《编译技术》课程设计文档

学号：76066001

姓名：张金源

2018 年　 11 月　 18 日

一. 需求说明

1. 文法说明

我获取的文法是 PL/0 文法，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。PL/0 的程序结构比较完全，赋值语句作为基本结构，构造概念有

- 顺序执行、条件执行和重复执行，分别由 begin/end,if then else 和 while do 语句表示。
- PL0 还具有子程序概念，包括过程说明和过程调用语句。
- 在数据类型方面，PL0 只包含唯一的整型，可以说明这种类型的常量和变量。
- 运算符有+，-，*，/，=，<>，<，>，<=，>=，(，)。
- 说明部分包括常量说明、变量说明和过程说明。

<程序>　　::=　<分程序>.
<分程序>　　::=　[<变量说明部分>][<常量说明部分>][<过程说明部分>]<复合语句>
<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>　::=　<非零数字>{<数字>}|0
<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>{;<过程首部><分程序>};
<过程首部>　　::=　procedure<标识符>;
<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>
<赋值语句>　　::=　<标识符> := <表达式>
<表达式>　::=　[+|-]<项>{<加法运算符><项>}//[+|-]只作用于第一个<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>
<当循环语句>　::=　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
<读语句>　　::=　read'('<标识符>{, <标识符>}')'
<写语句>　　::=　write'('<表达式>{, <表达式>}')'

<数字>　　::=　0|<非零数字>
<非零数字>　　::=　1|2|3...|8|9

2．文法解读

<程序>　　::=　<分程序>.

范例：

```
const a=4;

end.
```

分析：

程序的主题是分程序，“.”为结束符号，告诉编译器需要编译的程序到此为止；

<分程序>　　::=　[<变量说明部分>][<常量说明部分>][<过程说明部分>]<复合语句>
<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>
<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>{;<过程首部><分程序>};
<过程首部>　　::=　procedure<标识符>;
<复合语句>　　::=　begin<语句>{;<语句>}end

范例：

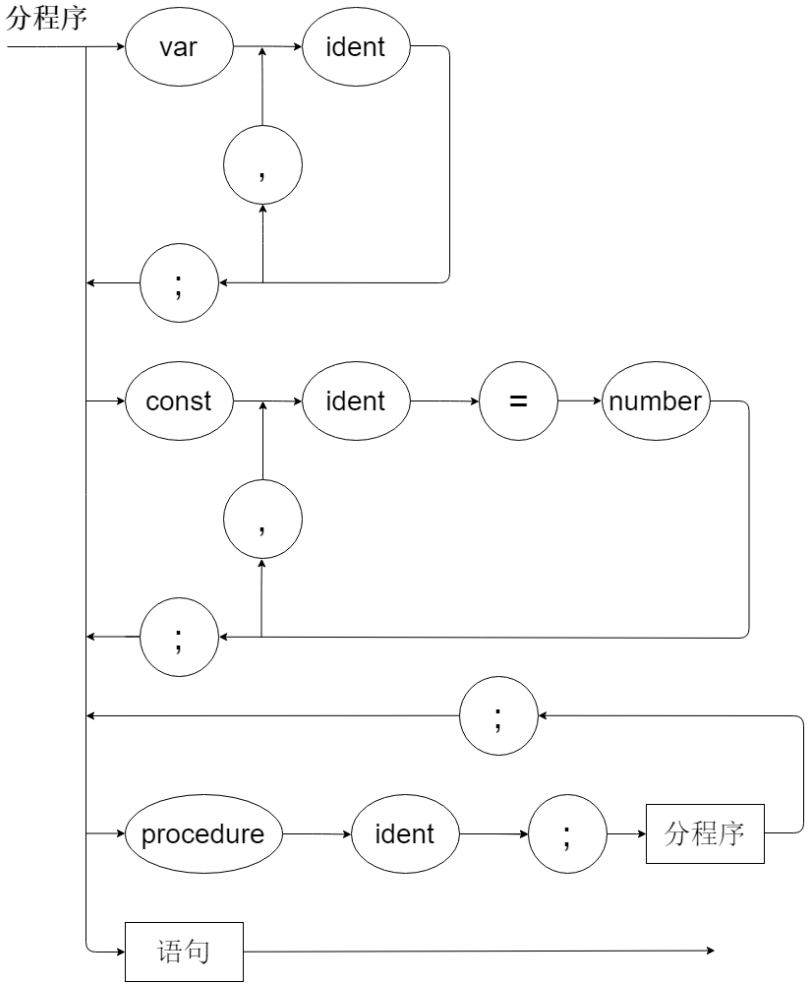
```
var x,y,z :integer; //变量说明部分
const a=6,b=5,c=3; //常量说明部分
procedure lab1; //过程声明部分
lab1 分程序（略）
begin
x:=a;
for y:=1 to 6
do
      x:= x+1;
end
```

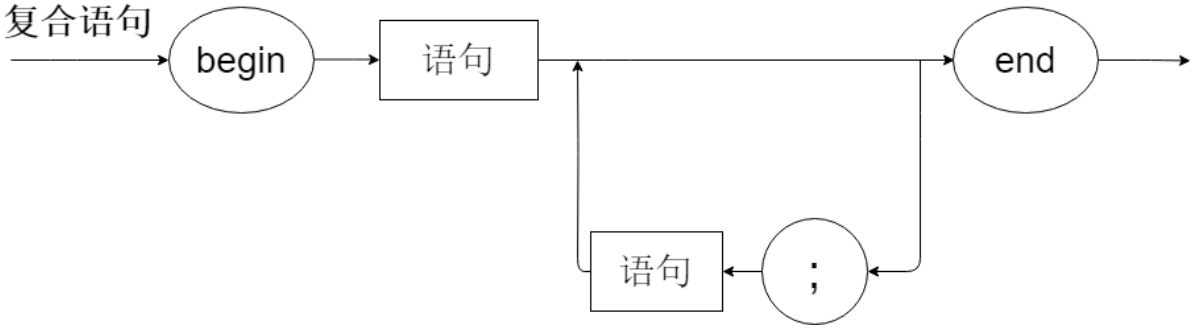
分析 1：

根据分程序的文法可知，分程序各个组成成分的声明顺序已经限定好了，不能随便更改他的声明顺序，如：”procedure lab1;const a=6,b=5,c=3;var x,y,z :integer;”，“var x; var y;”，这些声明顺序不合法上述的文法；

分析 2：

每一个分程序按顺序由变量说明部分，常量说明部分，过程说明部分，以及复合语句组成。其中变量，常量，过程说明，复合语句之间有严格的先后顺序，不能打乱。变量说明，常量说明，过程说明对于一个分程序来说均为可有可无的部分，只有最后的复合语句是必须存在的；





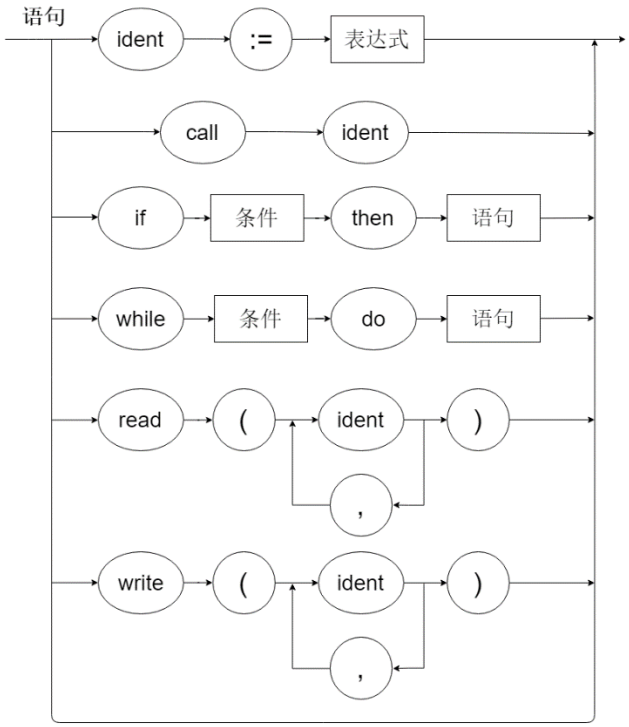
<语句>　　::=　<赋值语句>|<条件语句>|<当循环语句>|<过程调用语句>|<复合语句>|<读语句>|<写语句>|<空>
<赋值语句>　　::=　<标识符>:=<表达式>
<条件>　　::=　<表达式><关系运算符><表达式>|odd<表达式>
<关系运算符>　　::=　=|<>|<|<=|>|>=
<条件语句>　　::=　if<条件>then<语句>
<当循环语句>　　::=　while<条件>do<语句>
<过程调用语句>　::=　call<标识符>
<读语句>　　::=　read'('<标识符>{,<标识符>}')'
<写语句>　　::=　write'('<表达式>{,<表达式>}')'

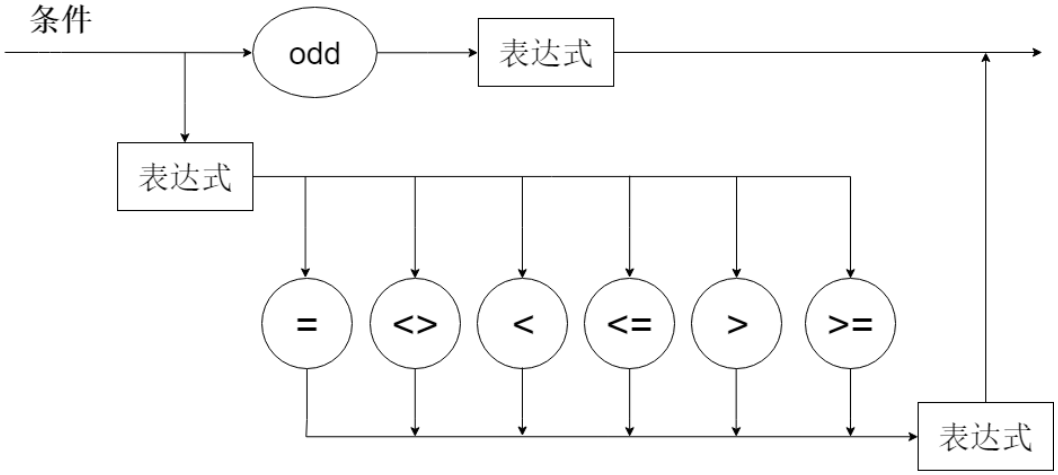
范例：

```
var x,y,m :integer;
x := 5; //赋值语句
y := 0;     //赋值语句
if x > 1 x := x  -  1; //条件语句
if odd x -1  y := y + 1; //条件语句用 odd
begin /*复合语
while x > 0       /*当循环语句
do write (x - 1); 写语句*/
end;  */
call lab1; //过程调用语句
read (m); //读语句
```

分析:

此部分主要说明了各种类型语句的结构组成，就是说明了能够接受的语句语法格式；某些语句会有多种形式，在进行分析时应考虑全面；在这部分我们文法有各种各样的语句语法格式；





<表达式>　　::=　　[+|-]<项>{<加法运算符><项>}　　//[+|-]只作用于第一个<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>　　::=　　*|/

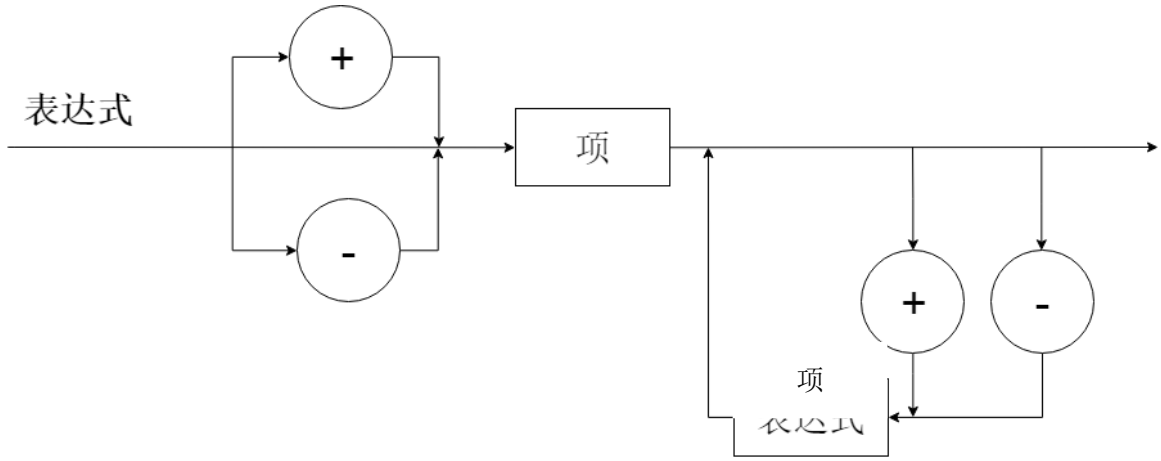
范例：

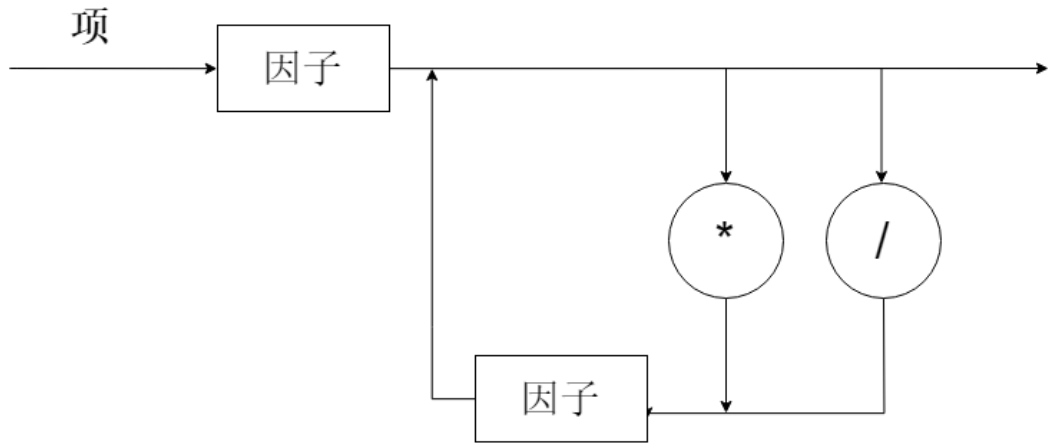
```
var x,y,z :integer;
```

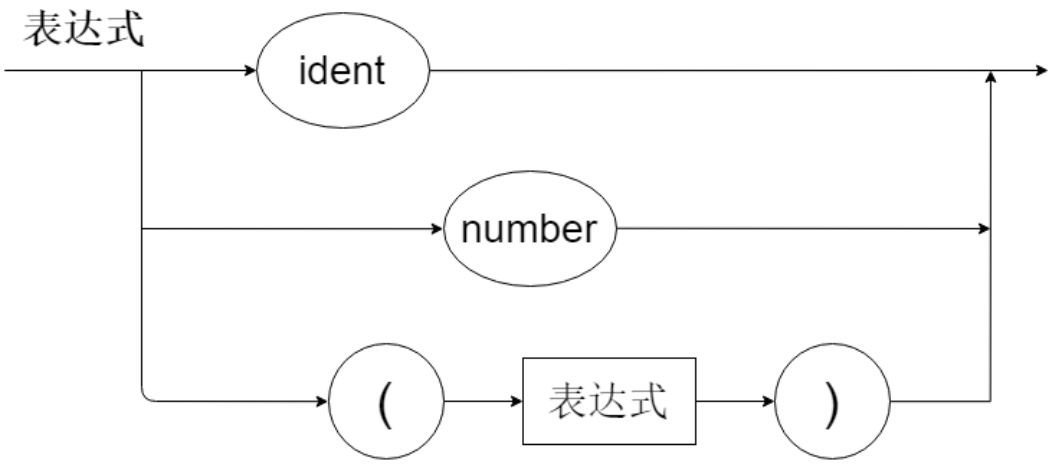
```
x := 2
y := 3
z := x + y;  /*   表达式的多种形式
z := x  -  y;
z := x * y;
z := x / y;
z := +2 + x;
z := -2 + x;
z := (x + 1)  -  y; */
```

分析：

此处表述了表达式由项组成的具体方式，包含多种情况，值得注意；第一个项前面的+、-是表示如果第一个项需要是负数或正数，在数字前加上符号；后面的+、-基本上为运算符的+、-，而在这文法也需要注意乘法运算和括号“（”“）”的优先级；







<无符号整数>　　::=　<非零数字>{<数字>}|0

范例：

```
var x,y :integer;
x := 1204;
y:= 0;        //1204, 0 为无符号整数
```

分析：

此处说明了无符号整数的组成方法，即数字的重复，但需要注意以 0 开头的数字他无法接受，如：011，因为 0 为开头是无符号整数文法的终结符；

<字母>　　::=　a|b|c|d...|x|y|z|A|B...|Z
<数字>　　::=　0|<非零数字>
<非零数字>　　::=　1|2|3...|8|9

分析：

此处说明了文法的所有终结符，而说明了字母，数字，非零数字的组成方式；

3．目标代码说明

P-code 是为目标代码。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。指令格式为 F，L，A

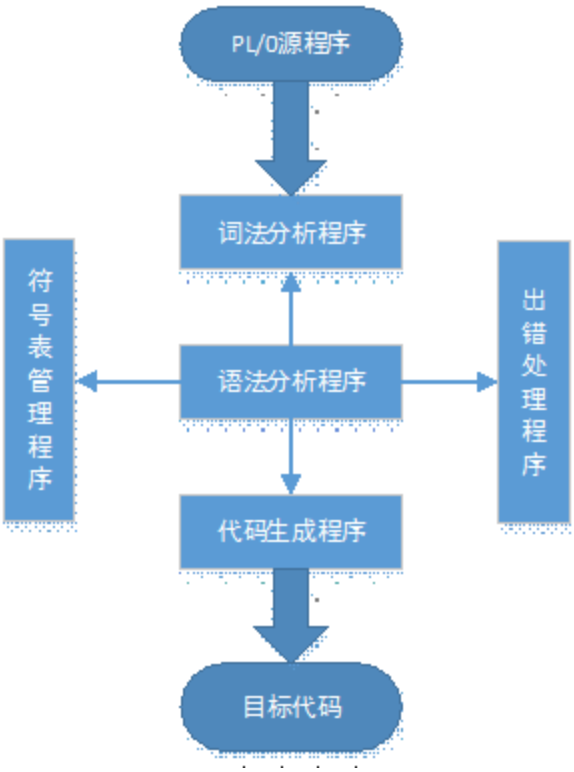
F: 操作吗， L: 层次差， A: 不同的指令含义。

P-code 指令含义

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(+[3] -[4] *[5] /[6])
lod 1,a	取变量放到数据栈栈顶(相对地址为 a,层次差为 1)
sto 1,a	将数据栈栈顶内容存入变量(相对地址为 a,层次差为 1)
cal 1,a	调用过程(入口指令地址为 a,层次差为 1)
inte 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
red 1,a	读数据并存入变量（相对地址 a,层次差为 1）
wrt 0,0	将栈顶内容输出

二.　　详细设计

1. 程序结构



首先程序分为 5 个阶段，第一个阶段是词法分析程序，就是把每个单词分析下来。然后语法分析，语法分析是用递归子程序法对每种语法成分进行分析。代码生成程序是能够输出目标代码（P-code）。出错处理程序是处理源代码的各个错误（报错）一般为数字来报并输出在哪出现错误。符号表管理是从词法分析来存储的，把每次读到的符号存到栈里边。这编译系统用了一遍扫描，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。目标代码为 p-code。

2. 类/方法/函数功能

本程序是用 C/C++实现，函数说明如下：

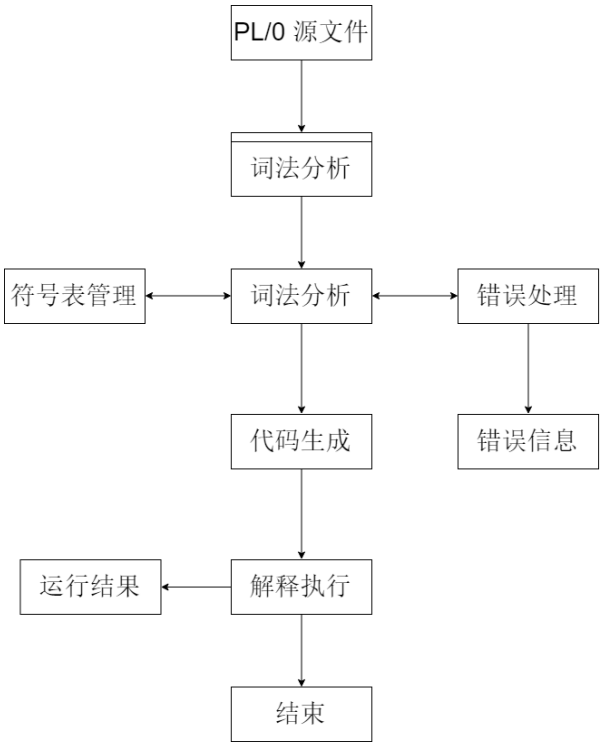
```
//出错处理
void error_msg(int n); /*出错处理*/

//词法分析
int getsym(); /*词法分析程序*/
int getch();   /*从文件读入*/
void init();  /*初始化*/
int test(bool*s1, bool*s2, int n); /*检测*/
```

```
//语法分析
int block(int lev, int tx, bool* fsys, int para);
int procdefine(bool ret, int para, int *pdx);
int condition(bool*fsys, int*ptx, int lev);
int expression(bool*fsys, int*ptx, int lev);
int statement(bool*fsys, int*ptx, int lev);
int vardeclaration(int* ptx, int lev, int* pdx);
int constdeclaration(int* ptx, int lev, int* pdx);
//运算处理
int factor(bool* fsys, int* ptx, int lev);
int term(bool*fsys, int*ptx, int lev);
//符号表操作
int position(char* idt, int tx);
void enter(enum object k, int* ptx, int lev, int* pdx, int length);
//目标代码处理（p-code）
int gen(enum fct x, int y, int z);
int base(int l, int* s, int b);
void interpret();/*执行目标代码*/
```

3．调用依赖关系

根据程序的逻辑结构,在实现上我将我的程序划分为 5 个模块:符号表、声明处理、语句处理、解释执行、错误处理。声明部分相对独立,其他部分相互嵌套调用。



3.1符号表

符号表是用来记录、查找源程序中的所有标识符的信息,通过符号表,为声明过程、语句处理、解释执行提供相应的信息。符号表的管理包括符号表的设计(详见二中 4 符号表管
理方案),登陆符号表,符号表定位,符号表重定位。

3.2声明处理

声明包括常量声明,变量声明、过程声明、函数声明,声明部分主要工作是根据语法分析,将合法的信息登入符号表,声明部分相对来说较为独立,其实质就是登陆符号表,在登陆符号表的时候,因为需要对标识符值类型进行判断,对于过程声明和函数声明还需要判断形参的个数和形参类型,所以需要返填符号表,需要动态登陆符号表。常量声明,变量声明,过程声明、函数声明各写了一个函数进行处理,分别为:
ConstDefine(),varDefine(),FuncProDeclare()。

3.3语句处理

语句处理是程序的核心部分,通过语法分析,查找符号表,生成相应的 P-code 代码,在解释执行时,根据生成的 P-code 代码解释执行。语句包括赋值语句、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、和空语句,各语句之间也
是相互嵌套,相互调用的。

3.4解释执行

程序的解释执行是通过生成的 P-code 完成的,P-code 指令详见“目标代码说明”部分,
语句处理部分生成的 P-code 指令储存在 codeTable 中,解释执行是循环遍历 codeTable 根据
规定的规则进行执行。

3.3错误处理

错误处理嵌套于声明过程和语句处理部分,包括语法错误和语义错误,在进行声明处理和语句处理时,如果不符合语法或语义,则报告相应错误,并跳到下一条语句或下一块。

4. 符号表管理方案

为了便于实现，这里用了符号表，符号表定义如下

```
struct tablestruct
{
        char name[al];                  /*名字*/
        enum object kind;                   /*类型：const，var，array or procedure*/
        int val;                            /*数值，仅const使用*/
        int level;                          /*层次*/
        int adr;                             /*偏移*/
        int size;
```

```
        bool ret;
        int parameter;
};
//符号表操作
int position(char* idt, int tx);
void enter(enum object k, int* ptx, int lev, int* pdx, int length);
```

5. 存储分配方案

参考 p10 编译器存储器的设计,设计了两个存储器、一个指令寄存器、三个地址寄存器。

5.1P-code 指令存储 code

建立一个 code 数组存放生成的 P-code 指令,大小为 1000,如果指令数量超过 1000,则溢出,报错。

5.2数据存储器 s

数据区采用动态存储分配,通过建立一个栈式数据存储器动态分配程序的数据空间(栈式动态存储分配),结构设计如下:

运算区间
局部变量
RA
DL
SL

SL：静态链,保存直接外层分程序数据区基地址。
DL：动态链,保存原调用过程的数据区的基地址。
RA：返回地址,保存应返回的调用点(下一条指令)的指令地址。
所有的算术运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儿保留在栈顶。栈顶单元的地址放在栈顶地址寄存器中,即栈顶地址寄存器作为栈顶指针永远指向数据栈的栈顶。用另一个指令寄存器存放当前要执行的指令。与之对应,设置一个程序地址寄存器存放下一条要执行的指令地址,每执行一条指令,程序地址寄存器的地址值加 1(数组下标加 1)。此外,再设置一个基址寄存器,用于存放当前的分程序数据区在数据栈中的起始地址。

运行栈以数组的形式运行。通过栈顶指针的移动，栈顶数据的加载和保存来运行程序。

```
int p = 0, b = 0, t = 0;              /*指令指针，指令基址，栈顶指针*/
struct instruction i;  /*存放当前指令*/
int s[500];   /*栈最大数为 500*/
```

6. 解释执行程序*

解释执行程序根据存储分配方案设计的数据在栈顶与次栈顶进行操作，具体执行何种操作有生成的 P-code 进行。解释程序函数如下

```
void interpret() /*解释程序*/
{
	int p = 0, b = 0, t = 0;			/*指令指针，指令基址，栈顶指针*/
	struct instruction i;  /*存放当前指令*/
	int s[stacksize];   /*栈最大数为500*/
	printf("Start pl0\n");

	s[0] = s[1] = s[2] = 0;
	do {
		i = code[p];		/*读当前指令*/
		p++;
		switch (i.f)
		{
		case lit:		/*将a的值取到栈顶*/
			s[t] = i.a;
			t++;
			break;
		case opr:		/*数字、逻辑运算*/
			switch (i.a)
			{
			case 0:
				t = b;
				p = s[t + 2];
				b = s[t + 1];
				break;
			case 1:
				s[t - 1] = -s[t - 1];
				break;
			case 2:
				t--;
				s[t - 1] = s[t - 1] + s[t];
				break;
			case 3:
				t--;
				s[t - 1] = s[t - 1] - s[t];
				break;
			case 4:
				t--;
				s[t - 1] = s[t - 1] * s[t];
				break;
			case 5:
				t--;
				s[t - 1] = s[t - 1] / s[t];
				break;
			case 6://odd
				s[t - 1] = s[t - 1] % 2;
				break;
```

```
        case 7:
                error_msg(26);
                break;
        case 8:
                t--;
                s[t - 1] = (s[t - 1] == s[t]);
                break;
        case 9:
                t--;
                s[t - 1] = (s[t - 1] != s[t]);
                break;
        case 10:
                t--;
                s[t - 1] = (s[t - 1]<s[t]);
                break;
        case 11:
                t--;
                s[t - 1] = (s[t - 1] >= s[t]);
                break;
        case 12:
                t--;
                s[t - 1] = (s[t - 1]>s[t]);
                break;
        case 13:
                t--;
                s[t - 1] = (s[t - 1] <= s[t]);
                break;
        case 14://14号操作为输出栈顶值操作
                printf("%d", s[t - 1]);
                fprintf(fa2, "%d", s[t - 1]);
                t--;
                break;
        case 15://15号操作为输出换行操作
                printf("\n");
                fprintf(fa2, "\n");
                break;
        case 16:
                printf("?");
                scanf("%d", &(s[t]));
                t++;
                break;
        }
        break;
case lod:       /*取相对当前过程的数据基地址为 a 的内存的值到栈顶*/
        s[t] = s[base(i.l, s, b) + i.a];
        t++;
        break;
case sto:
        t--;
        s[base(i.l, s, b) + i.a] = s[t];
        break;
case sta:
```

```
                t--;
                s[base(i.l, s, b) + i.a + s[t - 1]] = s[t];
                t--;
                break;
        case loa:
                s[t - 1] = s[base(i.l, s, b) + i.a + s[t - 1]];
                break;
        case cal:               /*调用子程序*/
                s[t] = base(i.l, s, b); /*将父过程基地址入栈*/
                s[t + 1] = b;           /*将本过程基地址入栈，此两项用于base函数*/
                s[t + 2] = p;           /*将当前指令指针入栈*/
                b = t;                 /*改变基地址指针值为新过程的基地址*/
                p = i.a;               /*跳转*/
                break;
        case inte:              /*分配内存*/
                t += i.a;
                break;
        case fre:
                t -= i.a;
                break;
        case jmp:           /*直接跳转*/
                p = i.a;
                break;
        case jpc:               /*条件跳转*/
                t--;
                if (s[t] == 0)
                {
                        p = i.a;
                }
                break;
        }
    } while (p != 0);
}
```

7. 四元式设计*

对于整个文法而言，可能出现的有关数据的操作和对应的四元式设计如下表：

操作种类	四元式设计
Var a;	Var,,,a
const	Const,,,a
A=b	=,b,,a
A:=b	:=,b,,a
关系运算符(=, <>,<,<=,>=,>)	关系运算符,,a,b
运算符(+,*,-, /)	运算符,b,a,T1

8. 目标代码生成方案*

```
int gen(enum fct x, int y, int z) //生成目标代码
{
        if (cx >= cxmax)              //cxmax = pcode largest number
        {
                printf("Program too long"); /*程序过长*/
                return -1;
        }
        code[cx].f = x;
        code[cx].l = y;
        code[cx].a = z;
        cx++;                                 //table of p-code
        return 0;
}
/*生成目标代码 x:instruction.f; y:instruction.l; z:instruction.a;*/
```

9. 出错处理

1）对于明显的错误,报错误信息并输出错误位置继续进行编译。
2）在每个语法分析子程序出口处,检查下一个取来的符号是否为该语法成分的合法后继符号。若不是,则报告出错误信息,并且跳读一段源程序,直至去来的符号属于该语法成分的合法后继符集合为止。为防止跳过太多的源程序,设置一个停止符号集合,只要取出的符号属于合法后继符号集合或者停止符号集合,程序就停止跳读。
当 err > 0 程序无法执行 interpret（）函数，程序输出”error in program PL/0”

```
void error_msg(int n)
{
char space[81];
memset(space, 32, 81);
space[cc - 1] = 0;

printf("****%s %d  ", space, n); /*每次遇到错误直接报错在当前识别的行*/

switch (n)
{
case 1:printf("should be =  not :=\n"); break;
case 2:printf("Const declaration error  \n"); break;
case 3:printf("Const must use =\n"); break;
case 4:printf("Error in declare const / var \n"); break;
case 5:printf("Missing\",\"\";\" (Comma, Semicolon)\n"); break;
case 6:printf("Procedure symbol wrong \n"); break;
case 7:printf("Should be statement\n"); break;
case 8:printf("Program's after statement symbol wrong\n"); break;
case 9:printf("Should be\".\"Period\n"); break;
case 10:printf("Statement Missing\";\"\n"); break;
case 11:printf("Undefine symbol / variable \n"); break;
case 12:printf("Symbol not variable\n"); break;
case 13:printf("Should be\":=\"\n"); break;
case 14:printf("call should with symbol(标识符)\n"); break;
case 15:printf("call should with procedure\n"); break;
```

```
case 16:printf("Should be then\n"); break;
case 17:printf("Should be\"end\" or \";\"\n"); break;
case 18:printf("While do missing do\n"); break;
case 19:printf("Missing Variable\n"); break;
case 20:printf("Operator error\n"); break;
case 21:printf("Inside operation can't procedure symbol\n"); break;
case 22:printf("Operator missing )\n"); break;
case 23:printf("After factor can't this symbol(符号)\n"); break;
case 24:printf("Factor can't use this symbol\n"); break;
case 25:printf("Can't run procedure\n"); break;
case 26:printf("Procedure declaration error\n"); break;
case 27:printf("Missing comma in procedure parameter\n"); break;
case 28:printf("Variable name not in table / stack\n"); break;
case 29:printf("Undefine procedure\n"); break;
case 30:printf("Out of number range\n"); break;
case 31:printf("Out of address range\n"); break;
case 32:printf("read out of range\n"); break;
case 33:printf("Missing ) in condition\n"); break;
case 34:printf("Missing ( in condition\n"); break;
//case 35:printf("Missing begin\n"); break;
default:printf("\n");
}
//      exit(0);
err++;
}
```

错误类	详细
1	should be = not :=
2	Const declaration error
3	Const must use =
4	Error in declare const
5	Missing","";" (Comma, Semicolon)
6	Procedure symbol wrong
7	Should be statement
8	Program's after statement symbol wrong
9	Should be”.” (Period)
10	Statement Missing";”
11	Undefine symbol / variable
12	Symbol not variable
13	Should be":="
14	call should with symbol(标识符)
15	call should with procedure
16	Should be then
17	Should be"end" or ";"
18	While do missing do

19	Missing Variable
20	Operator error
21	Inside operation can't procedure symbol
22	Operator missing “)”
23	After factor can't this symbol(符号)
24	Factor can't use this symbol
25	Can't run procedure
26	Procedure declaration error
27	Missing comma in procedure parameter
28	Variable name already in table / stack
29	Undefine procedure
30	Out of number range
31	Out of address range
32	read out of range
33	Missing) in condition
34	Missing (in condition

三．操作说明

1．运行环境

运行环境是能够进行 C/C++自编译的系统，编译器用 C 和 C++编写。VS 2017 为开发环境。

2．操作步骤

双击运行.exe 文件，输入需要编译的源文件路径或名称，回车若程序没有错误，将源文件进行编译之后程序会问用户是否要长生符号表文件和目标代码文件，要是用户输入“y”或”Y”程序会产生和在控制台输出正确的符号表，符号表文件（table.txt），目标代码（p-code），目标代码文件（p-code.txt），并解释执行（interpret()函数）且在控制台输出运行结果，若源文件有错误在控制台输出错误信息。

四．测试报告

1．测试程序及测试结果

1. Test1.txt

```
var a,b,c;
CONST d=10;
proCEdure swap;
var a,b,c,d;
begin
a := 3;
write(a,3,4);
end;
begin
 read(b);
 read(a,c);

 c:=a+B;
call swap;
 if c>15 then
 wrIte(a,b,c);
end.
```

测试结果正确

2. Test1_error.txt

```
var a,b,c,d;
CONST d=10;
proCEdure swap;
var a,b,c,d;
begin
a := 3;
write(a,3,4);
end;
begin
 read(b);
 read(a,c);

 c:=a+B;
call swap;
 if c>15 then
 wrIte(a,b,c);
end.
```

测试结果错误（const d, d 已在 var 声明）

3. Test2.txt

```
var
        f1,f2,f3,flag,Number;
begin
```

```
        f1:=1;
        f2:=1;
        flag:=3;
        read(Number);
        if Number=1 then write(f1);
        if Number>=2 then
        begin
                write(f1);write(f2);
                while flag<=Number do
          begin
                        f3:=f1+f2;
                        f1:=f2;
                        f2:=f3;
                        write(f3);
                        flag:=flag+1;
          end;
        end;
end.
```

正确，程序能编译 while 循环。

4. Text2_error.txt

```
var
        f1,f2,f3,flag,Number
begin
        f1:=1;
        f2:=1;
        flag:=3;
        read(Number);
        if Number=1 then write(f1);
        if Number>=2 then
        begin
                write(f1);write(f2);
                flag<=Number do
          begin
                        f3:=f1+f2;
                        f1:=f2;
                        f2:=f3;
                        write(f3);
                        flag:=flag+1;
          end;
        end;
end.
```

程序进入死循环因为缺了 while 结果错误

5. Text3.txt

```
var x,y,m,n,pf;
const true=1,false=0;
procedure prime;
 var i,f;
 procedure mod;
  begin
```

```
   x:=x-x/y*y;
  end;
 begin
   f:=true;
   i:=3;
   while i<m do
   begin
       x:=m;
       y:=i;
       call mod;
      if x=0 then f:=false;
     i:=i+2;
    end;

   if f=true then
     begin
      write(m);
     pf:=true;
    end
   end;
begin
  pf:=false;
  read(n);
  while n >= 2 do
  begin
    write(2);
    if n=2 then pf:=true;
     m:=3;
     while m<=n do
     begin
      call prime;
      m:=m+2;
     end;
    read(n);
   end;

   if pf=false then write(0)
end.
```

程序正确编译生成目标代码。

6. Test3_error.txt

```
var x,y,m,n,pf;
const true=1,false=0;
procedure prime;
 var i,f
 procedure mod;
  begin
   x:=x-x/y*y;
  end;
 begin
   f:=true;
   i:=3;
   while i<m do
```

```
  begin
     x:=m;
     y:=i;
     call mod;
    if x=0 then f:=false;
   i:=i+2;
  end;

  if f=true then
   begin
    write(m);
   pf:=true;
  end
 end;
begin
 pf:=false;
 read(n);
 while n >= 2 do
 begin
  write(2);
  if n=2 then pf:=true;
   m:=3;
   while m<=n do
   begin
    call prime;
    m:=m+2;
   end;
  read(n);
 end

  if pf=false then write(0)
end.
```

程序运行结果是错误，报错好多错误信息

7. example2.txt

```
var      b , c ;
const  a = 10 ;
procedure p;
     begin
          c := b + a ;

     end;

begin
           read(b) ;
           write(b) ;
       write(2+3*4-1) ;
           if  b = 0  then
              begin
                  read(b) ;
                  write(c)
              end ;
             call p ;
```

```
end.
```
结果正确。

8. example2_err.txt

```
var      b , c ;
const  a = 10 ;
procedure p;
     begin
          c := b + a ;

     end;

begin
           read(b ;
           write(b) ;
       write(2+3*4-1) ;
           if  b = 0  then
              begin
                  read(b) ;
                  write(c)
              end ;
             call p ;
end.
```
结果错误。

9. Test4.txt

```
var b;
begin
 read(b);
 if odd b then
  write(111);
end.
```
结果正确，程序能运行。

10.Test4_error.txt

```
var b;
const a = 99999999;
begin
 read(b);
 if odd b then
  write(111)
end.
```
程序结果错误，const a 超过地址最大数。

2．测试结果分析

测试 1 和 2：

在此测试重点是在程序能否识别重复声明的标识符，在测试 1 测试程序无误结果正确。在测第 2 程序的时候结果错误“28 Variable name already in table / stack”因为 d 在 var 已经声明了，但又在 const 声明。

测试 3 和 4：

在此测试重点在测试 while 语句，在测试 3 程序结果正确，程序能成功编译并输出目标代码，但在测试 4 程序报错因为缺了 while 语句，而且这程序进入了死循环，这个术语程序的 bug。

测试 5 和 6：

在此测试重点在测试程序的文法覆盖，在测试 5 程序结果正确，程序能成功编译并输出目标代码，但在测试 6 程序会报错各种各样的错误信息。

测试 7 和 8：

在此测试重点在测试程序对语法错误，在测试 7 程序结果正确，程序能成功编译并输出目标代码，但在测试 8 程序会报错是因为在 Read 语句缺了“）“。

测试 9 和 10：

在此测试重点在测试程序的 odd 和超过地址错误，在测试 9 程序结果正确，程序能成功编译并输出目标代码，但在测试 10 程序会报错是因为 const a 的值超过了地址上界。

五．总结感想

这次是我第一次学编译原理，遇到了编译原理的课程设计虽然很难但我学了好多新的东西，而且对编译器更加了解，一开始老师将课程设计的时候当时我自己有点模糊要选那个难度，因为我觉得我自己的编程能力不够强我不敢选第 3 难度，听起来也很恐惧，然后我选了第 2 难度，看了收到的文法我自己怕选了之后当做编译器时我怕做不出来，后来我改成了第 1 难度，虽然看起来文法很简单但对我自己来说实现编译程序完全不简单，需要考虑很多方面，像在测试过程中测试 1 我程序突然爆了，完全无法执行测试程序，后来我从网上看了一些文档，后来改了一点，但不测的时候只能得到 2 分因为程序当前无法是别’\t’，测试 2 的时候我程序又出了问题，虽然现在我最总版本还有一点 bug 但比之前已经好多了，上了课程设计的时候我觉得提高了我的编程能力，因为实现这个编译器的时候有些方法我从网上学的，然后有些方法从原代码翻译成 C 语言，虽然在做作业的时候需要经常熬夜，压力很大，但这门的对我来说很有意思而且在我国我觉得学不到这门课。希望编译原理和课设会越来越好。