**《编译原理》实验报告**

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| 年级、专业、班级 | | 2019级计算机科学与技术01班 | | 姓名 | 谢双骏 |
| 实验题目 | 目标代码生成程序的设计与实现 | | | | |
| 实验时间 | 2022.5.7 | | 实验地点 | DS3401 | |
| 实验成绩 |  | | 实验性质 | □验证性 □设计性 ■综合性 | |
| 教师评价：  □算法/实验过程正确；□源程序/实验内容提交 □程序结构/实验步骤合理；  □实验结果正确； ☑语法、语义正确； □报告规范；  其他：  评价教师签名： | | | | | |
| 1. 实验目的   在词法分析、语法分析及语义分析的接触上，为编译器实现语义分析、代码生成功能。 | | | | | |
| 1. 实验项目内容 2. 实验内容   需根据文法规则及语义约定，采用自顶向下的语法制导翻译技术，进行语义分析并生成目标代码（**MIPS** 形式）。完成编译器，将源文件（统一命名为testfile.txt）编译生成MIPS汇编并输出到文件（统一命名为mips.txt），具体要求包括：  a）需自行设计四元式中间代码，再从中间代码生成MIPS汇编，请设计实现输出中间代码的有关函数，本次作业不考核，后续会有优化前后中间代码的输出及评判(输出文件命名为学号\_姓名\_优化前/后中间代码.txt)。  b）若选择此项任务，后续可参加竞速排序，请提前预留代码优化有关的接口，并设计方便切换开启/关闭优化的模式  c）自行调试时，可使用Mars仿真器（使用方法见“Mars仿真器使用说明.docx”），提交到平台的编译器只需要能按统一的要求生成MIPS汇编代码文件即可。  d）**此类提交的编译器请仅读取testfile.txt文件并生成相应的MIPS代码，编译器自身不要读入标准输入中的内容。**  【输入形式】testfile. txt为符合文法要求的测试程序，另外可能存在来自于标准输入的数据。  【输出形式】 按如上要求将目标代码生成结果输出至mips.txt中，中文字符的编码格式要求是UTF-8。  【特别说明】   1. 本次作业是为了让同学们尽快实现一个完整的编译器，测试程序中仅涉及常量说明、变量说明、读语句、写语句、赋值语句，无函数定义及调用，无数组声明及引用。 2. 本次作业仅用正确的测试程序进行考核，但产生的编译器应具有错误处理能力。 3. 本次作业未要求优化生成代码，实现优化的同学会有加分。请在报告中写明采用优化方法和优化测试样例，建议尝试提交竞速排名作业进行测试。 | | | | | |
| 1. 实验过程或算法（源程序） 2. 实验概述   本实验完成了根据语义分析的结果输出中间代码和目标代码的程序。可以处理常量说明、变量说明、读语句、写语句、赋值语句，无函数定义及调用，无数组声明及引用的程序，可以通过平台所提供的测试样例，并且目标代码可以在MARS上正确执行。中间和目标代码结果分别位于mid\_code.txt和mips.txt中。  生成中间代码的代码主要在code\_gener.cpp文件中，建立在实验三所实现的语义分析器之上。因为语义分析器是基于递归调用的方法实现的，而中间代码主要针对赋值、读、写等语句，对于函数定义及调用、数组声明及引用等语句无需涉及。所以我们只需在语义分析器中选择对应的非终结符对应函数的部分代码段，在其中根据语义动作生成中间代码语句，并且全部加到全局变量mid\_codes容器中，最后输出到mid\_code.txt文本即可。本次平台测试涉及的语句，例如普通变量的赋值语句和声明语句实现都很简单。  在实现了mid\_code的转换之后，我在code\_gener.cpp文件中继续实现了中间代码到目标代码的转换。基本思路就是读进中间代码，结合符号表，对语句进行判断，根据语句不同类型做不同操作，生成最后的MIPS代码。在转换时还需要考虑一个问题，每次传进来的操作数是变量还是数字，对于一个z = x + y形式的语句，x和y的种类可能组合成很多可能，而根据每种类型可能会用不同的指令，比如x、y均为变量类型，那么只需要对x和y值的寄存器做add即可，如果二者中有一个为数字，可以直接使用addi指令，如果二者均为数字，则需要先用addi把x和y的值分别存进寄存器，再对两个寄存器做add指令。除此之外，还有一些细节需要结合mips指令的特点实现。  本次实验难度不大，结合实验三的语义分析基础，要实现的增量开发并不多，虽然我对mips指令集长时间未接触有点生疏，但总体比较顺利的解决。   1. 数据结构说明 2. 中间代码的操作类型   虽然列举了很多操作类型，但是最后使用的基本上就是赋值语句、读语句、写语句、变量声明、常量声明涉及到的操作   |  | | --- | | enum mid\_code\_op  {      PLUSOP, *//+*      MINUOP, *//-*      MULTOP, *//\**      DIVOP, *// /*      LSSOP, *//<*      LEQOP, *//<=*      GREOP, *//>*      GEQOP, *//>=*      EQLOP, *//==*      NEQOP, *//!=*      ASSIGNOP, *//=*      GOTO, *//无条件跳转*      BZ, *//不满足条件跳转*      BNZ, *//满足条件跳转*      PUSH, *//函数调用时参数传递*      CALL, *//函数调用*      RET, *//函数返回语句*      RETVALUE, *//有返回值函数返回的结果*      SCAN, *//读*      PRINT, *//写*      LABEL, *//标号*      CONST, *//常量*      ARRAY, *//数组*      VAR, *//变量*      VARINIT,      FUNC, *//函数定义*      PARAM, *//函数参数*      GETARRAY, *//取数组的值  t = a[]*      PUTARRAY, *//给数组赋值  a[] = t*      EXIT, *//退出 main最后*  }; |  1. 中间代码   中间代码采用了四元式的结构，分别为结果，左操作数，操作类型，右操作数   |  | | --- | | struct mid\_code  {      string res; *// 结果*      string left; *// 左操作数*      mid\_code\_op op; *// 运算符*      string right; *// 右操作数*  }; |  1. 中间代码列   所有生成的中间代码都会放在其中   |  | | --- | | vector<mid\_code> mid\_codes; |  1. MIPS代码的操作类型   MIPS代码的操作类型和名字含义相似，不过多赘述了，稍显复杂的操作类型已在注释中给出   |  | | --- | | enum mips\_op  {      add,      addi,      sub,      mult,      divop,      mflo,      mfhi,      j,      lw,      sw,      syscall,      la,      moveop,      dataSeg, *//.data*      textSeg, *//.text*      asciizSeg, *//.asciiz*      globlSeg, *//.globl*      label, *//产生标号*  }; |  1. MIPS代码   MIPS代码和mid code大同小异，只是多了一个立即数   |  | | --- | | struct mips\_code  {      mips\_op op; *//操作类型*      string res; *//结果*      string left; *//左操作数*      string right; *//右操作数*      int imm; *//立即数*  }; |  1. MIPS代码列   所有生成的MIPS代码都放在这个vector容器中   |  | | --- | | vector<mips\_code> mips\_codes; |  1. 字符常量   存放字符常量的容器   |  | | --- | | vector<string> string\_const; *//存放字符常量* |  1. 字符常量的下标   根据字符常量反查下标   |  | | --- | | unordered\_map<string, int> string\_index\_map; *//根据字符串反查下标*  int string\_cnt = 0; *//字符常量的个数* |  1. 函数名称  |  | | --- | | string func\_name = ""; |  1. 存储分配方法说明   本次程序不涉及寄存器的选择，不涉及寄存器和内存之间的交换，所涉及的变量均存储在栈中，具体的实现在后文会介绍。   1. 代码整体思路说明   代码的基本思想就是根据语法制导翻译生成中间代码，然后根据中间代码和符号表生成MIPS代码，由于时间问题，MIPS没有设计为完全使用寄存器，而是每次计算的时候，先用t0，t1，两个寄存器把两个操作数从对应的栈地址中读取出来，然后计算，用t2寄存器存放结果，之后把结果放到对应的栈地址中去。   1. 生成中间代码程序说明 2. 插入中间代码的函数   根据中间代码的四个属性插入到中间代码语句列中   |  | | --- | | void insert\_mid\_code(string *result*, string *left*, mid\_code\_op *op*, string *right*)  {      mid\_codes.emplace\_back(mid\_code{*result*, *left*, *op*, *right*});  } |  1. 输出中间代码到文件的函数   output\_mid\_code函数根据不同的操作类型将一个四元式转换为中间代码，输出到中间代码文件中  output\_mid\_code\_all函数调用output\_mid\_code函数处理所有的四元式   |  | | --- | | void output\_mid\_code(struct mid\_code *mc*)  {      switch (*mc*.op)      {      case PLUSOP:          mid\_code\_file << *mc*.res << " = " << *mc*.left << " + " << *mc*.right << "\n";          break;      case MINUOP:          mid\_code\_file << *mc*.res << " = " << *mc*.left << " - " << *mc*.right << "\n";          break;      case MULTOP:          mid\_code\_file << *mc*.res << " = " << *mc*.left << " \* " << *mc*.right << "\n";          break;      case DIVOP:          mid\_code\_file << *mc*.res << " = " << *mc*.left << " / " << *mc*.right << "\n";          break;      case ASSIGNOP:          mid\_code\_file << *mc*.res << " = " << *mc*.left << "\n";          break;      case RETVALUE:          mid\_code\_file << "RETVALUE " << *mc*.res << " = " << *mc*.left << "\n";          break;      case SCAN:          mid\_code\_file << "SCAN " << *mc*.res << "\n";          break;      case PRINT:          mid\_code\_file << "PRINT " << *mc*.res << " " << *mc*.left << "\n";          break;      case LABEL:          mid\_code\_file << *mc*.res << ": \n";          break;      case VARINIT:          mid\_code\_file << *mc*.res << " VARINIT" << endl;          break;      case FUNC:          mid\_code\_file << "FUNC " << *mc*.res << " " << *mc*.left << "()" << endl;          break;      case PARAM:          mid\_code\_file << "PARA " << *mc*.res << " " << *mc*.left << endl;          break;      default:          break;      }  }  void output\_mid\_code\_all()  {      for (auto mid\_code : mid\_codes)          output\_mid\_code(mid\_code);  } |  1. 常量定义的中间代码生成   在常量定义的时候，每声明一个常量，都需要生成一句中间代码，无论是字符常量还是整数常量   |  | | --- | | void constant\_def() {      if (symbol == INTTK) {          init\_now\_sym();          now\_sym.kind = 0;          now\_sym.type = 1;          do {              get\_token();              now\_sym.name = tokenBuffer;              get\_token();              get\_token();              integer();              now\_sym.value = last\_integer;              insert\_current\_symbol();  *//插入中间代码，表示声明一个整数常量*              insert\_mid\_code(now\_sym.name, "int", CONST, to\_string(last\_integer));          } while (symbol == COMMA);      }      else if (symbol == CHARTK) {          init\_now\_sym();          now\_sym.kind = 0;          now\_sym.type = 4;          do {              get\_token();              now\_sym.name = tokenBuffer;              get\_token();              get\_token();              now\_sym.str = tokenBuffer;              insert\_current\_symbol();  *//插入中间代码，表示声明一个常量字符*              insert\_mid\_code(now\_sym.name, "char", CONST, now\_sym.str);              get\_token();          } while (symbol == COMMA);      }  } |  1. 变量初始化的中间代码生成   变量初始化也需要生成中间代码   |  | | --- | | void variable\_def\_with\_init() {      indexCount = 0;      columnsCount = 0;      string nowVar;  *//插入中间代码，表示变量的初始化*      insert\_mid\_code(now\_sym.name, "", VARINIT, "");      get\_token();  *// 变量定义及初始化*  } |  1. 表达式的中间代码生成   表达式的中间代码生成，就是一直用中间变量代替每一个加减法运算，最后达到表示复合运算的效果，例如：  a+b-c+d的中间代码为t1=a+b,t2=t1-c,t3=t2+d   |  | | --- | | void expression(int &*type*, string &*ansTmp*)  {      bool isMinu = false;      string res, left, right;      if (symbol == PLUS || symbol == MINU)      {          isMinu = (symbol == MINU);          getADD();          type = 1;      }      item(type, left);      if (isMinu)      {          res = get\_tmp\_value();  *//向符号表插入一个新的中间变量*          insert\_tmp\_value(res, 1, 0);  *//如果项的开头第一个token是负号，用0-负号后内容作为新的变量来表示它*  *//例如 a=-b;---->t1=0-b;a=t1;*          insert\_mid\_code(res, "0", MINUOP, left);  *//那么左操作数就是0-负号的变量*          left = res;      }      while (symbol == PLUS || symbol == MINU)      {          isMinu = (symbol == MINU);          getADD();          type = 1;          item(type, right);          res = get\_tmp\_value();          insert\_tmp\_value(res, 1, 0);  *//如果是减号输出中间代码为减法*  *//例如a-b---->t1=a-b*          if (isMinu)              insert\_mid\_code(res, left, MINUOP, right);  *//如果是加号输出中间代码为加法*  *//例如a+b---->t1=a+b*          else              insert\_mid\_code(res, left, PLUSOP, right);          left = res;      }  *//把现在形成的最左操作数返回给上层*      ansTmp = left;  } |  1. 项的中间代码生成   项的生成和表达式类似，就是一直用中间变量代替每一个乘除法运算，最后达到表示复合运算的效果，例如：  a\*b\*c\*d的中间代码为t1=a\*b,t2=t1\*c,t3=t2\*d   |  | | --- | | void item(int &*type*, string &*ansTmp*)  {      string res, left, right;  *//因子的类型传回给项*      factor(*type*, left);      while (symbol == MULT || symbol == DIV)      {          int multOp = symbol;          get\_token();  *//因子的类型传回给项*  *type* = 1;          factor(*type*, right);  *//存放中间变量*          res = get\_tmp\_value();          insert\_tmp\_value(res, 1, 0);  *//如果token是乘号生成新的乘法中间代码*          if (multOp == MULT)              insert\_mid\_code(res, left, MULTOP, right);  *//如果token是除号生成新的除法中间代码*          else              insert\_mid\_code(res, left, DIVOP, right);          left = res;      }  *//把最后的中间变量返回给上层*  *ansTmp* = left;  } |  1. 写语句的中间代码生成   写语句分为三种情况，一种单纯打印字符串，一种打印表达式，一种两类都打，分开处理中间代码输出即可，具体细节见注释   |  | | --- | | void write\_statement()  {      get\_token();      get\_token();      if (symbol == STRCON)      {          string nowString = tokenBuffer;          int pos = 0;          while ((pos = nowString.find("\\", pos + 2)) != string::npos)          {              nowString.replace(pos, 1, "\\\\");          }          insert\_string\_const(nowString);          String();          if (symbol == COMMA)          {  *//如果在逗号之后，那么一定是打印字符串和表达式*  *//输出打印字符串的中间代码*              insert\_mid\_code(nowString, "3", PRINT, "");              get\_token();              int type = 2;              string tmpExpress;              expression(type, tmpExpress);  *//输出打印表达式的中间代码*              insert\_mid\_code(tmpExpress, to\_string(type), PRINT, "");  *//输出打印换行符的中间代码*              insert\_mid\_code("changeLine", "4", PRINT, "");          }          else          {  *//如果没有逗号，那么就是单纯的打印字符串*  *//输出打印字符串的中间代码*              insert\_mid\_code(nowString, "3", PRINT, "");  *//输出打印换行符的中间代码*              insert\_mid\_code("changeLine", "4", PRINT, "");          }      }      else      {  *//没有字符串，只打印表达式*          int type = 2;          string tmpExpress;          expression(type, tmpExpress);  *//输出打印表达式的中间代码*          insert\_mid\_code(tmpExpress, to\_string(type), PRINT, "");  *//输出打印换行符的中间代码*          insert\_mid\_code("changeLine", "4", PRINT, "");      }  } |  1. 赋值语句的中间代码生成   赋值语句的中间代码非常简单，直接输出相应类别的中间代码即可   |  | | --- | | void assign\_statement()  {      identifier();      string nowIdentifier = L\_Identifier;      get\_token();      int tmpType;      string tmpExpress;      expression(tmpType, tmpExpress);  *//插入中间代码，表示赋值给变量，标识符=表达式的值*      insert\_mid\_code(nowIdentifier, tmpExpress, ASSIGNOP, "");  } |  1. 读语句的中间代码生成   读语句的代码生成也很简单，直接输出相应类别的即可,表示需要变量需要读入值   |  | | --- | | void read\_statement()  {      get\_token();      get\_token();      identifier();      string nowIdentifier = L\_Identifier;  *// 读语句中间代码*      insert\_mid\_code(nowIdentifier, "", SCAN, "");  } |  1. main函数的中间代码生成   Main函数的代码生成也很简单，直接输出相应类别的即可,表示正在main函数中   |  | | --- | | void Main()  {      functionWithoutReturnValue.insert(MAIN);      functionReturnType[MAIN] = 0;      init\_now\_sym();      now\_sym.kind = 6;      now\_sym.type = 0;      now\_sym.name = MAIN;      insert\_current\_symbol();      nowTable = MAIN;      insert\_current\_level();      get\_token();      get\_token();      if (symbol != RPARENT)      {          error('l');      }      else      {          get\_token();      }  *//插入中间代码，表示现在在main函数中*      insert\_mid\_code("void", MAIN, FUNC, "");      get\_token();      compound\_statement();      get\_token();  } |  1. 生成MIPS代码说明 2. 插入字符常量的函数   把所有的字符常量都插入到容器中，以便最后在指令中加载   |  | | --- | | void insert\_string\_const(string *s*)  {      string\_const.push\_back(*s*);      string\_index\_map[*s*] = string\_cnt++;  } |  1. 插入MIPS代码的函数   输入MIPS代码的五个属性，构造一个MIPS代码   |  | | --- | | void insert\_mips\_code(mips\_op *op*, string *res*, string *left*, string *right*, int *imm* = 0)  {      mips\_codes.push\_back(mips\_code{*op*, *res*, *left*, *right*, *imm*});  } |  1. 插入load指令的函数   基本思想就是根据变量名，查询到当前变量是全局变量还是局部变量还是常数，然后找到其相对栈顶指针的位置，将值引入到相应的寄存器中。   |  | | --- | | void load\_instr(string *identifier*, string *reg*, bool *gene*, int &*va*, bool &*get*)  {      struct sym now\_identifier;  *//全局变量且是常量或变量*      if (symbol\_table[PROGRAM].find(*identifier*) != symbol\_table[PROGRAM].end())      {          now\_identifier = symbol\_table[PROGRAM][*identifier*];  *//是常量*          if (now\_identifier.kind == 0)          {              if (now\_identifier.type == 1)  *va* = now\_identifier.value;              else  *va* = now\_identifier.str[0];              if (*gene*)  *//常量直接输出MIPS立即数加减指令就可以了*                  insert\_mips\_code(addi, *reg*, "$0", "", *va*);  *get* = true;          }  *//是变量*          else          {  *//变量输出MIPS代码把变量从栈地址加载到寄存器中*              int addr = (symbol\_table[PROGRAM][*identifier*].addr << 2);              insert\_mips\_code(lw, *reg*, "$gp", "", addr);          }      }  *//局部变量且是常量或变量*      else if (symbol\_table[func\_name].find(*identifier*) != symbol\_table[func\_name].end())      {          now\_identifier = symbol\_table[func\_name][*identifier*];  *//是常量*          if (now\_identifier.kind == 0)          {              if (now\_identifier.type == 1)  *va* = now\_identifier.value;              else  *va* = now\_identifier.str[0];              if (*gene*)  *//常量直接输出MIPS立即数加减指令就可以了*                  insert\_mips\_code(addi, *reg*, "$0", "", *va*);  *get* = true;          }  *//是变量*          else          {  *//变量输出MIPS代码把变量从局部变量栈地址加载到寄存器中*              int addr = -(symbol\_table[func\_name][*identifier*].addr << 2);              insert\_mips\_code(lw, *reg*, "$fp", "", addr);          }      }  *//加载常数*      else      {  *va* = stoi(*identifier*);          if (*gene*)              insert\_mips\_code(addi, *reg*, "$0", "", *va*);  *get* = true;      }  } |  1. 插入store指令的函数   基本思想就是根据变量名，查询到当前变量是全局变量还是局部变量还是常数，然后找到其相对栈顶指针的位置，将相应的寄存器引入到栈地址中。   |  | | --- | | void save\_instr(string *identifier*, string *reg*)  {  *//全局变量且是常量或变量*      if (symbol\_table[PROGRAM].find(*identifier*) != symbol\_table[PROGRAM].end() &&          ((symbol\_table[PROGRAM][*identifier*].kind == 1) || symbol\_table[PROGRAM][*identifier*].kind == 2))      {            int addr = (symbol\_table[PROGRAM][*identifier*].addr << 2);  *//把全局变量的计算结果存进内存中*          insert\_mips\_code(sw, *reg*, "$gp", "", addr);      }  *//局部变量且是常量或变量*      else if (symbol\_table[func\_name].find(*identifier*) != symbol\_table[func\_name].end() &&               (symbol\_table[func\_name][*identifier*].kind == 1 || symbol\_table[func\_name][*identifier*].kind == 2))      {  *//把局部变量的计算结果存进内存中*          int addr = -(symbol\_table[func\_name][*identifier*].addr << 2);          insert\_mips\_code(sw, *reg*, "$fp", "", addr);      }  } |  1. 生成mips指令的函数   基本思路就是先输出加载data的MIPS指令,也就是加载字符串和换行符的指令；然后是text段的指令，先输出将所有的全局变量和局部变量分别加载到不同的栈中去的指令，再输出不同的中间代码翻译出的MIPS指令，最后输出程序终止的MIPS指令。具体的翻译规则都在注释中   |  | | --- | | void generate\_mips\_code()  {  *//首先是data段*      insert\_mips\_code(dataSeg, "", "", "", 0);  *//先把常量字符串加载进去*      for (int i = 0; i < string\_const.size(); i++)      {          insert\_mips\_code(asciizSeg, "string" + to\_string(i), string\_const[i], "");      }      insert\_mips\_code(asciizSeg, "changeLine", "\\n", "");  *//其次是text指令段*      insert\_mips\_code(textSeg, "", "", "");      bool get1 = false, get2 = false;      bool flag = false;      int pushCnt = 0;      int paramSize = 0;      int va = 0, addr = 0, len = 0, va2 = 0;      bool goMain = false;      stack<struct mid\_code> pushOpStack;  *//根据中间指令生成mips指令*      for (auto mc : mid\_codes)      {          get1 = false;          get2 = false;          switch (mc.op)          {  *//变量的初始化*          case VARINIT:          {  *//全局变量*              if (symbol\_table[PROGRAM].find(mc.res) != symbol\_table[PROGRAM].end())              {                  struct sym tmp = symbol\_table[PROGRAM][mc.res];  *//类别是整数*                  if (tmp.kind == 1)                  {  *//输出MIPS指令用临时变量把初始化的值保存下来*                      insert\_mips\_code(addi, "$t0", "$0", "", tmp.value);                  }  *//类别是字符*                  else                  {  *//输出MIPS指令用临时变量把初始化的值保存下来*                      insert\_mips\_code(addi, "$t0", "$0", "", tmp.str[0]);                  }  *//输出MIPS指令把初始化的值存到该全局变量的地址*                  save\_instr(mc.res, "$t0");              }  *//局部变量*              else if (symbol\_table[func\_name].find(mc.res) != symbol\_table[func\_name].end())              {                  struct sym tmp = symbol\_table[func\_name][mc.res];  *//类别是整数*                  if (tmp.kind == 1)                  {  *//输出MIPS指令用临时变量把初始化的值保存下来*                      insert\_mips\_code(addi, "$t0", "$0", "", tmp.value);                  }  *//类别是字符*                  else                  {  *//输出MIPS指令用临时变量把初始化的值保存下来*                      insert\_mips\_code(addi, "$t0", "$0", "", tmp.str[0]);                  }  *//输出MIPS指令把初始化的值存到该全局变量的地址*                  save\_instr(mc.res, "$t0");              }              break;          }  *//加法*          case PLUSOP:          {  *//输出MIPS指令把两个操作数分别加载到t0和t1*              load\_instr(mc.left, "$t0", false, va, get1);              load\_instr(mc.right, "$t1", false, va2, get2);  *//如果两个操作数都是常量*              if (get1 && get2)              {  *//输出MIPS指令addi，直接用两个操作数的值求和保存到中间寄存器中保存到中间寄存器中*                  insert\_mips\_code(addi, "$t2", "$0", "", va + va2);              }  *//如果第一个操作数是常量，第二个不是*              else if (get1 && !get2)              {  *//输出MIPS指令addi，直接用第一个操作数的值与第二个变量求和保存到中间寄存器中*                  insert\_mips\_code(addi, "$t2", "$t1", "", va);              }  *//如果第二个操作数是常量，第一个不是*              else if (!get1 && get2)              {  *//输出MIPS指令addi，直接用第二个操作数的值与第一个变量求和保存到中间寄存器中*                  insert\_mips\_code(addi, "$t2", "$t0", "", va2);              }  *//如果两个操作数都不是常量*              else              {  *//输出MIPS指令add，直接两个操作数求和保存到中间寄存器中*                  insert\_mips\_code(add, "$t2", "$t0", "$t1");              }  *//输出MIPS指令sw,把和保存到对应变量的栈地址中*              save\_instr(mc.res, "$t2");              break;          }  *//减法和加法基本一致，变化就是常量值取反或者把指令头改成sub*          case MINUOP:          {              load\_instr(mc.left, "$t0", false, va, get1);              load\_instr(mc.right, "$t1", false, va2, get2);              if (get1 && get2)              {                  insert\_mips\_code(addi, "$t2", "$0", "", va - va2);              }              else if (get1 && !get2)              {                  insert\_mips\_code(addi, "$t2", "$t1", "", -va);              }              else if (!get1 && get2)              {                  insert\_mips\_code(addi, "$t2", "$t0", "", -va2);              }              else              {                  insert\_mips\_code(sub, "$t2", "$t0", "$t1");              }              save\_instr(mc.res, "$t2");              break;          }  *//乘法*          case MULTOP:          {  *//输出MIPS指令读两个操作数*              load\_instr(mc.left, "$t0", true, va, get1);              load\_instr(mc.right, "$t1", true, va2, get2);  *//输出MIPS指令求积保存到中间寄存器中*              insert\_mips\_code(mult, "$t0", "$t1", "");              insert\_mips\_code(mflo, "$t2", "", "");  *//输出MIPS指令sw,把积保存到对应变量的栈地址中*              save\_instr(mc.res, "$t2");              break;          }  *//除法和减法基本一致，变化就是把指令头改成DIV*          case DIVOP:          {              load\_instr(mc.left, "$t0", true, va, get1);              load\_instr(mc.right, "$t1", true, va2, get2);              insert\_mips\_code(divop, "$t0", "$t1", "");              insert\_mips\_code(mflo, "$t2", "", "");              save\_instr(mc.res, "$t2");              break;          }  *//赋值语句*          case ASSIGNOP:          {  *//输出MIPS指令读左操作数的值到t0*              load\_instr(mc.left, "$t0", true, va, get1);  *//输出MIPS指令sw,把t0的值保存到对应变量的栈地址中*              save\_instr(mc.res, "$t0");              break;          }  *//读语句*          case SCAN:          {  *//全局变量*              if (symbol\_table[PROGRAM].find(mc.res) != symbol\_table[PROGRAM].end() &&                  (symbol\_table[PROGRAM][mc.res].kind == 1 || symbol\_table[PROGRAM][mc.res].kind == 2))              {  *//查变量的类型，以生成不同的mips指令*                  int callType;                  callType = (symbol\_table[PROGRAM][mc.res].type == 1) ? 5 : 12;  *//输出MIPS指令把输入读到v0寄存器*                  insert\_mips\_code(addi, "$v0", "$0", "", callType);                  insert\_mips\_code(syscall, "", "", "");  *//输出MIPS指令sw,把v0的值保存到对应变量的栈地址中*                  addr = (symbol\_table[PROGRAM][mc.res].addr << 2);                  insert\_mips\_code(sw, "$v0", "$gp", "", addr);              }  *//局部变量*              else if (symbol\_table[func\_name].find(mc.res) != symbol\_table[func\_name].end() &&                       (symbol\_table[func\_name][mc.res].kind == 1 || symbol\_table[func\_name][mc.res].kind == 2))              {  *//查变量的类型，以生成不同的mips指令*                  int callType;                  callType = (symbol\_table[func\_name][mc.res].type == 1) ? 5 : 12;  *//输出MIPS指令把输入读到v0寄存器*                  insert\_mips\_code(addi, "$v0", "$0", "", callType);                  insert\_mips\_code(syscall, "", "", "");  *//输出MIPS指令sw,把v0的值保存到对应变量的栈地址中*                  addr = -(symbol\_table[func\_name][mc.res].addr << 2);                  insert\_mips\_code(sw, "$v0", "$fp", "", addr);              }              break;          }  *//写语句*          case PRINT:          {              if (mc.left[0] == '1')              { *// int*  *//输出MIPS指令把整数变量的值取到a0寄存器*                  load\_instr(mc.res, "$a0", true, va, get1);  *//输出MIPS指令打印整数变量的值*                  insert\_mips\_code(addi, "$v0", "$0", "", 1);                  insert\_mips\_code(syscall, "", "", "");              }              else if (mc.left[0] == '2')              { *// char*  *//输出MIPS指令把字符变量的值取到a0寄存器*                  load\_instr(mc.res, "$a0", true, va, get1);  *//输出MIPS指令打印字符变量的值*                  insert\_mips\_code(addi, "$v0", "$0", "", 11);              }              else if (mc.left[0] == '3')              { *// string*  *//输出MIPS指令把预加载的字符串的地址赋给a0寄存器*                  insert\_mips\_code(la, "$a0", "string" + to\_string(string\_index\_map[mc.res]), "", 0);  *//输出MIPS指令打印字符串*                  insert\_mips\_code(addi, "$v0", "$0", "", 4);                  insert\_mips\_code(syscall, "", "", "");              } *// changeLine*              else if (mc.left[0] == '4')              {  *//输出MIPS指令把预加载的换行符的地址赋给a0寄存器*                  insert\_mips\_code(la, "$a0", "changeLine", "", 0);  *//输出MIPS指令打印换行符*                  insert\_mips\_code(addi, "$v0", "$0", "", 4);                  insert\_mips\_code(syscall, "", "", "");              }              break;          }  *//函数*          case FUNC:          {              if (!goMain)              {  *//进入main函数入口*                  insert\_mips\_code(j, "main", "", "");                  goMain = true;              }              insert\_mips\_code(label, mc.left, "", "", 0);              func\_name = mc.left;              if (mc.left == MAIN)              {                  len = symbol\_table[MAIN].size();  *//输出MIPS指令把栈指针和帧指针加载了*                  insert\_mips\_code(addi, "$fp", "$sp", "", 0);                  insert\_mips\_code(addi, "$sp", "$sp", "", -4 \* len - 8);              }              break;          }          default:              break;          }      }  *//输出MIPS指令退出程序*      insert\_mips\_code(addi, "$v0", "$0", "", 10);      insert\_mips\_code(syscall, "", "", "");  } |  1. 输出mips指令到文件的函数   output\_mips\_code函数就是把已经翻译好的一条mips指令按照标准的mips格式输出到mips.txt中，output\_mips\_code\_all函数调用output\_mips\_code函数把所有的mips指令输出到mips.txt中   |  | | --- | | void output\_mips\_code(struct mips\_code *mc*)  {      switch (*mc*.op)      {      case add:          mips\_code\_file << "add " << *mc*.res << "," << *mc*.left << "," << *mc*.right << "\n";          break;      case sub:          mips\_code\_file << "sub " << *mc*.res << "," << *mc*.left << "," << *mc*.right << "\n";          break;      case mult:          mips\_code\_file << "mult " << *mc*.res << "," << *mc*.left << "\n";          break;      case divop:          mips\_code\_file << "div " << *mc*.res << "," << *mc*.left << "\n";          break;      case addi:          mips\_code\_file << "addi " << *mc*.res << "," << *mc*.left << "," << *mc*.imm << "\n";          break;      case mflo:          mips\_code\_file << "mflo " << *mc*.res << "\n";          break;      case j:          mips\_code\_file << "j " << *mc*.res << "\n";          break;      case lw:          mips\_code\_file << "lw " << *mc*.res << "," << *mc*.imm << "(" << *mc*.left << ")\n";          break;      case sw:          mips\_code\_file << "sw " << *mc*.res << "," << *mc*.imm << "(" << *mc*.left << ")\n";          break;      case la:          mips\_code\_file << "la " << *mc*.res << "," << *mc*.left << "\n";          break;      case syscall:          mips\_code\_file << "syscall\n";          break;      case moveop:          mips\_code\_file << "move " << *mc*.res << "," << *mc*.left << "\n";          break;      case dataSeg:          mips\_code\_file << ".data\n";          break;      case textSeg:          mips\_code\_file << ".text\n";          break;      case asciizSeg:          mips\_code\_file << *mc*.res << ": .asciiz \"" << *mc*.left << "\"\n";          break;      case label:          mips\_code\_file << *mc*.res << ":\n";          break;      default:          break;      }  }  void output\_mips\_code\_all()  {      for (auto mips\_code : mips\_codes)          output\_mips\_code(mips\_code);  } | | | | | | |
| 1. 实验结果及分析和（或）源程序调试过程 2. 自定义测试 3. 代码  |  | | --- | | int test;  int test1=1;  int test2=2;  int test3=3;  void main(){  printf("I'm Xie, 20194196, here is the mips code!");  scanf(test);  printf(test);  } |  1. 中间代码  |  | | --- | | test1 VARINIT  test2 VARINIT  test3 VARINIT  FUNC void main()  PRINT I'm Xie, 20194196, here is the mips code! 3  PRINT changeLine 4  SCAN test  PRINT test 1  PRINT changeLine 4 |  1. MIPS代码  |  | | --- | | .data  string0: .asciiz "I'm Xie, 20194196, here is the mips code!"  changeLine: .asciiz "\n"  .text  addi $t0,$0,1  sw $t0,4($gp)  addi $t0,$0,2  sw $t0,8($gp)  addi $t0,$0,3  sw $t0,12($gp)  j main  main:  addi $fp,$sp,0  addi $sp,$sp,-8  la $a0,string0  addi $v0,$0,4  syscall  la $a0,changeLine  addi $v0,$0,4  syscall  addi $v0,$0,5  syscall  sw $v0,0($gp)  lw $a0,0($gp)  addi $v0,$0,1  syscall  la $a0,changeLine  addi $v0,$0,4  syscall  addi $v0,$0,10  syscall |  1. MARS执行结果   mars1  mars2   1. 平台测试 2. 结果   平台截图 | | | | | |
| 1. 实验总结 2. BUG记录   本次实验算是后三次实验里最简单的一个了，有实验三的基础而且涉及的语句也很少，因此从语法制导翻译到中间代码生成没有什么很大的BUG出现，小问题很多，基本上都是由于对MIPS指令的不熟悉造成的，例如，我最开始不知道怎么翻译MIPS的系统输入输出语句，后面看了文档，知道需要使用 syscall指令，参数所使用的寄存器为$v0,$a0,$a1，返回值使用$v0。   1. 实验心得   本次实验难度不大，只需要将常量说明、变量说明、读语句、写语句、赋值语句和mips指令集中的指令对应起来就好了，无函数定义及调用，无数组声明及引用，实验的难度可以说远低于实验二和实验三。完成的过程中我没有参考书上的方法和三元式定义（）书上的算法主要针对LR型文法），我自己在开发的递归调用的实验基础上，自行定义了一个四元式的中间代码，然后再开发了一个从该中间代码到MIPS代码的解释器，完成从高级语言到mips代码的翻译。到此，算是从无到有开发了一个简易编译器，因为整个编译原理的实验过程是在是太痛苦了，所以在平台上通过测试的那一刻还是非常开心的。既有解脱也有喜悦吧，在此抒情一下。   1. 未来发展 2. 因为采用的是LL文法+递归调用，所以从算法上，还可以更进一步，如果后面有机会的话，可以考虑将文法换到LR文法，效率提升应该会很大 3. 因为时间的原因，本次实验在代码优化和寄存器使用优化上还有很大的进步空间，改进时可以考虑采用PPT第十章上的优化方法来减少代码冗余，并提高寄存器使用效率。 | | | | | |