int level;
16     private int address;
17     private int size;

```

图表 61 符号表中表项结构

符号表中的每项记录中的level字段应填入说明该变量名或者过程名的分程序层次。主程序的层次为0，各个嵌套的分程序随嵌套深度递增，PL/0规定嵌套数最大为3；address字段应填入每层局部变量所分配单元的相应地址，对于过程，则应填入该过程的入口地址。

5.2.2. PL/0中对符号表操作的实现

5.2.2.1. 登录符号表：enter方法。

```

/**
 * enter方法是将符号登入符号表中
 * @param symbol 要登入的符号
 * @param type 符号的类型，常量，变量，以及过程，在程序中分别用0, 1, 2表示
 * @param level 变量和过程所在的层次
 * @param address 变量的地址
 * @return 符号表溢出则返回false，否则返回true
 */
public boolean enter(Symbol symbol,int type,int level,int address){
    tableIndex++;
    SymbolTableItem item = get(tableIndex);
    item.setName(symbol.getName());
    item.setType(type);
    if(type == 0){//constant
        item.setValue(symbol.getValue());
    }else if(type == 1){//variable
        item.setLevel(level);
        item.setAddress(address);
    }else if(type == 2){//procedure
        item.setLevel(level);
    }
    if(tableIndex > TABLE_MAX_SIZE) return false;
    return true;
}

```

图表 62 符号表登录

5.2.2.2. 符号表查找，position方法

在对各种语句进行分析处理时，凡遇到标识符，都要调用position方法去查找符号表。若该名字已定义，则返回它在符号表中的位置，并据此取得它的有关属性信息。如果返回值为0，表明没有在符号表中找到该标识符，也即是说该标识符未定义，报告错误。

```

/**
 * position方法主要是根据标识符名称查找符号表，并返回其在符号表中的位置。
 * 从后往前找，这样正好符合嵌套分程序定义及其作用域规定的要求
 * @param name 标识符名称
 * @return 返回该标识符在符号表中的索引，如果没找到，返回-1.
 */
public int position(String name){
    for(int i = tableIndex;i>0;i--){
        if(tables[i].getName().equals(name)){
            return i;
        }
    }
    return 0;
}

```

图表 63 position

6. PL/0错误处理

6.1. 错误处理概述

错误处理的原则是尽可能准确指出出错位置，错误性质，并尽可能进行校正。

一个具有良好查错功能的编译程序一般应做到如下几点：

- 1) 任何源程序的输入序列都不会导致编译工作的崩溃。
- 2) 应尽可能多的发现源程序中出现的语法和语义错误，并尽可能准确地指出出错位置和错误性质。
- 3) 应尽量遏制冗余的出错信息。

6.2. PL/0编译程序对语法错误的处理

- 1) 对于简单的错误，例如漏了分号之类的，指出错误位置，继续进行分析。
- 2) 在每个语法分析子程序的出口处，检测下一个取来的符号是否为该语法成分的合法的后继符号。如不是，报告错误信息，并跳读一段源程序，直到取来的符号属于该语法成分合法的后继符号集合为止。然后为了防止跳读的源程序太多而造成损失太大，程序中又设置了一个停止符号集合，即，只要取来的符号属于合法的后继符号或者停止符号，都可以使程序停止跳读。停止符号里面可以放置若干明显的能够使程序重新开始分析的保留字或者其他终结符号。
- 3) PL/0用test方法来实现上述这种测试和跳读工作。具体实现如下

```

/**
 * 当进入某个语法单位时，调用test方法，检查当前符号是否属于该语法单位的开始符号集合
 * 若不属于，则滤去开始符号和后跟符号集合外的所有符号。
 * 在语法分析结束时，调用test方法检查当前符号是否属于调用该语法单位应该有的
 * 后跟符号集合，若不属于，则滤去后跟符号和开始符号外的所有符号
 * @param first first符号或follow
 * @param second first符号或follow
 * @param error 错误代码
 */
public void test(HashSet<Integer> first, HashSet<Integer> second , int error){
    if(symbol!= null&&!first.contains(symbol.getSymbol())){
        errors.add(new Error(error,tokenizer.getLineIndex()));
        first.addAll(second); 将停用集合补充到后继符号集合中
        while(symbol!=null && !first.contains(symbol.getSymbol())){
            getNextSymbol();
        }
    }
}
}

```

图表 71 test

6.3. 错误列表

表格 71 错误列表

序号	错误信息	序号	错误信息
1	应该是=而不是:=	14	call后面应为标识符
2	=后面应为数	15	不可调用常量或者变量
3	标识符后应为=	16	应为then
4	const var procedure后应为标识符	17	应为分号或end
5	漏掉逗号或者分号	18	应为do
6	过程说明后的符号不正确	19	语句后的符号不正确
7	应为语句	20	应为关系运算符
8	程序体内语句部分后的符号不正确	21	表达式内不可有过程标识符
9	应为句号	22	漏右括号
10	语句之间漏分号	23	因子后不可以为此符号
11	标识符未说明	24	表达式不能以此符号开始
12	不可向常量或过程赋值	30	这个数太大
13	应为赋值运算符:=	40	应为左括号

由于以上列表中错误信息不全，故添加了额外的错误信息，见下表

表格 72 补充错误

-1	词法错误	32	嵌套数过多
31	read语句括号中不是变量	33	repeat后面漏掉了until

7. 目标代码的生成和解释执行

7.1. 目标代码的生成

7.1.1. PL/0的目标代码

PL/0编译程序所产生的目标代码是一个假想的栈式计算机的目标代码，称之为P-code指令代码，它不依赖于任何实际计算机，其指令集很简单，指令格式如下：



图表 81 P-code指令结构

其中f表示操作码；l表示变量或过程被引用的分程序与说明该变量或过程的分程序之间的层次差；a对于不同的指令有着不同的含义。常用的指令如下表

表格 81 指令列表

指令	功能
LIT	将常量值取到运行栈顶，a域为常数值
LOD	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
STO	将栈顶的内容送入某变量单元中。a,l域的含义同LOD指令
CAL	调用过程的指令。a域为被调用过程的目标程序入口地址，l为层差
INT	为被调用的过程在运行栈中开辟数据区。a域为开辟的单元个数
JMP	无条件转移指令，a为转向地址
JPC	条件转移指令，当栈顶的布尔值为假时，转向a域的地址，否则继续执行
OPR	关系运算和算术运算指令。具体的操作根据a域的值确定

7.1.2. 目标代码的生成

PL/0语言的代码生成是由Interpreter里的方法gen完成的。生成的代码顺序放在数组pcodes中，其中数组中每一个元素都是一条目标指令。

```

//生成pcode, 并且存入pcodes中
public void gen(int f,int l,int a){
    if(pcodeIndex >= MAX_PCINDEX){
        return;
    }
    PCode pCode = new PCode(f,l,a);
    pcodes[pcodeIndex] = pCode;
    //System.out.println(pcodeIndex+": "+pCode);
    pcodeIndex++;
}

```

图表 82 代码生成

7.2. 目标代码的解释与执行

P-code指令可通过解释执行程序interpret使其在PL/0计算机上运行，因此P-code解释执行程序也可以看做是PL/0计算机的算法描述。

7.2.1. PL/0计算机

PL/0计算机可看做由两个存储器，一个指令寄存器和三个地址寄存器组成。存储器code用来存放生成的P-code指令。数据存储器S实际上是一个堆栈，用来动态分配程序的数据空间。PL/0计算机没有专门的运算寄存器，因此所有计算都在数据堆栈栈顶的两个单元之间进行，并用运算结果取代原来的两个运算对象而保留在栈顶里。

指令寄存器I存放当前要执行的P-code指令。程序地址寄存器P存放下一条要执行的指令地址，当程序顺序执行时，每执行完一条指令后，p的值需要加一。基地址寄存器B用来存放当前运行的分程序数据区在数据栈S中的起始地址。

当程序运行进入某一个过程之后，即在数据栈S中为其分配一个数据区，运算操作就在该数据区上面的栈顶单元之间进行。若程序在当前过程中要调用其他过程，便在当前数据区上面再叠加分配一个新过程的数据区。为了实现上述功能，并能够在过程结束时恢复到原来调用处的运行环境，每个过程被调用时，在栈顶分配三个存储单元，这三个单元存放的内容如下：

- 1) SL: 静态链。指向定义该过程的直接外过程(或主程序)运行时最新数据段的基地址。
- 2) DL: 动态链。指向调用该过程前正在运行过程的数据段基地址。
- 3) RA: 返回地址。记录调用该过程时目标程序的断点，即调用过程指令的下一条指令的地址。

7.2.2. 特殊指令

- 1) INT 0 ,A

在过程目标程序的入口处，开辟A个数据单元的数据段。A为局部变量的个数+3 (SL, DL, RA)

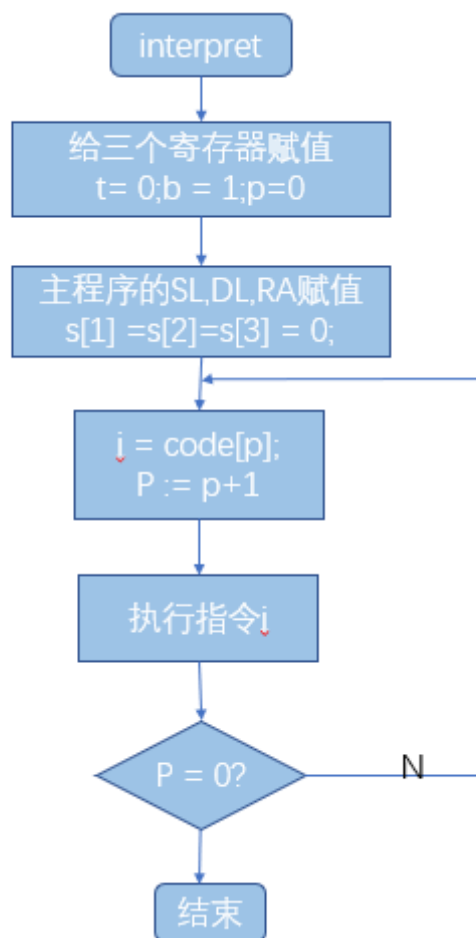
- 2) OPR 0 0

在过程目标程序的出口处，释放数据段，恢复调用该过程前正在运行的过程的数据段基地址寄存器B和栈顶寄存器T的值，并将返回地址送入到指令地址寄存器P中。

3) CALL A

调用过程，填写静态链，动态链、返回地址。给出被调用过程的基地址值，送入基地址存储器B中，目标程序的入口地址A的值送入指令地址寄存器P中，使指令从A开始执行。

7.2.3. 解释执行的流程图



图表 83 解释执行流程图

7.2.4. 实现

```
/**
 * 求上一层的基址
 * @param l 层数
 * @param b 基址
 * @param stack 运行栈
 * @return 找到的基址
 */
private int base(int l,int b,int[] stack){
    int b1 = b;
    while(l>0){
        b1 = stack[b1];
        l = l-1;
    }
    return b1;
}
```

图表 84 base

```
switch (pCode.getF()){
    case PCode.LIT: //将常数值取到栈顶
        t = t+1;
        stack[t] = pCode.getA();
        break;
    case PCode.OPR: ...
    case PCode.LOD: //将变量放到栈顶
        t = t+1;
        stack[t] = stack[base(pCode.getL(),b,stack)+pCode.getA()];
        break;
    case PCode.STO: //将栈顶内容送入变量
        stack[base(pCode.getL(),b,stack)+pCode.getA()] = stack[t];
        t = t-1;
        break;
    case PCode.CAL: //调用过程
        stack[t+1] = base(pCode.getL(),b,stack); //填写静态链SL
        stack[t+2] = b; //填写动态链
        stack[t+3] = p; //填写返回地址
        b = t+1; //被调用过程的基地址
        p = pCode.getA(); //设置入口地址
        break;
    case PCode.INT: //开辟空间
        t = t+pCode.getA(); break;
    case PCode.JMP: //无条件跳转
        p = pCode.getA(); break;
    case PCode.JPC: //当栈顶的值为假时，转向a域的地址，否则顺序执行
        if(stack[t] == 0) p = pCode.getA();
        t = t-1;
        break;
}
```

图表 85 interpret （部分）

对于命令**OPR**，**0**，**a**根据其a域的值定义不同的运算操作，见下表

表格 82 OPR指令详解

a域	0	1	2	3	4	5	6	8
操作	过程返回	将数字取负	加法操作	减法	乘法	除法	奇偶判断	相等判断
a域	9	10	11	12	13	14	15	16
操作	不等	小于	大于等于	大于	小于等于	输出栈顶元素	输出换行	输入元素

8. 总结语

PL/0语言是一种类Pascal语言，具备一般高级程序设计语言的典型特点，如常量定义，过程声明等。PL/0语言编译程序结构清晰，可读性强，充分体现了一个高级编译程序实现的基本组织、技术以及步骤，是一个适合小型编译程序的教学模型。

在本次实验中，利用Java语言实现了PL/0语言的编译系统。采用一趟扫描源程序方法，实现了PL/0的词法分析、语法分析、符号表处理、错误处理以及目标代码的生成与解释执行。

9. 附录 样例测试以及运行结果

9.1. 求两个数的最大公约数和最小公倍数

9.1.1. 源码

```
gcd.txt
1  const a=45,b=27;
2  var x,y,g,m;
3  procedure swap;
4  |   var temp;
5  |   begin
6  |       temp:=x;
7  |       x:=y;
8  |       y := temp;
9  |   end;
10
11 procedure mod;
12 |   x:=x-x/y*y;
13 |   begin
14 |       x:=a;
15 |       y:=b;
16 |       call mod;
17 |       while x<>0 do
18 |           begin
19 |               call swap;
20 |               call mod;
21 |           end;
22 |       g:=y;
23 |       m:=a*b/g;
24 |       write(g,m)
25 end.
```

9.1.2. 运行结果

源代码文件

错误信息

Show P-code

Run

/Users/yushijie/case/gcd.txt
编译成功, 没有错误

INT	0,4
LOD	1,3
STO	0,3
LOD	1,4
STO	1,3
LOD	0,3
STO	1,4
OPR	0,0
INT	0,3
LOD	1,3
LOD	1,3
LOD	1,4
OPR	0,5
LOD	1,4
OPR	0,4
OPR	0,3
STO	1,3
OPR	0,0
INT	0,7
LIT	0,45
STO	0,3
LIT	0,27
STO	0,4
CAL	0,11
LOD	0,3
LIT	0,0
OPR	0,9
JPC	0,34
CAL	0,2
CAL	0,11
END	0,27

-----PL0 start running-----
9 135

清除

9.2. 鸡兔同笼

9.2.1. 源码

```

1  const z = 0;
2  var head,foot,cock,rabbit,n;
3  begin
4      n:=z;
5      read(head,foot);
6      cock := 1;
7      while cock <= head do
8          begin
9              rabbit := head-cock;
10             if cock*2 + rabbit*4 = foot then
11                 begin
12                     write(cock,rabbit);
13                     n := n+1
14                 end;
15             cock := cock+1
16         end;
17         if n = 0 then write(cock,rabbit)
18     end.
```

9.2.2. 运行结果

源代码文件

错误信息

Show P-code

Run

/Users/yushijie/case/rabbit.txt

编译成功, 没有错误

LIT 0,4

OPR 0,4

OPR 0,2

LOD 0,4

OPR 0,8

JPC 0,37

LOD 0,5

OPR 0,14

LOD 0,6

OPR 0,14

OPR 0,15

LOD 0,7

LIT 0,1

OPR 0,2

STO 0,7

LOD 0,5

LIT 0,1

OPR 0,2

STO 0,5

JMP 0,10

LOD 0,7

LIT 0,0

OPR 0,8

JPC 0,51

LIT 0,0

OPR 0,14

LIT 0,0

OPR 0,14

OPR 0,15

OPR 0,0

-----PL0 start running-----

input a number:

9

input a number:






28

4 5

清除

9.3. REAPEAT测试

9.3.1. 源代码



repeat.txt x

```
1  const a = 10;
2  var b,c;
3  procedure p;
4      begin
5          repeat
6              write(b,c,b*c);
7              if b <> 0 then write(c/b) else write(b);
8              b := b+1;
9              c := c-1
10             until b > c
11         end;
12
13 procedure p1;
14     begin
15         write(a,b,c)
16     end;
17 begin
18     read(b,c);
19     call p;
20     call p1;
21 end.
```

9.3.2. 运行结果

源代码文件		
错误信息	Show P-code	Run
'Users/yushijie/case/repeat.txt' 编译成功，没有错误	INT 0,3 LOD 1,3 OPR 0,14 LOD 1,4 OPR 0,14 LOD 1,3 LOD 1,4 OPR 0,4 OPR 0,14 OPR 0,15 LOD 1,3 LIT 0,0 OPR 0,9 JPC 0,22 LOD 1,4 LOD 1,3 OPR 0,5 OPR 0,14 OPR 0,15 JMP 0,25 LOD 1,3 OPR 0,14 OPR 0,15 LOD 1,3 LIT 0,1 OPR 0,2 STO 1,3 LOD 1,4 LIT 0,1 OPR 0,3 STO 1,4	----PL0 start running---- input a number: 10 input a number: 60 10 60 600 6 11 59 649 5 12 58 696 4 13 57 741 4 14 56 784 4 15 55 825 3 16 54 864 3 17 53 901 3 18 52 936 2 19 51 969 2 20 50 1000 2 21 49 1029 2 22 48 1056
清除		

9.4. ELSE测试

9.4.1. 源代码

```
proc.txt x
1  const a = 10;
2  var b,c;
3  procedure p;
4  begin
5      b:= b-1;
6  end;
7  begin
8      read(b);
9      while b<>0 do
10     begin
11         call p;
12         write(b);
13     end;
14     begin
15         b := 15;
16         write(b);
17         if b = 0 then write(b) else write(a)
18     end
19 end.
```

9.4.2. 运行结果

源代码文件

错误信息

Show P-code

Run

/Users/yushijie/case/proc.txt
编译成功, 没有错误

INT	0,3
LOD	1,3
LIT	0,1
OPR	0,3
STO	1,3
OPR	0,0
INT	0,5
OPR	0,16
STO	0,3
LOD	0,3
LIT	0,0
OPR	0,9
JPC	0,20
CAL	0,2
LOD	0,3
OPR	0,14
OPR	0,15
JMP	0,11
LIT	0,15
STO	0,3
LOD	0,3
OPR	0,14
OPR	0,15
LOD	0,3
LIT	0,0
OPR	0,8
JPC	0,33
LOD	0,3
OPR	0,14
OPR	0,15

-----PL0 start running-----
input a number:
20
19
18
17
16
15
14
13
12
11
10
9
8
7
6
5
4
3
2
1
0
15
10

清除

9.5. 错误样例

9.5.1. 源码

```
1  const z = 1,p;  
2  var head,foot,cock,rabbit,n;  
3  begin  
4      n:=z;  
5      read (z,foot);  
6      cock :=  
7      while cock := head do  
8          begin  
9              rabbit = head-cock;  
10             if cock*2 + rabbit*4 = foot then  
11                 begin  
12                     write cock,rabbit);  
13                     n := n1  
14                 end;  
15             end;  
16 Aaaa  
17             cock := cock+1  
18         end;  
19 end.
```

9.5.2. 运行结果

