山东大学计算机学院实验报告

|  |  |  |
| --- | --- | --- |
| 实验题目：PL/0语言编译器 | | 学号： |
| 日期：5.27 | 班级：17.4 | 姓名： |
| Email：623581439@qq.com | | |
| 硬件环境：个人电脑 | | |
| 软件环境：C++ | | |
| 实验说明：  test文件夹为完整项目  input文件夹为输入pl0文件  lex文件夹为词法分析输出  table文件夹为符号表输出  parse文件夹为语法树输出  code文件夹为中间代码输出  vm虚拟机输入输出为标准输入输出  运行截图文件夹存放示例程序运行截图 | | |
| **实验一 PL/0语言建立一个词法分程序**  **实验要求**  把关键字、算符、界符称为语言固有的单词，标识符、常量称为用户自定义的单词。为此设置三个全程量：SYM,ID,NUM 。  SYM：存放每个单词的类别，为内部编码的表示形式。  ID：存放用户所定义的标识符的值，即标识符字符串的机内表示。  NUM：存放用户定义的数。  GETSYM要完成的任务：   1. 滤掉单词间的空格。 2. 识别关键字，用查关键字表的方法识别。当单词是关键字时，将对应的类别放在SYM中。如IF的类别为IFSYM，THEN的类别为THENSYM。 3. 识别标识符，标识符的类别为IDENT，IDENT放在SYM中，标识符本身的值放在ID中。关键字或标识符的最大长度是10。 4. 拼数，将数的类别NUMBER放在SYM中，数本身的值放在NUM中。 5. 拼由两个字符组成的运算符，如：>=、<=等等，识别后将类别存放在SYM。 6. 打印源程序，边读入字符边打印。 7. 由于一个单词是由一个或多个字符组成的，所以在词法分析程序GETSYM中定义一个读字符过程GETCH。   **实验内容**  1、词法分析所用产生式  <无符号整数> → <数字>{<数字>}  <标识符> → <字母>{<字母>|<数字>}  <加减运符> → +|-  <乘除运算符> → \*|/  <关系运算符> → =|#|<|<=|>|>=  <字母> → a|b|c…x|y|z  <数字> → 0|1|2…7|8|9  2、单词分为5类，运算符，界符，常数，标识符，关键字。运算符、界符、关键字一词一类。标识符为一类。常数为一类。使用枚举类enum token\_type表示类型。  3、结构体tokenStruct中有token\_type类成员变量表示单词类别，int成员变量表示常数数值，char数组成员变量存放标识符字符串。   1. **struct** tokenStruct 2. { 3. token\_type tokenID; 4. **int** numberValue; 5. **char** name[StrLength]; 6. };   4、词法分析的结果存放在token\_type类数组lexList中。lexListIndex表示当前长度。  5、char \*reservedWords[]数组定义了pl0语句中的保留字。  6、char specialSymbols[]定义了pl0中的运算符和或界符。  7、reserve函数判断字符串是否是保留字。   1. **int** reserve(**char** \*str) { 2. **for** (**int** i = 0; i < ReservedWordsLength; i++) 3. **if** (!strcmp(str, reservedWords[i])) 4. **return** i; 5. **return** -1; 6. }   8、speial函数判断字符是否是运算符或界符   1. **int** speial(**char** c) { 2. **for** (**int** i = 0; i < SpecialSymbolsLength; i++) 3. **if** (c == specialSymbols[i]) 4. **return** i; 5. **return** -1; 6. }   9、void lex(char \*infilename, char \*outfilename)进行词法分析，输入参数为输入文件名和输出文件名。  10、使用fgetc()输入字符。  11、当输入为字母时，可能为标识符或关键字，调用reserve函数判断。同时进行出错处理（如字符串长度过长）。如果判断为标识符，则插入符号表。  12、当输入为常数时，则计算出输入的常数值并存储。插入常数表。同时进行出错处理（如常数后紧跟字符、常数过大）。  13、当输入为运算符或界符时，因为运算符或界符可能包含多个字符，所以需要超前搜索。  14、巧妙使用int lookAhead记录是否超前搜索，简化了程序。  15、data.h存放全局变量和各阶段分析都要用到的宏定义。Lex.h存放词法分析用到的宏定义。 | | |
| **实验二 PL/0语言建立一个语法分程序**  **2.1 语法分析**  **实验要求**  PL/0编译程序采用一遍扫描的方法，所以语法分析和代码生成都有在BLOCK中完成。BLOCK的工作分为两步：  1 说明部分的处理  说明部分的处理任务就是对每个过程（包括主程序，可以看成是一个主过程）的说明对象造名字表。填写所在层次（主程序是0层，在主程序中定义的过程是1层，随着嵌套的深度增加而层次数增大。PL/0最多允许3层），标识符的属性和分配的相对地址等。标识符的属性不同则填写的信息不同。  所造的表放在全程量一维数组TABLE中，TX为指针，数组元素为结构体类型数据。LEV给出层次，DX给出每层的局部量的相对地址，每说明完一个变量后DX加1。  例如：一个过程的说明部分为：  const a=35,b=49;  var c,d,e;  procedure p;  var g;  对它的常量、变量和过程说明处理后，TABLE表中的信息如下：   |  |  |  |  | | --- | --- | --- | --- | | NAME: a  NAME: b  NAME: c  NAME: d  NAME: e  NAME: p | KIND: CONSTANT  KIND: CONSTANT  KIND: VARIABLE  KIND: VARIABLE  KIND: VAEIABLE  KIND: PROCEDURE | VAL: 35  VAL: 49  LEVEL: LEV  LEVEL: LEV  LEVEL: LEV  LEVEL: LEV | ADR: DX  ADR: DX+1  ADR: DX+2  ADR: | | NAME: g  。  。  。 | KIND: VARIABLE  。  。  。 | LEVEL: LEV+1  。  。  。 | ADR: DX  。  。  。 |   对于过程名的ADR域，是在过程体的目标代码生成后返填过程体的入口地址。TABLE表的索引TX和层次单元LEV都是以BLOCK的参数形式出现，在主程序调用BLOCK时实参的值为0。每个过程的相对起始位置在BLOCK内置初值DX=3。  2 语句处理和代码生成  对语句逐句分析，语法正确则生目标代码,当遇到标识符的引用则去查TABLE表,看是否有过正确的定义，若有则从表中取出相关的信息,供代码生成用。PL/0语言的代码生成是由过程GEN完成。GEN过程有三个参数，分别代表目标代码的功能码、层差、和位移量。生成的目标代码放在数组CODE中。CODE是一维数组，数组元素是结构体类型数据。  PL/0语言的目标指令是一种假想的栈式计算机的汇编语言，其格式如下：  f l a  其中f代表功能码，l代表层次差，a代表位移量。目标指令有8条：  ① LIT：将常数放到运栈顶，a域为常数。  ② LOD：将变量放到栈顶。a域为变量在所说明层中的相对位置，l为调用层与说明层的层差值。  ③ STO：将栈顶的内容送到某变量单元中。a,l域的含义与LOD的相同。  ④ CAL：调用过程的指令。a为被调用过程的目标程序的入中地址，l为层差。  ⑤ INT：为被调用的过程（或主程序）在运行栈中开辟数据区。a域为开辟的个数。  ⑥ JMP：无条件转移指令，a为转向地址。  ⑦ JPC：条件转移指令，当栈顶的布尔值为非真时，转向a域的地址，否则顺序执行。为0跳转，为1顺序执行  ⑧ OPR：关系和算术运算。具体操作由a域给出。运算对象为栈顶和次顶的内容进行运算，结果存放在次顶。a域为0时是退出数据区。  **实验内容**  1 语法分析所用产生式  〈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>  <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>；{<过程说明部分>}  <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>|<空>  <赋值语句> → <标识符>:=<表达式>  <复合语句> → begin<语句>{ ；<语句>}<end>  <条件> → <表达式><关系运算符><表达式>|odd<表达式>  <表达式> → [+|-]<项>{<加减运算符><项>}  <项> → <因子>{<乘除运算符><因子>}  <因子> → <标识符>|<无符号整数>|(<表达式>)  <条件语句> → if<条件>then<语句>  <过程调用语句> → call<标识符>  <当型循环语句> → while<条件>do<语句>  <读语句> → read(<标识符>{ ，<标识符>})  <写语句> → write(<标识符>{，<标识符>})  2 语法分析标识符表项由结构体symbol定义。int kind表示标识符种类，char name[12]存放标识符名，val存放常量标识符值，int level存放标识符层数，int addr存放偏移值。   1. /\*For constants, you must store kind, name and value. 2. For variables, you must store kind, name, L and M. 3. For procedures, you must store kind, name, L and M.\*/ 4. **struct** symbol 5. { 6. **int** kind;       // const = 1, var = 2, proc = 3 7. **char** name[12];  // name up to 11 chars 8. **int** val;        // number (ASCII value) 9. **int** level;      // L level 10. **int** addr;       // M address 11. };   3 语法分析目标代码表项由结构体instruction表示   1. **struct** instruction{ 2. ins\_type ins; //Opcode 3. **int** l;  //L 4. **int** m;  //M 5. };   4 全局变量  table为符号表，code为目标代码表，tx表示符号表序号，cx表示代码表序号，lev表示当前层数 ，dx表示当前过程偏移，lexemeListIndex是词法分析结果当前读入位置，curToken是最新读入tokenStruct，blank是打印空格数。   1. symbol table[MAX\_SYMBOL\_TABLE\_SIZE];    //符号表 2. instruction code[CODE\_SIZE];            //目标代码 4. **int** tx, cx, lev = -1;                   //tx表示符号表序号，cx表示代码表序号，lev表示当前层数 5. //tx已经占据，cx没有占据 7. **int** dx = 3;                             //表示当前过程偏移 9. **int** lexemeListIndex = 0;                //词法分析结果当前读入位置 10. tokenStruct curToken;                   //最新读入tokenStruct 12. **int** blank = 0;                          //打印空格数   5 自上而下的语法分析需要为每一个非终结符构造一个语法分析函数。   1. **void** program();             //程序 2. **void** block();               //分程序 3. **void** constDeclaration();    //常量说明部分 4. **void** constDefinition();     //常量定义 5. **void** varDeclaration();      //变量说明部分 6. **void** varDefinition();       //变量定义 7. **void** procedureDeclaration();//过程说明部分 8. **void** procedureHead();       //过程首部 9. **void** statement();           //语句 10. **void** condition();           //条件 11. **void** expression();          //表达式 12. **void** term();                //项 13. **void** factor();              //因子 15. **void** assignment();          //赋值语句 16. **void** callProcedure();       //过程调用语句 17. **void** conditionStatement();  //条件语句 18. **void** compoundStatement();   //复合语句 19. **void** loopStatement();       //循环语句 20. **void** writeStatement();      //写语句 21. **void** readStatement();       //读语句   6 getNextToken()从词法分析结果中获得下一个输入非终结符。存放到全局变量curToken中。全局变量int lexemeListIndex 表示当前读入终结符的个数。   1. **void** getNextToken() { 2. curToken.tokenID = lexList[lexemeListIndex].tokenID; 4. **if** (curToken.tokenID == 2) 5. strcpy(curToken.name, lexList[lexemeListIndex].name); 6. **else** **if** (curToken.tokenID == 3) 7. curToken.numberValue = lexList[lexemeListIndex].numberValue; 9. lexemeListIndex++; 10. }   7 enter()向标识符表中添加变量、常量、过程。根据类型赋值不同的成员变量。  symbol table[MAX\_SYMBOL\_TABLE\_SIZE]为符号表。   1. **void** enter(**int** kind) { 2. tx++; 4. strcpy(table[tx].name, curToken.name); 5. table[tx].kind = kind; 7. //const 8. **if** (kind == 1) 9. table[tx].val = curToken.numberValue; 10. **else** **if** (kind == 2) { 11. table[tx].level = lev; 12. table[tx].addr = dx; 13. dx++; 14. } **else** **if** (kind == 3) 15. table[tx].level = lev; 16. }   8 全局变量int tx为当前标识符表最后一项。Int lev表示当前层数，int dx表示当前层的偏移量，  9 position()函数进行查标识符表，只能查找当前层或当前层的所有父层，如果找到则返回位置，如果没有找到返回0。   1. **int** position() { 2. **int** curLev = lev; 4. **for** (**int** i = tx; i > 0; i--) { 5. **if** (table[i].kind == 3 && table[i].level == curLev - 1) 6. curLev--; 8. **if** (table[i].level > curLev) 9. **continue**; 11. **if** (strcmp(table[i].name, curToken.name) == 0) 12. **return** i; 13. } 14. **return** 0; 15. }   10 error(int errorCase)函数输出语法分析错误。便于调试。  11 语法分析树输出。通过打印制表符\t，表示树的层级。通过全局变量blank存放需打印制表符\t的个数。printBlank()进行打印制表符\t。在每个语法分析函数终结符处进行输出，打印语法树   1. **void** printBlank() { 2. **for** (**int** i = 0; i < blank; i++) 3. fprintf(ofp, "\t"); 4. }   12 emit()生成目标代码到代码表。   1. **void** emit(ins\_type ins, **int** l, **int** m) { 2. **if** (cx >= CODE\_SIZE) { 3. printf("Program too long! cx > CODE\_SIZE\n"); 4. exit(1); 5. } **else** { 6. code[cx].ins = ins; 7. code[cx].l = l; 8. code[cx].m = m; 9. cx++; 10. } 11. }   12处理非终结符<程序>   1. **void** program() { 2. printBlank(); 3. fprintf(ofp, "program\n"); 4. blank++; 6. **int** cx0 = cx; 7. emit(JMP, 0, 0); 9. getNextToken(); 10. block(); 12. code[cx0].m = mainCode; 14. **if** (curToken.tokenID != periodsym) 15. error(1);       //Period expected. 17. printBlank(); 18. fprintf(ofp, ".\n"); 20. blank--; 21. }   13 处理非终结符<分程序>   1. **void** block() { 2. printBlank(); 3. fprintf(ofp, "block\n"); 4. blank++; 6. lev++; 7. **if** (lev > MAX\_LEXI\_LEVELS) 8. error(2);      //errors for above max lexi level 10. **int** oldDx = dx; 11. dx = 3; 13. **int** tx0 = tx; 15. **do** { 16. **if** (curToken.tokenID == constsym) { 17. constDeclaration(); 18. } 19. **if** (curToken.tokenID == varsym) { 20. varDeclaration(); 21. } 22. **while** (curToken.tokenID == procsym) { 23. procedureDeclaration(); 24. } 25. } **while** ((curToken.tokenID == constsym) || (curToken.tokenID == varsym) || (curToken.tokenID == procsym)); 27. **if** (lev == 0) 28. mainCode = cx; 30. table[tx0].addr = cx; 32. emit(**INT**, 0, dx); 34. statement(); 36. emit(OPR, 0, 0); 38. dx = oldDx; 39. lev--; 41. blank--; 42. }   14 处理非终结符<常量说明部分>   1. **void** constDeclaration() { 2. printBlank(); 3. fprintf(ofp, "constDeclaration\n"); 4. blank++; 6. printBlank(); 7. fprintf(ofp, "const\n"); 9. getNextToken(); 10. constDefinition(); 11. **while** (curToken.tokenID == commasym) { 12. printBlank(); 13. fprintf(ofp, ",\n"); 15. getNextToken(); 16. constDefinition(); 17. } 18. **if** (curToken.tokenID == semicolonsym) { 19. printBlank(); 20. fprintf(ofp, ";\n"); 22. getNextToken(); 23. } **else** { 24. error(3);       //常量声明部分缺少; 25. } 27. blank--; 28. }   15 处理非终结符   1. **void** constDefinition() { 2. printBlank(); 3. fprintf(ofp, "constDefinition\n"); 4. blank++; 6. **if** (curToken.tokenID == identsym) { 7. printBlank(); 8. fprintf(ofp, "%s\n", curToken.name); 10. getNextToken(); 11. **if** ((curToken.tokenID == eqsym) || (curToken.tokenID == becomessym)) { 12. **if** (curToken.tokenID == becomessym) 13. error(6);       //Use = instead of := in constDefinition 15. printBlank(); 16. fprintf(ofp, ":=\n"); 18. getNextToken(); 19. **if** (curToken.tokenID == numbersym) { 20. printBlank(); 21. fprintf(ofp, "%d\n", curToken.numberValue); 23. enter(1); 24. getNextToken(); 25. } **else** { 26. error(8);       //常量定义赋值的不是整数 27. } 29. } **else** { 30. error(7);           //常量未赋值 31. } 32. } **else** { 33. error(5);       //常量定义缺少标识符 34. } 36. blank--; 37. }   **2.3 实例与输出**  输入PL0文件   1. **const** a=10; 2. var b,c; 3. procedure p; 4. begin 5. c:=b+a 6. end; 7. begin 8. read(b); 9. **while** b#0 **do** 10. begin 11. call  p; 12. write(2\*c); 13. read(b) 14. end 15. end .   输出符号表   1. kind:0  val:0   lev:-1  addr:7  name: 2. kind:1  val:10  lev:0   addr:0  name:a 3. kind:2  val:0   lev:0   addr:3  name:b 4. kind:2  val:0   lev:0   addr:4  name:c 5. kind:3  val:0   lev:0   addr:1  name:p   输出语法树   1. program 2. block 3. constDeclaration 4. **const** 5. constDefinition 6. a 7. := 8. 10 9. ; 10. varDeclaration 11. var 12. b 13. , 14. c 15. ; 16. procedureDeclaration 17. procedureHead 18. procedure 19. p 20. ; 21. block 22. statement 23. compoundStatement 24. begin 25. statement 26. assignment 27. c 28. := 29. expression 30. term 31. factor 32. b 33. + 34. term 35. factor 36. a 37. end 38. ; 39. statement 40. compoundStatement 41. begin 42. statement 43. readStatement 44. read 45. ( 46. b 47. ) 48. ; 49. statement 50. loopStatement 51. **while** 52. condition 53. expression 54. term 55. factor 56. b 57. # 58. expression 59. term 60. factor 61. 0 62. **do** 63. statement 64. compoundStatement 65. begin 66. statement 67. callProcedure 68. call 69. p 70. ; 71. statement 72. writeStatement 73. write 74. ( 75. expression 76. term 77. factor 78. 2 79. \* 80. factor 81. c 82. ) 83. ; 84. statement 85. readStatement 86. read 87. ( 88. b 89. ) 90. end 91. end 92. .   输出中间代码   1. 6 0 7 2. 5 0 3 3. 2 1 3 4. 1 0 10 5. 8 0 1 6. 3 1 4 7. 8 0 0 8. 5 0 5 9. 8 0 12 10. 3 0 3 11. 2 0 3 12. 1 0 0 13. 8 0 9 14. 7 0 22 15. 4 0 1 16. 1 0 2 17. 2 0 4 18. 8 0 3 19. 8 0 11 20. 8 0 12 21. 3 0 3 22. 6 0 10 23. 8 0 0 | | |
| **实验三 建立一个解释执行目标程序的函数**  **实验要求**  编译结束后，记录源程序中标识符的TABLE表已退出内存，内存中只剩下用于存放目标程序的CODE数组和运行时的数据区S。S是由解释程序定义的一维整型数组。解释执行时的数据空间S为栈式计算机的存储空间。遵循后进先出的规则，对每个过程（包括主程序）当被调用时，才分配数据空间，退出过程时，则所分配的数据空间被释放。  为解释程序定义四个寄存器：  I：指令寄存器，存放当前正在解释的一条目标指令。  P：程序地址寄存器，指向下一条要执行的目标指令（相当于CODE数组的下标）。  T：栈顶寄存器，每个过程运行时要为它分配数据区（或称为数据段），该数据区分为两部分。  静态部分：包括变量存放区和三个联单元。  动态部分：作为临时工作单元和累加器用。需要时临时分配，用完立即释放。栈顶寄存器T指出了当前栈中最新分配的单元（T也是数组S的下标）。  B：基地址寄存器，指出每个过程被调用时，在数据区S中给出它分配的数据段起始地址，也称为基地址。每个过程被调用时，在栈顶分配三个联系单元。这三个单元的内容分别是：  SL：静态链，它是指向定义该过程的直接外过程运行时数据段的基地址。  DL：动态链，它是指向调用该过程前正在运行过程的数据段的基地址。  RA：返回地址，记录调用该过程时目标程序的断点，即当时的程序地址寄存器P的值。  具体的过程调用和结束，对上述寄存器及三个联系单元的填写和恢复由下列目标指令完成。  INT 0 a  a:为局部量个数加3  OPR 0 0  恢复调用该过程前正在运行过程（或主程序）的数据段的基地址寄存器的值，恢复栈顶寄存器T的值，并将返回地址送到指令寄存器P中。  CAL l a  a为被调用过程的目标程序的入口，送入指令地址寄存器P中。  CAL指令还完成填写静态链，动态链，返回地址，给出被调用过程的基地址值，送入基址寄存器B中。  **实验内容**  1 全局变量  sp表示栈顶寄存器，bp表示基寄存器，pc表示程序地址寄存器，codeLength表示目标代码长度，arrayStruct存放目标代码，ir为指令寄存器，stack为堆栈。   1. **int** sp;     //栈顶 2. **int** bp;     //基址 3. **int** pc;     //程序地址寄存器 5. **int** codeLength;                             //目标代码长度 6. instruction arrayStruct[CODE\_SIZE];         //存放读入目标代码 8. instruction ir;                             //指令寄存器 10. **int** stack[MAX\_STACK\_HEIGHT];                //堆栈   2 vm()读入目标代码并执行，一个周期有取值、执行两部分组成。去掉注释可打印单步过程。   1. **void** vm(**char** \*codefilename) { 2. pc = 0; 3. bp = 0; 4. sp = -1; 5. memset(stack,0, **sizeof**(stack)); 7. **FILE** \*ifp; 8. ifp = fopen(codefilename, "r"); 9. codeLength = 0; 11. **while** (fscanf(ifp, "%d %d %d", &arrayStruct[codeLength].ins, &arrayStruct[codeLength].l, 12. &arrayStruct[codeLength].m) != EOF) { 13. codeLength++; 14. } 16. //    printReg(); 17. //    printStack(); 19. **for** (; pc < codeLength - 1;) { 20. ir = arrayStruct[pc++]; 22. //        printIr(); 24. executeCycle(); 26. //        printReg(); 27. //        printStack(); 28. } 29. }   3 base()回溯静态链。如果当前指令层号为0则返回bp值，如果不为0，则进行回溯。每一个过程第一个地址存放静态链。   1. **int** base() { 2. **int** l = ir.l; 3. **int** SL = bp; 5. **while** (l > 0) { 6. SL = stack[SL]; 7. l--; 8. } 9. **return** SL; 10. }   4 executeCycle()执行一条指令。switch根据8种指令码确定分支。CAL分支建立静态链。   1. **void** executeCycle() { 2. **switch** (ir.ins) { 3. **case** LIT: 4. stack[++sp] = ir.m; 5. **break**; 6. **case** LOD: 7. stack[++sp] = stack[base() + ir.m]; 8. **break**; 9. **case** STO: 10. stack[base() + ir.m] = stack[sp--]; 11. **break**; 12. **case** CAL: 13. stack[sp + 1] = base();         //SL 14. stack[sp + 2] = bp;             //DL 15. stack[sp + 3] = pc;             //返回地址 17. bp = sp + 1; 18. pc = ir.m; 19. **break**; 20. **case** **INT**: 21. sp = sp + ir.m; 22. **break**; 23. **case** JMP: 24. pc = ir.m; 25. **break**; 26. **case** JPC: 27. **if** (!stack[sp--]) 28. pc = ir.m; 29. **break**; 30. **case** OPR: 31. opr(); 32. **break**; 33. **default**: 34. printf("Illegal INS!\n"); 35. } 36. }   5 opr()处理当指令码为OPR时，根据m觉得具体操作。可以进行运算操作和逻辑运算操作。当m为0时则为RET操作，会收过程栈桢。   1. **void** opr() { 2. **int** temp; 3. **switch** (ir.m) { 4. **case** ADD: 5. temp = stack[sp - 1] + stack[sp]; 6. stack[--sp] = temp; 7. **break**; 8. **case** SUB: 9. temp = stack[sp - 1] - stack[sp]; 10. stack[--sp] = temp; 11. **break**; 12. **case** DIV: 13. temp = stack[sp - 1] / stack[sp]; 14. stack[--sp] = temp; 15. **break**; 16. **case** MINUS: 17. stack[sp] = -stack[sp]; 18. **break**; 19. **case** MUL: 20. temp = stack[sp - 1] \* stack[sp]; 21. stack[--sp] = temp; 22. **break**; 23. **case** EQ: 24. temp = (stack[sp - 1] == stack[sp]); 25. stack[--sp] = temp; 26. **break**; 27. **case** NE: 28. temp = (stack[sp - 1] != stack[sp]); 29. stack[--sp] = temp; 30. **break**; 31. **case** GE: 32. temp = (stack[sp - 1] >= stack[sp]); 33. stack[--sp] = temp; 34. **break**; 35. **case** GT: 36. temp = (stack[sp - 1] > stack[sp]); 37. stack[--sp] = temp; 38. **break**; 39. **case** LE: 40. temp = (stack[sp - 1] <= stack[sp]); 41. stack[--sp] = temp; 42. **break**; 43. **case** LT: 44. temp = (stack[sp - 1] < stack[sp]); 45. stack[--sp] = temp; 46. **break**; 47. **case** ODD: 48. temp = stack[sp] % 2; 49. stack[sp] = temp; 50. **break**; 51. **case** WRITE: 52. printf("%d\n", stack[sp--]); 53. **break**; 54. **case** READ: 55. scanf("%d", &stack[++sp]); 56. **break**; 57. **case** 0: 58. sp = bp - 1; 59. pc = stack[sp + 3]; 60. bp = stack[sp + 2]; 61. **break**; 62. **default**: 63. printf("Illegal OPR!\n"); 64. } 65. }   6 单步执行信息输出  printIr()输出指令，printReg()输出寄存器值，printStack()输出堆栈信息。   1. **void** printIr()              //输出指令 2. { 3. printf("ir:(%d %d %d)\t", ir.ins, ir.l, ir.m); 4. **if**(ir.ins == OPR) 5. printf("(%s %d %s)\n", insStr[ir.ins], ir.l, oprStr[ir.m]); 6. **else** 7. printf("(%s %d %d)\n", insStr[ir.ins], ir.l, ir.m); 8. } 10. **void** printReg()             //输出寄存器值 11. { 12. printf("pc:%d\t", pc); 13. printf("bp:%d\t", bp); 14. printf("sp:%d\t", sp); 15. } 17. **void** printStack()           //输出堆栈信息 18. { 19. printf("stack:("); 20. **int** max = sp >= bp ? sp : bp; 21. **for** (**int** i = 0; i <= max; i++) 22. **if** (i) 23. printf(" %d", stack[i]); 24. **else** 25. printf("%d", stack[i]); 26. printf(")\n"); 27. }   **实例及输出**  1 PL0\_code.in文件进行乘除、求公因数、乘方运算  6 \* 6 = 36  7 / 3 = 2 余数1  24和18求最大公约数6  5! = 120    2 PL0\_code1.in文件输入一个数b，判断b是否为0，如果为0则退出。如果b不为0则输出(b + 10) \* 2  输入5，输出30  输入10，输出40  输入1，输出22  输入0，终止    3 PL0\_code2.in输出1到10的乘方    4 PL0\_code3.in输出100以内的素数 | | |
| 5 PL0\_code0.in    6 PL0\_code4.in  输入一个数b,b!=0时输出（b + 10） \* 2,b=0时退出  单步执行结果   1. pc:0    bp:0    sp:-1   stack:(0) 2. ir:(6 0 19)     (JMP 0 19) 3. pc:19   bp:0    sp:-1   stack:(0) 4. ir:(5 0 6)      (**INT** 0 6) 5. pc:20   bp:0    sp:5    stack:(0 0 0 0 0 0) 6. ir:(8 0 12)     (OPR 0 READ) 7. 5 8. pc:21   bp:0    sp:6    stack:(0 0 0 0 0 0 5) 9. ir:(3 0 4)      (STO 0 4) 10. pc:22   bp:0    sp:5    stack:(0 0 0 0 5 0) 11. ir:(8 0 12)     (OPR 0 READ) 12. 10 13. pc:23   bp:0    sp:6    stack:(0 0 0 0 5 0 10) 14. ir:(3 0 5)      (STO 0 5) 15. pc:24   bp:0    sp:5    stack:(0 0 0 0 5 10) 16. ir:(2 0 4)      (LOD 0 4) 17. pc:25   bp:0    sp:6    stack:(0 0 0 0 5 10 5) 18. ir:(8 0 11)     (OPR 0 WRITE) 19. 5 20. pc:26   bp:0    sp:5    stack:(0 0 0 0 5 10) 21. ir:(2 0 5)      (LOD 0 5) 22. pc:27   bp:0    sp:6    stack:(0 0 0 0 5 10 10) 23. ir:(8 0 11)     (OPR 0 WRITE) 24. 10 25. pc:28   bp:0    sp:5    stack:(0 0 0 0 5 10) 26. ir:(2 0 3)      (LOD 0 3) 27. pc:29   bp:0    sp:6    stack:(0 0 0 0 5 10 0) 28. ir:(8 0 11)     (OPR 0 WRITE) 29. 0 30. pc:30   bp:0    sp:5    stack:(0 0 0 0 5 10) 31. ir:(4 0 1)      (CAL 0 1) 32. pc:1    bp:6    sp:5    stack:(0 0 0 0 5 10 0) 33. ir:(5 0 4)      (**INT** 0 4) 34. pc:2    bp:6    sp:9    stack:(0 0 0 0 5 10 0 0 31 0) 35. ir:(1 0 10)     (LIT 0 10) 36. pc:3    bp:6    sp:10   stack:(0 0 0 0 5 10 0 0 31 0 10) 37. ir:(1 0 2)      (LIT 0 2) 38. pc:4    bp:6    sp:11   stack:(0 0 0 0 5 10 0 0 31 0 10 2) 39. ir:(8 0 3)      (OPR 0 MUL) 40. pc:5    bp:6    sp:10   stack:(0 0 0 0 5 10 0 0 31 0 20) 41. ir:(3 1 3)      (STO 1 3) 42. pc:6    bp:6    sp:9    stack:(0 0 0 20 5 10 0 0 31 0) 43. ir:(1 0 10)     (LIT 0 10) 44. pc:7    bp:6    sp:10   stack:(0 0 0 20 5 10 0 0 31 0 10) 45. ir:(1 0 3)      (LIT 0 3) 46. pc:8    bp:6    sp:11   stack:(0 0 0 20 5 10 0 0 31 0 10 3) 47. ir:(8 0 4)      (OPR 0 DIV) 48. pc:9    bp:6    sp:10   stack:(0 0 0 20 5 10 0 0 31 0 3) 49. ir:(3 1 4)      (STO 1 4) 50. pc:10   bp:6    sp:9    stack:(0 0 0 20 3 10 0 0 31 0) 51. ir:(2 1 3)      (LOD 1 3) 52. pc:11   bp:6    sp:10   stack:(0 0 0 20 3 10 0 0 31 0 20) 53. ir:(2 1 4)      (LOD 1 4) 54. pc:12   bp:6    sp:11   stack:(0 0 0 20 3 10 0 0 31 0 20 3) 55. ir:(8 0 8)      (OPR 0 LE) 56. pc:13   bp:6    sp:10   stack:(0 0 0 20 3 10 0 0 31 0 0) 57. ir:(7 0 18)     (JPC 0 18) 58. pc:18   bp:6    sp:9    stack:(0 0 0 20 3 10 0 0 31 0) 59. ir:(8 0 0)      (OPR 0 RET) 60. pc:31   bp:0    sp:5    stack:(0 0 0 20 3 10) 61. ir:(2 0 3)      (LOD 0 3) 62. pc:32   bp:0    sp:6    stack:(0 0 0 20 3 10 20) 63. ir:(8 0 14)     (OPR 0 ODD) 64. pc:33   bp:0    sp:6    stack:(0 0 0 20 3 10 0) 65. ir:(7 0 40)     (JPC 0 40) 66. pc:40   bp:0    sp:5    stack:(0 0 0 20 3 10) | | |
| 结论分析与体会：   1. 完成了PL0编译器的词法分析工作。结果采用精简形式打印输出。 2. 完成了语法分析工作，打印输出符号表和语法树。 3. 对变量的符号表查询进行了优化，保证只能在正确的作用域使用变量。 4. 完成了目标代码生成工作，主要通过实现emit()和修改词法分析程序完成。 5. 完成了虚拟机，可以执行目标代码。正确运行示例程序。必要时可打印输出单步执行时的寄存器和堆栈信息。 6. 本次实验，通过课本的理论知识和编程实践相结合，对编译原理有了更加系统的认识，对词法分析、语法分析等阶段有了更深刻的概念。特别是虚拟机中对静态链和动态链的处理，使我对程序执行过程中数据空间、堆栈的处理有了更加清晰的认识。 | | |
| 代码：  1 data.h   1. // 2. // Created by 62358 on 2020/4/18. 3. // 5. #ifndef COMPILE\_DATA\_H 6. #define COMPILE\_DATA\_H 8. #define StrLength 15 9. #define CODE\_SIZE 1000 11. #define MAX\_LEXI\_LEVELS 3 12. #define MAX\_SYMBOL\_TABLE\_SIZE 1000  15. #define MAX\_STRPRINT\_LENGTH 34 17. **enum** token\_type 18. { 19. nulsym = 1, 20. identsym,       //标识符 21. numbersym,      //常数 22. plussym,        //+ 23. minussym,       //- 24. multsym,        //\* 25. slashsym,       /// 26. oddsym,         //odd 27. eqsym,          //= 28. neqsym,         //# 29. lessym,         //< 30. leqsym,         //<= 31. gtrsym,         //> 32. geqsym,         //>= 33. lparentsym,     //( 34. rparentsym,     //) 35. commasym,       //, 36. semicolonsym,   //; 37. periodsym,      //. 38. becomessym,     //:= 39. beginsym,       //begin 40. endsym,         //end 41. ifsym,          //if 42. thensym,        //then 43. whilesym,       //while 44. dosym,          //do 45. callsym,        //call 46. constsym,       //const 47. varsym,         //var 48. procsym,        //procedure 49. writesym,       //write 50. readsym,        //read 51. elsesym         //else 52. }; 54. **const** **extern** **char** \*strPrint[MAX\_STRPRINT\_LENGTH];  57. **struct** tokenStruct 58. { 59. token\_type tokenID; 60. **int** numberValue; 61. **char** name[StrLength]; 62. }; 64. **extern** tokenStruct lexList[5000]; 65. **extern** **int** lexListIndex; 67. **void** lex(**char** \* infilename,**char** \*outfilename); 69. **enum** ins\_type 70. { 71. LIT = 1, 72. LOD , 73. STO, 74. CAL, 75. **INT**, 76. JMP, 77. JPC, 78. OPR 79. }; 81. **enum** op\_type 82. { 83. ADD = 1, 84. SUB, 85. MUL, 86. DIV, 87. GT, 88. LT, 89. GE, 90. LE, 91. NE, 92. EQ, 93. WRITE, 94. READ, 95. MINUS, 96. ODD 97. }; 99. **struct** instruction{ 100. ins\_type ins; //Opcode 101. **int** l;  //L 102. **int** m;  //M 103. }; 105. **extern** instruction code[CODE\_SIZE]; 107. **void** parse(**char** \*parsefilename,**char** \*tablefilename,**char** \*codefilename); 109. **void** vm(**char** \*codefilename);   113. #endif //COMPILE\_DATA\_H   2 lex.h   1. // 2. // Created by 62358 on 2020/4/18. 3. // 5. #ifndef COMPILE\_LEX\_H 6. #define COMPILE\_LEX\_H 8. #include "data.h" 9. #define ReservedWordsLength 14 10. #define SpecialSymbolsLength 14 11. #define NumMaxLength 4 13. **extern** **int** reserve(**char** \*str); 15. **extern** **int** speial(**char** c); 17. #endif //COMPILE\_LEX\_H   3 lex.cpp   1. // 2. // Created by 62358 on 2020/4/18. 3. // 5. #include <cstdio> 6. #include <cstring> 7. #include <cctype> 8. #include "data.h" 9. #include "lex.h" 11. **const** **char** \*reservedWords[] = {"const", "var", "procedure", "call", "begin", "end", "if", "then", "else", "while", "do", 12. "read", "write", "odd"}; 13. **const** **char** specialSymbols[] = {'+', '-', '\*', '/', '(', ')', '=', ',', '.', '<', '>', ';', ':','#'}; 14. **const** **char** \*strPrint[] = {"", "", "", "", "+", "-", "\*", "/", "odd", "=", "#", "<", "<=", ">", ">=", "(", ")", ",", 15. ";", ".", ":=", "begin", "end", "if", "then", "while", "do", "call", "const", "var", 16. "procedure", "write", "read", "else"}; 18. tokenStruct lexList[5000]; 19. **int** lexListIndex = 0; 21. **int** reserve(**char** \*str) { 22. **for** (**int** i = 0; i < ReservedWordsLength; i++) 23. **if** (!strcmp(str, reservedWords[i])) 24. **return** i; 25. **return** -1; 26. } 28. **int** speial(**char** c) { 29. **for** (**int** i = 0; i < SpecialSymbolsLength; i++) 30. **if** (c == specialSymbols[i]) 31. **return** i; 32. **return** -1; 33. } 35. **void** lex(**char** \*infilename,**char** \*outfilename) { 36. lexListIndex = 0; 37. **FILE** \*ifp; 38. **FILE** \*ofp; 40. ifp = fopen(infilename, "r"); 41. ofp = fopen(outfilename, "w"); 43. //检察错误 44. **int** errorHolder; 45. //读入字符 46. **int** c; 47. **int** comments = 0; 48. //是否需要读入下一个字符,1表示已经读入且不需要读入，0表示需要读入 49. **int** lookAhead = 1; 51. c = fgetc(ifp); 53. **while** (c != EOF) { 54. //处理空白字符 55. **if** (c == ' ' || c == '\t' || c == '\r' || c == '\n') { 56. c = fgetc(ifp); 57. lookAhead = 1; 58. **continue**; 59. } 60. **if** (isalpha(c)) { 61. //临时存放输入字符串 62. **char** characterString[StrLength]; 63. memset(characterString, 0, **sizeof**(characterString)); 65. //读入位置 66. **int** index = 0; 67. characterString[index++] = c; 68. lookAhead = 1; 70. **while** (isalpha(c = fgetc(ifp)) || isdigit(c)) { 71. **if** (index >= StrLength - 1) { 72. printf("Error 3: Name too long.\n"); 73. //读入多余字符,超过StrLength-1个字符予以忽略 74. **while** (isalpha(c = fgetc(ifp)) || isdigit(c)); 75. errorHolder = 1; 76. **break**; 77. } 78. characterString[index++] = c; 79. } 81. //出现错误，不接受输入单词 82. **if** (errorHolder == 1) { 83. errorHolder = 0; 84. **continue**; 85. } 87. **switch** (reserve(characterString)) { 88. //const 89. **case** 0: 90. lexList[lexListIndex].tokenID = constsym; 91. **break**; 92. //var 93. **case** 1: 94. lexList[lexListIndex].tokenID = varsym; 95. **break**; 96. //procedure 97. **case** 2: 98. lexList[lexListIndex].tokenID = procsym; 99. **break**; 100. //call 101. **case** 3: 102. lexList[lexListIndex].tokenID = callsym; 103. **break**; 104. //begin 105. **case** 4: 106. lexList[lexListIndex].tokenID = beginsym; 107. **break**; 108. //end 109. **case** 5: 110. lexList[lexListIndex].tokenID = endsym; 111. **break**; 112. //if 113. **case** 6: 114. lexList[lexListIndex].tokenID = ifsym; 115. **break**; 116. //then 117. **case** 7: 118. lexList[lexListIndex].tokenID = thensym; 119. **break**; 120. //else 121. **case** 8: 122. lexList[lexListIndex].tokenID = elsesym; 123. **break**; 124. //while 125. **case** 9: 126. lexList[lexListIndex].tokenID = whilesym; 127. **break**; 128. //do 129. **case** 10: 130. lexList[lexListIndex].tokenID = dosym; 131. **break**; 132. //read 133. **case** 11: 134. lexList[lexListIndex].tokenID = readsym; 135. **break**; 136. //write 137. **case** 12: 138. lexList[lexListIndex].tokenID = writesym; 139. **break**; 140. //odd 141. **case** 13: 142. lexList[lexListIndex].tokenID = oddsym; 143. **break**; 144. **default**: 145. lexList[lexListIndex].tokenID = identsym; 146. strcpy(lexList[lexListIndex].name, characterString); 147. **break**; 148. } 149. lexListIndex++; 150. } **else** **if** (isdigit(c)) { 151. **int** number = c - '0'; 152. **int** d; 153. **int** place = 1; 154. lookAhead = 1; 156. **while** (isdigit(c = fgetc(ifp))) { 157. **if** (place > NumMaxLength) { 158. printf("Error 2: Number too long.\n"); 159. **while** (isdigit(c = fgetc(ifp))); 160. errorHolder = 1; 161. **break**; 162. } 163. d = c - '0'; 164. number = 10 \* number + d; 165. place++; 166. } 168. //常数不能含有单词 169. **if** (isalpha(c)) { 170. printf("Error 1: Variable does not start with letter.\n"); 171. **while** (isalpha(c = fgetc(ifp)) || isdigit(c)); 172. **continue**; 173. } 175. //处理常数过长 176. **if** (errorHolder == 1) { 177. errorHolder = 0; 178. **continue**; 179. } 181. lexList[lexListIndex].tokenID = numbersym; 182. lexList[lexListIndex++].numberValue = number; 183. } **else** { 184. lookAhead = 0; 185. **switch** (speial(c)) { 186. //+ 187. **case** 0: 188. lexList[lexListIndex++].tokenID = plussym; 189. **break**; 190. //- 191. **case** 1: 192. lexList[lexListIndex++].tokenID = minussym; 193. **break**; 194. //\* 195. **case** 2: 196. lexList[lexListIndex++].tokenID = multsym; 197. **break**; 198. //注释/\*  \*/ 199. **case** 3: 200. c = fgetc(ifp); 201. lookAhead = 1; 202. **if** (c == '\*') { 203. comments = 1; 204. lookAhead = 0; 205. c = fgetc(ifp); 206. **while** (comments) { 207. **if** (c == '\*') { 208. c = fgetc(ifp); 209. **if** (c == '/') 210. comments = 0; 211. } **else** 212. c = fgetc(ifp); 213. } 214. } **else** 215. lexList[lexListIndex++].tokenID = slashsym; 216. **break**; 217. //( 218. **case** 4: 219. lexList[lexListIndex++].tokenID = lparentsym; 220. **break**; 221. //) 222. **case** 5: 223. lexList[lexListIndex++].tokenID = rparentsym; 224. **break**; 225. //= 226. **case** 6: 227. lexList[lexListIndex++].tokenID = eqsym; 228. **break**; 229. //, 230. **case** 7: 231. lexList[lexListIndex++].tokenID = commasym; 232. **break**; 233. //. 234. **case** 8: 235. lexList[lexListIndex++].tokenID = periodsym; 236. **break**; 237. //<> <= < 238. **case** 9: 239. c = fgetc(ifp); 240. **if** (c == '=') { 241. lexList[lexListIndex++].tokenID = leqsym; 242. lookAhead = 0; 243. } **else** { 244. lexList[lexListIndex++].tokenID = lessym; 245. lookAhead = 1; 246. } 247. **break**; 248. //>= > 249. **case** 10: 250. c = fgetc(ifp); 251. **if** (c == '=') { 252. lexList[lexListIndex++].tokenID = geqsym; 253. lookAhead = 0; 254. } **else** { 255. lexList[lexListIndex++].tokenID = gtrsym; 256. lookAhead = 1; 257. } 258. **break**; 259. //; 260. **case** 11: 261. lexList[lexListIndex++].tokenID = semicolonsym; 262. **break**; 263. //:= 264. **case** 12: 265. c = fgetc(ifp); 266. **if** (c == '=') { 267. lexList[lexListIndex++].tokenID = becomessym; 268. lookAhead = 0; 269. } **else** { 270. lookAhead = 1; 271. printf("Error 4: Invalid symbols :.\n"); 272. } 273. **break**; 274. **case** 13: 275. lexList[lexListIndex++].tokenID = neqsym; 276. **break**; 277. **default**: 278. lookAhead = 0; 279. printf("Error 4: Invalid symbols %c.\n",c); 280. **break**; 281. } 282. } 283. **if**(!lookAhead) 284. { 285. c = fgetc(ifp); 286. lookAhead = 1; 287. } 288. } 290. fprintf(ofp,"%d",lexList[0].tokenID); 291. **if**(lexList[0].tokenID == 2) 292. fprintf(ofp,"|%s",lexList[0].name); 293. **else** **if**(lexList[0].tokenID == 3) 294. fprintf(ofp,"|%d",lexList[0].numberValue); 295. **for** (**int** i = 1; i < lexListIndex; i++) 296. { 297. fprintf(ofp, "|%d", lexList[i].tokenID); 298. **if** (lexList[i].tokenID == 2) 299. { 300. fprintf(ofp, "|%s", lexList[i].name); 301. } 302. **else** **if** (lexList[i].tokenID == 3) 303. { 304. fprintf(ofp, "|%d", lexList[i].numberValue); 305. } 306. } 307. fclose(ifp); 308. fclose(ofp); 310. }   4 parse.h   1. // 2. // Created by 62358 on 2020/4/28. 3. // 5. #ifndef COMPILE\_PARSE\_H 6. #define COMPILE\_PARSE\_H 8. #include "data.h" 10. /\*For constants, you must store kind, name and value. 11. For variables, you must store kind, name, L and M. 12. For procedures, you must store kind, name, L and M.\*/ 13. **struct** symbol 14. { 15. **int** kind;       // const = 1, var = 2, proc = 3 16. **char** name[12];  // name up to 11 chars 17. **int** val;        // number (ASCII value) 18. **int** level;      // L level 19. **int** addr;       // M address 20. }; 22. **void** program();             //程序 23. **void** block();               //分程序 24. **void** constDeclaration();    //常量说明部分 25. **void** constDefinition();     //常量定义 26. **void** varDeclaration();      //变量说明部分 27. **void** varDefinition();       //变量定义 28. **void** procedureDeclaration();//过程说明部分 29. **void** procedureHead();       //过程首部 30. **void** statement();           //语句 31. **void** condition();           //条件 32. **void** expression();          //表达式 33. **void** term();                //项 34. **void** factor();              //因子 36. **void** assignment();          //赋值语句 37. **void** callProcedure();       //过程调用语句 38. **void** conditionStatement();  //条件语句 39. **void** compoundStatement();   //复合语句 40. **void** loopStatement();       //循环语句 41. **void** writeStatement();      //写语句 42. **void** readStatement();       //读语句 43. **void** identifier(); 45. **void** enter(**int** kind); 46. **void** error(**int** errorCase); 47. **void** emit(ins\_type ins, **int** l, **int** m); 48. **int** position(); 50. **void** getNextToken(); 52. **void** printBlank(); 54. #endif //COMPILE\_PARSE\_H   5 parse.cpp   1. // 2. // Created by 62358 on 2020/4/28. 3. // 5. /\*Clayton Walker 6. COP 3402 7. Due: 7/21/13 8. Parser 9. \*/ 11. #include <stdio.h> 12. #include <string.h> 13. #include <stdlib.h> 14. #include <ctype.h> 15. #include "data.h" 16. #include "parse.h" 18. symbol table[MAX\_SYMBOL\_TABLE\_SIZE];    //符号表 19. instruction code[CODE\_SIZE];            //目标代码 21. **int** tx, cx, lev = -1;                   //tx表示符号表序号，cx表示代码表序号，lev表示当前层数 22. //tx已经占据，cx没有占据 24. **int** dx = 3;                             //表示当前过程偏移 26. **int** lexemeListIndex = 0;                //词法分析结果当前读入位置 27. tokenStruct curToken;                   //最新读入tokenStruct 29. **int** blank = 0;                          //打印空格数 30. **FILE** \*ofp; 31. **FILE** \*codeFp; 32. **FILE** \*tableFp; 34. **int** mainCode; 36. **void** parse(**char** \*parsefilename,**char** \*tablefilename,**char** \*codefilename) { 37. memset(table,0, **sizeof**(table)); 38. tx = 0, cx = 0, lev = -1; 39. dx = 3; 40. lexemeListIndex = 0; 41. blank = 0; 42. table[0].level = -1; 44. ofp = fopen(parsefilename, "w"); 45. codeFp = fopen(codefilename,"w"); 46. tableFp = fopen(tablefilename,"w"); 48. program(); 50. **for** (**int** i = 0; i <= tx; i++) { 51. fprintf(tableFp,"kind:%d\tval:%d\tlev:%d\taddr:%d\tname:%s\n", table[i].kind, table[i].val, table[i].level, 52. table[i].addr, table[i].name); 53. } 55. **for** (**int** i=0; i<cx;i++) { 56. fprintf(codeFp,"%d %d %d\n", code[i].ins, code[i].l, code[i].m); 57. } 59. fclose(ofp); 60. fclose(codeFp); 61. } 63. **void** program() { 64. printBlank(); 65. fprintf(ofp, "program\n"); 66. blank++; 68. **int** cx0 = cx; 69. emit(JMP, 0, 0); 71. getNextToken(); 72. block(); 74. code[cx0].m = mainCode; 76. **if** (curToken.tokenID != periodsym) 77. error(1);       //Period expected. 79. printBlank(); 80. fprintf(ofp, ".\n"); 82. blank--; 83. } 85. **void** block() { 86. printBlank(); 87. fprintf(ofp, "block\n"); 88. blank++; 90. lev++; 91. **if** (lev > MAX\_LEXI\_LEVELS) 92. error(2);      //errors for above max lexi level 94. **int** oldDx = dx; 95. dx = 3; 97. **int** tx0 = tx; 99. **do** { 100. **if** (curToken.tokenID == constsym) { 101. constDeclaration(); 102. } 103. **if** (curToken.tokenID == varsym) { 104. varDeclaration(); 105. } 106. **while** (curToken.tokenID == procsym) { 107. procedureDeclaration(); 108. } 109. } **while** ((curToken.tokenID == constsym) || (curToken.tokenID == varsym) || (curToken.tokenID == procsym)); 111. **if** (lev == 0) 112. mainCode = cx; 114. table[tx0].addr = cx; 116. emit(**INT**, 0, dx); 118. statement(); 120. emit(OPR, 0, 0); 122. dx = oldDx; 123. lev--; 125. blank--; 126. } 128. **void** constDeclaration() { 129. printBlank(); 130. fprintf(ofp, "constDeclaration\n"); 131. blank++; 133. printBlank(); 134. fprintf(ofp, "const\n"); 136. getNextToken(); 137. constDefinition(); 138. **while** (curToken.tokenID == commasym) { 139. printBlank(); 140. fprintf(ofp, ",\n"); 142. getNextToken(); 143. constDefinition(); 144. } 145. **if** (curToken.tokenID == semicolonsym) { 146. printBlank(); 147. fprintf(ofp, ";\n"); 149. getNextToken(); 150. } **else** { 151. error(3);       //常量声明部分缺少; 152. } 154. blank--; 155. } 157. **void** constDefinition() { 158. printBlank(); 159. fprintf(ofp, "constDefinition\n"); 160. blank++; 162. **if** (curToken.tokenID == identsym) { 163. printBlank(); 164. fprintf(ofp, "%s\n", curToken.name); 166. getNextToken(); 167. **if** ((curToken.tokenID == eqsym) || (curToken.tokenID == becomessym)) { 168. **if** (curToken.tokenID == becomessym) 169. error(6);       //Use = instead of := in constDefinition 171. printBlank(); 172. fprintf(ofp, ":=\n"); 174. getNextToken(); 175. **if** (curToken.tokenID == numbersym) { 176. printBlank(); 177. fprintf(ofp, "%d\n", curToken.numberValue); 179. enter(1); 180. getNextToken(); 181. } **else** { 182. error(8);       //常量定义赋值的不是整数 183. } 185. } **else** { 186. error(7);           //常量未赋值 187. } 188. } **else** { 189. error(5);       //常量定义缺少标识符 190. } 192. blank--; 193. } 195. **void** varDeclaration() { 196. printBlank(); 197. fprintf(ofp, "varDeclaration\n"); 198. blank++; 200. printBlank(); 201. fprintf(ofp, "var\n"); 203. getNextToken(); 204. varDefinition(); 205. **while** (curToken.tokenID == commasym) { 206. printBlank(); 207. fprintf(ofp, ",\n"); 209. getNextToken(); 210. varDefinition(); 211. } 212. **if** (curToken.tokenID == semicolonsym) { 213. printBlank(); 214. fprintf(ofp, ";\n"); 216. getNextToken(); 217. } **else** { 218. error(4); //Semicolon or comma missing. 219. } 221. blank--; 222. } 224. **void** varDefinition() { 225. **if** (curToken.tokenID == identsym) { 226. printBlank(); 227. fprintf(ofp, "%s\n", curToken.name); 229. enter(2); 230. getNextToken(); 231. } **else** 232. error(9); 233. } 235. **void** procedureDeclaration() { 236. printBlank(); 237. fprintf(ofp, "procedureDeclaration\n"); 238. blank++; 240. procedureHead(); 241. block(); 243. **if** (curToken.tokenID == semicolonsym) { 244. printBlank(); 245. fprintf(ofp, ";\n"); 247. getNextToken(); 248. } **else** 249. error(33);                  //; not find in procedureDeclaration() 251. blank--; 252. } 254. **void** procedureHead() { 255. printBlank(); 256. fprintf(ofp, "procedureHead\n"); 257. blank++; 259. printBlank(); 260. fprintf(ofp, "procedure\n"); 262. getNextToken(); 263. **if** (curToken.tokenID == identsym) { 264. printBlank(); 265. fprintf(ofp, "%s\n", curToken.name); 267. enter(3); 268. getNextToken(); 270. **if** (curToken.tokenID == semicolonsym) { 271. printBlank(); 272. fprintf(ofp, ";\n"); 274. getNextToken(); 275. } **else** 276. error(32);                  //; not find in procedureHead() 277. } **else** 278. error(31);                      //identsym not find in procedureHead() 280. blank--; 281. } 283. **void** statement() { 284. printBlank(); 285. fprintf(ofp, "statement\n"); 286. blank++; 288. //赋值语句 289. **if** (curToken.tokenID == identsym) { 290. assignment(); 291. } **else** **if** (curToken.tokenID == callsym) { 292. callProcedure(); 293. } **else** **if** (curToken.tokenID == ifsym) { 294. conditionStatement(); 295. } **else** **if** (curToken.tokenID == beginsym) { 296. compoundStatement(); 297. } **else** **if** (curToken.tokenID == whilesym) { 298. loopStatement(); 299. } **else** **if** (curToken.tokenID == writesym) { 300. writeStatement(); 301. } **else** **if** (curToken.tokenID == readsym) { 302. readStatement(); 303. } 305. blank--; 306. } 308. **void** assignment() { 309. printBlank(); 310. fprintf(ofp, "assignment\n"); 311. blank++; 313. printBlank(); 314. fprintf(ofp, "%s\n", curToken.name); 316. **int** i = position(); 317. **if** (i == 0) 318. error(10);          //identsym not find in table in assignment 319. **else** **if** (table[i].kind != 2) 320. error(11);          //identsym in assignment must be var 322. getNextToken(); 323. **if** (curToken.tokenID == becomessym) { 324. printBlank(); 325. fprintf(ofp, ":=\n"); 327. getNextToken(); 328. expression(); 329. } **else** 330. error(12);          //becomessym not find in assignment 332. emit(STO, lev - table[i].level, table[i].addr); 334. blank--; 335. } 337. **void** callProcedure() { 338. printBlank(); 339. fprintf(ofp, "callProcedure\n"); 340. blank++; 342. printBlank(); 343. fprintf(ofp, "call\n"); 345. getNextToken(); 346. **if** (curToken.tokenID != identsym) 347. error(13);          //identsym not find in callProcedure 348. **else** { 349. printBlank(); 350. fprintf(ofp, "%s\n", curToken.name); 352. **int** i = position(); 353. **if** (i == 0) 354. error(14);      //identsym not find in table in callProcedure 355. **else** **if** (table[i].kind != 3) 356. error(15);      //identsym called must be procedure 357. **else** { 358. getNextToken(); 360. emit(CAL, lev - table[i].level, table[i].addr); 361. } 362. } 364. blank--; 365. } 367. **void** conditionStatement() { 368. printBlank(); 369. fprintf(ofp, "conditionStatement\n"); 370. blank++; 372. printBlank(); 373. fprintf(ofp, "if\n"); 375. getNextToken(); 376. condition(); 378. **if** (curToken.tokenID == thensym) { 379. printBlank(); 380. fprintf(ofp, "then\n"); 382. getNextToken(); 383. } **else** 384. error(16);                  //then not find in conditionStatement 386. **int** cx0 = cx;                           //回填 387. emit(JPC, 0, 0); 389. statement(); 391. code[cx0].m = cx; 393. blank--; 394. } 396. **void** compoundStatement() { 397. printBlank(); 398. fprintf(ofp, "compoundStatement\n"); 399. blank++; 401. printBlank(); 402. fprintf(ofp, "begin\n"); 404. getNextToken(); 406. statement(); 407. **while** (curToken.tokenID == semicolonsym) { 408. printBlank(); 409. fprintf(ofp, ";\n"); 411. getNextToken(); 412. statement(); 413. } 415. **if** (curToken.tokenID == endsym) { 416. printBlank(); 417. fprintf(ofp, "end\n"); 419. getNextToken(); 420. } **else** { 421. error(17);                  //end not find in compoundStatement() 422. } 424. blank--; 425. } 427. **void** loopStatement() { 428. printBlank(); 429. fprintf(ofp, "loopStatement\n"); 430. blank++; 432. printBlank(); 433. fprintf(ofp, "while\n"); 435. getNextToken(); 437. **int** cx0 = cx;           //标号 439. condition(); 441. **int** cx1 = cx;           //回填 442. emit(JPC, 0, 0); 444. **if** (curToken.tokenID == dosym) { 445. printBlank(); 446. fprintf(ofp, "do\n"); 448. getNextToken(); 449. } **else** 450. error(18);                      //do not find in loopStatement() 452. statement(); 454. emit(JMP, 0, cx0); 456. code[cx1].m = cx; 458. blank--; 459. } 461. **void** writeStatement() { 462. printBlank(); 463. fprintf(ofp, "writeStatement\n"); 464. blank++; 466. printBlank(); 467. fprintf(ofp, "write\n"); 469. getNextToken(); 471. **if** (curToken.tokenID == lparentsym) { 472. printBlank(); 473. fprintf(ofp, "(\n"); 475. getNextToken(); 476. expression(); 478. emit(OPR, 0, WRITE); 480. **while** (curToken.tokenID == commasym) { 481. printBlank(); 482. fprintf(ofp, ",\n"); 484. getNextToken(); 485. expression(); 487. emit(OPR, 0, WRITE); 488. } 490. **if** (curToken.tokenID == rparentsym) { 491. printBlank(); 492. fprintf(ofp, ")\n"); 494. getNextToken(); 495. } **else** 496. error(21);              //) not find in writeStatement() 497. } **else** 498. error(20);                  //( not find in writeStatement() 500. blank--; 501. } 503. **void** readStatement() { 504. printBlank(); 505. fprintf(ofp, "readStatement\n"); 506. blank++; 508. printBlank(); 509. fprintf(ofp, "read\n"); 511. getNextToken(); 513. **if** (curToken.tokenID == lparentsym) { 514. printBlank(); 515. fprintf(ofp, "(\n"); 517. getNextToken(); 518. identifier(); 520. **while** (curToken.tokenID == commasym) { 521. printBlank(); 522. fprintf(ofp, ",\n"); 524. getNextToken(); 525. identifier(); 526. } 528. **if** (curToken.tokenID == rparentsym) { 529. printBlank(); 530. fprintf(ofp, ")\n"); 532. getNextToken(); 533. } **else** 534. error(25);              //) not find in readStatement() 535. } **else** 536. error(24);                  //( not find in readStatement() 538. blank--; 539. } 541. **void** identifier() { 542. **if** (curToken.tokenID == identsym) { 543. printBlank(); 544. fprintf(ofp, "%s\n", curToken.name); 546. **int** i = position(); 547. **if** (i == 0) 548. error(22);                      //identifier is not find in table in identifier() 549. **else** **if** (table[i].kind != 2) 550. error(23);                      //identifier is not var in identifier() 552. getNextToken(); 554. emit(OPR, 0, READ); 555. emit(STO, lev - table[i].level, table[i].addr); 556. } **else** { 557. error(19);                          //identifier not find in identifier() 558. } 559. } 561. **void** condition() { 562. printBlank(); 563. fprintf(ofp, "condition\n"); 564. blank++; 566. **if** (curToken.tokenID == oddsym) { 567. printBlank(); 568. fprintf(ofp, "odd\n"); 570. getNextToken(); 571. expression(); 573. emit(OPR, 0, ODD); 574. } **else** { 575. expression(); 577. **if** ((curToken.tokenID != eqsym) && (curToken.tokenID != neqsym) && (curToken.tokenID != lessym) && 578. (curToken.tokenID != leqsym) && (curToken.tokenID != gtrsym) && (curToken.tokenID != geqsym)) 579. error(26);                  //relationSwitch not find in condition() 580. **else** { 581. token\_type relop = curToken.tokenID; 583. printBlank(); 584. fprintf(ofp, "%s\n", strPrint[curToken.tokenID]); 586. getNextToken(); 587. expression(); 589. **switch** (relop) { 590. **case** eqsym: 591. emit(OPR, 0, EQ); 592. **break**; 593. **case** neqsym: 594. emit(OPR, 0, NE); 595. **break**; 596. **case** lessym: 597. emit(OPR, 0, LT); 598. **break**; 599. **case** leqsym: 600. emit(OPR, 0, LE); 601. **break**; 602. **case** gtrsym: 603. emit(OPR, 0, GT); 604. **break**; 605. **case** geqsym: 606. emit(OPR, 0, GE); 607. **break**; 608. **default**: 609. **break**; 610. } 611. } 612. } 614. blank--; 615. } 617. **void** expression() { 618. printBlank(); 619. fprintf(ofp, "expression\n"); 620. blank++; 622. **if** (curToken.tokenID == plussym || curToken.tokenID == minussym) { 623. printBlank(); 624. fprintf(ofp, "%s\n", strPrint[curToken.tokenID]); 626. token\_type relop = curToken.tokenID; 628. getNextToken(); 629. term(); 631. **if** (relop == minussym) 632. emit(OPR, 0, MINUS); 633. } **else** { 634. term(); 635. } 637. **while** (curToken.tokenID == plussym || curToken.tokenID == minussym) { 638. printBlank(); 639. fprintf(ofp, "%s\n", strPrint[curToken.tokenID]); 641. token\_type relop = curToken.tokenID; 643. getNextToken(); 644. term(); 646. **if** (relop == plussym) 647. emit(OPR, 0, ADD); 648. **else** 649. emit(OPR, 0, SUB); 650. } 652. blank--; 653. } 655. **void** term() { 656. printBlank(); 657. fprintf(ofp, "term\n"); 658. blank++; 660. factor(); 661. **while** (curToken.tokenID == multsym || curToken.tokenID == slashsym) { 662. printBlank(); 663. fprintf(ofp, "%s\n", strPrint[curToken.tokenID]); 665. token\_type relop = curToken.tokenID; 667. getNextToken(); 668. factor(); 670. **if** (relop == multsym) 671. emit(OPR, 0, MUL); 672. **else** 673. emit(OPR, 0, DIV); 674. } 676. blank--; 677. } 679. **void** factor() { 680. printBlank(); 681. fprintf(ofp, "factor\n"); 682. blank++; 684. **if** (curToken.tokenID == identsym) { 685. printBlank(); 686. fprintf(ofp, "%s\n", curToken.name); 688. **int** i = position(); 689. **if** (i == 0) 690. error(27);                  //identsym not find in table in factor() 691. **else** **if** (table[i].kind == 3) 692. error(28);                  //identsym in factor() cannot be procedure 693. **else** **if** (table[i].kind == 1) 694. emit(LIT,0,table[i].val); 695. **else** **if** (table[i].kind == 2) 696. emit(LOD,lev - table[i].level,table[i].addr); 698. getNextToken(); 699. } **else** **if** (curToken.tokenID == numbersym) { 700. printBlank(); 701. fprintf(ofp, "%d\n", curToken.numberValue); 703. emit(LIT,0,curToken.numberValue); 705. getNextToken(); 706. } **else** **if** (curToken.tokenID == lparentsym) { 707. printBlank(); 708. fprintf(ofp, "(\n"); 710. getNextToken(); 711. expression(); 713. **if** (curToken.tokenID == rparentsym) { 714. printBlank(); 715. fprintf(ofp, ")\n"); 717. getNextToken(); 718. } **else** 719. error(29);                      //) not find in factor() 720. } **else** 721. error(30);                          //factor() cannot be right 723. blank--; 724. } 726. **void** getNextToken() { 727. curToken.tokenID = lexList[lexemeListIndex].tokenID; 729. //var 730. **if** (curToken.tokenID == 2) 731. strcpy(curToken.name, lexList[lexemeListIndex].name); 732. **else** **if** (curToken.tokenID == 3) 733. curToken.numberValue = lexList[lexemeListIndex].numberValue; 735. lexemeListIndex++; 736. } 738. **void** printBlank() { 739. **for** (**int** i = 0; i < blank; i++) 740. fprintf(ofp, "\t"); 741. } 743. **void** enter(**int** kind) { 744. tx++; 746. strcpy(table[tx].name, curToken.name); 747. table[tx].kind = kind; 749. //const 750. **if** (kind == 1) 751. table[tx].val = curToken.numberValue; 752. **else** **if** (kind == 2) { 753. table[tx].level = lev; 754. table[tx].addr = dx; 755. dx++; 756. } **else** **if** (kind == 3) 757. table[tx].level = lev; 758. } 760. **int** position() { 761. **int** curLev = lev; 763. **for** (**int** i = tx; i > 0; i--) { 764. **if** (table[i].kind == 3 && table[i].level == curLev - 1) 765. curLev--; 767. **if** (table[i].level > curLev) 768. **continue**; 770. **if** (strcmp(table[i].name, curToken.name) == 0) 771. **return** i; 772. } 773. **return** 0; 774. } 776. **void** error(**int** errorCase) { 777. **switch** (errorCase) { 778. **case** 1: 779. printf("Error 1: "); 780. printf("Period(.) expected in end of program\n");               //program不以句号结尾 781. **break**; 782. **case** 2: 783. printf("Error 2: "); 784. printf("Level is larger than the maximum allowed lexicographical levels!\n");   //层数过大 785. **break**; 786. **case** 3: 787. printf("Error 3: "); 788. printf("Semicolon(;) missing in constDeclaration.\n");     //常量声明部分缺少; 789. **break**; 790. **case** 4: 791. printf("Error 4: "); 792. printf("Semicolon(;) missing in varDeclaration.\n");        //变量声明部分缺少; 793. **break**; 794. **case** 5: 795. printf("Error 5: "); 796. printf("Identifier missing in constDefination.\n");         //常量定义缺少标识符 797. **break**; 798. **case** 6: 799. printf("Error 6: "); 800. printf("Use = instead of := in constDefination.\n"); 801. **break**; 802. **case** 7: 803. printf("Error 7: "); 804. printf("Constant not assigned in constDefination.\n");      //常量定义未赋值 805. **break**; 806. **case** 8: 807. printf("Error 8: "); 808. printf("Constant not assigned by num in constDefination.\n");    //常量定义赋值的不是整数 809. **break**; 810. **case** 9: 811. printf("Error 9: "); 812. printf("Identifier missing in varDefination.\n");               //变量定义缺少标识符 813. **break**; 814. **case** 10: 815. printf("Error 10: "); 816. printf("identsym not find in table in assignment.\n");          //赋值语句中变量没有在表中找到 817. **break**; 818. **case** 11: 819. printf("Error 11: "); 820. printf("identsym in assignment must be var.\n");                //赋值语句中必须为变量 821. **break**; 822. **case** 12: 823. printf("Error 12: "); 824. printf("becomessym not find in assignment.\n");                 //赋值语句中没有:= 825. **break**; 826. **case** 13: 827. printf("Error 13: "); 828. printf("identsym not find in callProcedure.\n");                //过程调用必须有过程名 829. **break**; 830. **case** 14: 831. printf("Error 14: "); 832. printf("identsym not find in table in callProcedure.\n");       //过程名没有找到 833. **break**; 834. **case** 15: 835. printf("Error 15: "); 836. printf("identsym called must be procedure in callProcedure().\n");                 //调用的过程不是过程 837. **break**; 838. **case** 16: 839. printf("Error 16: "); 840. printf("then not find in conditionStatement().\n");                //条件语句中没有then 841. **break**; 842. **case** 17: 843. printf("Error 17: "); 844. printf("end not find in compoundStatement().\n");                   //复合语句中没有end 845. **break**; 846. **case** 18: 847. printf("Error 18: "); 848. printf("do not find in loopStatement().\n");                        //循环语句中没有找到do 849. **break**; 850. **case** 19: 851. printf("Error 19: "); 852. printf("identifier not find in identifier().\n");                   //identifier()没有标识符 853. **break**; 854. **case** 20: 855. printf("Error 20: "); 856. printf("( not find in writeStatement().\n");                        //写语句中没找到( 857. **break**; 858. **case** 21: 859. printf("Error 21: "); 860. printf(") not find in writeStatement().\n");                        //写语句中没找到) 861. **break**; 862. **case** 22: 863. printf("Error 22: "); 864. printf("identifier is not find in table in identifier().\n");       //identifier()中标识符查表没有找到 865. **break**; 866. **case** 23: 867. printf("Error 23: "); 868. printf("identifier is not var in identifier().\n");                 //identifier()标识符不是var 869. **break**; 870. **case** 24: 871. printf("Error 24: "); 872. printf("( not find in readStatement().\n");                         //读语句中没有( 873. **break**; 874. **case** 25: 875. printf("Error 25: "); 876. printf(") not find in readStatement().\n");                         //读语句中没有) 877. **break**; 878. **case** 26: 879. printf("Error: 26 "); 880. printf("relationSwitch not find in condition().\n");                //条件中没有关系运算符 881. **break**; 882. **case** 27: 883. printf("Error: 27 "); 884. printf("identsym not find in table in factor().\n");                //因子 标识符没有在表中 885. **break**; 886. **case** 28: 887. printf("Error: 28 "); 888. printf("identsym in factor() cannot be procedure.\n");              //因子 标识符不能为过程 889. **break**; 890. **case** 29: 891. printf("Error: 29 "); 892. printf(") not find in factor().\n");              //因子 )没有 893. **break**; 894. **case** 30: 895. printf("Error: 30 "); 896. printf("factor() cannot be right.\n");              //因子 不匹配 897. **break**; 898. **case** 31: 899. printf("Error: 31 "); 900. printf("identsym not find in procedureHead().\n");              //过程头中标识符找不到 901. **break**; 902. **case** 32: 903. printf("Error: 32 "); 904. printf("; not find in procedureHead().\n");              //过程头中 ;找不到 905. **break**; 906. **case** 33: 907. printf("Error: 33 "); 908. printf("; not find in procedureDeclaration().\n");              //过程说明部分 ;找不到 909. **break**; 910. **default**: 911. **break**; 912. } 913. //stops program when error occurs 914. printf("%d\t%s\n", curToken.tokenID, strPrint[curToken.tokenID]); 915. exit(1); 916. } 918. **void** emit(ins\_type ins, **int** l, **int** m) { 919. **if** (cx >= CODE\_SIZE) { 920. printf("Program too long! cx > CODE\_SIZE\n"); 921. exit(1); 922. } **else** { 923. code[cx].ins = ins; 924. code[cx].l = l; 925. code[cx].m = m; 926. cx++; 927. } 928. }   6 vm.h   1. // 2. // Created by 62358 on 2020/5/26. 3. // 5. #ifndef TEST\_VM\_H 6. #define TEST\_VM\_H 8. #define MAX\_STACK\_HEIGHT 2000 10. **void** executeCycle(); 12. **int** base(); 14. **void** opr(); 16. **void** printStackFrame(); 18. **void** printIr(); 20. **void** printReg(); 22. **void** printStack(); 24. **char**\* insStr[] = {"","LIT","LOD","STO","CAL","INT","JMP","JPC","OPR"}; 25. **char**\* oprStr[] = {"RET","ADD","SUB","MUL","DIV","GT","LT","GE","LE","NE","EQ","WRITE","READ","MINUS","ODD"}; 27. #endif //TEST\_VM\_H   7 vm.cpp   1. // 2. // Created by 62358 on 2020/5/26. 3. // 5. #include <cstdio> 6. #include <cstring> 7. #include "data.h" 8. #include "vm.h" 10. **int** sp;     //栈顶 11. **int** bp;     //基址 12. **int** pc;     //程序地址寄存器 14. **int** codeLength;                             //目标代码长度 15. instruction arrayStruct[CODE\_SIZE];         //存放读入目标代码 17. instruction ir;                             //指令寄存器 19. **int** stack[MAX\_STACK\_HEIGHT];                //堆栈 21. **void** vm(**char** \*codefilename) { 22. pc = 0; 23. bp = 0; 24. sp = -1; 25. memset(stack,0, **sizeof**(stack)); 27. **FILE** \*ifp; 28. ifp = fopen(codefilename, "r"); 29. codeLength = 0; 31. **while** (fscanf(ifp, "%d %d %d", &arrayStruct[codeLength].ins, &arrayStruct[codeLength].l, 32. &arrayStruct[codeLength].m) != EOF) { 33. codeLength++; 34. } 36. //    printReg(); 37. //    printStack(); 39. **for** (; pc < codeLength - 1;) { 40. ir = arrayStruct[pc++]; 42. //        printIr(); 44. executeCycle(); 46. //        printReg(); 47. //        printStack(); 48. } 49. } 51. **void** executeCycle() { 52. **switch** (ir.ins) { 53. **case** LIT: 54. stack[++sp] = ir.m; 55. **break**; 56. **case** LOD: 57. stack[++sp] = stack[base() + ir.m]; 58. **break**; 59. **case** STO: 60. stack[base() + ir.m] = stack[sp--]; 61. **break**; 62. **case** CAL: 63. stack[sp + 1] = base();         //SL 64. stack[sp + 2] = bp;             //DL 65. stack[sp + 3] = pc;             //返回地址 67. bp = sp + 1; 68. pc = ir.m; 69. **break**; 70. **case** **INT**: 71. sp = sp + ir.m; 72. **break**; 73. **case** JMP: 74. pc = ir.m; 75. **break**; 76. **case** JPC: 77. **if** (!stack[sp--]) 78. pc = ir.m; 79. **break**; 80. **case** OPR: 81. opr(); 82. **break**; 83. **default**: 84. printf("Illegal INS!\n"); 85. } 86. } 88. **void** opr() { 89. **int** temp; 90. **switch** (ir.m) { 91. **case** ADD: 92. temp = stack[sp - 1] + stack[sp]; 93. stack[--sp] = temp; 94. **break**; 95. **case** SUB: 96. temp = stack[sp - 1] - stack[sp]; 97. stack[--sp] = temp; 98. **break**; 99. **case** DIV: 100. temp = stack[sp - 1] / stack[sp]; 101. stack[--sp] = temp; 102. **break**; 103. **case** MINUS: 104. stack[sp] = -stack[sp]; 105. **break**; 106. **case** MUL: 107. temp = stack[sp - 1] \* stack[sp]; 108. stack[--sp] = temp; 109. **break**; 110. **case** EQ: 111. temp = (stack[sp - 1] == stack[sp]); 112. stack[--sp] = temp; 113. **break**; 114. **case** NE: 115. temp = (stack[sp - 1] != stack[sp]); 116. stack[--sp] = temp; 117. **break**; 118. **case** GE: 119. temp = (stack[sp - 1] >= stack[sp]); 120. stack[--sp] = temp; 121. **break**; 122. **case** GT: 123. temp = (stack[sp - 1] > stack[sp]); 124. stack[--sp] = temp; 125. **break**; 126. **case** LE: 127. temp = (stack[sp - 1] <= stack[sp]); 128. stack[--sp] = temp; 129. **break**; 130. **case** LT: 131. temp = (stack[sp - 1] < stack[sp]); 132. stack[--sp] = temp; 133. **break**; 134. **case** ODD: 135. temp = stack[sp] % 2; 136. stack[sp] = temp; 137. **break**; 138. **case** WRITE: 139. printf("%d\n", stack[sp--]); 140. **break**; 141. **case** READ: 142. scanf("%d", &stack[++sp]); 143. **break**; 144. **case** 0: 145. sp = bp - 1; 146. pc = stack[sp + 3]; 147. bp = stack[sp + 2]; 148. **break**; 149. **default**: 150. printf("Illegal OPR!\n"); 151. } 152. } 154. **int** base() { 155. **int** l = ir.l; 156. **int** SL = bp; 158. **while** (l > 0) { 159. SL = stack[SL]; 160. l--; 161. } 162. **return** SL; 163. } 165. **void** printStackFrame() { 166. printf("pc:%d\t", pc); 167. printf("bp:%d\t", bp); 168. printf("sp:%d\t", sp); 170. printf("stack:("); 171. **int** max = sp >= bp ? sp : bp; 172. **for** (**int** i = 0; i <= max; i++) 173. **if** (i) 174. printf(" %d", stack[i]); 175. **else** 176. printf("%d", stack[i]); 177. printf(")\n"); 179. printf("ir:(%d %d %d)\n", ir.ins, ir.l, ir.m); 180. } 182. **void** printIr()              //输出指令 183. { 184. printf("ir:(%d %d %d)\t", ir.ins, ir.l, ir.m); 185. **if**(ir.ins == OPR) 186. printf("(%s %d %s)\n", insStr[ir.ins], ir.l, oprStr[ir.m]); 187. **else** 188. printf("(%s %d %d)\n", insStr[ir.ins], ir.l, ir.m); 189. } 191. **void** printReg()             //输出寄存器值 192. { 193. printf("pc:%d\t", pc); 194. printf("bp:%d\t", bp); 195. printf("sp:%d\t", sp); 196. } 198. **void** printStack()           //输出堆栈信息 199. { 200. printf("stack:("); 201. **int** max = sp >= bp ? sp : bp; 202. **for** (**int** i = 0; i <= max; i++) 203. **if** (i) 204. printf(" %d", stack[i]); 205. **else** 206. printf("%d", stack[i]); 207. printf(")\n"); 208. }   8 main.cpp   1. #include <iostream> 3. #include "data.h" 4. #include <cstring> 6. **void** lextest() { 7. **char** c1[30] = "input\\test0.txt"; 8. **char** c2[30] = "lexoutput\\lexoutput0.txt"; 9. **char** c3[30] = "lextest\\lexoutput0.txt"; 11. **for** (**int** i = 0; i < 10; i++) { 12. c1[10] = i + '0'; 13. c2[19] = i + '0'; 14. c3[17] = i + '0'; 16. lex(c1, c2); 18. **FILE** \*file1 = fopen(c2, "r"); 19. **FILE** \*file2 = fopen(c3, "r"); 20. **while** (fgetc(file1) != fgetc(file2)) { 22. printf("false"); 23. **return**; 24. } 25. printf("true\n"); 26. fclose(file1); 27. fclose(file2); 28. } 29. } 31. **int** main() { 33. **char** \*c1[6] = {"input\\PL0\_code.in", "input\\PL0\_code0.in", "input\\PL0\_code1.in", "input\\PL0\_code2.in", 34. "input\\PL0\_code3.in","input\\PL0\_code4.in"}; 35. **char** \*c2[6] = {"lex\\PL0\_code.out", "lex\\PL0\_code0.out", "lex\\PL0\_code1.out", 36. "lex\\PL0\_code2.out", "lex\\PL0\_code3.out","lex\\PL0\_code4.out"}; 37. **char** \*c3[6] = {"parse\\PL0\_code.out", "parse\\PL0\_code0.out", "parse\\PL0\_code1.out", "parse\\PL0\_code2.out", 38. "parse\\PL0\_code3.out","parse\\PL0\_code4.out"}; 39. **char** \*c4[6] = {"table\\PL0\_code.out", "table\\PL0\_code0.out", "table\\PL0\_code1.out", "table\\PL0\_code2.out", 40. "table\\PL0\_code3.out","table\\PL0\_code4.out"}; 41. **char** \*c5[6] = {"code\\PL0\_code.out", "code\\PL0\_code0.out", "code\\PL0\_code1.out", "code\\PL0\_code2.out", 42. "code\\PL0\_code3.out","code\\PL0\_code4.out"}; 44. //    for(int i = 0;i < 6;i++) 45. //    { 46. //        lex(c1[i], c2[i]); 47. //        parse(c3[i],c4[i],c5[i]); 48. //    } 50. vm("code\\PL0\_code3.out");   54. **return** 0; 55. } | | |