

**编译实践-PL\0编译系统实现**

|  |  |
| --- | --- |
| 姓名： | 姜楠 |
| 专业： | 计算机科学与技术 |
| 学院： | 软件学院 |
| 提交时间： | 2013年12月25日 |

北京航空航天大学·软件学院

**编译实践-PL\0编译系统实现**

1. 实验要求
   1. 以个人为单位进行开发，不得多人合作完成。
   2. 共32个学时。个人无计算机者可以申请上机机时。
   3. 细节要求：

* 输入：符合PL/0文法的源程序（自己要有5个测试用例，包含出错的情况，还要用老师提供的测试用例进行测试）
* 输出：P-Code
* 错误信息：参见教材第316页表14.4。
* P-Code指令集：参见教材第316页表14.5。
* 语法分析部分要求统一使用递归下降子程序法实现。
* 编程语言使用C、C++、C#或Java等。
* 上交材料中不但要包括源代码（含注释）和可执行程序，还应有完整文档。

1. PL/0语言描述

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。
  1. PL/0语言文法的EBNF表示

<程序> ::= <分程序>.

<分程序> ::= [<常量说明部分>][变量说明部分>]{<过程说明部分>}<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> ::= <数字>{<数字>}

<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> ::= procedure<标识符>;

<语句> ::= <赋值语句>|<条件语句>|<当型循环语句>|<过程调用语句>|<读语句>|<写语句>|<复合语句>|<重复语句>|<空>

<赋值语句> ::= <标识符>:=<表达式>

<表达式> ::= [+|-]<项>{<加法运算符><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>[else<语句>]

<当型循环语句> ::= 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

<重复语句> ::= repeat<语句>{;<语句>}until<条件>

<读语句> ::= read'('<标识符>{,<标识符>}')'

<写语句> ::= write'('<标识符>{,<标识符>}')'

<字母> ::= a|b|...|X|Y|Z

<数字> ::= 0|1|2|...|8|9

注意：

数据类型：无符号整数

标识符类型 ：简单变量(var)和常数(const)

数字位数：小于14位

标识符的有效长度：小于10位

过程嵌套：小于3层

* 1. PL/0语言的语法图描述



图1-1 程序语法描述图



图1-2 分程序语法描述图



图1-6 项语法描述图



图1-7 因子语法描述图

1. PL/0编译系统结构



图 1-8 PL/0编译程序和解释执行过程

|  |
| --- |
| PL/0编译程序函数定义层次结构：  pl0  error  getsym  getch  gen  test  block  enter  position  constdeclaration  vardeclaration  listcode  st：atement  expression  term  factor  condition  interpret  base |

下面介绍这些过程（函数）的作用。

|  |  |
| --- | --- |
| pl0 | 主程序 |
| error | 出错处理，打印出错位置和错误代码 |
| getsym | 词法分析，读取一个单词 |
| getch | 取字符 |
| gen | 生成P-code指令，送入目标程序区 |
| test | 测试当前单词符号是否合法 |
| block | 分程序分析处理 |
| enter | 登记符号表 |
| position | 查找标识符在符号表中的位置 |
| constdeclaration | 常量定义处理 |
| vardeclaration | 变量定义处理 |
| listcode | 列出p-code指令清单 |
| statement | 语句部分分析处理 |
| expression | 表达式分析处理 |
| term | 项分析处理 |
| factor | 因子分析处理 |
| condition | 条件分析处理 |
| interpret | P-code解释执行程序 |
| base | 通过静态链求出数据区的基地址 |

1. PL/0编译程序的词法分析

PL/0编译系统中所有的字符，字符串的类型为，如下表格：

|  |  |
| --- | --- |
| 保留字 | begin, end, if,then, else, const, procedure,  var,do,while, call,read, write, repeat, until |
| 算数运算符 | + ,—，\*，/ |
| 比较运算符 | <> , < ,<= , >, >= ,= |
| 赋值符 | := , = |
| 标识符 | 变量名，过程名，常数名 |
| 常数 | 10,25等整数 |
| 界符 | ‘，’，‘.’，‘；’，‘（’，‘）’ |

PL/0的词法分析程序Scanner.getsym()由语法分析程序调用，主要功能为：

* 跳过空格字符。
* 识别单词符号，返回单词类型（按照在Symbol.java中定义的编译系统的字符编号，返回类型码）
* 特别的，对于编译系统的保留字符（例如：const, if, then等）需要查找系统的保留字符表word[]，为了加快查找速度，调用系统的二分搜索法Arrays.binarySearch().
* 另外，如果读取的字符为数字，需要将该字符转换成整数值（调用公式num = 10 \* num + (ch - '0');），再存入符号表的Value区域.

Scanner.getsym()是调用扫描输入的源程序。主要功能如下：

* 优化读取字符效率，每次读取一行源程序，存入缓冲区line，因此设置lineLength为源程序当前行的长度，chCount标志当前正在读取的字符位置
* 采用“单符号先行”技术，在识别完每个符号的类型后，必须再度入下一个字符，以保证下一次再调用getsym()时，curCh保存的是该符号的首字符



图1- 词法分析程序的状态转换图

1. PL/0编译程序的符号表管理

* 符号表结构

|  |
| --- |
| * 符号表中每一条记录所对应的结构：   public class Item {  public static final int constant = 0;  public static final int variable = 1;  public static final int procedure = 2;  String name; //名字  int type; //类型，const var or procedure  int value; //数值，const使用  int level; //所处层，var和procedure使用  int addr; //地址，var和procedure使用  int size; //需要分配的数据区空间，仅procedure使用  } |

|  |
| --- |
| 符号表类SymbolTable中用数组存储符号表，再分配一个指针tablePtr指向当前符号表的末尾。  public class SymbolTable {  //有效的符号表大小  public int tablePtr = 0;  //名字表  public Item[] table = new Item[tableMax];  ... ...  } |

举例：

|  |
| --- |
| PL/0代码样例：  CONST A=35，B=49； VAR C，D，E； PROCEDURE P；  VAR G，X，Y，Z ； |

此时的符号表内容：

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| NAME：A | KIND:CONSTANT | VAL:35 |  |  |
| NAME：B | KIND:CONSTANT | VAL:49 |  |  |
| NAME：C | KIND:VARIABLE | LEVEL:LEV | ADDR:DX |  |
| NAME：D | KIND:VARIABLE | LEVEL:LEV | ADDR:DX+1 |  |
| NAME：E | KIND:VARIABLE | LEVEL:LEV | ADDR:DX+2 |  |
| NAME：P | KIND:PROCEDURE | LEVEL:LEV | ADDR: | SIZE:7 |
| NAME：G | KIND:VARIABLE | LEVEL:LEV+1 | ADDR:DX |  |

* 符号表管理
  + 登记(在符号表中插入一项)

|  |
| --- |
| /\*\*  \* 把某个符号登录到名字表中，从1开始填，0表示不存在该项符号  \* @param sym 要登记到名字表的符号  \* @param k 该符号的类型：const, var ,procedure  \* @param lev 名字所在的层次  \* @param dx 当前应分配的变量的相对地址，注意dx要加一  \*/  public void enter(Symbol sym,int type,int lev, int dx) |

* + 查询

|  |
| --- |
| /\*\*  \* 在名字表中查找某个名字的位置  \*从后往前查， 这样符合嵌套分程序名字定义和作用域的规定  \* @param idt 要查找的名字  \* @return 如果找到则返回名字项的下标，否则返回0  \*/  public int position(String idt) |

1. PL/0编译程序的语法分析



图1- 语法调用关系图

采用不带回溯的递归子程序法，对于语言的文法要求：

1. 该文法必须是非左递归。
2. 文法的非终结符，其规则右部所生成的first集合两两不相交
3. 若文法具有形如，则

递归子程序设计实例

|  |
| --- |
| * <expression>::=[+|-]<term>{(+|-)<项>}   void expression(BitSet fsys, int lev) {  if (symtype == plus || symtype == minus) {  int adop = symtype;  nextsym();  term(nxtlev, lev);  if (adop == minus)  gen(OPR, 0, 1);  } else  term(nxtlev, lev);  //分析{<加法运算符><项>}  while (symtype == plus || symtype == minus) {  int adop = symtype;  nextsym();  term(nxtlev, lev);  gen(OPR, 0, adop);  }  } |

|  |
| --- |
| * <term>::=<factor>{(\*|/)<term>}   void term(BitSet fsys, int lev) {  factor(nxtlev, lev);  //分析{<乘法运算符><因子>}  while (symtype == mul || symtype == div) {  int mop = sym.symtype;  nextsym();  factor(nxtlev, lev);  gen(OPR, 0, mop);  }  } |

|  |
| --- |
| * <factor>::=<ident>|<number>|’(’<experssion>’)’   void factor(BitSet fsys, int lev) {  if (symtype == ident) {  int index = table.position(sym.id);  if (index > 0) {  Item item = table.get(index);  switch (item.type) {  case constant:  gen(LIT, 0, item.value);  break;  case variable:  gen(LOD, lev - item.lev, item.addr);  break;  }  }  nextsym();  } else if (symtype == number) {  gen(LIT, 0, num);  nextsym();  } else if (symtype == lparen) {  nextsym();  expression(nxtlev, lev);  if (symtype == rparen)  nextsym();  } |

1. PL/0编译程序的目标代码结构和代码生成
   * **代码结构**

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。

|  |  |  |
| --- | --- | --- |
| F | L | A |

指令格式

F ：操作码

L ：层次差 （标识符引用层减去定义层）

A ：不同的指令含义不同

表5 P-code 指令的含义

|  |  |
| --- | --- |
| 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(+ - \* /) |
| LOD l, a | 取变量放到数据栈栈顶(相对地址为a,层次差为l) |
| STO l, a | 将数据栈栈顶内容存入变量(相对地址为a,层次差为l) |
| CAL l, a | 调用过程(入口指令地址为a,层次差为l) |
| INT 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 |

|  |
| --- |
| //pcode类的结构  public class Pcode{  //虚拟机代码指令  public int f;  //引用层与声明层的层次差  public int l;  //指令参数  public int a;  }  //存放虚拟机代码的数组  public Pcode[] pcodeArray;  //生成虚拟机代码  public void gen(int f, int l, int a) {  pcodeArray[arrayPtr++] = new Pcode(f, l, a);  } |

* + 代码生成与地址返填

对于if then [else],while do和repeat until语句，要生成跳转指令，故采用地址返填技术。

* + - if-then-else语句的目标代码生成模式：

|  |  |
| --- | --- |
| if <condition> then <statement>[else] | |
|  | <condition> |
|  | JPC addr1 |
|  | <statement> |
| addr1: | [else] |

* + - while-do语句的目标代码生成模式：

|  |  |
| --- | --- |
| while <condition> do <statement> | |
| addr2: | <condition> |
|  | JPC addr3 |
|  | <statement> |
|  | JPC addr2 |
| addr3: |  |

* + - repeat-until语句的目标代码生成模式：

|  |  |
| --- | --- |
| repeat <statement> until <condition> | |
| addr4: | <statement> |
|  | <condition> |
|  | JPC addr4 |

注意：由于OPR指令设计复杂，故进一步解释：

|  |
| --- |
| (1).OPR 0 0 |
| RETUEN  (stack[sp + 1] 🡨 base(L);  sp 🡨 bp - 1;  bp 🡨 stack[sp + 2];  pc 🡨 stack[sp + 3];) |
| (2).OPR 0 1 |
| NEG  (- stack[sp] ) |
| (3).OPR 0 2 |
| ADD  (sp 🡨 sp – 1 ;  stack[sp] 🡨 stack[sp] + stack[sp + 1]) |
| (4).OPR 0 3 |
| SUB  (sp 🡨 sp – 1 ;  stack[sp] 🡨stack[sp] - stack[sp + 1]) |
| (5).OPR 0 4 |
| MUL  (sp 🡨 sp – 1 ;  stack[sp] 🡨 stack[sp] \* stack[sp + 1]) |
| (6).OPR 0 5 |
| DIV  (sp 🡨 sp – 1 ;  stack[sp] 🡨 stack[sp] / stack[sp + 1]) |
| (7).OPR 0 6 |
| ODD  (stack[sp] 🡨 stack % 2) |
| (8).OPR 0 7 |
| MOD  (sp 🡨 sp – 1 ;  stack[sp] 🡨 stack[sp] % stack[sp + 1]) |
| (9).OPR 0 8 |
| EQL  (sp 🡨 sp – 1 ;  stack[sp] 🡨 stack[sp] == stack[sp + 1]) |
| (10).OPR 0 9 |
| NEQ  (sp 🡨 sp – 1 ;  stack[sp] 🡨 stack[sp] != stack[sp + 1]) |
| (11).OPR 0 10 |
| LSS  (sp 🡨 sp – 1 ;  stack[sp] 🡨 stack[sp] < stack[sp + 1]) |
| (12).OPR 0 11 |
| GEQ  (sp 🡨 sp – 1 ;  stack[sp] 🡨 stack[sp] >= stack[sp + 1]) |
| (13).OPR 0 12 |
| GTR  (sp 🡨 sp – 1 ;  stack[sp] 🡨 stack[sp] > stack[sp + 1]) |
| (14).OPR 0 13 |
| LEQ  (sp 🡨 sp – 1 ;  stack[sp] 🡨 stack[sp] <= stack[sp + 1]) |
| (15).OPR 0 14 |
| print (stack[sp]);  sp 🡨 sp – 1; |
| (16).OPR 0 15 |
| print ('\n'); |
| (17).OPR 0 16 |
| scan(stack[sp]);  sp 🡨 sp + 1; |
|  |

1. PL/0编译程序的语法错误处理

8.1错误处理的原则

* + 尽可能准确指出错误位置和错误属性
  + 尽可能进行校正
  + 短语层恢复技术
    - 在进入某个语法单位时，调用TEST函数, 检查当前符号是否属于该语法单位的开始符号集合.
    - 在语法单位分析结束时，调用TEST函数, 检查当前符号是否属于调用该语法单位时应有的后跟符号集合.

|  |
| --- |
| Test()函数的定义：  /\*\*  \* @param s1 需要的符号  \* @param s2 不需要的符号，添加一个补救集合  \* @param errcode 错误号  \*/  void test(BitSet s1, BitSet s2, int errcode) {  if (!s1.get(sym.symtype)) {  Err.report(errcode);  //当检测不通过时，不停地获取符号，直到它属于需要的集合  s1.or(s2); //把s2集合补充进s1集合  while (!s1.get(sym.symtype)) {  nextsym();  }  }  } |

注意：FOLLOW集合随着调用的深度增加，逐层增加，且与调用的位置相关。

|  |
| --- |
| 举例：  在write语句的下一层:  <statement>::=write ‘(‘<identity>{,identity}’)’  fsys={[rparen, comma]+fsys};  在factor语句的下一层  <factor>::=… …|’(’<expression>’)’  fsys={[rparen]+fsys}; |

表1- PL/0文法非终结符的开始符号集与后继符号集

|  |  |  |
| --- | --- | --- |
| 非终结符 | FIRST(S) | FOLLOW(S) |
| 分程序 | const var procedure ident if call begin while read write repeat | . ; |
| 语句 | ident call begin if while read write until | . ; end |
| 条件 | odd + - ( ident number | then do |
| 表达式 | = + - ( ident number | . ; R end then do |
| 项 | ident number ( | . ; R + - end then do |
| 因子 | ident number ( | . ; R + - \* / end then do |

PL/0编译系统中，所定义的36种错误类型，如下列举：

|  |  |
| --- | --- |
| PL/0语言的出错信息表 | |
| 出错编号 | 出错原因 |
| 1 | 常数说明中的"="写成"∶="。 |
| 2 | 常数说明中的"="后应是数字。 |
| 3 | 常数说明中的标识符后应是"="。 |
| 4 | const ,var, procedure后应为标识符。 |
| 5 | 漏掉了'，'或'；'。 |
| 6 | 过程说明后的符号不正确(应是语句开始符，或过程定义符)。 |
| 7 | 应是语句开始符。 |
| 8 | 程序体内语句部分的后跟符不正确。 |
| 9 | 程序结尾丢了句号'.'。 |
| 10 | 语句之间漏了'；'。 |
| 11 | 标识符未说明。 |
| 12 | 赋值语句中，赋值号左部标识符属性应是变量。 |
| 13 | 赋值语句左部标识符后应是赋值号'∶='。 |
| 14 | call后应为标识符。 |
| 15 | call后标识符属性应为过程。 |
| 16 | 条件语句中丢了'then'。 |
| 17 | 丢了'end"或'；'。 |
| 18 | while型循环语句中丢了'do'。 |
| 19 | 语句后的符号不正确。 |
| 20 | 应为关系运算符。 |
| 21 | 表达式内标识符属性不能是过程。 |
| 22 | 表达式中漏掉右括号')'。 |
| 23 | 因子后的非法符号。 |
| 24 | 表达式的开始符不能是此符号。 |
| 31 | 数越界。 |
| 32 | read语句括号中的标识符不是变量。 |
| 33 | 格式错误，应为右括号 |
| 34 | 格式错误，应为左括号 |
| 35 | read()中的变量未声明 |
| 36 | 变量字符过长 |

1. PL/0编译程序的目标代码解释执行和存储分配

* 类pcode解释器的结构

1. . 目标代码存放在数组pcodeArray中
2. . 定义一维整型数组runtimeStack作为运行栈
3. .栈顶寄存器（指针）sp;
4. .基址寄存器（指针）bp;
5. .程序地址寄存器 pc;
6. .指令寄存器 index.

* 运行栈的存储分配

1. . SL:静态链，指向定义该过程的直接外过程(或主程序)运行时最新数据段的基地址。
2. .DL:动态链，指向调用该过程前正在运行过程的数据段基地址。
3. .RA:返回地址，记录调用该过程时目标程序的断点，即调用过程指令的下一条指令的地址

例如，假定有过程 A，B，C，其中过程 C 的说明局部于过程 B，而过程 B 的 说明局部于过程 A，程序运行时，过程 A 调用过程 B，过程 B 则调用过程 C，过 程 C 又调用过程 B，如下图所示：

  

图9-1过程说明嵌套图 过程调用图 表示 A 调用 B

从静态链的角度我们可以说A是在第一层说明,B是在第二层说明,C则是在第三层说明。

若在B中存取A中说明的变量a,由于编译程序只知道A,B间的静态层差为1,如果这时沿着动态链下降一步,将导致对C的局部变量的操作。

为防止这种情况发生，设置第二条链，将各个数据区连接起来。我们称之为动态链（dynamic link）DL。

这样，编译程序所生成的代码地址，指示着静态层差和数据区的相对修正量。下面是过程 A、B 和 C 运行时刻的数据区图示：



P-code解释执行过程：

|  |
| --- |
| (1).LIT 0 A |
| sp 🡨 sp +1;  stack[sp] 🡨 A; |
| (2).LOD L A |
| sp 🡨 sp +1;  stack[sp] 🡨 stack[ base(L) + A]; |
| (3).STO L A |
| stack[ base(L) + A] 🡨 stack[sp];  sp 🡨 sp -1; |
| (4).CAL L A |
| stack[sp + 1] 🡨 base(L);  stack[sp + 2] 🡨 bp;  stack[sp + 3] 🡨 pc;  bp 🡨 sp + 1;  pc 🡨 A; |
| (5).INT 0 A |
| sp 🡨 sp + A;  (6).JMP 0 A |
| pc = A; |
| (7).JPC 0 A |
| if stack[sp] == 0  {  pc 🡨 A;  sp 🡨 sp - 1;  } |
| (8).OPR 0 0 |
| RETUEN  (stack[sp + 1] 🡨 base(L);  sp 🡨 bp - 1;  bp 🡨 stack[sp + 2];  pc 🡨 stack[sp + 3];) |
| (9).OPR 0 1 |
| NEG  (- stack[sp] ) |
| (10).OPR 0 2 |
| ADD  (sp 🡨 sp – 1 ;  stack[sp] 🡨 stack[sp] + stack[sp + 1]) |
| (11).OPR 0 3 |
| SUB  (sp 🡨 sp – 1 ;  stack[sp] 🡨stack[sp] - stack[sp + 1]) |
| (12).OPR 0 4 |
| MUL  (sp 🡨 sp – 1 ;  stack[sp] 🡨 stack[sp] \* stack[sp + 1]) |
| (13).OPR 0 5 |
| DIV  (sp 🡨 sp – 1 ;  stack[sp] 🡨 stack[sp] / stack[sp + 1]) |
| (14).OPR 0 6 |
| ODD  (stack[sp] 🡨 stack % 2) |
| (15).OPR 0 7 |
| MOD  (sp 🡨 sp – 1 ;  stack[sp] 🡨 stack[sp] % stack[sp + 1]) |
| (16).OPR 0 8 |
| EQL  (sp 🡨 sp – 1 ;  stack[sp] 🡨 stack[sp] == stack[sp + 1]) |
| (17).OPR 0 9 |
| NEQ  (sp 🡨 sp – 1 ;  stack[sp] 🡨 stack[sp] != stack[sp + 1]) |
| (18).OPR 0 10 |
| LSS  (sp 🡨 sp – 1 ;  stack[sp] 🡨 stack[sp] < stack[sp + 1]) |
| (19).OPR 0 11 |
| GEQ  (sp 🡨 sp – 1 ;  stack[sp] 🡨 stack[sp] >= stack[sp + 1]) |
| (20).OPR 0 12 |
| GTR  (sp 🡨 sp – 1 ;  stack[sp] 🡨 stack[sp] > stack[sp + 1]) |
| (21).OPR 0 13 |
| LEQ  (sp 🡨 sp – 1 ;  stack[sp] 🡨 stack[sp] <= stack[sp + 1]) |
| (22).OPR 0 14 |
| print (stack[sp]);  sp 🡨 sp – 1; |
| (23).OPR 0 15 |
| print ('\n'); |
| (24).OPR 0 16 |
| scan(stack[sp]);  sp 🡨 sp + 1; |
|  |

系统运行环境

硬件配置：lenovo-g470

软件配置：netbeans-7.4

软件运行环境：java JDK-1.7

附录：

样例测试

|  |  |
| --- | --- |
| //test.pl0 | //generated p-code |
| const z=0;  var head,foot,cock,rabbit,n;  begin  n := z;  cock := 1;  while cock <= head do  begin  rabbit :=head-cock;  if cock\*2+rabbit\*4=foot then  begin  write(cock,rabbit);  n:=n+1  end;  cock:=cock+1  end;  if n=0 then write(0,0)  end. | 0 JMP 0 21  1 JMP 0 2  2 INT 0 4  3 LOD 1 3  4 STO 0 3  5 LOD 1 4  6 STO 1 3  7 LOD 0 3  8 STO 1 4  9 OPR 0 0  10 JMP 0 11  11 INT 0 3  12 LOD 1 3  13 LOD 1 3  14 LOD 1 4  15 OPR 0 5  16 LOD 1 4  17 OPR 0 4  18 OPR 0 3  19 STO 1 3  20 OPR 0 0  21 INT 0 7  22 LIT 0 45  23 STO 0 3  24 LIT 0 27  25 STO 0 4  26 CAL 0 11  27 LOD 0 3  28 LIT 0 0  29 OPR 0 9  30 JPC 0 34  31 CAL 0 2  32 CAL 0 11  33 JMP 0 27  34 LOD 0 4  35 STO 0 5  36 LIT 0 45  37 LIT 0 27  38 OPR 0 4  39 LOD 0 5  40 OPR 0 5  41 STO 0 6  42 LOD 0 5  43 OPR 0 14  44 LOD 0 6  45 OPR 0 14  46 OPR 0 15  47 OPR 0 0 |