**PL0语言基础设计报告**

By 邵维科，李展鹏，于敬禹

**1.实验内容解说：**首先是（1）词法分析正规式：

*digit*  -> [0-9]

*number* ->

*letter* -> [A-Za-z]

*identifier* -> *letter*

*rel\_logicop*  -> < | > | <> | <= | >= | := | && | ‘||’| ! | [ | ]

*op*  -> + | - | \* | / | , | . | ; | ( | ) | = | &

*word*  -> begin | const | call | do | end | if |else | odd | procedure |while | for |var | then

上面是词法分析的正规式，覆盖了可能出现的合法的单词和符。

（2）下面描述语言的文法：

program ::=

body**.**

body ::=

“const” { identifier “=”number“,”}<identifier “=” number “;”>

| “var”{identifier[ {“[”expression

“]”}],}<identifier[{“[”expression“]”}]“;”>

|“procedure”identifier{“(”{(identifier{“[”expression“]”}|“&”identifier)“,”}{(identifier{“[”expression“]”}|“&”identifier)}“)”}“;” body “;”

| stmt\_mul

stmt\_mul ::=

{stmt “;”}

stmt ::=

identifier[{“[”number“]”}] “:=” expression

|identifier{“(”{(identifier{“[”expression“]”}|“&”identifier)“,”}{(identifier{“[”expression“]”}|“&”identifier)}“)”}

| “begin” stmt\_mul “end”

| “if” <or\_condition> then stmt “else” stmt

| “while” <or\_condition> “do” stmt

|“for”“(“ ((expression“;”)|“;”)[expression] “;”[expression] “)”stmt

| “return”expression

or\_condition:=

and\_condition {“||” and\_condition}

and\_condition:=

condition {“&&” condition}

condition ::=

“odd” expression

| expression (“=”| “<>”| “<”| “>”|

“<=”| “>=”)expression

| expression

expression ::=

term {(“+”| “-“) term}

term ::=

factor {(“\*”| “/”) term}

factor ::=

identifier[{“[”number“]”}]

| number

| “-”factor

| “!”factor

| “(”or\_condition“)”

上面使用BNF来描述的PL0的文法。

**2.各小题设计：**

**第二题：**

（1）（2）对逻辑算符的操作首先在getsym函数中增加识别这三个算符的代码，然后对这三个算符按照优先级分别添加了产生式和对应的函数。由于底层的汇编指令是用C语言实现的，所以对逻辑值加入算术值的运算当中有很好的支持。因此，除了编译时计算的常量表达式以外（这部分的支持会在之后加入），其他算术逻辑操作实际上已经通行于逻辑值和算术值。在底层，添加了逻辑与或非的指令。

（3）最后，在短路计算中，添加了全局变量int \*list[2]用来存放指向truelist和falselist的指针，而这两个列表是动态分配的。采取这样的存储和信息交换方式的原因是truelist和falselist的内容是局部生成的，在递归调用的不同层级上需要同时存储两个list的不同信息，从而list的存储空间应当通过动态分配获得。另一方面，在两个函数之间只会传递一组list信息，因此承载list指针的空间可以设为全局变量。在具体实现上，与课本不同的是，只有&&和||向上传递list，而！直接定位在后一个语句。这是逻辑量和算术量的运算可以相互混杂所导致的。！算符作为左侧单目算符，其优先级仅次于括号，需要放置于产生式factor处。但是由于两种运算可以互通，！算符不能从factor中剥离并另外复制一套并行的产生式，所以当递归下降分析进行到！算符所属的表达式分析结束需要返回的时候，紧接其后的运算很可能是算术运算，这样跳转语句就应该定位在这个算术运算上面。因此，最多到！运算符以后，list信息的传递就应该结束了。通过更仔细的分析削去无用的list传递，可以说明，在！运算符之前，（or\_condition）计算完成时list信息的传递就已经结束，因此最后的回填动作就位于这个产生式处。

最后，阐述一下错误处理在本小题中的具体体现。在本小题中，基本不需要具体实现错误处理，因为在前后的产生式中已经对输入流中可能出现的乱码做了定位（即test(ksys,ksys,19)）并且，本题的产生式结构较为简单，关键符号缺失的影响可以简单地视为输入流中产生了缺失符号后那段残留符号的乱码，从而归并到前面的情况中给予处理。

**第三题：**

数组类型的添加思路和添加逻辑运算符时一样，需要综合前端到后端的各关键点进行迭代式的推演和细化。实现的目标是多维数组的声明和调用，其中声明中的各维度需要显示地使用常数表达式，调用则无限制。为了实现这个愈发特性，在前端使用添加控制参数的方式实现了常数表达式。控制参数CONST\_CHECK置CONST\_EXPR时，函数的返回值即为常数表达式的值，置UNCONST\_EXPR时，函数产生相应的指令。实际上，两者共享同一文法，仅是翻译方案有所不同。但是，在扩展程序以后（完成了第二题）控制参数的值在程序中变化的时间与维持不变的时间倒转，从而更适合以全局变量的形式传递信息，这点会在之后的版本修改。另一方面，在后端，在基本数据类型中添加了数组类型，在符号表中添加了动态分配空间的表项int \*ptr（该项在之后的函数传参中也有应用）用以储存数组的维数和各维度的大小。同时，为了动态指定装载和存储的地址，添加了指令LODI和STOI。

在数组的实现中，错误输入的处理是一个重点。由于在当前版本中，下标运算符还不作为运算符处理，而是作为一个文法结构来处理，所以需要对下标运算符的作用对象错误单独作出提示，并且还要对各种结构数目的超上限给出提示。这里就很好地体现了我们改进的错误处理系统的灵活性和准确性。这个改进（或者说改正）在之后有专门的阐述，这里我们仅说，对于错误处理系统中各参量概念的澄清和准确划分使得根据产生式的局部形态控制输入流的行止成为可能。

**第四题：**

（1）利用加入数组时增加的符号表项ptr，记录过程名字附属的参数的信息。由于现在参数只有整型，只需一个域记录个数即可。之后添加地址和过程参数时还需要记录各参数的类型。

（2）增加了声明带参函数的代码。参数在活动记录中正向储存（与C不同），并且在过程头声明时直接加入符号表，其地址直接绑定到调用函数存放实参的地方。因此，越早声明的参数相对偏移（为负）越小，其值要在扫描完所有参数后才能确定。

（3）在声明和调用含参过程的代码中提供了多种错误检测，包括超过参数个数上限，参数过多、过少，参数不能为数组、缺少分隔符等。

（4）删除了call符号，改为直接使用名字调用。

（5）提供对无参函数直接使用函数名的调用和使用调用运算符的调用两种方式，前者与之前的语法兼容。

**第五题：**

（1）在原有的if语句的基础上增加else语句，这需要设计文法使else不会出现悬空现象，但试了书上和几种其他方法后，文法始终无法契合LL文法的要求，在这种情况下，只有退而求其次，将else设计成语句的必要成分，不影响程序将要实现的功能，只是是在书写时有时会有些繁琐，出现不必要的else，其后其实是空语句。

（2）exit语句要求遇到执行这个语句时直接退出程序的执行，为了实现这个功能，保存每次遇到exit时code数组的下标，生成一个跳转代码JMP，跳转到程序体的最后一个code值之后，因为所有exit都要跳到最后，不可能只用一个变量来标记exit语句，为了保存下标不致下标覆盖，将下标设计成整型数组，每个exit对应其中一项。目前的exit语句只能支持直接返回整型常量，以后还会继续增加他的功能。

（3）return语句和exit相似，但是只是退出当前的过程，同样保存code数组下标，生成JMP指令，在block函数的最后对每个JMP指令的跳转地址赋值后，再将下标数组初始化，保证每个return只作用于最靠近它的过程

（4）for语句的设计比较保守，括号中第一项只能是一个赋值语句，第二项是条件语句，最后跟上一个赋值语句。为了方便说明，以a,b,c,d分别表示括号内三个语句和for的循环部分。设计中需要在b的后面设置两个跳转指令，第一个是JPC，在条件为假是退出循环，跳到d的后面，第二个是JMP，无条件跳到d的第一条指令，然后在d的后面设置JMP跳到c的第一条指令,在c的后面设置JMP无条件跳到条件语句b的第一条指令。这样可以实现循环。

**3.其他：**

（1）对于错误处理的修改和理解，

test函数的基本左右是和各个函数的形参（即Follow和关键字集）相配合，判断当前sym是不是属于他第一个形参的符号，在不是的情况下报错并且跳过错误代码，查找属于它第二个形参的符号。所以要想使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，就需要第一个形参精确输入需要查找的符号的集合，第二个形参输入一个把所有后续可能的关键符号都包括在内的集合。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test函数两个形参一精确、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。

为了更好实现错误恢复，基本所有非终结符号对应的函数的形参都从一个调整为两个。同时还增加了允许多参数合并关键字和FOLLOW集的unite\_mul函数和多参数的setinsert\_mul函数。

（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。

（4）修正了原有程序的bug。

**实验分工说明**：实验中第一小题很容易完成，其余四题分工完成，第2,3,4题由邵维科同学完成，第5题由李展鹏同学完成。在实验中的错误检测和恢复方法由邵维科同学做了修改。