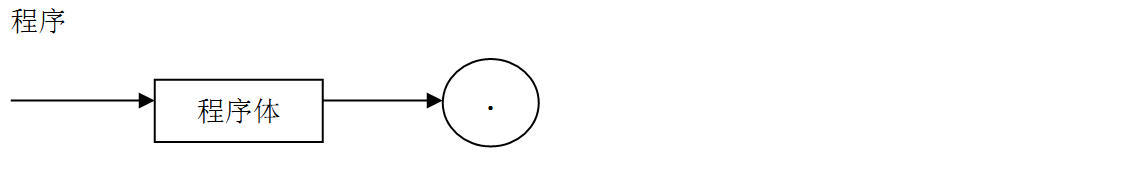
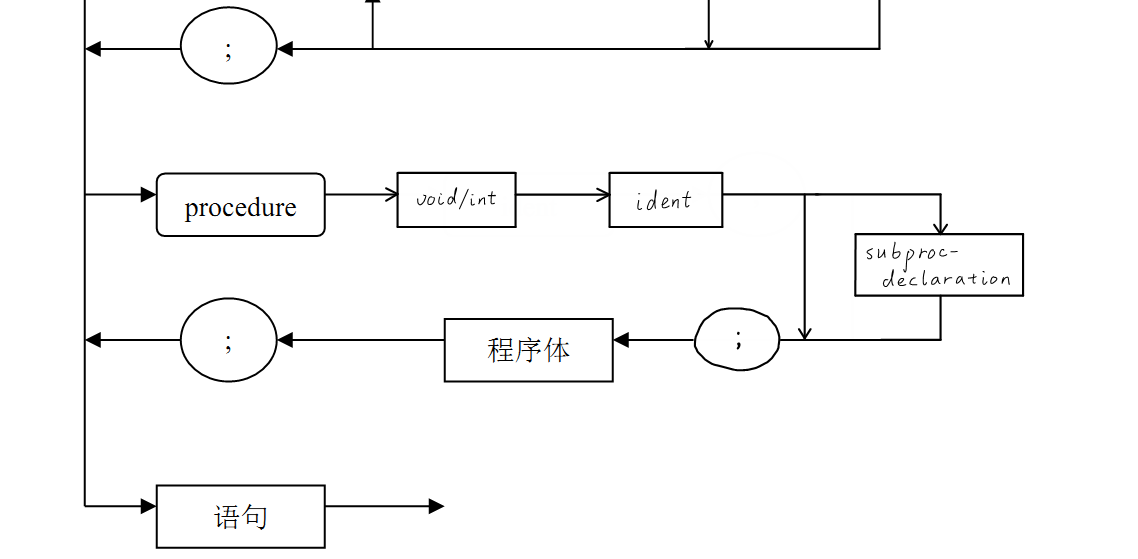
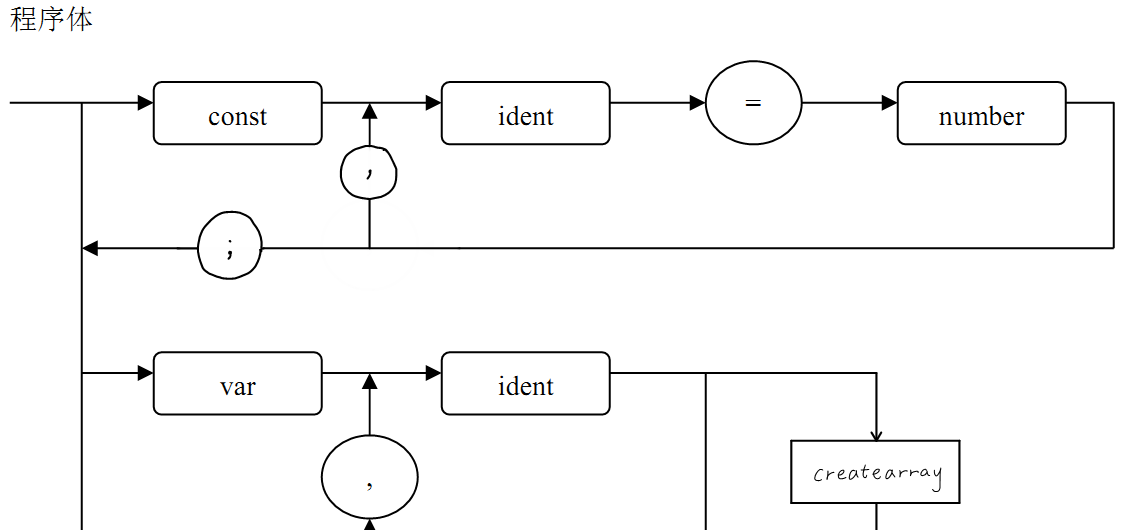
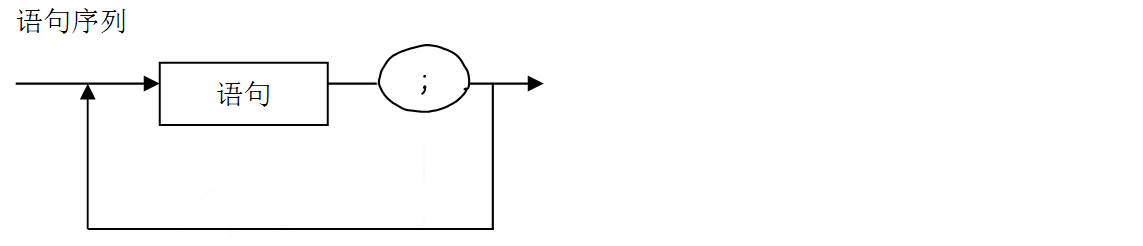
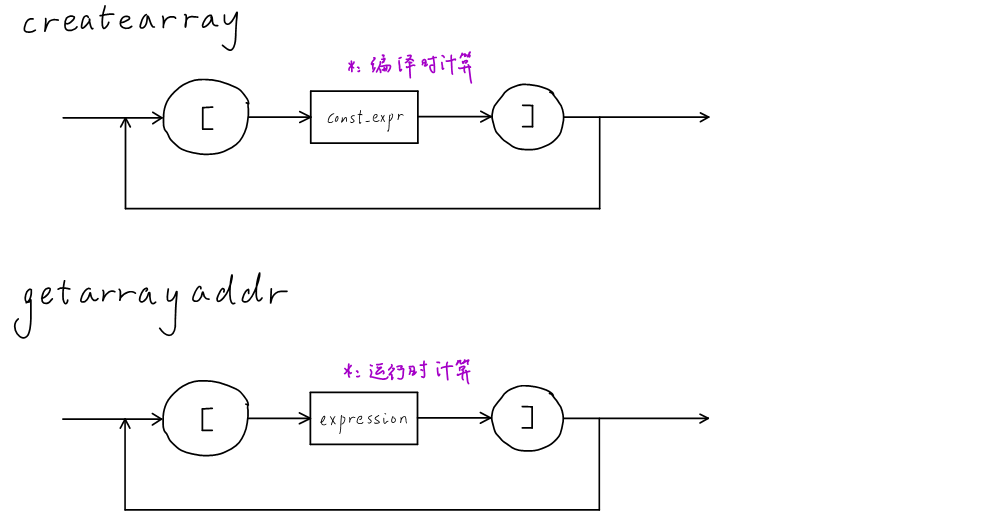
**PL0语言提高设计报告**

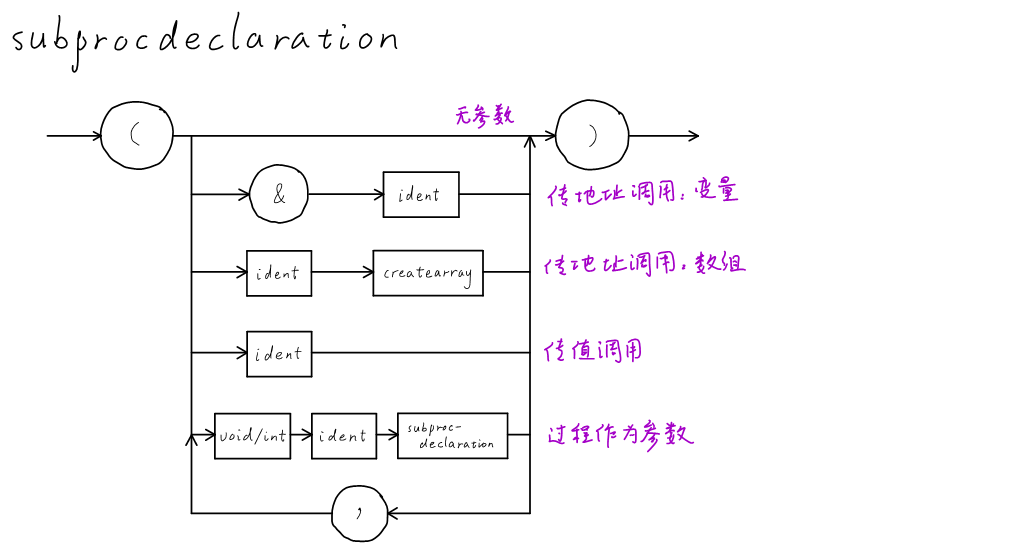
By 邵维科，李展鹏，于敬禹

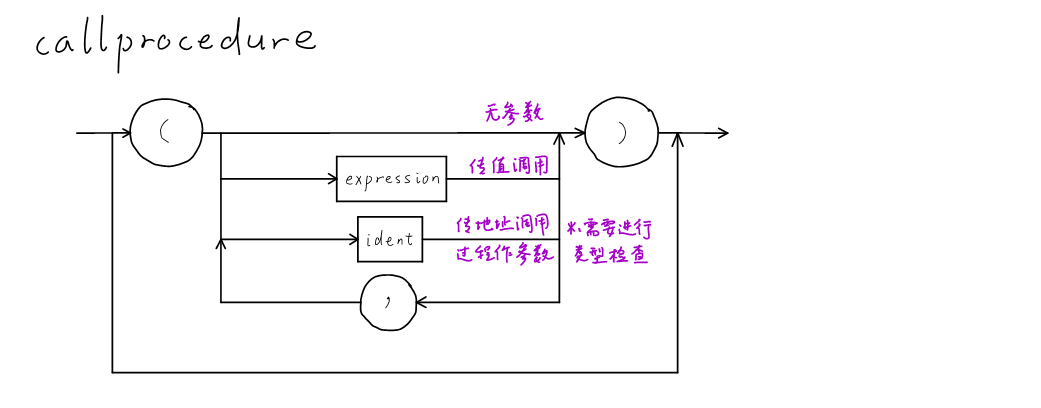
**1.提高设计语法图：**

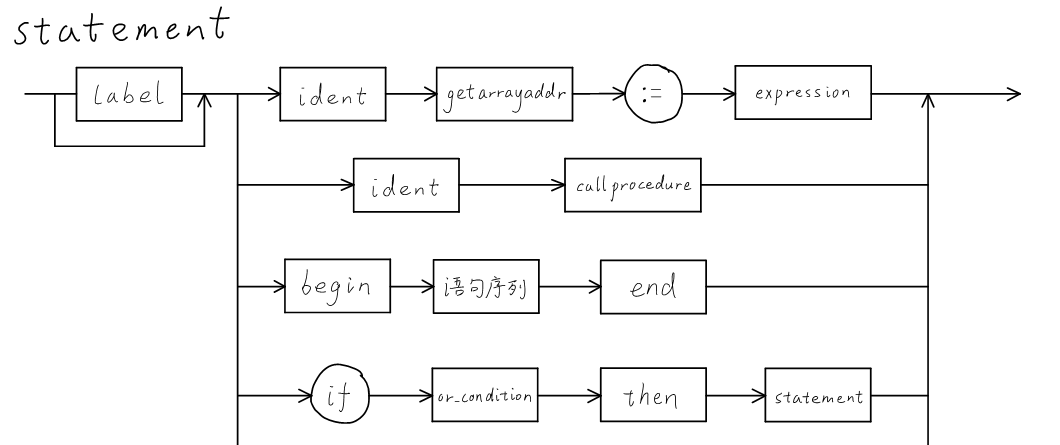


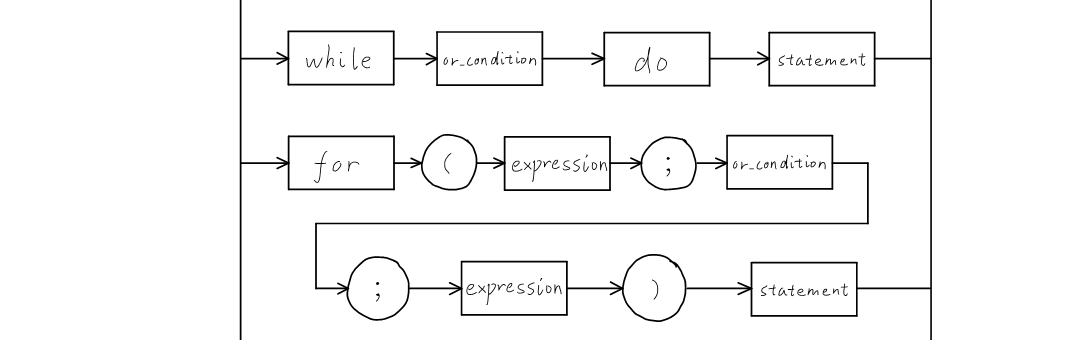


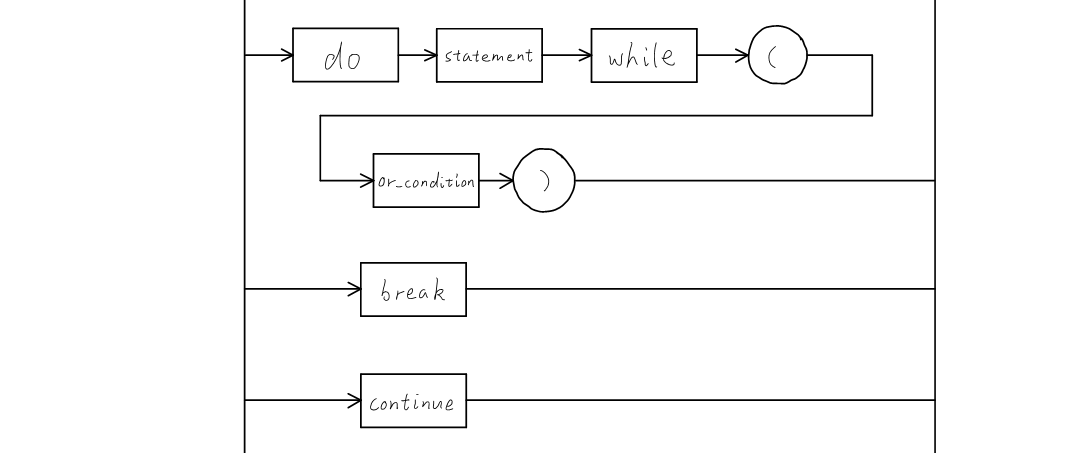


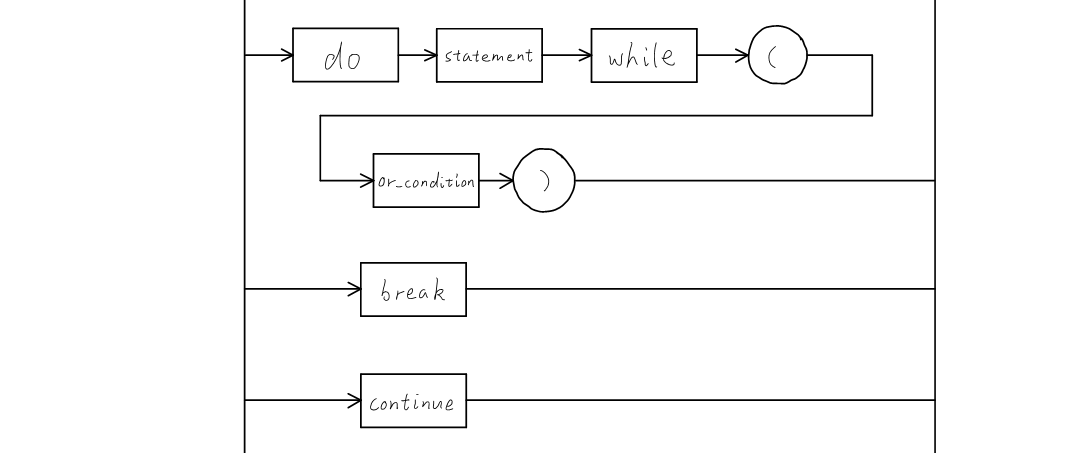


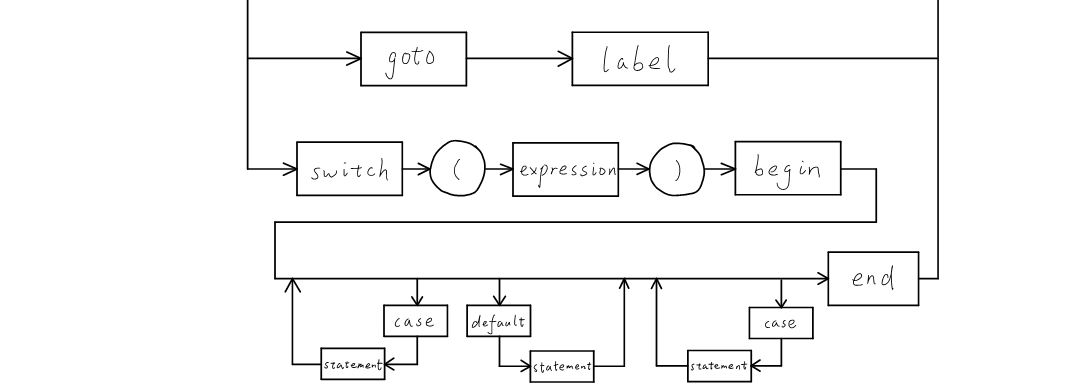


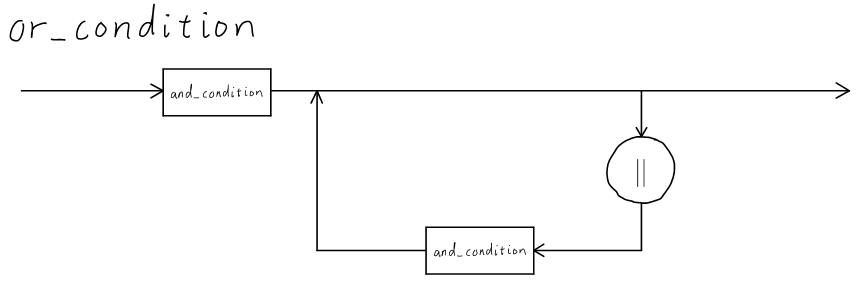


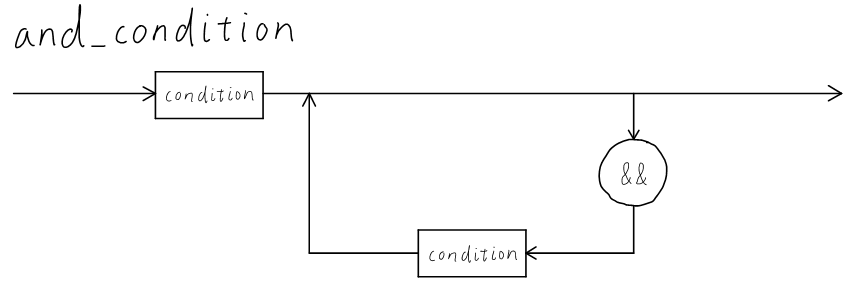


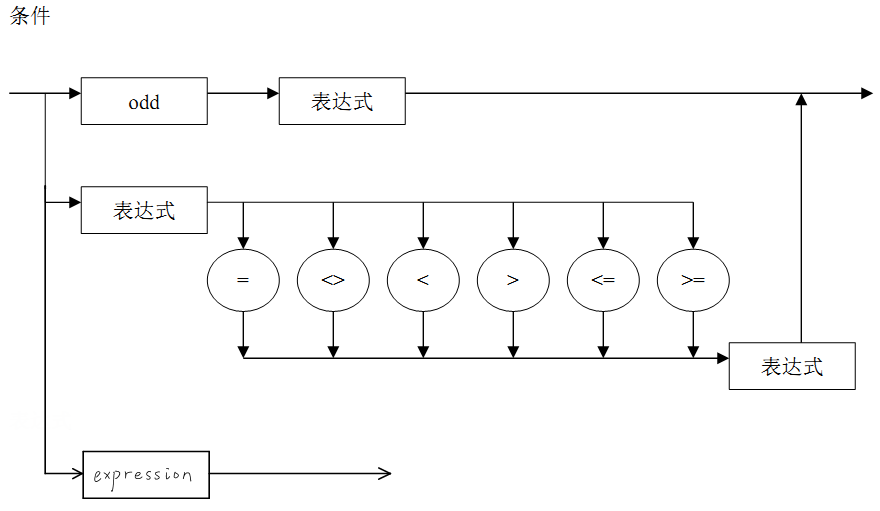


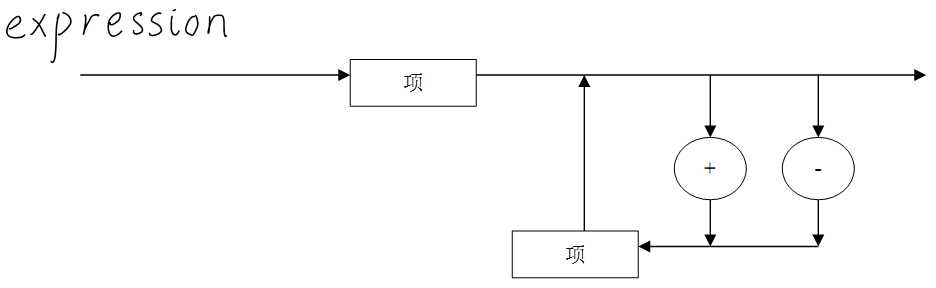


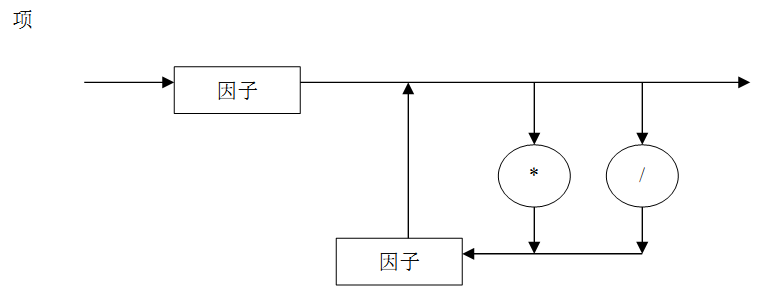


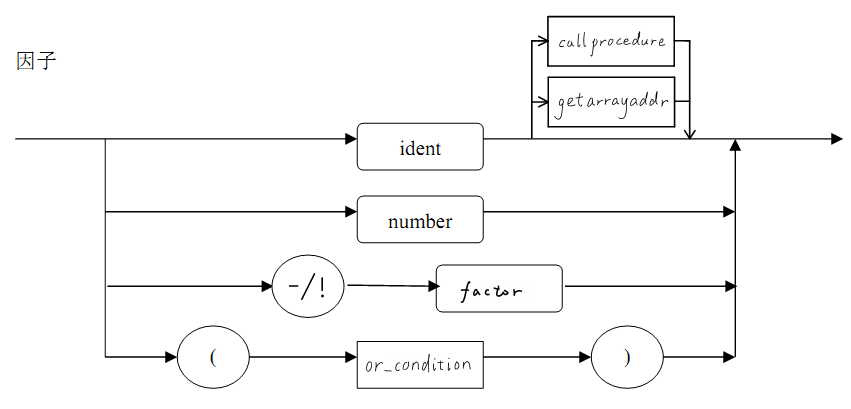












**2.各小题设计：**

**第九题：**

添加传地址调用的关键就在于怎样处理含有地址信息的变量，或者左值变量。如果在不向程序员开放左值变量的基础上实现传地址调用，可以有两个实现思路：第一，在编译阶段认定一些变量是左值变量，编译器可以处理它们的地址，但是这个权限不开放给程序员；第二，不把左值变量单独作为一个类型，而仅仅只是把使用地址或者使用值作为一般的变量的一个属性，在使用这个变量的时候根据符号表中的这个属性来决定调用的是传值的指令还是传地址的指令。本程序的实现使用的是前一种方式，即在程序的各个环节都承认地址（指针）可以单独作为一种变量，但是这种变量只能在函数的形式参数中声明，从而限制了它的使用方式。另一方面，在后端，指令的设计一方面要考虑语言的限制，另一方面又要尽可能具有扩展性地支持前端的语言特性。在本题中，原本变量的寻址方式是静态链+相对位移寻址，而题目要求的传地址调用就是对这种寻址方式的一种扩展。因为被调用函数和传递的参数对于调用者都是可见的，所以它们一定在调用者的祖先域中。而这二者之间的关系可以是被调用函数在参数的祖先域中，或参数在被调用函数的祖先域中的任一种。所以对于被调用函数而言，参数并不一定是可见的，更遑论静态链寻址。因此，传地址调用传递的只能是绝对地址而不能是相对地址，在原本的寻址方式中需要添加绝对寻址的方式。本实现中的LODS/STOS指令回应了这种需求。并且，指令本身读取栈顶内容作为绝对地址的特性也考虑到了尽可能多类型的变量的传地址调用的要求。正因为从类型设计所体现的独立性到指令设计所体现的适应性，在这个实现中可以对已有的各种类型进行传地址调用，包括数组元素，而之后继续添加新类型以后也可以比较方便地拓展。

传地址调用的具体实现分为三个部分：第一，函数的声明。由于产生式的更改，函数的声明部分极大地复杂化了。因此，就有必要从产生式中提取出相对简单的部分作为独立的函数提供调用。另外，由于添加了数组作为函数的参量，函数的复合类型变得更为深入，需要进一步设计复合和清除复合的具体实现。第二，调用序列的编写。在调用前需要读取各个参数并且把它们放在对应的位置，这之中需要进行简单的类型检查，还需要对可能出现的错误输入做出特定的回应。为了放置变量的地址，在实现中添加了LEA指令。第三，在函数块内使用在函数头上声明的形式参数。由于形式参数中的传地址调用与其他的传值调用有本质的不同，所以在类型上全设置为新添加的POINTER类型，变量与数组的指针在决定一个类型是否为复合类型的属性ptr上做出区分（是否为NULL）。通过对一个名字的类型的检查，可以决定在调用的时候是产生原来传值的指令LOD/LODI/STO/SOTI还是传地址的指令LODS/STOS。

**第十题：**

实现过程作为参数的关键在于如何声明和初始化一个作为参数的过程。实际上，允许把过程作为参数就是允许过程声明中的嵌套结构，由此过程的声明本身极大地复杂化了，过程的复合类型也被进一步地复杂化。事实上，过程头中的嵌套结构的内部和最外层有着不同：最外层是一个过程的第一次声明，声明的这个过程具有函数体，过程头里面的形参会在函数体中被使用；而内部的嵌套过程则不同，这些过程实际上只是将来某个将要被输入的参数的“别名”，它们内部的形参并不会实际被使用，而仅仅只是指示了这个过程的类型。而实际被使用的形参是被输入的那个参数的声明中出现的那些形参。因此，这种嵌套结构对应的产生式在实现中被单独剥离出来作为一个函数而被过程的第一次声明调用。

在实现过程作为参数的过程中，PLO本身的类型系统进一步得到了完善。在数组的实现中偶然添加入的指针属性ptr被进一步确认为复合类型的重要组成部分。事实上，本实现中的复合类型就是基础类型加上编译时给ptr动态分配的附加内容。通过ptr指针，实现了复合类型的嵌套。在实现中，ptr里可以分配数组的各个维度，也可以分配函数的各个参数的类型。

对过程作为参数的实现同样涉及三个部分：首先是函数声明本身的更改。基于和传地址调用中需要添加寻址方式相同的理由，即被调用函数可能在参数的祖先域中，参数不一定对被调用函数可见。从而要通过静态链寻址来设置作为参数的过程的静态链就不一定能成功，而这个过程是通过CAL指令来实现的，所以在指令上也需要有所增加。因此，需要在传递过程的入口地址的同时传递静态链。所以过程作为参数时就需要在栈中占用两个字的空间来存放它的内容，而其他变量只需要一个字。因此在过程声明时为各个参数分配空间的过程中就需要参考参数的类型来决定分配多少空间。这里对于参数是沿读取过程正向放置还是反向放置就会有实现上的不同。第二，在调用序列中放置静态链的实现可以使用LEA指令完成，放置入口地址则可以使用LIT完成。第三，在过程块内使用作为参数的过程时，正如上面所分析的，不能继续使用CAL指令来为被调用的过程放置静态链，由此新设置了一个CALS指令，直接从栈中读取过程的静态链和入口地址。

**第十一题：**

Goto：goto语句需要label的支持，将label看成一种ID，其类型为ID\_LABEL，label存放在table中，当成常量存放，其对应的指令地址存放在value成员里，在分析进行到goto时，在table中找到label对应的位置，生成跳转到value的JMP指令即可。

Break和continue：两个语句本质上只是一个JMP指令，直接跳转到指定的指令地址，为了分析跳转地址，在编译程序中，设置了一个栈，栈的元素是一个两成员的结构体，两个成员分别记录控制跳转语句类型和记录语句指令地址。break要求在循环语句中直接跳转出循环，而continue要求跳转到循环判断部分继续下次循环。对于三种循环，都需要记录循环的结尾和开头，for和while语句的结尾在循环部分的最后，而do-while语句的结尾是判断条件的结尾；for和while语句的开头都在循环部分之前，do-while的则相当于在判断条件之前。代码中设置head和tail分别记录这两个值。具体来说，在三种语句中break都跳转到tail，必须在分析完所有语句之后才能回填break的跳转地址；在for和while语句中，可以在分析continue时就得到head值，从而直接生成完整的指令，在do-while语句中需要完成全部分析在回填。我在代码中设置了env变量，记录循环语句的类型，对于break，还可以辨析循环和switch的区别。考虑到循环的嵌套，全局变量head，env在进入新的内层循环时会被更新，因此在进入新的循环时，需要征用栈的一个元素分别存放head和env的旧值，并重新赋值。同时设置一个count变量来记录栈顶的增加量，也即需要回填的break和continue的数量，用于恢复到外层的栈顶和head及env的值。最后在break语句的分析中，分为switch和循环两种情况分别分析，前者只需生成跳转指令，后者需要在栈中加入数据，记录控制语句类型和指令地址，并且增加count。对于continue语句，在for和while语句中，直接生成跳到head的指令，在do-while语句中需要回填tail的值，并增加count值。

**第十二题：**

Do-while：这个语句和while语句相似，但是是直接执行循环部分然后再判断循环条件，所以需要在进入循环时先设定循环部分开头指令的label，分析这一部分后进入条件判断，生成JPC和JMP，前一个指令在条件为假时跳转到循环语句后面的语句，后一个指令用于在条件为真时跳转到循环开始的部分。

Switch：这个语句的设计中加入了一个数组switchtab，数组元素是结构体，有四个成员，分别记录每个case后的常量，case后语句的指令位置，case部分语句中用于跳出switch的break是否存在和break语句指令的位置。Switch语句的分支部分被begin和end包围，读入begin之后，会首先判断下一个sym是否是case，default或end，然后会在一个循环中不断判断，直到读出end跳出循环，在这个循环中完成对每个case和default后面语句的分析。其中在循环的开始要判断switchtab是否已满，必要时会追加空间，然后在内部的另一个循环中解析case，default之后的语句，循环的条件是读入的sym不是case，default或end，因为语句都应当在这三个关键词之间。为了实现default的作用（在没有匹配时的默认选项，它的跳转指令放在所有case的跳转之后），default的信息都不会放入switchtab里，而是用特别的de\_break,cx\_br分别记录break的存在和其位置，再有一个cx2记录default后语句的起始指令位置。在生成上述的对应表之后，就可以生成相应的跳转指令。这里用的跳转指令是新加入的JET，当栈顶的值等于l时跳转到a对应的指令位置，而对default对应的跳转指令会放在最后作为默认选项。然后在分析break语句的相关信息，依次判断每个case和default后是否有break，有的话就把对应的跳转指令的a值置成default跳转指令后一个指令的地址。

**3.其他特性：**

（1）对于错误处理体系的修改：

本程序中，错误处理的关键在test函数。因此，整个错误处理体系是围绕test函数的功能而搭建的。在解释test函数的基本作用之前，需要先理清两个概念：Follow集合和KeyWord(关键字)集合。Follow集合的概念和课程中讲解的完全相同，这里不再展开。而KeyWord集合则是test函数的错误处理功能不会出错的关键，也是test函数适应各个具体场景、实现特定产生式的需求的关键。这个集合是Follow集合的一个扩充，是在某一个特定产生式下，test函数承认的最大的文法符号集合。程序中的每一个函数（代表着它们对应的产生式）都有两个形参fsys和ksys，这就是这个产生式所决定的Follow集合和KeyWord集合。test函数正是与这两个形参和在产生式对应的函数中按照需求人为设定的符号集合相配合，判断在当前产生式的环境下，哪些文法符号是错误的输入，在输入错误的情况下又应该跳过多少符号，在什么符号处停下分析。具体来说，test函数需要输入两个集合s1、s2作为判断错误输入和判断停止分析的符号集合。test函数会判断当前正在分析的符号sym属不属于s1，如果不属于s1，则说明sym是错误输入，那么就需要跳过错误的部分继续进行之后无错部分的分析。而什么时候停止跳过符号是由s2决定的，test函数会跳过符号直到当前符号sym属于s2为止。

因此，对于两个输入集合s1和s2的要求是不一样的。如果想要使test函数发挥正确的作用，既能准确找到错误，又不会跳过过多代码，就需要s1是一个允许的正确符号的精确集合，而s2是一个包括所有后续可能的停止分析处的符号集合。所以，程序员一般需要在使用test的时候根据当时的产生式单独设置s1（这里可能会用到fsys），再令s2是s1、fsys、ksys的并（可能根据需求有所改变）。

而调用test函数的非终结符可能在不同的产生式里，每一个产生式中该非终结符的后续符号都可能是不同的，所以要通过传参的形式，由产生式左部的非终结符产生右部非终结符的后续符号集合。针对s1和s2两个形参一精确、一宽泛的要求，非终结符之间也传递两个形参。要求精确的Follow集合严格按照产生式中非终结符后面的符号来产生；而要求宽泛的关键字集合是三个集合的并：非终结符的Follow集、非终结符所在的产生式的左部非终结符的关键字集和一个因各个产生式具体的错误处理方法而异的集合。修改文法符号以后，需要考虑产生式的错误处理方法、Follow集合和关键字集合需不需要修改，用test函数做的错误检查需不需要修改。

（2）增加了当前作用域内冲突声明的检测。在当前作用域内已声明过的非过程名字再次声明时会报错（No.31），在当前作用域及其先祖作用域内已声明过的过程名字再次声明时会报错（No.31）。扩大过程名字的检测范围是为了更好地支持递归调用，禁止声明父过程的名字，防止声明错误在编译阶段检测不出，导致在运行时产生难以预料的错误的情况发生。为此，修改了函数position的实现，增加参数tx\_beg，记录当前查找的作用域的开始位置。该参数只有在检测冲突声明时才置非0，过去已存在的调用均为0。为了传递正确的tx\_beg，又添加了全局变量tx\_b，它在每次声明过程前修改，并且进入过程体后由block\_dx记录它的值，因为一个过程体内可能声明多个子过程，每个过程都要更改tx\_b的值。所以要提前记录当前域的tx\_b值，以使与过程声明间杂的变量声明可以拿到正确的tx\_b值。

（3）扩展了set.c中set集合的操作。增加了查重功能，对于重复的符号在集合合并、扩大时只添加一个，减少查找特定符号的时间。

**实验分工说明**：第9,10题由邵维科同学完成，第11,12题由李展鹏同学完成。