**《编译原理》实验报告**

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| **姓名** | | **蔡嘉轩** | | | **年级** | **2018** |
| **学号** | | **20185670** | | | **专业、班级** | **计科 卓越班** |
| **实验名称** | **目标代码生成程序的设计与实现** | | | | | |
| **实验时间** | **2021.05.23** | | **实验地点** | **DS1421** | | |
| **实验成绩** |  | | **实验性质** | **□验证性 □设计性 □综合性** | | |
| 教师评价：  完成实验内容，达到实验要求，实验数据和结果正确，报告内容详实。  程序质量得分：7，实验报告得分：2，实验项目总得分：  评价教师签名：张敏 | | | | | | |
| 一、实验目的  在词法分析、语法分析及语义分析的接触上，编写一个程序对使用类C语言书写的源代码翻译为中间代码/目标代码，并打印翻译结果 | | | | | | |
| 二、实验项目内容  一）实验内容  设计并实现实验一中的C语言子集的目标代码生成程序，并打印分析结果。要求实现以下功能：  1、必做项：   1. 能够输出抽象语法树/四元式的中间代码，应至少包括以下代码类型：赋值语句、算术运算操作（加减乘除），跳转语句、分支与循环语句及其他基本语 2. 在四元式基础上，生成汇编语言形式的目标代码   2、可选项：   1. 生成过程/函数的目标代码   二）实验要求：   1. 自由选用程序设计语言（java，python，c/c++）作为实现语言，手工编写语法分析程序。 2. 提交实验报告及源代码。实验报告需严格遵循学校文档规范，内容包含对应文法、语法分析测试用例。   三）C语言子集:  数据类型: **int**, 无符号整数, 取值范围0-9999  int a;  int a,b;  int a = 1;  算术运算符: **+**,**-**  a = b + 1;  a = b + c;  赋值运算符: **=**  a = 1;  关系运算符: **==** ,**>**,**<,<>,>=,<=**  a = (b==c);  a = (b>c);  a = (b<c);  逻辑运算符: **&&**, **||, !**  a = (b&&c);  a = (b||c);  a = (!b);  条件语句: **if**  if(a==b)  {  };  循环语句: **while**  while(a==b)  {  };  输入,输出: **get**,**put**  get(a);  put(a);  语句结束符: **;**  **加分项：**  条件语句 **if else**  if(a==b)  {  };  else  {  };  四）目标语言  在本次实验中假设数据类型只包含整数类型，不包含如浮点数、数组、结构和指针等其它数据类型， 目标语言选定为MIPS32指令序列，中间代码及MIPS32指令对应关系如下表所示。其中reg(x)表示变量x所分配的寄存器。 | | | | | | |
| 三、实验过程或算法（源程序）   1. **中间代码/目标代码生成程序总体说明：**   完成的实验要求：按照实验指导书要求，完成了本次“必选”和“可选”中所有的要求。本实验使用语法制导翻译技术实现中间代码的生成，本次选择的中间代码的表示形式是四元式。完成中间代码生成后，对中间代码进行了翻译，使之成为最终可以在目标机器（MIPS指令集）上运行的目标代码。本次实验中实现的可翻译为目标代码的语句具体有：  a) 基本的赋值语句；  b) 算术运算操作（加减乘除）；  c) 跳转语句；  d) IF分支语句；  e) WHILE循环语句；  f) 变量声明语句；  g) 函数返回语句等其他基本语句；  h) 过程/函数调用语句。  中间代码/目标代码生成程序总体说明总体说明：实验三中使用自顶向下的语法制导翻译技术实现了对C语言程序子集的语义分析并，在本次实验四中同样使用了自顶向下的语法制导翻译技术，基于实验三的思想与生成的语法树，实现了中间代码与目标代码的生成。  在此阶段采用自顶向下的思想与深度优先遍历DFS的方法，对实验三中实现的语义分析程序进行部分修改，使之可以适应中间代码与目标代码生成的要求，对语法分析树进行进一步分析，实现了基本的赋值语句、算术运算操作（加减乘除）、跳转语句、IF分支语句、WHILE循环语句、变量声明语句、函数返回语句等其他基本语句、过程/函数调用语句等语句的中间代码/目标代码翻译，即完成了实验要求中“必选”和“可选”的所有要求。  实验代码文件的结构如下图所示；       1. **存储分配方法说明：**   本次实验中存储分配方法参考《编译原理与技术》教材上的管理方法，实现了对内存和寄存器的管理与存储分配。本次实验中定义变量的初始存储地址均位于内存中，实现上以基址加偏移量确定其在内存中的具体位置。  为了管理涉及内存的存储分配，本次实验实现的目标代码中实现了对变量内存存储地址偏移量offset的管理，在后续连接器进行代码连接的时候，对于每个变量地址的处理是：将此offset值加上代码段的基址，通过加法运算得到的结果即为变量存储空间在内存中的真实地址。  为了管理涉及寄存器的存储分配，参考教材上的方法，维护了两个向量，分别为可用寄存器regs和寄存器状态regstatus，分别用来存储当前所有可用的寄存器的名字，管理每个寄存器对应的分配状态。实现函数getregs获取当前临时变量应该得到的寄存器名，recallregs用于还回寄存器，即如果临时变量在之后的代码中不需要再被引用，即解除它占有的寄存器的存储分配状态，以供之后重新分配给其他临时变量。  **1)** **涉及内存的存储分配**  为了实现对变量内存存储地址偏移量offset的管理，需要在变量对应的类型中加入新的字段，如下列代码所示：   1. class Variable{ 2. public: 3. string name; 4. string type; 5. string value; 6. int offset; 7. static int total\_offset; 8. Variable(){} 9. Variable(string t){ 10. type = t; 11. offset = total\_offset; 12. total\_offset -= 4; 13. }   使用字段offset，代表每个变量所在位置与基址之间的偏移量。同时，与实验二中实现语法树节点编号自增加的方式类似，定义了静态变量total\_offset，用于代表当前的总偏移量，当新的变量分配内存空间时当前总偏移量减变量的长度，反之当变量释放时当前总偏移量加变量的长度。由于本次实验中实现的变量类型为INT型，所以长度均为4。  修改作用域重定位代码，实现在作用域重定位过程中，所有位于作用域中的变量全部释放的功能。   1. *//重定位域* 2. void relocate\_scope(){ 3. scope\_list.pop\_back(); 4. scope\_ptr = scope\_list.back(); 5. int size = identifiers.variable\_list[scope\_ptr].size(); 6. Variable::total\_offset += 4\*size; 7. }   **2)** **涉及寄存器的存储分配**  为了实现对涉及寄存器的存储进行管理，需要在中间代码翻译成目标代码的过程中进行数据结构的维护。分别维护两个向量，分别为可用寄存器regs和寄存器状态regstatus，分别用来存储当前所有可用的寄存器的名字，管理每个寄存器对应的分配状态。   1. *//查询得出MIP目标机器一般使用8个寄存器，分别编号为7～0* 2. vector<int> regs = {7，6，5，4，3，2，1，0}; 4. *//寄存器状态存储数据结构* 5. map<string, int> regstatus;   通过查询MIPS目标机器的设计要求，发现寄存器数目一般是32个，故在此处设置32个寄存器，分别编号为31~0号，同时设置regstatus容器，用于存放管理每个寄存器对应的分配状态。  实现函数getregs获取当前临时变量应该得到的寄存器名，recallregs用于还回寄存器，即如果临时变量在之后的代码中不需要再被引用，即解除它占有的寄存器的存储分配状态，以供之后重新分配给其他临时变量。   1. int getregs(string id){ 2. *//分配寄存器* 3. int av\_reg = regs.back(); 4. regs.pop\_back(); 5. regstatus[id] = av\_reg; 6. return av\_reg; 7. } 8. int recallregs(string id){ 9. *//收回寄存器* 10. int al\_reg = regstatus[id]; 11. regstatus.erase(id); 12. regs.push\_back(al\_reg); 13. return al\_reg; 14. }   Getregs函数用于获取现在应该分配给临时变量的寄存器，输入的参数为临时变量的名称；recallregs函数用于释放临时变量拥有的寄存器，并返回该寄存器，输入的参数为临时变量的名称。通过查询寄存器分配状态表，获取到该变量拥有的寄存器，将其释放并返回，同时更新寄存器分配状态表。   1. **错误处理：**   实验二中，实现了语法错误的分析、检查与处理；实验三中，实现了语义错误的分析、检查与处理。本次实验中，针对中间代码/目标代码生成程序对于错误处理的特殊需要，对之前实现的错误处理方法进行了略微的改动，实现了下列类型的错误检查与处理：   * 类型检查 * 函数重复定义处理 * 函数嵌套定义处理 * 同定义域中变量重复定义处理 * 类型合法性检查   由于本次实验对错误检查与处理的方式同上次实验有数处改动，所以还是在此处结合部分关键代码说明本次实验错误检查与处理的方法。类似于我在实验二与三中对错误进行管理的方法，定义了一个类型为vector<vector<string>>的错误流，用于存放产生的错误。在检测到错误时，将错误相关的信息存入错误流，并且向后继续分析，从而避免错误对正常分析产生影响，程序不会遇到错误就停止。本实验中按照实验指导的要求实现了如下几种的错误处理：  **1)类型合法性**  对int、bool和string类型的变量与常量进行类型检查，确保不同类型的变量之间不能相互赋值，常量不能赋给与其类型不符的变量，确保布尔类型值与字符串类型值不能参与算数运算，不同类型的量不能相互运算。相关代码如下：   1. if(v.type != identifiers.variable\_list[scope][id\_name].type && v.type!="" && v.type!="error"){ 2. seman\_error\_stream.push\_back(vector<string>{"[Illegal assignment]: '" + id\_name + "': Cannot assign '" + v.type +"' to '" + identifiers.variable\_list[scope][id\_name].type +"'.", Assign.start\_pos,Assign.end\_pos});   当进行赋值操作时，会检查产生式右部Expression的类型，如果类型与左边待赋值目标标识符的类型不一致，则会报错：无法把xxx类型的值赋给yyy类型的值。   1. if(operand.first == "int"){ 2. if(Expression.children[0].children[0].symbol == "++"){ 3. return pair<string, string>("int", to\_string(stoi(operand.second)+1)); 4. } 5. else if(Expression.children[0].children[0].symbol == "--"){ 6. return pair<string, string>("int", to\_string(stoi(operand.second)-1)); 7. } 8. else { 9. seman\_error\_stream.push\_back(vector<string>{"Illegal unary opereator: " + Expression.children[0].children[0].symbol,Expression.start\_pos,Expression.end\_pos}); 10. return pair<string, string>("error",""); 11. } 12. } 13. else { 14. seman\_error\_stream.push\_back(vector<string>{"[Illegal operand]: '" + Expression.children[1].children[0].children[0].symbol + "': Cannot use type '" + operand.first + "' in Arithmetic Operation.",Expression.start\_pos,Expression.end\_pos});   在算数表达式中，本次实验实现过程中定义在算术表达式中仅支持使用int类型的变量作为操作数进行运算，所以如果操作数的类型不是int的话，就会报错：无法在算术表达式使用xxx类型的操作数。  **2)函数、过程重定义**  函数和过程不允许重复定义，由于所有的函数都定义在同一个作用域内，所以此处暂时不涉及作用域管理。   1. if(identifiers.function\_exist\_in\_scope(0, f.name)){ 2. seman\_error\_stream.push\_back(vector<string>{"[Redecleration of Function/Process]: Function '" + f.name + "' already exists in the current scope.",Function\_decleration.start\_pos,Function\_decleration.end\_pos}); 3. } 4. else{ 5. if(scope\_ptr != 0){ 6. seman\_error\_stream.push\_back(vector<string>{"[Nested decleration of Function/Process]: You can not declare: '" + f.name + "' because it's nestedly declared.",Function\_decleration.start\_pos,Function\_decleration.end\_pos}); 7. }   当声明函数、过程的时候，对当前的符号表进行检查，如果发现已经有同名的函数、过程，则判断为重复定义，在错误流中存储重复定义函数的相关信息。  **3)函数、过程嵌套定义：**  函数和过程不允许嵌套定义，即不允许在某个函数/过程内部再定义一个函数/过程，此检查的实现方法较为有趣。   1. f.name = Function\_decleration.children[1].children[0].symbol; 3. if(identifiers.function\_exist\_in\_scope(0, f.name)){ 4. seman\_error\_stream.push\_back(vector<string>{"[Redecleration of Function/Process]: Function '" + f.name + "' already exists in the current scope.",Function\_decleration.start\_pos,Function\_decleration.end\_pos}); 5. } 6. else{ 7. if(scope\_ptr != 0){ 8. seman\_error\_stream.push\_back(vector<string>{"[Nested decleration of Function/Process]: You can not declare: '" + f.name + "' because it's nestedly declared.",Function\_decleration.start\_pos,Function\_decleration.end\_pos}); 9. }   经过分析可知所有的函数声明，如果是在程序的最外层进行编写的，则其scope一定为0，不可能为其他值。经过分析，如果某个函数定义的scope值不为0，则说明其是在其他的函数内部进行声明的，即产生了函数嵌套定义的情况，需要在错误流中报错，存储函数嵌套定义的相关错误信息。  **4)同一作用域内的变量重定义：**  不允许在统一作用域内重复定义一个变量，它的变量名与已经存在的某个变量名重复。这样的规则称为在同一作用域内不允许变量的重定义。为实现这个错误类型的检查，必须对作用域进行合理适当的管理，使得作用域数据结构能够被正常的维护，且在错误检查的过程中能够正确地查询并使用对应数据结构中的作用域数据。   1. if(identifiers.variable\_exist\_in\_scope(scope\_ptr, v.name)){ 2. seman\_error\_stream.push\_back(vector<string>{"[Redecleration of Variable]: Variable '" + v.name + "' already exists in the current scope.",Variable\_Definition\_Without\_Initial\_Assignment.start\_pos,Variable\_Definition\_Without\_Initial\_Assignment.end\_pos}); 3. } 4. else identifiers.variable\_list[scope\_ptr][v.name] = v; 5. } 6. (……) 7. if(identifiers.variable\_exist\_in\_scope(scope\_ptr, v.name)){ 8. seman\_error\_stream.push\_back(vector<string>{"[Redecleration of Variable]: Variable '" + v.name + "' already exists in the current scope.",Variable\_Definition\_With\_Initial\_Assignment.start\_pos,Variable\_Definition\_With\_Initial\_Assignment.end\_pos}); 9. } 11. else identifiers.variable\_list[scope\_ptr][v.name] = v; 12. }   在带有初始赋值的变量定义语句与不带有初始赋值的变量定义语句中都必须对同一作用域内的变量名重复定义问题进行处理。处理的方法是使用函数variable\_exist\_in\_scope,参数为当前所在的作用域scope\_ptr和待定义的变量名，使用这两个参数去对应的作用域中查询是否存在同名的变量，如果存在说明出现了变量重复定义的问题，需要在错误流中报错，存储变量重复定义的相关信息。   1. bool variable\_exist\_in\_scope(int scope, string name){ 2. if(variable\_list.find(scope) != variable\_list.end() && variable\_list[scope].find(name) != variable\_list[scope].end()){ 3. return true; 4. } 5. return false; 6. }   函数variable\_exist\_in\_scope的实现思路是，在作用域列表中进行遍历查询，如果查到对应的作用域，就进入匹配相同名字的变量，如果匹配成功说明这个作用域内存在待搜索的变量名。   1. **程序说明：**   本次实验的程序主要模块为语法树分析模块、符号表建立与维护模块、中间代码生成模块、目标代码生成模块。接下来对这几个模块，分别结合部分关键代码进行实现细节说明。  **1)语法分析树模块**  本次实验中，基于实验三中实现的自顶向下的语义分析过程与输出的语法分析树，扩展了语法制导翻译方案中的语义动作，增加了中间代码四元式生成的函数，并在合适的位置执行生成中间代码四元式的函数从而向中间代码序列中添加新的四元式，完成中间代码的生成。   1. class Semantic{ 2. public: 3. Semantic(Node &node){ 4. S = node; 5. *//语义分析* 6. *//处理根节点（方法：遍历子节点）* 7. scope\_ptr = 0; 8. quaternions.add\_quaternion("LABEL", "NIL", "NIL", "main"); 9. for(auto i : S.children){ 10. if(i.symbol == "Function\_Body") 11. proc\_function\_body(i); 12. else if (i.symbol == "External\_Statements") 13. proc\_external\_statements(i); 14. } 16. *//输出错误* 17. *//error\_output();* 19. *//输出符号表* 20. *//identifiers.output();* 22. *//输出中间代码* 23. quaternions.output\_quaternions("/Users/caijiaxuan/workspace/xcodecpp/CP\_LAB4/CP\_LAB4/inter.txt"); 24. } 25. Quaternions quaternions;   上述代码段是本次中间代码/目标代码对应的类型Semantic的构造函数，构造函数的基本过程是，使用传进的node初始化使用的根节点。在实际调用过程中，这个传进的node为实验二中生成的语法分析树的根节点。然后将当前的域指针值设为0，接下来遍历根节点的每个子节点，判断子节点类型是外部语句还是函数体，并根据判断的情况对外部语句和函数体分别进行识别。识别结束后，即完成语义分析，此时调用error\_output函数进行语义错误信息输出，然后生成对应的中间代码四元式，最后调用quaternions.output函数进行符号表输出。  **2)符号表建立与维护模块**   1. class Variable{ 2. public: 3. string name; 4. string type; 5. string value; 7. Variable(){} 9. private: 10. }; 11. class Function{ 12. public: 13. string name; 14. string return\_type; 15. vector<pair<string, string>> parameters; 17. Function(){} 19. private: 20. }; 21. class Identifiers{ 22. public: 23. void output(){ 24. cout<<"----------------------- Symbol Table -----------------------"<<endl<<endl; 25. cout<<"------------------------- Variables ------------------------"<<endl<<endl; 26. for(auto i : variable\_list){ 27. cout<<"------------------------------------------------------------"<<endl; 28. cout<<"Scope: " << i.first<<endl; 29. for(auto k : i.second){ 30. string name = k.first; 31. Variable &v = k.second; 32. cout<<"variable: " <<name<<" ,type: "<<v.type<<" ,value:"<<v.value <<endl; 33. } 34. cout<<"------------------------------------------------------------"<<endl<<endl; 35. } 37. cout<<"------------------------- Functions ------------------------"<<endl<<endl; 38. for(auto i : function\_list){ 39. cout<<"------------------------------------------------------------"<<endl; 40. cout<<"Scope: " << i.first<<endl; 41. for(auto k : i.second){ 42. string name = k.first; 43. Function &f = k.second; 44. cout<<"function: " <<name<<" ,return type: "<<f.return\_type; 45. for(auto j : f.parameters){ 46. cout<<" ,params: "<<j.first<<" ,"<<j.second<<endl; 47. } 48. } 49. cout<<endl<<"------------------------------------------------------------"<<endl<<endl; 50. }  53. } 54. map<int,map<string,Variable>> variable\_list; 55. map<int,map<string,Function>> function\_list; 57. bool variable\_exist\_in\_scope(int scope, string name){ 58. if(variable\_list.find(scope) != variable\_list.end() && variable\_list[scope].find(name) != variable\_list[scope].end()){ 59. return true; 60. } 61. return false; 62. } 64. bool function\_exist\_in\_scope(int scope, string name){ 65. if(function\_list.find(scope) != function\_list.end() && function\_list[scope].find(name) != function\_list[scope].end()){ 66. return true; 67. } 68. return false; 69. } 71. int find\_variable(vector<int> scope\_list, string variable\_name){ 72. for (int i = scope\_list.size() - 1; i>=0; i--){ 73. int scope = scope\_list[i]; 74. if(variable\_exist\_in\_scope(scope, variable\_name)) 75. return scope; 76. } 77. return -1; 78. } 80. private: 82. };   符号表管理模块定义了三个类，分别是Variable类、Function类和Identifiers类，用于管理符号表。前两个类是数据类，而Identifiers类是用于管理符号表的类。定义了三个函数variable\_exist\_in\_scope，function\_exist\_in\_scope和find\_variable，分别用于判断当前域中是否存在输入名字的变量、当前域中是否存在输入名字的函数和用于查找输入名字的变量在哪个域中。符号表管理模块还定义了两个输出函数，分别用来输出符号表中的函数信息和变量信息。  **3)中间代码生成模块**  **3.1)中间代码数据类型与中间代码管理器类型定义**  本次实验中首先定义了代表中间代码的类型Quaternion和中间代码管理器Quaternions类，类型定义如下：   1. class Quaternion{ 2. public: 3. string Operator, Operand1, Operand2, Result; 5. Quaternion(string op, string o1, string o2, string res){ 6. Operator = op; 7. Operand1 = o1; 8. Operand2 = o2; 9. Result = res; 10. } 12. }; 13. class Quaternions{ 14. public: 15. vector<Quaternion> quaternions; 16. int temp\_ptr = 0; 18. void output\_quaternions(string filepath){ 19. ofstream file(filepath); 20. for(auto i : quaternions){ 21. file << i.Operator << " " << i.Operand1 << " " << i.Operand2 << " " << i.Result << endl; 22. } 23. } 25. void add\_quaternion(string op, string o1, string o2, string res){ 26. quaternions.push\_back(Quaternion(op,o1,o2,res)); 27. } 29. string get\_temp(){ 30. return "TEMP"+to\_string(temp\_ptr++); 31. } 32. };   中间代码类型Quaternion中定义了四个字符串类型的值域，分别代表四元式中的操作符、操作数1、操作数2和四元式的结果。构造函数读入四个字符串类型的参数并分别对上述的四个值域进行赋值得到初始的四元式数据。  中间代码管理器Quaternions类定义了一个Quaternion类型的向量，用来存放生成的全部四元式，定义了临时变量指针temp\_ptr，其作用是指示当前需要生成的临时变量的编号，生成方式与实验二中生成语法树节点的ID类似，每当新建一个临时变量的时候，就将临时变量指针加一。  定义了函数output\_quaternions，输入形参为字符串类型的变量filepath，用于表示将四元式序列输出到的文件的路径。对四元式的输出即对中间代码管理器Quaternions中的每一条四元式都按照格式进行输出。  定义了函数add\_quaternion，输入形参为四个字符串类型的值域，分别对应四元式的操作符、操作数1、操作数2和结果。加入一条四元式即向中间代码管理器中的向量中插入一条新的四元式。  定义了函数get\_temp，用于获取当前应该生成的临时变量的名字。方法前面已经简述，即使用临时变量指针temp\_ptr，每当新建一个临时变量的时候，其名字就是对应temp\_ptr的值，新建完毕后就将临时变量指针加一。  **3.2)操作数与表达式处理**   1. string proc\_expression(const Node& Expression){ 2. string res; 3. if (Expression.children.size() == 1){ 4. res = proc\_operand(Expression.children[0]); 5. } 6. else{ 7. string op1 = proc\_operand(Expression.children[0]), op2 = proc\_operand(Expression.children[2]); 8. string opr = Expression.children[1].children[0].symbol; 9. string res = quaternions.get\_temp(); 10. quaternions.add\_quaternion(opr, op1, op2, res); 11. } 12. return "NIL"; 13. } 15. string proc\_operand(const Node& Operand){ 16. if(Operand.children[0].symbol == "Number\_Constant"){ 17. string val = Operand.children[0].children[0].symbol; 18. string res = quaternions.get\_temp(); 19. quaternions.add\_quaternion("LOAD\_CONST", val, "NIL", res); 20. } 21. else if(Operand.children[0].symbol == "Identifier"){ 22. string id\_name = Operand.children[0].children[0].symbol; 23. int scope = identifiers.find\_variable(scope\_list, id\_name); 24. if(scope){ 25. string res = quaternions.get\_temp(); 26. int offset = identifiers.variable\_list[scope][id\_name].offset; 27. quaternions.add\_quaternion("LOAD\_ID", to\_string(offset), "NIL", res); 28. } 29. else if(Operand.children[0].symbol == "("){ 30. return proc\_expression(Operand.children[1]); 31. } 32. } 33. return "NIL"; 34. }   操作数与表达式的语法制导翻译方案与实验三中的类似，需要在实验三实现的语法制导翻译方案上进一步扩充，实现中间代码生成的语义动作。  在处理操作数时，如果类型为常数类型，则对应的变量类型操作与实验三中的一致，在此处不再赘述。对于中间代码的生成，常数对应的运算符为LOAD\_CONSTANT，对应的操作数1为操作数的语义值，操作数2为空。如果类型为标识符类型，则需要在符号表中读取对应的符号信息，对应的变量类型操作与实验三中的一致，在此处不再赘述。对于中间代码的生成，常数对应的运算符为LOAD\_ID，对应的操作数1为内存偏移量，操作数2为空。  处理表达式时，默认此处的表达式如果不是一个单独的操作数，则为操作数+二元运算符+操作数的类型，分别对三个运算部分进行处理。对应的结果为当前应该生成的临时变量名，调用get\_temp函数获取当前应该生成的临时变量的名字，对应的四元式即为语义分析的操作符、操作数1、操作数2和临时变量名。  **3.3)赋值语句处理**   1. void proc\_assign(const Node& Assign){ 2. string id\_name = Assign.children[0].children[0].symbol; 4. int scope = identifiers.find\_variable(scope\_list, id\_name); 6. if(scope){ 7. string exp = proc\_expression(Assign.children[2]); 8. int offset = identifiers.variable\_list[scope][id\_name].offset; 9. quaternions.add\_quaternion(":=", exp, "NIL", to\_string(offset)); 10. } 11. }   对于赋值语句的处理，首先读取当前赋值语句左部标识符的字符串值，然后寻找对应标识符变量所在的作用域，如果能够找到作用域，则调用proc\_expression函数获取右部表达式的语义值并且生成对应的中间代码，然后计算当前变量所在内存空间中的偏移量，使用赋值运算符:=、表达式语义结果对应字符串和偏移量生成对应的四元式。  **3.4)返回语句处理**   1. void proc\_return(const Node& Return){ 2. string res = proc\_expression(Return.children[1]); 3. quaternions.add\_quaternion("RETURN", "NIL", "NIL", res); 4. }   对于返回语句的处理较为简单，首先通过调用proc\_expression函数得到“return”关键字右边表达式的语义值对应的字符串，然后使用RETURN运算符和“return”关键字右边表达式的语义值对应的字符串生成对应的四元式。  **3.5)函数调用处理**   1. void proc\_funcall(const Node& Funcall){ 2. string func\_name = Funcall.children[0].children[0].symbol; 3. if(func\_name == "get"){ 4. for(auto i : Funcall.children[2].children){ 5. string v\_name = i.children[0].children[0].children[0].symbol; 6. int scope = identifiers.find\_variable(scope\_list, v\_name); 7. int offset = identifiers.variable\_list[scope][v\_name].offset; 8. quaternions.add\_quaternion("READ", "NIL", "NIL", to\_string(offset)); 9. } 10. } 11. else if (func\_name == "put"){ 12. string res = proc\_operand(Funcall.children[2].children[0].children[0]); 13. quaternions.add\_quaternion("WRITE", "NIL", "NIL", res); 14. } 15. }   对于函数调用的处理首先需要对函数调用的实参进行处理，这里主要实现了测试样例1中要求的计算器对应的get和put函数，对于其他的函数的实现与此类似，不在此处赘述。对于get函数调用的每个实参，进行分析，分别获得它们的类型、名字，根据名字查找实参所在的作用域，计算实参在内存中地址的偏移量，然后使用READ操作符和偏移量生成对应的四元式。对于put函数调用则与get函数的性质类型但过程相反，首先读取结果应该存放的位置，然后使用WRITE运算符和结果存放位置生成对应的四元式。  **3.6)变量定义语句**   1. void proc\_variable\_definition\_without\_initial\_assignment(const Node& Variable\_Definition\_Without\_Initial\_Assignment){ 2. *//在符号表中构建新变量的项目* 3. Variable v; 4. v.type = Variable\_Definition\_Without\_Initial\_Assignment.children[0].children[0].symbol; 5. v.name = Variable\_Definition\_Without\_Initial\_Assignment.children[1].children[0].children[0].symbol; 6. v.value = ""; 8. if(identifiers.variable\_exist\_in\_scope(scope\_ptr, v.name)){ 9. seman\_error\_stream.push\_back(vector<string>{"[Redecleration of Variable]: Variable '" + v.name + "' already exists in the current scope.",Variable\_Definition\_Without\_Initial\_Assignment.start\_pos,Variable\_Definition\_Without\_Initial\_Assignment.end\_pos}); 10. } 11. else 12. identifiers.variable\_list[scope\_ptr][v.name] = v; 13. } 15. void proc\_variable\_definition\_with\_initial\_assignment(const Node& Variable\_Definition\_With\_Initial\_Assignment){ 16. *//先处理Variable Description再在符号表中构建新变量的项目* 17. string res = proc\_expression(Variable\_Definition\_With\_Initial\_Assignment.children[2].children[1]); 18. Variable v; 19. v.type = Variable\_Definition\_With\_Initial\_Assignment.children[0].children[0].symbol; 20. v.name = Variable\_Definition\_With\_Initial\_Assignment.children[1].children[0].children[0].symbol; 21. v.value = " "; 23. if(identifiers.variable\_exist\_in\_scope(scope\_ptr, v.name)){ 24. seman\_error\_stream.push\_back(vector<string>{"[Redecleration of Variable]: Variable '" + v.name + "' already exists in the current scope.",Variable\_Definition\_With\_Initial\_Assignment.start\_pos,Variable\_Definition\_With\_Initial\_Assignment.end\_pos}); 25. } 26. else 27. identifiers.variable\_list[scope\_ptr][v.name] = v; 28. int offset = identifiers.variable\_list[scope\_ptr][v.name].offset; 29. quaternions.add\_quaternion(":=", "0", "NIL", to\_string(offset)); 30. }   对于变量定义，如果是不带初始赋值的变量定义，那么就首先检查该作用域内是否存在重名变量，然后将该变量加入符号表中。将该变量的值设置为默认值空值，然后获取该变量的偏移量offset。使用操作符:=，操作数空和offset生成对应的四元式并加入四元式管理器中。  如果是带有初始赋值的变量赋值语句，那么首先需要检查该作用域内是否存在重名变量，然后将该变量加入符号表中，接着调用赋值语句处理函数proc\_Assign，获取赋值右部的语义值然后生成对应的四元式代码。  **3.7)分支语句**   1. void proc\_if(const Node& If){ 2. string if\_name = "IF" + to\_string(If.id); 3. proc\_expression(If.children[2]); 4. quaternions.quaternions.back().Result = if\_name; 6. for(auto i : If.children){ 7. if(i.symbol == "Else\_If") proc\_else\_if\_header(i); 8. else if(i.symbol == "Else") proc\_else\_header(i); 9. } 11. string exit\_name = "EXIT" + to\_string(If.id); 12. quaternions.add\_quaternion("GOTO", "NIL", "NIL", exit\_name); 14. proc\_if\_body(If, exit\_name); 15. for(auto i : If.children){ 16. if(i.symbol == "Else\_If") proc\_else\_if\_body(i, exit\_name); 17. else if(i.symbol == "Else") proc\_else\_body(i, exit\_name); 18. } 19. quaternions.add\_quaternion("LABEL", "NIL", "NIL", exit\_name); 20. }   处理分支语句时，依次对条件语句进行遍历，并如上述代码所示，按照中间代码生成的要求设置各自语句的跳转入口。  **3.8)循环语句**  本次实验中实现了While循环语句，其基本处理过程如下面代码中给出的函数所示：   1. void proc\_while(const Node& While){ 2. string while\_name = "WHILE" + to\_string(While.id); 3. string while\_block\_name = "WHILE BLOCK" + to\_string(While.id); 4. string while\_exit\_name = "EXIT" + to\_string(While.id); 6. quaternions.add\_quaternion("LABEL", "NIL", "NIL", while\_name); 7. proc\_expression(While.children[2]); 8. quaternions.quaternions.back().Result = while\_block\_name; 9. quaternions.add\_quaternion("GOTO", "NIL", "NIL", while\_exit\_name); 10. quaternions.add\_quaternion("LABEL", "NIL", "NIL", while\_block\_name); 11. proc\_block(While.children[4]); 12. quaternions.add\_quaternion("GOTO", "NIL", "NIL", while\_name); 13. quaternions.add\_quaternion("LABEL", "NIL", "NIL", while\_exit\_name); 14. }   在循环语句的处理中，需要进行大量的标签设置与跳转指令的实现。首先创建while标签while\_name、循环代码块标签while\_block\_name、退出地址跳转标签while\_exit\_name。然后调用proc\_expression函数处理布尔表达式，最后使用上述得到的条件进行四元式的生成。  **4)目标代码生成模块**   1. class ObjectCodeTranslator{ 2. public: 3. vector<Quaternion> internal\_code\_sequence; 4. vector<string> object\_code\_sequence; 6. void output\_object\_code(string fp){ 7. ofstream f(fp); 8. for(auto i : object\_code\_sequence) 9. f << i; 10. f.close(); 11. } 13. *//查询得出MIP目标机器一般使用32个寄存器，分别编号为31～0* 14. vector<int> regs = {31，30，29，28，27，26，25，24，23，22，21，20，19，18，17，16，15，14，13，12，11，10，9，8，7，6，5，4，3，2，1，0}; 16. *//寄存器状态存储数据结构* 17. map<string, int> regstatus; 19. ObjectCodeTranslator(vector<Quaternion> internal){ 20. internal\_code\_sequence = internal; 21. translate(); 22. } 24. void translate(){ 25. for(auto i : internal\_code\_sequence){ 26. transfer(i); 27. } 28. } 30. int getregs(string id){ 31. *//分配寄存器* 32. int av\_reg = regs.back(); 33. regs.pop\_back(); 34. regstatus[id] = av\_reg; 35. return av\_reg; 36. } 37. int recallregs(string id){ 38. *//收回寄存器* 39. int al\_reg = regstatus[id]; 40. regstatus.erase(id); 41. regs.push\_back(al\_reg); 42. return al\_reg; 43. } 45. void transfer(Quaternion inter){ 46. ostringstream oss; 47. string Operator = inter.Operator; 48. string Operand1 = inter.Operand1; 49. string Operand2 = inter.Operand2; 50. string Result = inter.Result; 52. if(Operator == "+"){ 53. oss<<"ADD $t" << getregs(Result) << " $t" <<recallregs(Operand1) << " $t" << recallregs(Operand2) << endl; 54. } 55. else if(Operator == "-"){ 56. oss<<"SUB $t" << getregs(Result) << " $t" <<recallregs(Operand1) << " $t" << recallregs(Operand2) << endl; 57. } 58. else if(Operator == "\*"){ 59. oss<<"MULT $t" << recallregs(Operand1) << " $t" << recallregs(Operand2) << endl; 60. oss<<"MFLO $t" <<getregs(Result) <<endl; 61. } 62. else if(Operator == "/"){ 63. oss<<"DIV $t" << recallregs(Operand1) << " $t" << recallregs(Operand2) << endl; 64. oss<<"MFLO $t" <<getregs(Result) <<endl; 65. } 66. else if(Operator == "RETURN"){ 67. oss << "MOVE $v0 $t" << recallregs(Result) << endl; 68. oss << "MOVE $ra $s0" << endl; 69. oss << "jr $ra" << endl; 70. } 71. else if(Operator == "LOAD\_CONST"){ 72. oss << "LI $t" << getregs(Result) << " " << Operand1 << endl; 73. } 74. else if(Operator == "LOAD\_ID"){ 75. oss << "LW $t" << getregs(Result) << " " << stoi(Operand1) << "($SP)" << endl; 76. } 77. else if(Operator == "LABEL"){ 78. oss << endl << Result << ":" <<endl; 79. } 80. else if(Operator == "READ"){ 81. oss << "JAL READ" <<endl; 82. oss << "SW $v0" << stoi(Result) << "($SP)" <<endl; 83. } 84. else if(Operator == "WRITE"){ 85. oss << "MOVE $a0 $t" << recallregs(Result) <<endl; 86. oss << "JAL WRITE" << endl; 87. } 88. else if(Operator == ">"){ 89. oss << "BGT $t" << recallregs(Operand1) << " $t" << recallregs(Operand2) << " " << Result <<endl; 90. } 91. else if(Operator == "<"){ 92. oss << "BLT $t" << recallregs(Operand1) << " $t" << recallregs(Operand2) << " " << Result <<endl; 93. } 94. else if(Operator == ">="){ 95. oss << "BGE $t" << recallregs(Operand1) << " $t" << recallregs(Operand2) << " " << Result <<endl; 96. } 97. else if(Operator == "<="){ 98. oss << "BLE $t" << recallregs(Operand1) << " $t" << recallregs(Operand2) << " " << Result <<endl; 99. } 100. else if(Operator == "="){ 101. oss << "BEQ $t" << recallregs(Operand1) << " $t" << recallregs(Operand2) << " " << Result <<endl; 102. } 103. else if(Operator == ":="){ 104. if(Operand1 == "0"){ 105. oss << "SW $zero" << " " << stoi(Result) << "($sp)" << endl; 106. } 107. else{ 108. oss << "SW $t" << recallregs(Operand1) << " " << stoi(Result) << "($sp)" << endl; 109. } 110. } 111. else if(Operator == "GOTO"){ 112. oss << "j " << Result << endl; 113. } 114. else if(Operator == "EXIT"){ 115. oss << "LI $v0 10" << endl <<"syscall" << endl; 116. } 117. object\_code\_sequence.push\_back(oss.str()); 118. } 120. };   目标代码生成类使用之前生成的中间代码，实现存储管理和目标代码的生成。首先定义了两个向量用于维护四元式序列和目标代码序列，定义了output\_object\_code函数进行目标代码的输出，步骤是使用输入的文件路径生成文件流，向文件流中输出每条目标代码的内容。  定义了有关寄存器涉及寄存器的存储进行管理，需要在中间代码翻译成目标代码的过程中进行数据结构的维护。分别维护两个向量，分别为可用寄存器regs和寄存器状态regstatus，分别用来存储当前所有可用的寄存器的名字，管理每个寄存器对应的分配状态。  通过查询MIPS目标机器的设计要求，发现寄存器数目一般是32个，故在此处设置32个寄存器，分别编号为31~0号，同时设置regstatus容器，用于存放管理每个寄存器对应的分配状态。  实现函数getregs获取当前临时变量应该得到的寄存器名，recallregs用于还回寄存器，即如果临时变量在之后的代码中不需要再被引用，即解除它占有的寄存器的存储分配状态，以供之后重新分配给其他临时变量。  Getregs函数用于获取现在应该分配给临时变量的寄存器，输入的参数为临时变量的名称；recallregs函数用于释放临时变量拥有的寄存器，并返回该寄存器，输入的参数为临时变量的名称。通过查询寄存器分配状态表，获取到该变量拥有的寄存器，将其释放并返回，同时更新寄存器分配状态表。  接下来定义了transfer函数，用于将单条四元式转换为MIPS目标代码，根据中间代码向目标代码的转换规则对不同的语句分别进行翻译即可。 | | | | | | |
| 四、实验结果及分析和（或）源程序调试过程   1. 必选测试样例: 2. 简易计算器 3. //输入数据num1,num2,op，根据op确定操作进行运算，最后输出运算结果ans 4. int num1,num2,op,ans; 5. get(num1,num2,op); 6. if(op==0) 7. { 8. ans = num1 + num2; 9. }; 10. if(op==1) 11. { 12. ans = num1 - num2; 13. }; 14. if(op==2) 15. { 16. ans = num1 & num2; 17. }; 18. if(op==3) 19. { 20. ans = num1 | num2; 21. }; 22. put(ans);   **目标代码结果：**   1. main: 2. J BLOCK 3. LI $v0 10 4. SYSCALL 5. BLOCK: 6. SW $zero 0($sp) 7. SW $zero -4($sp) 8. SW $zero -8($sp) 9. SW $zero -12($sp) 10. J READ 11. SW $v0 0($sp) 12. J READ 13. SW $v0 -4($sp) 14. J READ 15. SW $v0 -8($sp) 16. LW $t0 -8($sp) 17. LI $t1 0 18. BEQ $t0 $t1 IF56 19. J EXIT55 20. IF56: 21. LW $t0 0($sp) 22. LW $t1 -4($sp) 23. ADD $t0 $t0 $t1 24. SW $t0 -12($sp) 25. J EXIT55 26. EXIT55: 27. LW $t0 -8($sp) 28. LI $t1 1 29. BEQ $t0 $t1 IF88 30. J EXIT87 31. IF88: 32. LW $t0 0($sp) 33. LW $t1 -4($sp) 34. SUB $t0 $t0 $t1 35. SW $t0 -12($sp) 36. J EXIT87 37. EXIT87: 38. LW $t0 -8($sp) 39. LI $t1 2 40. BEQ $t0 $t1 IF120 41. J EXIT119 42. IF120: 43. LW $t0 0($sp) 44. LW $t1 -4($sp) 45. SW $t0 -12($sp) 46. J EXIT119 47. EXIT119: 48. LW $t0 -8($sp) 49. LI $t2 3 50. BEQ $t0 $t2 IF152 51. J EXIT151 52. IF152: 53. LW $t0 0($sp) 54. LW $t2 -4($sp) 55. SW $t0 -12($sp) 56. J EXIT151 57. EXIT151: 58. LW $t0 -12($sp) 59. MOVE $a0 $t0 60. J WRITE 61. READ: 62. LI $v0 4 63. LA $a0 prompt 64. SYSCALL 65. LI $v0 5 66. SYSCALL 67. J $ra 68. WRITE: 69. LI $v0 1 70. SYSCALL 71. LI $v0 4 72. LA $a0 end 73. SYSCALL 74. JR $ra 75. 跑马灯 76. *//循环输入op，改变输出结果out，输入0则结束程序* 77. int num0,num1,out,op; 78. num1 = 3333; 79. num2 = 6666; 80. num3 = 9999; 81. op = 1; 82. while(op>0) 83. { 84. if(op==1) 85. { 86. out = num1; 87. }; 88. if(op==2) 89. { 90. out = num2; 91. }; 92. if(op==2) 93. { 94. out = num3; 95. }; 96. put(out); 97. get(op); 98. };   **中间代码结果：**   1. LABEL NIL NIL main 2. GOTO NIL NIL BLOCK 3. EXIT NIL NIL NIL 4. LABEL NIL NIL BLOCK 5. := 0 NIL 0 6. := 0 NIL -4 7. := 0 NIL -8 8. := 0 NIL -12 9. := 0 NIL -16 10. LOAD\_CONST 3333 NIL temp0 11. := temp0 NIL 0 12. LOAD\_CONST 6666 NIL temp1 13. := temp1 NIL -4 14. LOAD\_CONST 9999 NIL temp2 15. := temp2 NIL -8 16. LOAD\_CONST 1 NIL temp3 17. := temp3 NIL -16 18. LABEL NIL NIL WHILE74 19. LOAD\_ID -16 NIL temp4 20. LOAD\_CONST 0 NIL temp5 21. > temp4 temp5 WHILE BLOCK74 22. GOTO NIL NIL EXIT74 23. LABEL NIL NIL WHILE BLOCK74 24. LOAD\_ID -16 NIL temp7 25. LOAD\_CONST 1 NIL temp8 26. = temp7 temp8 IF91 27. GOTO NIL NIL EXIT90 28. LABEL NIL NIL IF91 29. LOAD\_ID 0 NIL temp10 30. := temp10 NIL -12 31. GOTO NIL NIL EXIT90 32. LABEL NIL NIL EXIT90 33. LOAD\_ID -16 NIL temp11 34. LOAD\_CONST 2 NIL temp12 35. = temp11 temp12 IF118 36. GOTO NIL NIL EXIT117 37. LABEL NIL NIL IF118 38. LOAD\_ID -4 NIL temp14 39. := temp14 NIL -12 40. GOTO NIL NIL EXIT117 41. LABEL NIL NIL EXIT117 42. LOAD\_ID -16 NIL temp15 43. LOAD\_CONST 3 NIL temp16 44. = temp15 temp16 IF145 45. GOTO NIL NIL EXIT144 46. LABEL NIL NIL IF145 47. LOAD\_ID -8 NIL temp18 48. := temp18 NIL -12 49. GOTO NIL NIL EXIT144 50. LABEL NIL NIL EXIT144 51. LOAD\_ID -12 NIL temp19 52. WRITE NIL NIL temp19 53. READ NIL NIL -16 54. GOTO NIL NIL WHILE74 55. LABEL NIL NIL EXIT74 56. LOAD\_CONST 0 NIL temp20 57. RETURN NIL NIL temp20   **目标代码结果：**   1. main: 2. JAL BLOCK 3. LI $v0 10 4. SYSCALL 5. BLOCK: 6. MOVE $s0 $ra 7. SW $zero 0($sp) 8. SW $zero -4($sp) 9. SW $zero -8($sp) 10. SW $zero -12($sp) 11. SW $zero -16($sp) 12. LI $t0 3333 13. SW $t0 0($sp) 14. LI $t0 6666 15. SW $t0 -4($sp) 16. LI $t0 9999 17. SW $t0 -8($sp) 18. LI $t0 1 19. SW $t0 -16($sp) 20. WHILE74: 21. LW $t0 -16($sp) 22. LI $t1 0 23. BGT $t0 $t1 WHILE BLOCK74 24. J EXIT74 25. WHILE BLOCK74: 26. LW $t0 -16($sp) 27. LI $t1 1 28. BEQ $t0 $t1 IF91 29. J EXIT90 30. IF91: 31. LW $t0 0($sp) 32. SW $t0 -12($sp) 33. J EXIT90 34. EXIT90: 35. LW $t0 -16($sp) 36. LI $t1 2 37. BEQ $t0 $t1 IF118 38. J EXIT117 39. IF118: 40. LW $t0 -4($sp) 41. SW $t0 -12($sp) 42. J EXIT117 43. EXIT117: 44. LW $t0 -16($sp) 45. LI $t1 3 46. BEQ $t0 $t1 IF145 47. J EXIT144 48. IF145: 49. LW $t0 -8($sp) 50. SW $t0 -12($sp) 51. J EXIT144 52. EXIT144: 53. LW $t0 -12($sp) 54. MOVE $a0 $t0 55. J WRITE 56. J READ 57. SW $v0 -16($sp) 58. J WHILE74 59. EXIT74: 60. LI $t0 0 61. MOVE $v0 $t0 62. MOVE $ra $s0 63. JR $ra 64. READ: 65. LI $v0 4 66. LA $a0 prompt 67. SYSCALL 68. LI $v0 5 69. SYSCALL 70. JR $ra 71. WRITE: 72. LI $v0 1 73. SYSCALL 74. LI $v0 4 75. LA $a0 end 76. SYSCALL 77. JR $ra 78. 自定义测试样例: 79. int f(int a){ 80. return a+1; 81. } 82. int y = 1; 83. int main(){ 84. int x; 85. bool b = true; 86. *//x = f(1)+1;* 87. x = 2 + 1; 88. int z; 89. z = --x; 90. f(3); 91. while(true){ 92. z += 2; 93. } 94. return 0; 95. }   **中间代码结果：**   1. LABEL NIL NIL main 2. LOAD\_CONST 1 NIL temp0 3. := temp0 NIL 0 4. GOTO NIL NIL BLOCK 5. EXIT NIL NIL NIL 6. LABEL NIL NIL BLOCK 7. := 0 NIL -4 8. := NIL NIL -8 9. LOAD\_CONST 2 NIL temp1 10. LOAD\_CONST 1 NIL temp2 11. + temp1 temp2 temp3 12. := temp3 NIL -4 13. := 0 NIL -12 14. := NIL NIL -12 15. LABEL NIL NIL WHILE BLOCK125 16. GOTO NIL NIL EXIT125 17. LABEL NIL NIL WHILE BLOCK125 18. GOTO NIL NIL WHILE125 19. LABEL NIL NIL EXIT125 20. LOAD\_CONST 0 NIL temp4 21. RETURN NIL NIL temp4   **目标代码结果：**   1. main: 2. LI $t0 1 3. SW $t0 0($sp) 4. J BLOCK 5. LI $v0 10 6. SYSCALL 7. BLOCK: 8. MOVE $s0 $ra 9. SW $zero -4($sp) 10. SW $t0 -8($sp) 11. LI $t0 2 12. LI $t0 1 13. ADD $t0 $t0 $t0 14. SW $t0 -4($sp) 15. SW $zero -12($sp) 16. SW $t0 -12($sp) 17. WHILE BLOCK125: 18. J EXIT125 19. WHILE BLOCK125: 20. J WHILE125 21. EXIT125: 22. LI $t0 0 23. MOVE $v0 $t0 24. MOVE $ra $s0 25. JR $ra 26. READ: 27. LIO $v0 4 28. LA $a0 prompt 29. SYSCALL 30. LI $v0 5 31. SYSCALL 32. JR $ra 33. WRITE: 34. LI $v0 1 35. SYSCALL 36. LI $v0 4 37. LA $a0 end 38. SYSCALL 39. JR $ra | | | | | | |
| 五、实验总结  问题：中间代码生成时，出现无法向文件流输出文件的问题  问题描述：起初编码完成后发现无论如何运行都无法向文件中输出中间代码，后来在StackOverflow上查询发现是MACOS系统XCode版本问题，对于filestream的操作不兼容。  解决方法：切换平台运行程序，正常输出。  总结：个人进行实验时认为本次实验中最大的难点是如何处理错误以及如何分配寄存器。通过本学期所有实验的实现，我手工完成了一个简单的编译系统的实现，这给我带来了很大的成就感，也使我更加深入地了解了编译系统的各种原理。感谢整个过程中助教学长和老师给予的悉心指导和耐心帮助！！  **提交说明：** | | | | | | |