

第12组PL0编译器基础部分实验报告

 | 编译原理 | 2017年11月25日

# 小组成员：

林鑫 PB15111616

宋鑫 PB15111640

谢鹏 PB15111636

王嘉男 PB15111678

崔焱 PB15000114

**第12组PL0编译器基础部分实验报告**

**指令集**

一条指令由指令码f，层差l和操作数a构成。

指令码f及对应的功能如下：（pl0.c#1867）

LIT： 将操作数a加载到栈顶，并使栈顶加1；

OPR： 根据操作数a进行相关运算，运算结果保存在栈顶，操作数a及相应的功能如下：

OPR\_NEG： 栈顶的数取反；

OPR\_ADD： 次栈顶与栈顶的数相加，并使栈顶减1；

OPR\_MIN： 次栈顶与栈顶的数相减，并使栈顶减1；

OPR\_MUL： 次栈顶与栈顶的数相乘，并使栈顶减1；

OPR\_DIV： 次栈顶与栈顶的数相除，并使栈顶减1；

OPR\_MOD： 次栈顶与栈顶的数取余，并使栈顶减1；

OPR\_SHL： 次栈顶左移保存在栈顶的位数，并使栈顶减1；

OPR\_SHR： 次栈顶右移保存在栈顶的位数，并使栈顶减1；

OPR\_BIT\_NOT： 栈顶按位非运算；

OPR\_BIT\_AND： 次栈顶与栈顶按位与运算，并使栈顶减1；

OPR\_BIT\_XOR： 次栈顶与栈顶按位异或运算，并使栈顶减1；

OPR\_BIT\_OR： 次栈顶与栈顶按位或运算，并使栈顶减1；

OPR\_ODD： （弃用）栈顶为奇数将栈顶置1；

OPR\_EQU： （弃用）次栈顶等于栈顶将栈顶置1，并使栈顶减1；

OPR\_NEQ： （弃用）次栈顶不等于栈顶将栈顶置1，并使栈顶减1；

OPR\_LES： （弃用）次栈顶小于栈顶将栈顶置1，并使栈顶减1；

OPR\_LEQ： （弃用）次栈顶小于或等于栈顶将栈顶置1，并使栈顶减1；

OPR\_GTR： （弃用）次栈顶大于栈顶将栈顶置1，并使栈顶减1；

OPR\_GEQ： （弃用）次栈顶大于或等于栈顶将栈顶置1，并使栈顶减1；

OPR\_LOGIC\_NOT：（弃用）栈顶逻辑或运算；

OPR\_LOGIC\_AND：（弃用）次栈顶与栈顶逻辑与运算，并使栈顶减1；

OPR\_LOGIC\_OR： （弃用）次栈顶与栈顶逻辑或运算，并使栈顶减1；

OPR\_RET： （弃用）过程返回，清除过程调用建立的栈帧；

LEA： 根据层差l与偏移地址a寻址变量在栈中的绝对地址保存到栈顶，并使栈顶加1；

LOD： 根据层差l与偏移地址a寻址变量，将变量的值加载到栈顶，并使栈顶加1；

STO： 根据层差l与偏移地址a寻址变量，将栈顶的值保存到变量对应的地址处，栈顶不变；

CAL： 调用过程，建立函数栈帧，根据层差l建立函数的访问链、调用链，保存函数返回地址，并使pc指向函数的入口绝对地址a；

INT： 将栈顶增加（或减少）a表示的大小；

JMP： 无条件跳转，将pc指向相对当前指令偏移为a的指令地址；

JPC： （弃用）如果栈顶为0则跳转，将pc指向相对当前指令偏移为a的指令地址，并使栈顶减1；

JZ： 如果栈顶为0则跳转，将pc指向相对当前指令偏移为a的指令地址，并使栈顶减1；

JNZ： 如果栈顶非0则跳转，将pc指向相对当前指令偏移为a的指令地址，并使栈顶减1；

JE ： 如果次栈顶等于栈顶则跳转，将pc指向相对当前指令偏移为a的指令地址，并使栈顶减2；

JNE： 如果次栈顶不等于栈顶则跳转，将pc指向相对当前指令偏移为a的指令地址，并使栈顶减2；

JG ： 如果次栈顶大于栈顶则跳转，将pc指向相对当前指令偏移为a的指令地址，并使栈顶减2；

JGE： 如果次栈顶大于或等于栈顶则跳转，将pc指向相对当前指令偏移为a的指令地址，并使栈顶减2；

JL ： 如果次栈顶小于栈顶则跳转，将pc指向相对当前指令偏移为a的指令地址，并使栈顶减2；

JLE： 如果次栈顶小于或等于栈顶则跳转，将pc指向相对当前指令偏移为a的指令地址，并使栈顶减2；

JOD： 如果栈顶为奇数则跳转，将pc指向相对当前指令偏移为a的指令地址，并使栈顶减1；

RET： 过程返回，清除过程调用是建立的栈帧，恢复调用者的栈帧基址b，恢复栈顶top，将pc指向返回地址指向的指令地址；

OUT： 输出指令，输出元素个数由操作数a指定，将栈顶的a个元素输出，并使栈顶减a；

ALOD：将栈顶保存的绝对地址处的内容加载到栈顶，栈顶不变；

ASTO：将栈顶内容保存到次栈顶保存的绝对地址处，将栈顶内容转存到次栈顶，栈顶减1。

词法

合法词法记号对应如下：（pl0.h#24）

基本元素：

1. SYM\_IDENTIFER：([a-z A-Z])(\_ | [a-z A-Z] | [0-9])\*，即由字母开始的字母、数字或下划线构成的序列
2. SYM\_NUMBER： ([0-9])+，即由数字构成的序列

运算类：

1. SYM\_INC： ++
2. SYM\_DEC： --
3. SYM\_PLUS： +
4. SYM\_MINUS： -
5. SYM\_TIMES： \*
6. SYM\_SLASH： /
7. SYM\_MOD： %
8. SYM\_BIT\_AND： &
9. SYM\_BIT\_OR： |
10. SYM\_BIT\_NOT： ~
11. SYM\_BIT\_XOR： ^
12. SYM\_SHL： <<
13. SYM\_SHR： >>

赋值相关：

1. SYM\_ASSIGN： =
2. SYM\_PLUS\_ASSIGN： +=
3. SYM\_MINUS\_ASSIGN： -=
4. SYM\_TIMES\_ASSIGN： \*=
5. SYM\_SLASH\_ASSIGN： /=
6. SYM\_MOD\_ASSIGN： %=
7. SYM\_BIT\_AND\_ASSIGN： &=
8. SYM\_BIT\_OR\_ASSIGN： |=
9. SYM\_BIT\_XOR\_ASSIGN： ^=
10. SYM\_SHL\_ASSIGN： <<=
11. SYM\_SHR\_ASSIGN： >>=

关系运算：

1. SYM\_EQU： ==
2. SYM\_NEQ： !=
3. SYM\_LES： <
4. SYM\_LEQ： <=
5. SYM\_GTR： >
6. SYM\_GEQ： >=

逻辑运算：

1. SYM\_LOGIC\_AND： &&
2. SYM\_LOGIC\_OR： ||
3. SYM\_LOGIC\_NOT： !

其它：

1. SYM\_LPAREN： (
2. SYM\_RPAREN： )
3. SYM\_LINDEX： [
4. SYM\_RINDEX： ]
5. SYM\_COMMA： ,
6. SYM\_SEMICOLON： ;
7. SYM\_QUES： ?
8. SYM\_COLON： :
9. SYM\_BEGIN： {
10. SYM\_END： }
11. SYM\_PERIOD： $

关键字：

1. SYM\_ODD： odd
2. SYM\_IF： if
3. SYM\_ELIF： elif
4. SYM\_ELSE： else
5. SYM\_FOR： for
6. SYM\_WHILE： while
7. SYM\_DO： do
8. SYM\_BREAK： break
9. SYM\_CONTINUE： continue
10. SYM\_CALL： call（弃用）
11. SYM\_RETURN： return
12. SYM\_EXIT： exit
13. SYM\_CONST： const
14. SYM\_VAR： var
15. SYM\_PROCEDURE： procedure
16. SYM\_PRINT： print

词法分析器的实现（pl0.c#101）：

当遇到“/\*”时，自动忽略接下来的字符，直到遇到”\*/”;当遇到“//”时，忽略当前行，读取下一行代码。从源程序中去除注释并不断读入字符，读入的字符分为3类，即字母或下划线、数字和其它符号。如果第一个字符为字母，可以判断将读入的符号为标识符或关键字，读入下一个字符，如果读入的字符为字母、数字或下划线则继续读入字符，否则停止读入，将读入的字符序列与关键字列表进行匹配，如果序列与某一关键字匹配，则读入的符号为该关键字，否则读入的符号为标识符，并将该标识符的内容由变量id带回；如果第一个字符是数字，可以判断将读入的符号是一个整数，继续读入下一个字符直到读入的字符不是数字为止，读入每一个数字时更新已读入的整数的内容，停止读入字符时判断读入的整数是否超过最大合法整数，如果不是错误处理，否则返回符号为整数，并将整数内容由num带回；其它情况下，将读入的符号是一个操作符，对于有歧义的操作符，如<、<=、<<、<<=，+、++、+=等，每匹配一个字符就继续读入下一个字符，直到匹配最长的合法操作符，将该操作返回；否则，对于第一个字符无歧义的操作符，如(，)，[，]等，将读入的字符与无歧义符号构成的列表进行匹配，如果与某一个符号匹配，将该符号返回，否则可以判断读入的字符是一个非法字符，进行错误处理；

**语法分析**

program → {block} “$”

block → const-declaration

| var-declaration

|proc-declaration

|array-declaration

|statement

constdeclaration → “const” ident "=" number {"," ident "=" number} “;"

var-declaration → "var" ident {"," ident} “;"

proc-declaration → "procedure" ident "("{var ident{“,” var ident }}”)” “{”block”}”

statement → |{statement};

| ident”=”expression;

|[“++”|”--”]ident;

| "{" statement { ";" statement } "}"

| "if" condition statement [{“elif" statement }“else" statement]

| "while" “(”condition”)” statement

|”for””(”{ expression }”;”{ expression }”;””)””{”statement”}”

| “print” ident;

| return {expression};

|exit;

|continue;

|break;

condition → "odd" expression

| expression rel-op expression.

rel-op → "=="|“!="|"<"|"<="|">"|">=“

expression → [ "+"|"-"] term { ("+"|"-") term}.

term → factor {("\*"|"/") factor}

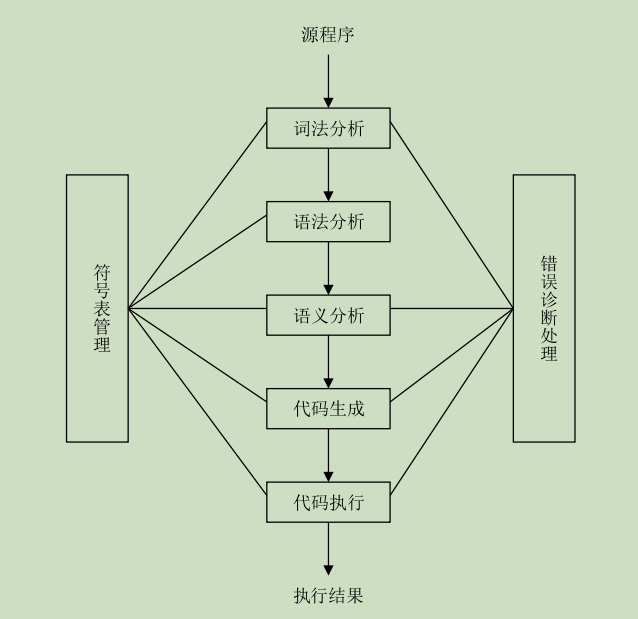
factor → ident | number | "(" expression ")“.

number → digit {digit}.

ident → [letter|’\_’]{letter | digit|’\_’}.

digit → "0" | "1" | "2" | "3" | "4" | "5" | "6" | "7" | "8" | "9“.

Letter → "a" | "b" | … | "y" | "z" | "A" | "B" | ... | "Y" | "Z".



由于PL/0编译程序采用一趟扫描方法，所以语法分析过程BLOCK是整个编译过程的核心。上图是编译程序的总体流程。在流程图中可以看出，主程序置初值后先调用读单词过程GETSYM取一个单词，然后再调用语法分析过程BLOCK，直到遇源程序的结束符"$"为止。语法分析过程BLOCK是整个编译过程的核心，是指开始由主程序调用GETSYM取一个单词，再调用语法分析过程BLOCK，BLOCK由当前单词根据语法规则再调用其它过程，如说明处理、代码生成或出错处理等过程进行分析，当分析完一个单词后，BLOCK再调用GETSYM取下一个单词，一直重复到当前单词为结束符"$"表明源程序已分析结束。若未取到结束符"$"，而源程序已没有输入符号，这时表明源程序有错误，无法再继续分析。

语句

语句在statement函数（pl0.c#1436）中实现，语句合法的开始符号由statbegsys定义，即SYM\_INC，SYM\_DEC，SYM\_IDENTIFIER，SYM\_PRINT，SYM\_RETURN，SYM\_EXIT，SYM\_CONTINUE，SYM\_BREAK，SYM\_IF， SYM\_WHILE，SYM\_DO（未实现），SYM\_FOR，SYM\_BEGIN，statement函数根据不同的开始符号进行相应的处理。

由于处理break和continue执行时需要知道跳转到的指令地址，因此statement函数导游另个codelist：nextlist用于存放break语句对应的指令地址，需要用循环或switch（未实现）的下一条指令地址回填；looplist用于存放continue语句对应的指令地址，需要用循环的第一条指令地址回填。对于不被switch或循环包围的statement，nextlist与looplist为NULL。

如果读入的开始符号为SYM\_INC、SYM\_DEC、SYM\_IDENTIFIER，语句为表达式语句，直接调用expression函数进行处理，如果结果为逻辑真值在expression中将逻辑真值转化为算术运算结果，最后生成一条INT指令将栈顶清除；如果读入开始符号为SYM\_PRINT，调用args\_list函数处理待输出的参数列表，然后生成OUT指令输出栈顶的count个数，输出个数count有args\_list取回，执行OUT指令时将栈顶count个数据清除；如果读入开始符号为SYM\_EXIT，生成JMP指令，将pc指向绝对地址0处，执行下一条指令时直接终止执行；如果开始符号为SYM\_CONTINUE，如果looplist为NULL，说明statement不被循环包围，continue语句出错，否则生成JMP跳转指令，指令地址存入looplist；同理。如果开始符号为SYM\_BREAK，若nextlist为NULL，说明statement不被switch或循环包围，break语句出错，否则，生成JMP跳转指令，指令地址存入nextlist；如果开始符号为SYM\_RETURN，根据全局变量curr\_proc追踪过程参数个数，如果return后由表达式，根据表达式生成指令，否则生成（LIT，0，0），生成STO指令将返回值存入对应地址，再生成RET指令从过程调用返回，SYM\_INC、SYM\_DEC、SYM\_IDENTIFIER、SYM\_PRINT、SYM\_EXIT、SYM\_RETURN、SYM\_BREAK、SYM\_CONTINUE语句必须以SYM\_SEMICOLON结束，否则出错；如果读入开始符号为SYM\_BEGIN，调用multi\_statement处理多条语句，multi\_statement函数（pl0.c#1703）通过循环调用statement函数，处理多条语句，直到读入的符号不是语句的合法开始符号为止，multi\_statement返回之后，下一个符号应为SYM\_END，否则出错。

如果读入符号为SYM\_IF，调用condition函数（pl0.c#1411）读入条件，调用statement处理条件为真语句块，用语句块的第一条指令地址回填truelist，生成一条JMP指令跳转到if语句块的下一条语句，指令地址存入临时变量nextlist；如果下一个符号为SYM\_ELIF，处理方式与SYM\_IF相似，还需用条件的第一条指令地址回填上一个SYM\_IF/SYM\_ELIF条件的falselist，并继续读入符号，直到读入符号不是SYM\_ELIF；如果符号是SYM\_ELSE，调用statement，用语句块的第一条执行地址回填最后一个条件的falselist，用if语句块的下一条指令地址回填nextlist；如果符号不是SYM\_ELSE，if语句块结束的下一条指令地址回填最后一个条件的falselist和nextlist。

如果读入符号为SYM\_WHILE，调用condition函数读入条件，调用带nextlist和looplist的statement函数处理循环体，最后生成无条件条状指令回到条件的第一条指令进入下一次循环，用循环体的第一条指令地址回填条件的truelist，用while语句块的下一条指令地址回填条件的falselist和nextlist，用条件的第一条指令回填looplist；

如果读入符号为SYM\_FOR，先调用expression函数处理初始化表达式，读入SYM\_COMMA时继续调用expression直到读入SYM\_SEMICOLON；调用expression函数处理循环条件，并调用num2logic生成truelist和falselist；按照读入的顺序，调用expression生成循环控制变量调整部分的表达式，记录该部分指令的开始地址start与结束地址end，然后调用带nextlist和looplist的statement函数处理循环体。完成所有指令的生成之后，需要进行指令的移动，由于nextlist与looplist中的指令会移动，而truelist与falselist中的指令不会移动，因此在指令全部移动结束后回填truelist和falselist，在循环体指令移动前回填nextlist和looplist，先将start和end之间的指令依次移动都循环体指令之后，回填nextlist和looplist，由于之后循环体指令和循环变量调整指令同步移动，不会改变彼此之间的相对偏移，在将循环体及之后的全部指令移动到条件指令之后，以最终的地址回填truelist和falselist，在指令的最后生成一条无条件跳转指令，回到条件的第一条指令，用于下一次进入循环，至此，for循环的所有指令均已生成并完成调整。

数组的定义在vardeclaration中实现。若ident后接的的字符不是”[”，则其为变量，送入符号表，否则为数组。数组最高维数为10维,dim初始化为0。对于每出现一次”[”代表数组的维数增加一，即将dim加1。若“[”后接的为数字num，则将该维度数组允许的最大值设为num-1；若接的是标识符，则在符号表中查找，只有当其为符号表中存在的常量时，将常量的值减一作为该维度数组的最大值，其它任何情况都报错。在处理完所有的“[”后如果接下来不是“]”则报错。至此数组的维数以及各维度元素数目已经确定。数组的维数、各维度元素上界、数目都会存进arraytable中编号为ax处。接下来根据数组元素总数，为其分配存储空间，将dx增大相应的数目。ax也是该数组在符号表中的索引。

**条件短路运算**

条件短路运算依靠条件跳转指令实现，因此弃用OPR指令中所有关系运算和逻辑运算指令，使用条件跳转指令JZ、JNZ、JE、JNE、JG、JGE、JL、JLE、JOD代替，在处理关系运算的函数rel\_expr和逻辑运算的函数logic\_and\_expr、logic\_or\_expr函数接口中添加truelist和falselist，分别用于存放跳转到真出口和假出口的跳转指令地址，在确定真出口和假出口之后，用出口指令地址回填跳转指令，由于处理语法的函数系列是递归调用的，所可能出现在下层的函数不经过任何实际操作而转到上层函数的情况，因此在处理语法的所有函数中添加truelist和falselist用于传递真出口链和假出口链。

truelist和falselist变量类型为codelist，codelist直接使用symset实现，修改symset上的操作为createlist、destroylist、isempty、inlist、insertlist、deletelist、unitelist使其适应新的使用情况。backpatch函数（pl0.c#440）用于回填codelist，codelist中存放的指令地址都是跳转指令，backpatch用跳转的目标地址num填补跳转指令的跳转地址a，并清空codelist，由于跳转指令都是跳转到相对地址，实际上backpatch用num与每条指令的cx的差值回填跳转地址a。

在rel\_expr中，每一条关系运算表达式被翻译成两条条件跳转指令（JX，0，true\_exit）（X为相关条件），（JMP，0，false\_exit），并分别将地址填入truelist和falselist。由于编译器实现中允许逻辑真和逻辑假分别作为1和0参与算术运算，也允许算术运算结果根据是否为0作为逻辑真值应用，因此num2logic和logic2num函数进行算术结果与逻辑真值的转换。逻辑真值表现为truelist和falselist，算术结果表现为栈顶的内容，num2logic（pl0.c#469）根据栈顶的内容生成填充truelist和falselist，当truelist为空时，表示用栈顶内容作为真值，生成两条指令（JZ，0，false\_exit）和（JMP，0，true\_exit），logic2num（pl0.c#456）根据truelist和falselist在栈顶添加内容，当truelist不空时，表示用逻辑真值作为算术运算参数，生成3条指令（LIT，0，1）回填truelist，（JMP，0，2）跳过下一条指令，（LIT，0，0）回填falselist。对于非逻辑运算且非关系运算的函数，如果函数实际上没有操作，如在polyn函数中没有读入+/-，就直接将truelist和falselist向调用者提交，否则，在需要算术运算的场合执行logic2num指令，在需要逻辑运算的场合如logic\_and\_expr执行num2logic，进行转换（如果需要的话，否则num2logic及logic2num不做任何操作）。

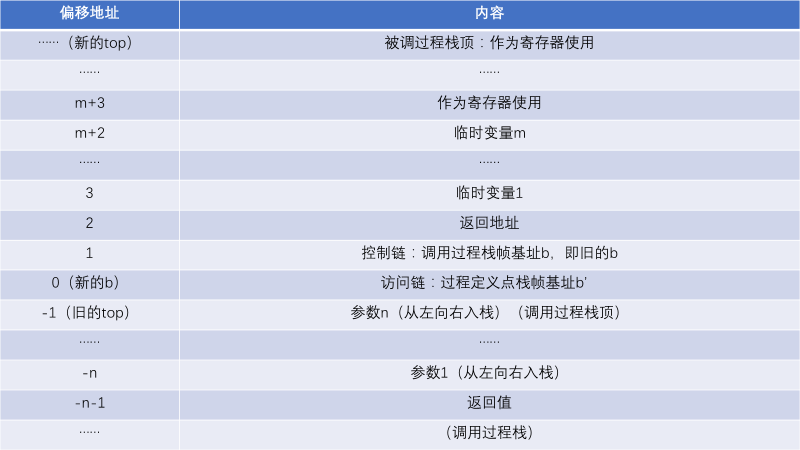
条件短路运算在logic\_and\_expr函数、logic\_or\_expr函数与factor函数中对SYM\_LOGIC\_NOT处理的部分实现。在logic\_and\_expr（pl0.c#1182）中，如果表达式真值为假则跳转到假出口，否则继续进行下一个表达式的处理，logic\_and\_expr读入SYM\_LOGIC\_AND时将表达式返回的sub\_falselist内容合并到falselisst中，用下一个表达式的第一条指令地址回填上一个表达式返回的sub\_truelist，将最后一个表达式的sub\_truelist合并到truelist。在logic\_or\_expr（pl0.c#1219）中，如果表达式真值为真，直接跳转到真出口，否则处理下一个表达式，logic\_or\_expr读入SYM\_LOGIC\_OR时将表达式返回的sub\_truelist合并到truelist中，用下一个表达式的第一条指令地址回填上一个表达式返回的sub\_falselist，将最后一个表达式的sub\_falselist合并到falselist中。对于逻辑非，在factor函数中读入SYM\_LOGIC\_NOT时（pl0.c#923），将下层表达式返回的sub\_falselist合并到truelist中，将sub\_truelist合并到falselist中，实现条件反转。

由于rel\_expr生成的指令一般为（JX，0，true\_exit），（JMP，0，false\_exit）形式，对于逻辑与关系和许多条件、循环语句而言，会产生大量诸如（JX，0，2），（JMP，0，false\_exit）形式的指令，为删除冗余的跳转指令，进行条件反转，将指令转换成（JNX，0，false\_exit）的形式，表达式为真的情形直接往下执行，删除不必要的跳转，conditionreverse函数（pl0.c#483）完成这一操作，该函数判断上两条指令是否满足上述形式，如果满足，删除上一条指令，将其从falselist中删除，将上第二条指令的条件反转（从JX变为JNX，如JZ变为JNZ，JG变为JLE等），添加到falselist中，实现条件反转。而对于逻辑或关系而言，会产生诸如（JX，0，false\_exit），（JMP，0，1）形式的指令，第二条跳转指令为多余的跳转指令，需要检查上一条指令是否满足上述形式，如果满足，就删除该指令，并将其从falselist中删除，以去除冗余跳转。

**过程定义与调用**

过程的定义在block函数中实现（pl0.c#1717），过程调用在factor函数中实现（pl0.c#847）。

过程调用时栈的组织结构如下：



被调过程栈帧位于调用过程栈帧之上，栈帧偏移0、1、2的内容由调用指令CAL生成，偏移3及以上的内容由被调过程生成，偏移-1及以下的内容由调用过程生成。变量b的新内容为被调过程栈帧基址，即偏移为0处的绝对地址，b原来的内容作为控制链保存在偏移为1处，控制链通过CAL指令的层差l追踪过程定义点的基址保存在偏移为0处，过程返回后执行的指令地址保存在偏移地址为2处。被调过程执行结束后，由指令显式将栈顶内容保存到偏移为-n-1的返回值处，执行RET指令由被调过程清除栈帧中偏移为0及以上的内容，恢复b和top，返回调用过程后由指令显式清除参数，将栈顶指针top指向返回值处。

过程参数使用一个参数表proctable存储（pl0.h#245），proctable存储一个过程的参数个数argc和每个参数类型type，扩展符号表项，添加元素index用于定位过程的proctable表表项或数组的arraytable表表项。在读入一个新的过程名时，将过程名添加到符号表table中，同时在参数表proctable中添加一个新的表项，表项内容在处理过程参数时进行更新。

过程定义在block函数中实现，将过程体作为一个block进行处理，整个程序也作为一个block处理。在block函数（pl0.c#1777）中读入一个SYM\_PROCEDURE（procedure）关键字和过程名之后，将过程名添加符号表table，在参数表proctable中添加表项，将level增加1，保存相关全局变量，调用args\_decl函数处理过程参数声明。args\_decl函数(pl0.c#644)将全局变量dx置为0（旧值已保存）已记录过程中的栈分配情况，维护一个已处理参数个数的计数器count，count初始值为0，args\_decl读入一个左括号之后如果读入一个右括号，说明过程没有参数，否则调用函数argdeclaration处理一个参数声明，并返回参数的类型，计数器count增加1，并在proctable表项中type数组中保留参数类型，如果读入下一个符号是逗号，继续处理参数声明，直到读入符号为右括号为止。argdeclaration函数（pl0.c#629）中读入一个SYM\_VAR，直接调用函数vardeclaration（pl0.c#559）处理变量声明（vardeclaration函数修改后可以同时处理变量声明与数组声明），将过程参数声明与普通变量声明进行相同的处理，返回符号表table的最后一个表项类型作为当前参数的类型返回。args\_decl函数读入所有参数声明之后，将count保存到proctable表项的argc，并根据count和dx修改符号表table最后count个元素，即过程参数的偏移地址，使其偏移地址由最后一个参数为-1向前依次递减，与上图中栈内的参数偏移匹配。

过程参数定义之后，读入一个SYM\_BEGIN左大括号，将过程体作为一个block，直接调用block函数处理过程体，将当前过程在proctable中的index保存在全局变量curr\_proc中方便调用。block函数（pl0.c#1717）要求过程中所有的相关常量、变量、数组、过程在block的开头声明，用一个循环处理所有声明，循环中调用constdeclaration处理常量声明，vardeclaration处理变量或数组声明，使用相同的方式处理过程声明。所有声明结束后，调用multi\_statement函数处理过程体，过程体作为多句statement语句处理。multi\_statement函数（pl0.c#1703）处理多句statement语句，直至读入一个不能作为statement语句开始的符号。multi\_statement返回时，通常下一个符号为SYM\_END（右大括号）指示过程体结束。block开始时在过程入口地址处生成一条跳转指令，跳转到过程体的第一条指令，过程体的第一条指令为一条INT指令，将栈顶指针增加一定大小，为过程栈帧中的访问链、控制链、返回地址以及局部变量统一分配栈空间，之后才是过程体的第一条语句所生成的指令。过程定义在block函数返回，即过程体结束之后自动生成返回指令序列，包括一条（LIT，0，0）指令，将默认返回值0加载到栈顶，一条STO指令，将默认返回值保存到返回值地址处，以及一条RET指令。自动生成的返回指令序列用于确保在过程体内没有显式返回语句时过程仍能正确返回，如果过程体中有显式返回语句，则在自动生成的返回语句执行之前，控制即交还给调用者，自动生成的返回语句不会执行，造成副作用。过程定义的block返回之后，将会将之前保存的全局变量还原，进行上层block的处理。

过程执行中遇到return语句，如果return关键字之后有一个表达式，则会针对该表达式生成相应指令，指令执行结束后将结果留在栈顶，否则生成一条（LIT，0，0）指令，将默认返回值0留在栈顶，然后生成STO指令，将栈顶结果保存到栈中返回值预留的地址处，最后生成一条RET指令清除被调过程栈帧。STO的层差l为0，通过保留的curr\_proc寻址proctable表项，根据其argc值确定返回值偏移a。

过程调用在factor函数（pl0.c#847）中实现，读入一个标识符并确认为过程之后，调用proc\_call函数处理函数调用。proc\_call（pl0.c#728）函数生成INT指令为返回值预留空间，调用args\_list函数处理过程实际参数，进行简单的参数检查，生成过程调用指令CAL，在过程调用结束后生成一条INT指令清除栈上的过程实际参数，使栈顶指向过程的返回值。args\_list函数（pl0.c#690）维护一个过程实参个数计数器count，依次读入过程实参中的各个表达式，根据表达式生成指令序列，指令执行结束后会将计算结果留在栈顶，成为栈上对应位置的实际参数，读入右括号时读入所有实际参数，将实际参数个数count返回。由于过程调用目前只实现将简单的变量作为过程参数，因此类型检查只需要将args\_list返回的实际参数个数与proctable中记录的形式参数个数argc进行检查，判断两者是否相同。如果要实现其它类型的数据作为过程参数的参数检查，可以在args\_list中将读入的参数类型与proctabletype数组中对应位置的参数类型进行检查。

过程调用的执行过程如下：调用过程执行（INT，0，1）为返回值预留空间，计算实际参数指令将计算结果依次留在栈上作为实际参数，执行指令CAL先建立访问链、控制链、保存返回地址，将pc指向过程入口，进入被调过程，被调过程执行JMP指令跳转到实际的执行入口，执行INT指令统一为访问链、控制链、返回地址和局部变量分配空间，执行过程体中指令序列，要返回时，计算返回值并留在栈顶（无返回值的返回指令会执行一条（INT，0，1）指令将默认返回值0留在栈顶），执行STO指令将栈顶的返回值保存到之前预留的返回值地址处，执行RET指令将top和b还原以清除被调栈帧，将控制返回调用过程，调用过程执行INT指令将栈顶的参数清空，使栈顶指针指向被调过程的返回值，至此，过程调用结束。