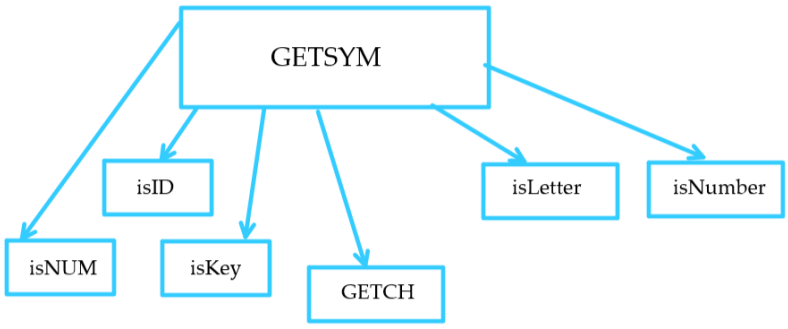
1. 词法分析

词法分析代码见附录 1。该词法分析实现了对读入数据中的空格回车等符号的处理，仅对有效字符进行词法分析，且对大小写不敏感，最终输出带有行号的存储单词的 SYM 数组。该程序设置了四个全局表，KeyWordList 中存储关键字、界符、算符，ID 数组存储标识符，NUM 存储常数，SYM 存储最终生成的单词。单词以二元组的形式存在，关键字、界符、算符的类型(即二元组的第一项)都为 1，ID 的类型为 2，NUM 的类型为 3。二元组第二项为在对应的表中的索引。

由于词法分析时需要向后试探一个字符，故而设置了一个全局量 buffer。GETCH 函数用于每次读入字符，该函数只有在缓冲区 buffer 中没有字符时才会从源文件读入字符,同时也是在该函数中完成对大小写的兼容。词法分析的核心函数为 GETSYM，该函数负责模拟正则文法的自动机，从而将字符组成单词。IsLetter 和 isNumber 函数帮助 GETSYM 判断读入的字符的类型，isNUM,isID,isKey 分别判断是否在对应的表中出现，若出现则直接生成单词，若没出现则需要填入。字符组成一个字符串时也需要通过 isKey 判断是否是标识符。设置了一个全局量 lineCounter, GETSYM 每读入一个'\n'就递增该值,维护读入字符所在的行数，用于后续错误处理。

词法分析的相关代码结构如下：



此外该部分进行了一定程度的错误处理，根据标识符和常量的形式，对数字后面跟字母的符号进行检测，并输出错误信息。（例如：88a,1temp 等非法标识符）

2. 语法分析

语法与语义分析代码见附录 2.根据 PL0 语法制作的程序框图如下图 2,3 所示。其中图中英文说明和实验题目中产生式的非终结符的对应关系如下表格 1 所示。(其中没有描绘的 sentence()用于对各个语句进行选择)。

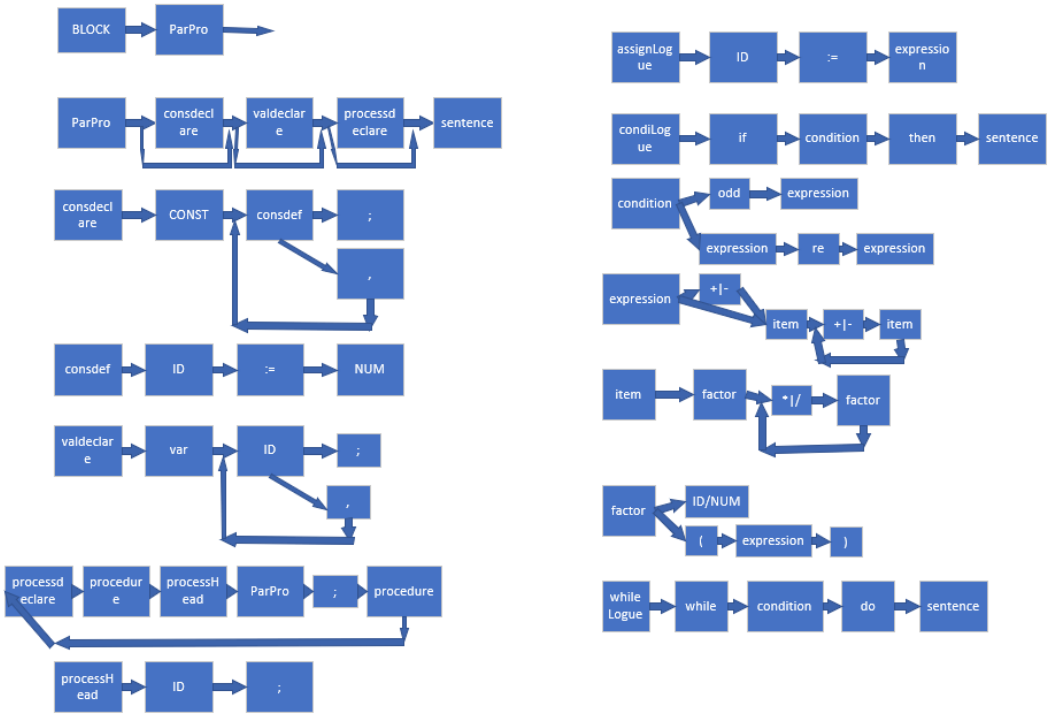


图 1

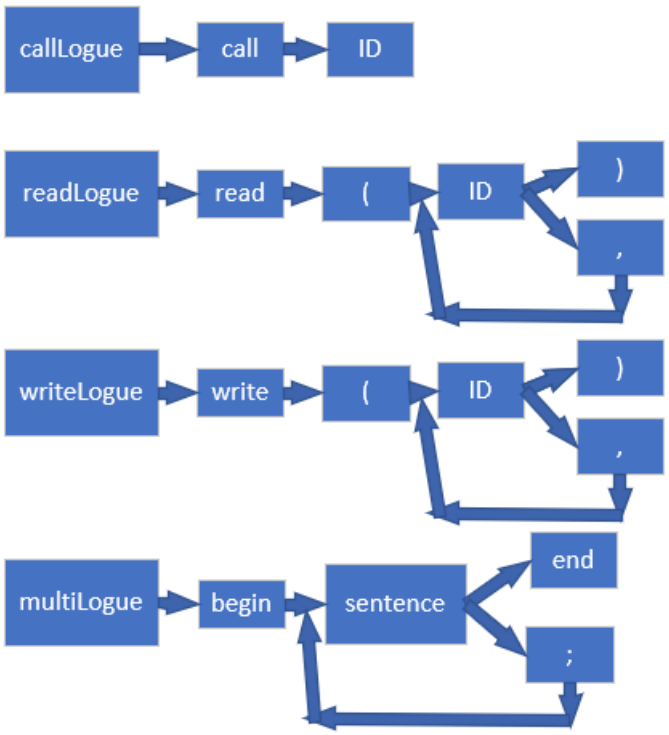


图 2

表格 1

非终结符名称	对应图中符号
程序	BLOCK
分程序	ParPro
常量说明部分	consdeclare

变量说明部分	valdeclare
过程说明部分	processdeclare
语句	sentence
常量定义	consdef
标识符	ID
常数	NUM
过程首部	processHead
赋值语句	assignLogue
条件语句	condiLogue
条件	condition
关系运算符	re
当型循环语句	whileLogue
过程调用语句	callLogue
读语句	readLogue
写语句	writeLogue
复合语句	multiLogue
表达式	expression
项	item
因子	factor

采用递归下降的方式识别程序词法分析后生成的单词列表，递归下降翻译的程序代码见附录 2（该部分附录包括中间代码生成的程序）。符号表在语法分析的时候建立，由于 PL0 语言本身非常简单，所以符号表以顺序表的形式建立，一个 procedure 的常量和变量紧跟在符号表的 procedure 项之后，在后续的程序代码中由一个过程在符号表中最初始记录的位置标识该过程，即用某个过程的 procedure 变量标识该过程。而后对该过程中某个符号的查找可以从标识的位置顺次查询；可以通过两个过程标识的位置进行次序比较等。在建立符号表的时候，维持一个全局量 pointer，说明当前过程对应的 procedure 项在符号表中的位置，标识当前过程，维护 pointerStack 存储父子关系，从而设置符号表中的儿子项指向父亲项；即在建立符号表的时候拉起一条父子链。（将第 0 项看作 main 过程的 procedure 项所在的位置，即使该项的内容不是 procedure）.同时根据父子链，在建立符号表时，每读入一个标识符，就判断它的祖先的域中是否有和它同名的标识符，若有则输出错误信息。

在各个递归下降子程序中，通过给 SYM 数组加入新项来保存各个子程序之间的调用关系。最终根据 SYM 数组的内容生成语法分析树。对下述简单代码构建的语法树如下：（为了方便展示，下面的各个运行结果都是针对该简单代码，程序本身可以执行一些更复杂的代码，但是不便于展示）

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

        c := b + a;
    end;
begin
    read(b);

    while b # 0 do
    begin

        call p;
        write(c);
        read(b);

    end
end.
```

代码 2

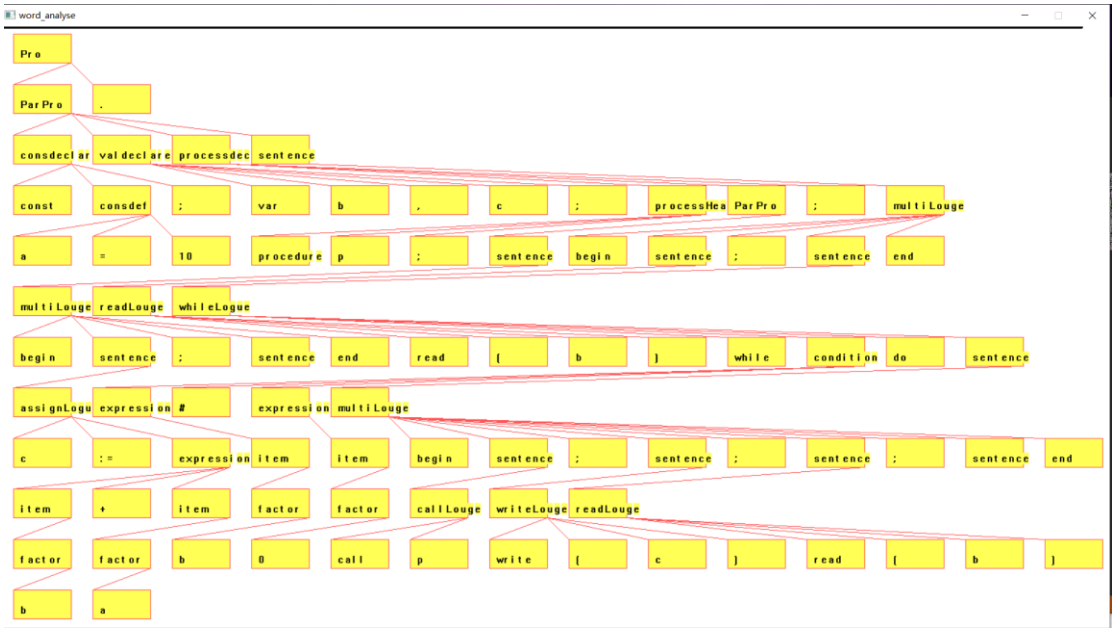


图 3

此外，由于图形化语法输出能输出的规模有限，故而还提供了命令行输出语法树的方式.

3. 语义分析与中间代码产生

语义分析时，生成的中间代码类型如下：

```
1  LIT 0 a          将常数值取值到栈顶 a为常数值
2  LOD l a          将变量值取值到栈顶 a为偏移量 l为层差
3  STO l a          将栈顶内容送入某变量单元中 a为偏移量 l为层差
4  CAL l a          调用过程 a为过程地址 l为层差
5  INT 0 a          在运行栈中为被调用的过程开辟a个单元的数据区
6  JMP 0 a          无条件跳转至a地址
7  JPC 0 a          条件为假 跳转至a地址 否则顺序执行
8  OPR 0 0          过程结束调用 返回调用点并退栈
9  OPR 0 1          栈顶元素取反
10 OPR 0 2          次栈顶与栈顶相加 退栈两个元素 结果进栈
11 OPR 0 3          次栈顶减栈顶 退栈两个元素 结果进栈
12 OPR 0 4          次栈顶乘以栈顶 退栈两个元素 结果进栈
13 OPR 0 5          次栈顶除以栈顶 退栈两个元素 结果进栈
14 OPR 0 6          栈顶元素奇偶判断 结果进栈顶
15 OPR 0 7
16 OPR 0 8          是否相等 退栈两个元素 结果进栈
17 OPR 0 9          是否不等 退栈两个元素 结果进栈
18 OPR 0 10         次栈顶是否小于栈顶 退栈两个元素 结果进栈
19 OPR 0 11         次栈顶是否大于等于栈顶 退栈两个元素 结果进栈
20 OPR 0 12         次栈顶是否大于栈顶 退栈两个元素 结果进栈
21 OPR 0 13         次栈顶是否小于等于栈顶 退栈两个元素 结果进栈
22 OPR 0 14         栈顶值输出至屏幕
23 OPR 0 15         屏幕输出换行
24 OPR 0 16         从命令行读入一个输入至于栈顶
```

该语义分析程序对一些 PI0 不需要考虑的特性提供了支持。考虑了 call p 的时候 p 还没有声明的情况，回填的时候也考虑了回填 call 指令的情况，在符号表中通过 p 的 called 数组存放调用 p 的过程的信息和生成的 call 指令的信息。实验指导书要求 pl0 最多允许三层，但是语义分析过程中对过程间父子关系或者兄弟关系的确定的方法是对具有更多层的代码也兼容的。

根据中间代码的形式，我们需要知道层次差与偏移量。故而在上述语法分析过程中设置了全局量 CURLEVEL 记录当前层。在建立符号表的时候，会根据 CURLEVEL 存入符号的当前层信息。生成中间代码的时候，根据当前的 CURLEVEL 和符号表中存储的信息，可以得到层差。由于符号表的结构，从符号在符号表中的位置向前遍历，直到找到 procedure 项或者索引为 0，即可得到偏移量。生成中间代码时，根据 judgeNode 判断标识符的有效性，即标识符是否声明，是否在当前过程作用域或者祖先作用域中声明，否则输出错误信息。为了拉链反填等操作，需要知道下一条生成的代码的行号，故而维护全局量 CODELine,指向下一条生成的中间代码的地址。

- **表达式的翻译模式：**

  在 expression 函数中先判断表达式的第一项是否为负号，若为负号，则在读入项(item)之后产生 opr 0 1 代码完成栈顶元素取反。然后进入加减运算的循环，每个循环中读入运算符后先记录，然后读入项,之后，再根据之前记录的运算符，生成相应的中间代码，而后继续循环。由于上述的读入项，是通过调用子函数 item()完成的，故而在读入完成，即 item()返回时,要计算的量已经被压入栈，后面的中间代码只需要对栈顶进行操作。

  在 item()函数中通过 factor()读取一个因子，而后进入乘除循环，和加减循环的处理方式类似，先记录运算符，再通过 factor()读入因子，之后根据运算符产生中间代码，继续循环。

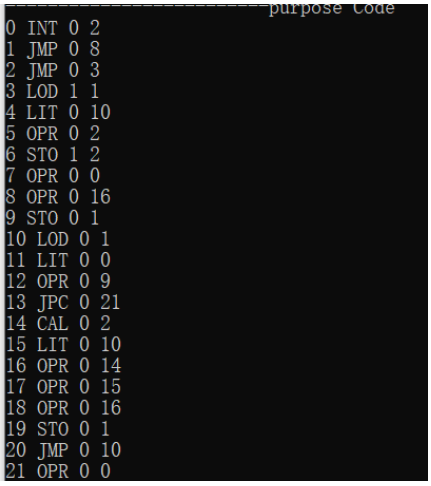
  factor()中，若读入标识符，则先判断标识符合法性，查表得到标识符的类型，若为 const，则生成 LIT 代码，否则生成 LOD 代码，将对应的常量或者变量压入栈。若读入常数，则生成 LIT 代码。若要读入(expression)，则将中间代码生成交给调用的 expression().

- **赋值语句的翻译模式：**

首先查表判断读入的标识符类型，若为 const，则输出错误信息，因为 const 类型并不能被赋值。读入":="并调用 expression 之后，插入目标代码 STO，说明将存放 expression 运算结果的栈顶存入变量。

- **过程调用语句的翻译模式：**
  过程调用时先判断读入的标识符是否在符号表中被声明，标识符是否为一个 procedure 名字。然后判断该过程是否可以被当前过程调用。判断符号表中调用过程和当前过程的位置，若在当前位置之前，则通过 findsameFather()函数找到两个过程的最邻近共同祖先，判断该祖先是否是调用过程的父亲；若找不到祖先，或者不是调用过程的父亲，则输出错误信息。若在当前过程之后，判断是不是当前过程的子孙。
- **条件跳转语句翻译模式：**
  condiLogue 调用 condition()读入条件之后，输出条件为假的跳转 JPC 0,-1。之后调用 sentence()处理 then 之后的语句，调用返回后，得到下一条中间代码的地址，反填回上述跳转语句 JPC 中的-1 项。condition()中在读入要判断的表达式后，输出相应的中间代码进行判断。
- **当型循环语句翻译模式：**
  在条件判断之前记录下一条生成的代码的位置 beforeCondi, condition()调用返回之后，记录下一条代码的位置 aftDoLine. 生成 JPC 跳转代码。do 之后的 sentence 调用返回之后，生成 JMP 0,beforeCondi 代码，跳转到 while 后的条件判断。生成该条语句之后，根据 aftDoLine 找到之前生成的 JPC 跳转代码的位置，回填当前的下一条代码的位置。完成判断语句为假时的 falselist。
- **过程说明语句和分程序的翻译模式：**
  在 processdeclare()函数中，每次读入分程序的分号之后，生成 OPR 0,0，表示一个过程结束。在分程序 ParPro()中，先调用 consdeclare 和 valdeclare 生成当前过程的符号表，在建符号表的时候维护变量 countInta,计数变量的个数。然后生成 INT 代码为变量分配空间，同时生成的 INT 代码作为 call 该过程时的入口。生成 INT 之后再生成 JMP 代码，用于跳转到程序语句入口，且在不需要分配空间时(无 INT 指令时)，作为 call 的入口。该代码会在 sentence()中被回填。
- **<语句>的翻译模式：**
  为了判断当前过程进入 sentence()函数时，是否已经有了程序入口，在符号表中的当前过程的 procedure 项中增添 codePort 项。sentence()函数根据 codePort 进行判断，若还没有语句入口，则设置下一条代码为语句入口，回填 ParPro()中生成的 JMP 语句。
- **读写语句的翻译模式：**
  read 之后每读入一个标识符，先判断其合法性，生成 OPR 0,16 将字符读入栈顶，再用 STO 写入标识符。write 每读入一个标识符，判断标识符合法性之后，采用 LOD 将标识符放到栈顶或者用 LIT 将 const 放到栈顶，然后生成 OPR 0,14 输出，OPR 0,15 换行。

之后，根据上述内容将中间代码生成的程序实现。对上述代码 1 生成的中间代码如下：



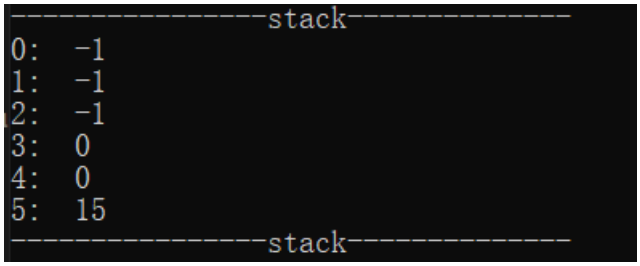
4. 代码解释执行

中间代码的执行程序如附录 3 所示。设置全局量 registerL 为当前指令，registerP 指向下一条执行的指令，T 作为栈顶寄存器，B 作为基地址寄存器；设置栈 newPointerStack，存储调用过程的信息。维护全局量 pointer，指示当前指令所在的过程。在符号表中新增 isassigned 项，判断该项是否被赋值，若 LOD 未赋值的变量则产生错误信息。在符号表中增加 assignedValue，存储该项的当前值。

静态链：通过 pointer 得到当前过程所在的 procedure,通过查找符号表中 procedure 项的 father 链，得到当前过程的所有祖先。在符号表中新增 B 项，用于存储该 procedure 在栈中的 base。静态链通过在符号表中查找祖先的 B 项得到。

动态链：在 CAL 指令改变栈的 base 的时候，用变量存储原来的 base，在 CAL 指令改变 base 之后，将动态链压入栈中。OPR 0,0 调用返回的时候从当前 base 向上找到动态链，将动态链填回基地址寄存器 B。下面描述具体指令的实现情况：

LOD 指令根据层差和偏移量从符号表中取出 assignedValue, 而 STO 指令用于给对应的 assignedValue 赋值。OPR 指令对栈顶进行增改操作。CAL 指令修改栈的基地址，填入下一个栈帧所需的基本信息（静态链，返回地址，动态链），并修改 registerP 的值。JMP 和 JPC 指令修改 registerP 的值。上述代码解释执行时，调用的 p 将退出时的运行栈如下图：



该代码量为 2000+，采用 easyX 图形库完成对语法树的输出，具有较为完善的错误检测能力。由于编译器希望能检测尽可能多的错误，故而每次出错之后只会返回-1，而不会终止运行。设计思路有一些地方比较冗余，譬如设计的时候考虑了 call 的时候 call 的标识符还没有声明的情况，但是由于 PL0 比较简单，并不会出现这种情况。同时为了兼容词法分析中的单词表，需要建立符号表和其他表格之间的对应关系，增大了计算量。