**简单C语言编译器**

202083290503 胡天乐

202083290374 刘 辉

202083290153 王子荀

202083290400 鲁哲豪

1. 实验介绍

本实验是南京信息工程大学编译原理课程实践课课程设计，要求实现一个简易的C语言编译器。

项目已开源至GitHub： <https://github.com/NTMRWSB/Simple-C-Compiler>

项目主要内容包括：

1. 基本功能

词法分析 20分

语法分析（构建语法树）40分

中间代码生成（如四元式）10分

类型检查 20分

1. 附加功能

代码优化 5分

错误分析 5分

汇编程序 10分

1. 实验要求
2. 基本要求

变量类型：int

语句：注释，声明，赋值，循环（while和for），判断（if），输入输出

算术运算：+，-，\*，/，%，^

关系运算：==，<，>，<=，>=，!=

逻辑运算：&&（与），||（或），!（非）

拓展功能

支持过程和函数调用

支持数组运算

支持指针运算

支持结构体

1. 小组成员及分工

202083290503 胡天乐（词法分析、汇编代码生成）

202083290374 刘 辉（词法分析、语法分析）

202083290153 王子荀（语法分析、测试）

202083290400 鲁哲豪（语法分析、中间代码生成）

1. 本组主要自创与改进的部分说明
   1. 运用lex和yacc工具进行词法/语法分析，产生中间代码文件Innercode。
   2. 使用C++从中间代码生成汇编语言，运用C++中STL库的map、set等容器，从中间代码文件高效地转化为汇编代码。
   3. 可输出移入-归约冲突和归约-归约冲突，也可以设置优先级以避免冲突。
   4. 重构了语法树代码，改为抽象语法树，去除语法树中的冗余信息，保留必要的信息，提升了其性能和简洁性
   5. 将原本的链表符号表改为链式哈希表，确定好变量的定义域和复位义问题，并且效率更佳。
   6. 根据本组以上所有修改内容，重新书写了Makefile文件。
2. 实验内容
3. 环境准备及使用说明

运行环境：Windows WSL2.0（Ubuntu）或 Ubuntu 22.04 LTS，请不要在ARM平台下编译该项目，可能出现意想不到的错误。运行前，需要安装必要的编译组件、Flex/Bison、NASM、CMAKE，并在文件根目录下运行。相关指令如下：

|  |
| --- |
| sudo apt install build-essential  sudo apt install flex bison nasm  sudo apt install cmake |

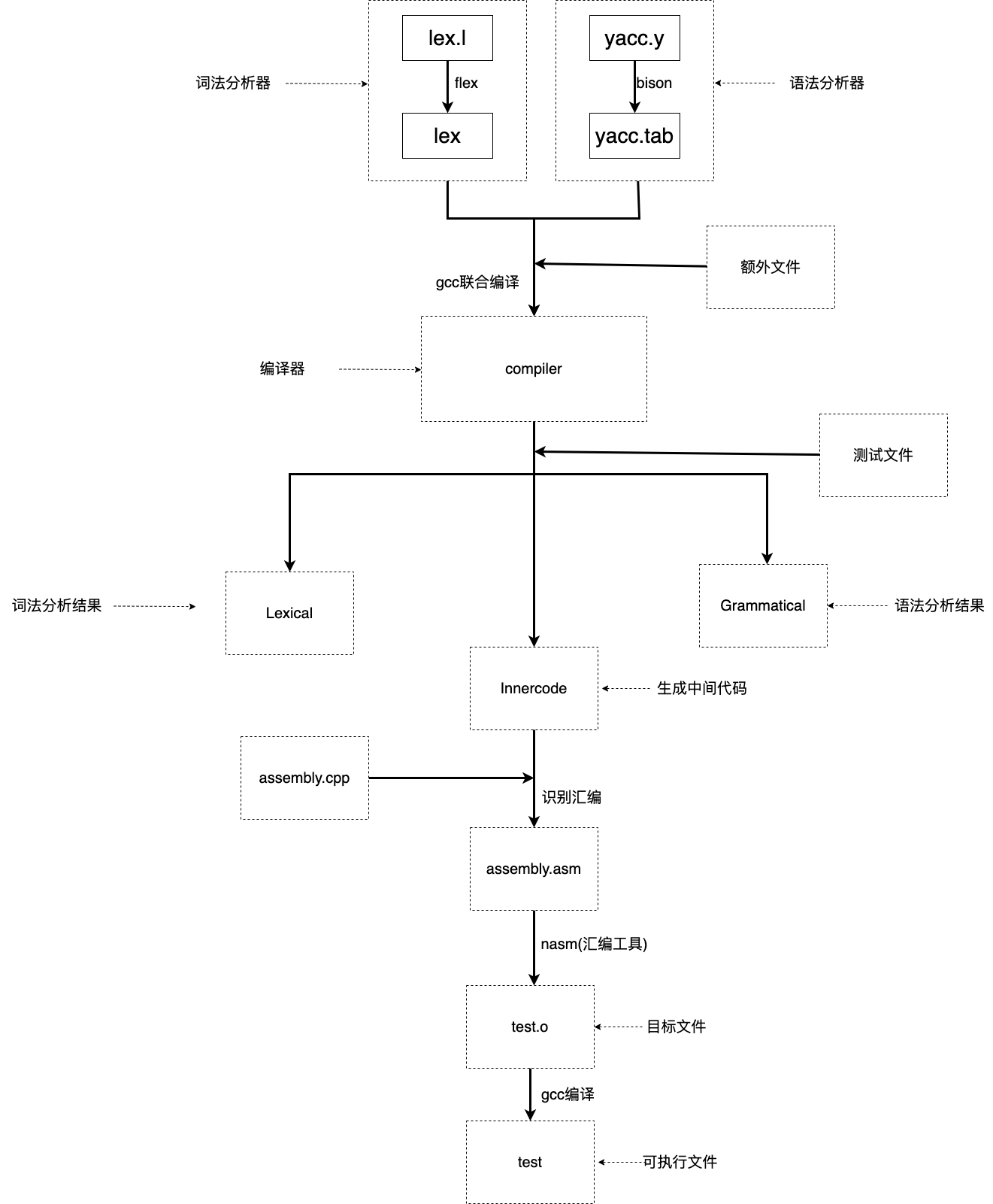
有两种方式可以在文件根目录下运行：

|  |
| --- |
| make // 默认文件名为test.c  make file=<filename> // filename指要编译的C代码文件 |

分析结果产生在当前目录下

|  |
| --- |
| 词法分析：Lexical  语法分析：Grammatical  中间代码：InnerCode  汇编代码：assembly.asm  可执行文件：<filename> |

1. 项目架构图



1. 词法分析

经过调研与选型，我们决定使用lex作为词法分析器。通过词法分析器，生成源代码相应的词法分析文件。

部分类型说明：

| 词素 | 词法单元 | 介绍 |
| --- | --- | --- |
| return | RET | 返回 |
| if/while/for | IF/WHILE/FOR | 判断及两种循环标识符 |
| break/continue | BREAK/CONTINUE | 流程控制 |
| int | INT | 整型变量 |
| input\_int/output\_int | INPUT/OUTPUT | 输入/输出 |
| id | ID | 变量名 |
| + - \* / | + - \* / | 四则运算符 |
| % ^ | % ^ | 模运算及乘方运算符 |
| && || ！ | AND/OR/NOT | 逻辑运算符 |

词法分析结果:

|  |
| --- |
| 单词 词素 属性  INT int  MAIN main  LP (  RP )  LCB {  INT int  ID a 0x5604106792a0  ASSIGN =  INT10 8 8  SEMICOLON ;  INT int  ID b 0x5604106792a0  ASSIGN =  INT10 10 10  SEMICOLON ;  IF if  LP (  ID a 0x5604106792a0  EQ ==  INT10 8 8  RP )  LCB {  OUTPUT output\_int  LP (  ID a 0x5604106792a0  RP )  SEMICOLON ;  RCB }  INT int  ID i 0x5604106792a0  ASSIGN =  INT10 2 2  SEMICOLON ;  FOR for  LP (  SEMICOLON ;  ID i 0x5604106792a0  LT <  INT10 3 3  SEMICOLON ;  ID i 0x5604106792a0  ASSIGN =  ID i 0x5604106792a0  ADD +  INT10 1 1  RP )  LCB {  OUTPUT output\_int  LP (  ID b 0x5604106792a0  RP )  SEMICOLON ;  RCB }  INT int  ID c 0x5604106792a0  ASSIGN =  ID b 0x5604106792a0  MOD %  ID a 0x5604106792a0  SEMICOLON ;  OUTPUT output\_int  LP (  ID c 0x5604106792a0  RP )  SEMICOLON ;  INT int  ID d 0x5604106792a0  ASSIGN =  INT10 5 5  POW ^  INT10 2 2  SEMICOLON ;  OUTPUT output\_int  LP (  ID d 0x5604106792a0  RP )  SEMICOLON ;  RETURN return  INT10 0 0  SEMICOLON ;  RCB } |

1. 语法分析

使用yacc接受lex产生的token，并对程序的语法进行语法分析，最终打印出抽象语法树。

* 1. 部分语句文法

|  |
| --- |
| //if语句文法  //设置优先级解决移入-归约冲突  %precedence ')'  %precedence ELSE  ......  ......  if\_expression  : if\_identifier ')' statement  | if\_identifier ')' statement ELSE statement  ;  if\_identifier  : IF '(' operate\_expression  ;  //for语句文法  for\_expression  : FOR '(' nullable\_expression ';' nullable\_expression ';' nullable\_expression ')' statement  ;  //while文法  while\_expression  : WHILE '(' operate\_expression ')' statement  ; |

* 1. 语法树构造

使用n叉树来构造抽象语法树。

|  |
| --- |
| //语法树数据结构  typedef struct Tree{  char\* content; //词素  char\* name; //名字  int line; //行号  int num; //子节点数量  int headline; //语句段起始行号  int nextline; //下一个语句段行号  char\* inner; //存储中间变量  char\* code; //中间代码  struct Tree\*\* leaves; //叶子节点  struct Tree\* next; //链表指针  struct Declator\* declator; //修饰符  }Tree;  //初始化树  Tree\* initTree(int num);  //创建树  Tree\* createTree(char\* name, int number, ...);  //在lex文件里创建终结符节点  Tree\* terminator(char\* name, int yylineno);  //二元运算符创建树  Tree\* binaryOpr(char\* name, Tree\* t1, Tree\* t2, Tree\* t3);  //赋值运算符创建树  Tree\* assignOpr(char\* name, Tree\* t1, Tree\* t2, Tree\* t3);  //一元运算符创建树  Tree\* unaryOpr(char\* name, Tree\* t1, Tree\* t2);  //if语句创建树  Tree\* ifOpr(char\* name,int headline,int nextline,Tree\* op,Tree\* stmt);  //if-else语句创建树  Tree\* ifelseOpr(char\* name, int headline, int next1, int next2, Tree\* op, Tree\* stmt1, Tree\* stmt2);  //while语句创建树  Tree\* whileOpr(char\* name,int head1, int head2,int nextline,Tree\* op,Tree\* stmt);  //for语句创建树  Tree\* forOpr(char\* name,int head1, int head2, int nextline, Tree\* op1, Tree\* op2, Tree\* op3, Tree\* stmt);  Tree\* retNull(char\* name,Tree\* ret);  Tree\* retOpr(char\* name,Tree\* ret,Tree\* op);  //输入输出函数创建树  Tree\* unaryFunc(char\* name,Tree\* func, Tree\* op);  // 添加修饰符  Tree\* addDeclator(char\* name, Tree\* t1, Tree\* t2);  //打印树  void printTree(Tree\* tree);  //释放树指针  void freeTree(Tree\* tree);  //语法树创建函数  Tree\* createTree(char\* name, int number, ...){ //c语言可变长参数  int i;  va\_list valist;  va\_start(valist, number);  if(number == 1){  return va\_arg(valist, Tree\*);  }  Tree\* tree = initTree(1);  if(!tree){  printf("Out of space \n");  exit(0);  }  tree->num = number;  int len = strlen(name);  tree->name = (char\*)malloc(len + 1);  memcpy(tree->name, name, len + 1);  char \*str= (char\*)malloc(MAX);  tree->leaves = (Tree\*\*)malloc(sizeof(Tree\*)\*number);  for(i = 0; i < number; i++){  tree->leaves[i] = va\_arg(valist, Tree\*); //创建叶子节点  if(tree->leaves[i]->code)  strcat(str,tree->leaves[i]->code); //初始化叶子节点并合并中间代码  }  if(strlen(str)>0)  tree->code = str;  return tree;  } |

语法树示例：

|  |
| --- |
| Main Func  INT int  MAIN main  LP (  RP )  LCB {  sentence  declare\_expression  INT int  assignment\_expression =  ID a  INT10 8  sentence  declare\_expression  INT int  assignment\_expression =  ID b  INT10 10  sentence  if\_expression  equality\_expression ==  ID a  INT10 8  statement  LCB {  output  OUTPUT output\_int  ID a  RCB }  sentence  declare\_expression  INT int  assignment\_expression =  ID i  INT10 2  sentence  for\_expression  null  relational\_expression <  ID i  INT10 3  assignment\_expression =  ID i  additive\_expression +  ID i  INT10 1  sentence  declare\_expression  INT int  assignment\_expression =  ID c  multiplicative\_expression %  ID b  ID a  sentence  output  OUTPUT output\_int  ID c  sentence  declare\_expression  INT int  assignment\_expression =  ID d  multiplicative\_expression ^  INT10 5  INT10 2  sentence  output  OUTPUT output\_int  ID d  return\_expression  RETURN return  INT10 0  RCB } |

* 1. 符号表创建

为提高存取效率，我们将原本的链表结构的符号表进行了修改，改用链式哈希表实现。也便于就解决变量复位义和作用域问题。

|  |
| --- |
| typedef struct Declator{ //修饰符(array or pointer)  int type;  struct Tree\* length;  struct Declator\* next;  }Declator;    typedef struct Data{  char\* id\_name; //变量名  int type; //变量类型  void\* adress; //存储地址  int size; //大小  int scope; //作用域  struct Declator\* declator;  }Data;    typedef struct HashNode{  Data\* data;  struct HashNode\* next;  }HashNode;    typedef struct HashMap{  int size;  HashNode\*\* hash\_table;  }HashMap;  //Hash Function  unsigned int RSHash(const char\* str, unsigned int len);  //创建哈希表  HashMap\* createHashMap(int size);  //变量转化为data类型  Data\* toData(int type, char\* str, struct Declator\* declator, int scope);  //释放节点  void freeHashNode(HashNode\* hashNode);  //分配内存  void getSize(HashNode\* hashNode);  //创建哈希节点  HashNode\* createHashNode(Data\* data);  //add data to hashMap  HashNode\* put(HashMap\* hashMap, Data\* data);  //将树中的变量存入符号表中  void putTree(HashMap\* hashMap, struct Tree \*tree);  //find data in hashMap, return NULL when error  HashNode\* get(HashMap\* hashMap, Data\* data);  //语句块退出时销毁当前块符号表  void destoryPartOfHashMap(HashMap\* hashMap, int scope);  void destoryHashMap(HashMap\* hashMap); |

* 1. 中间代码生成

根据抽象语法树生成中间代码。产生时，子节点的中间代码依次向上合并，最终在根节点产生出完整的中间代码。

|  |
| --- |
| typedef struct Node{  char\* inner; //中间代码变量  char\* op; //存储算符  char\*\* var; //存储多个变量名  int num; //存储变量个数（子节点个数）  struct Node\* next;  }Node;    //获取双目运算中间代码  struct Node\* getNodeByDoubleVar(char\* op, char\* var0, char\* var1,int inner\_count);  //获取单目运算中间代码  struct Node\* getNodeBySingleVar(char\* op, char\* var,int inner\_count);  //拼接多个字符串  char\* mergeCode(int num, ...);  //生成行号  char\* lineToString(int number);  //int转换为char\*  char\* toString(int number);  //替换字符串  int swap(char \*text, char \*a, char \*b);  /\*  获取双目运算符的中间代码  在链表中寻找相同的运算  若运算已存在，则直接返回中间代码变量，无需生成  若不存在，则创建新的中间代码变量，并向列表中插入该运算的信息  \*/  Node\* getNodeByDoubleVar(char\* op, char\* var0, char\* var1,int inner\_count){  Node \*p = head;  while (p)  {  if (p->num = 2 && !strcmp(p->var[0],var0) && !strcmp(p->var[1],var1) && !strcmp(p->op,op))  {  Node \*new\_node = (Node\*)malloc(sizeof(Node));  new\_node->op = NULL;  new\_node->inner = p->inner;  return new\_node;//返回相同的中间代码指针  }  p=p->next;  }  Node \*new\_node = (Node\*)malloc(sizeof(Node));  new\_node->var = (char\*\*)malloc(sizeof(char\*)\*2);//二目运算符  new\_node->inner = (char\*)malloc(20);//中间代码最高20位  new\_node->num = 2;  sprintf(new\_node->inner,"t%d",inner\_count++);    new\_node->var[0] = (char\*)malloc(strlen(var0)+1);  new\_node->var[1] = (char\*)malloc(strlen(var1)+1);  new\_node->op = (char\*)malloc(strlen(op)+1);    memcpy(new\_node->var[0], var0, strlen(var0)+1);  memcpy(new\_node->var[1], var1, strlen(var1)+1);  memcpy(new\_node->op, op, strlen(op)+1);    if(head){  new\_node->next = head;  }    head = new\_node;  return head;  } |

中间代码运行结果：

|  |
| --- |
| 1: a = 8  2: b = 10  3: t1 = a == 8  4: if t1 goto 6  5: goto 8  6: arg a  7: call output  8: i = 2  9: t2 = i < 3  10: if t2 goto 12  11: goto 17  12: arg b  13: call output  14: t3 = i + 1  15: i = t3  16: goto 9  17: t4 = b % a  18: c = t4  19: arg c  20: call output  21: t5 = 5 ^ 2  22: d = t5  23: arg d  24: call output  25: return 0 |

1. 类型检查

由于该编译器目前只支持int型变量，无需进行类型检查。

1. 代码优化

在生成中间代码的过程中，我们逐步进行了中间代码格式的优化。例如，生成中间代码时会寻找过去是否已经生成了相同运算的中间代码，可以减少部分不必要的代码。

1. 错误分析

错误分析部分包括以下内容：

lex文件中实现了词素无法匹配的问题。

yacc文件中实现了在读入源程序时的代码匹配问题，产生错误时会返回行号。

符号表的实现解决了变量的复位义和未定义问题。

1. 汇编程序

读取生成的中间代码文件，产生相应的nasm64位汇编。使用C++实现汇编语言的生成。

实现思路：

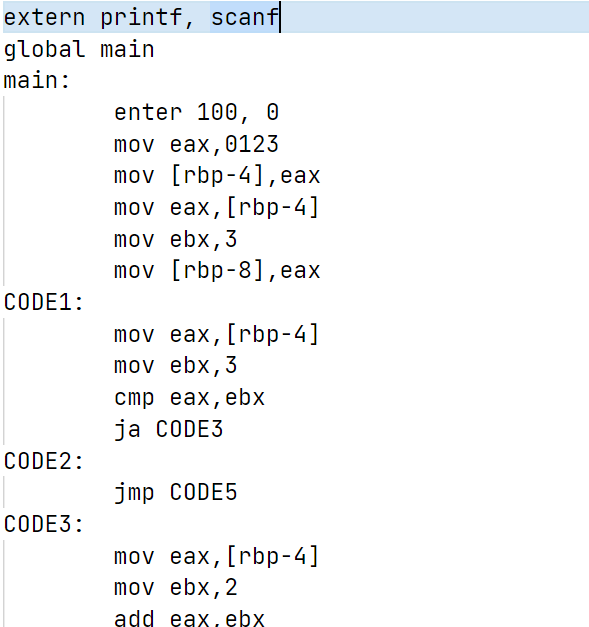
打开中间代码文件并读取。

根据中间代码插入相应的四元组向量。

1. 根据四元式的操作符类型生成相应的代码块和汇编代码（注意跳转）。
2. 写汇编文件。

|  |
| --- |
| #include <algorithm>  #include <fstream>  #include <iostream>  #include <map>  #include <set>  #include <sstream>  #include <string>  #include <vector>  using namespace std;  string emptystr = "empty";  // 四元组结构体  struct Quadruple {  string s0, s1, s2, s3;  };  // 语句类型和汇编语言的对应表  map<string, string> asmMap = {  {"j==", "JE"}, {"%", ""}, {"^", ""}, {"=", "mov"}, {"+", "add"},  {"-", "sub"}, {"\*", "mul"}, {"/", "div"}, {">=", "GE"}, {"<=", "LE"},  {">", "GT"}, {"<", "LT"}, {"==", ""}, {"!=", "ne"}, {"||", "and"},  {"&&", "or"}, {"!", "not"}, {"j", "jmp"}, {"ret", "RET"}, {"arg", "arg"},  {"call", "call"}};  bool isNumber(string str) {  return str.find\_first\_not\_of("0123456789") == string::npos;  }  template <typename T>  int indexV(vector<T> v, T t) {  if (find(v.begin(), v.end(), t) == v.end())  return -1;  else  return find(v.begin(), v.end(), t) - v.begin();  }  template <typename T>  bool isInVector(vector<T> v, T t) {  return find(v.begin(), v.end(), t) != v.end();  }  int main() {  vector<Quadruple> quad; // 四元组向量，记录多个四元组  // 1 打开Innercode文件  ifstream ifile("Innercode"); // 读名为Innercode的文件  if (!ifile) {  cout << "找不到名为Innercode的文件！" << endl;  return 1; // 非正常退出  }  // 2 根据Innercode创建四元组向量  string line; // line用于读取Innercode中的每一行  while (getline(ifile, line)) {  // cout << line << endl;  // 2.1 读入一行，替换此行所有换行符（这个方法是在algorithm头文件里）  line.erase(remove(line.begin(), line.end(), '\n'), line.end());  // 2.2 按空格分割此行的内容，放入临时变量wds里  vector<string> wds; // wds=words，储存语句的内容  stringstream input(line); // 读取str到字符串流中  string temp;  while (getline(input, temp, ' ')) {  wds.push\_back(temp);  }  // 2.3 判断类型并插入四元组向量  if (wds[1] == "if") // 若相等compare返回0  quad.push\_back({"j==", wds[2], "1", wds[4]});  else if (wds[1] == "goto")  quad.push\_back({"j", "\_", "\_", wds[2]});  else if (wds[1] == "arg" || wds[1] == "call")  quad.push\_back({wds[1], "\_", "\_", wds[2]});  else if (wds[1] == "return")  quad.push\_back({"ret", "\_", "\_", wds[2]});  else if (wds.size() > 3 && wds[3] == "!")  quad.push\_back({"!", wds[4], "\_", wds[1]});  else if (wds[1][0] == 't')  quad.push\_back({wds[4], wds[3], wds[5], wds[1]});  else  quad.push\_back({wds[2], wds[3], "\_", wds[1]});  }  // ~1 关闭Innercode文件  ifile.close();  // 3 为所有转移（j==与j）记录CODE位置  int n = 0;  set<int> t;  for (auto element : quad) {  if (element.s0[0] == 'j') { // 如果是条件转移"j=="或无条件转移"j"  t.insert(n);  t.insert(stoi(element.s3) - 1);  }  ++n;  }  int m = 0;  map<int, string> quad\_in;  for (int it : t) {  quad\_in[it] = "CODE" + to\_string(++m);  }  // 4 创建assembly.asm文件，写入固定内容[1]  ofstream ofile("assembly.asm", fstream::out);  string tab = " "; // 默认的缩进（8空格）  string inx = "[rbp-";  ofile << "extern printf, scanf" << endl  << "global main" << endl  << "main:" << endl  << tab + "enter 100, 0" << endl;  // 5 写入实际的汇编内容  vector<string> data;  for (int i = 0; i < 100; ++i) data.push\_back(emptystr);  vector<string> judge = {">", "<", "==", ">=", "<=", "!="};  for (int i = 0; i < quad.size(); ++i) {  if (quad\_in.count(i)) ofile << quad\_in[i] + ":" << endl;  if (quad[i].s0 == "j==") { // 条件转移，则需要记录条件  if (isInVector(judge, quad[i - 1].s0)) {  if (isNumber(quad[i - 1].s1))  ofile << tab + "mov eax," + quad[i - 1].s1 << endl;  else  ofile << tab + "mov eax," + inx  << to\_string(indexV(data, quad[i - 1].s1) \* 4 + 4) + "]"  << endl;  if (isNumber(quad[i - 1].s2))  ofile << tab + "mov ebx," + quad[i - 1].s2 << endl;  else  ofile << tab + "mov ebx," + inx  << to\_string(indexV(data, quad[i - 1].s2) \* 4 + 4) + "]"  << endl;  ofile << tab + "cmp eax,ebx" << endl;  if (quad[i - 1].s0 == ">")  ofile << tab + "" + "ja" + " " + quad\_in[stoi(quad[i].s3) - 1]  << endl;  else if (quad[i - 1].s0 == "<")  ofile << tab + "" + "jb" + " " + quad\_in[stoi(quad[i].s3) - 1]  << endl;  else if (quad[i - 1].s0 == ">=")  ofile << tab + "" + "jae" + " " + quad\_in[stoi(quad[i].s3) - 1]  << endl;  else if (quad[i - 1].s0 == "<=")  ofile << tab + "" + "jbe" + " " + quad\_in[stoi(quad[i].s3) - 1]  << endl;  else if (quad[i - 1].s0 == "==")  ofile << tab + "" + "je" + " " + quad\_in[stoi(quad[i].s3) - 1]  << endl;  else if (quad[i - 1].s0 == "!=")  ofile << tab + "" + "jne" + " " + quad\_in[stoi(quad[i].s3) - 1]  << endl;  } else {  if (isNumber(quad[i].s2))  ofile << tab + "mov eax," + quad[i].s2;  else  ofile << tab + "mov eax," + inx  << to\_string(indexV(data, quad[i].s2) \* 4 + 4) + "]" << endl;  ofile << tab + "cmp eax,1" << endl  << tab + "je " + quad\_in[stoi(quad[i].s3) - 1] << endl  << endl;  }  } else if (quad[i].s0 == "=") { // 赋值  data[indexV(data, emptystr)] = quad[i].s3;  if (isNumber(quad[i].s1))  ofile << tab + "mov eax," + quad[i].s1 << endl;  else  ofile << tab + "mov eax," + inx  << to\_string(indexV(data, quad[i].s1) \* 4 + 4) + "]" << endl;  ofile << tab + "mov " + inx  << to\_string(indexV(data, quad[i].s3) \* 4 + 4) + "],eax" << endl;  } else if (quad[i].s0 == "j") { // 转移  ofile << tab + "jmp " + quad\_in[stoi(quad[i].s3) - 1] << endl;  } else if (quad[i].s0 == "ret") { // 返回  ofile << tab + "leave" << endl << tab + "ret" << endl;  } else if (quad[i].s0 == "arg" || quad[i].s0 == "call") { // 函数  if (quad[i].s0 == "arg") {  ofile << tab + "sub rsp, 8" << endl  << tab + "mov rsi," + inx  << to\_string(indexV(data, quad[i].s3) \* 4 + 4) + "]" << endl;  } else if (quad[i].s0 == "call" && quad[i].s3 == "output") {  ofile << tab + "mov rdi, out\_format" << endl  << tab + "xor rax, rax" << endl  << tab + "call printf" << endl  << tab + "xor rax, rax" << endl  << tab + "add rsp, 8" << endl;  } else if (quad[i].s0 == "call" && quad[i].s3 == "input") {  ofile << tab + "mov rsi, number" << endl  << tab + "mov rdi, in\_format" << endl  << tab + "xor rax, rax" << endl  << tab + "call scanf" << endl  << tab + "mov rbx, [number]" << endl  << tab + "add rsp, 8" << endl;  }  } else { // 运算  if (isNumber(quad[i].s1))  ofile << tab + "mov eax," + quad[i].s1 << endl;  else if (quad[i].s1 != "\_")  ofile << tab + "mov eax," + inx  << to\_string(indexV(data, quad[i].s1) \* 4 + 4) + "]" << endl;  if (isNumber(quad[i].s2))  ofile << tab + "mov ebx," + quad[i].s2 << endl;  else if (quad[i].s2 != "\_")  ofile << tab + "mov ebx," + inx  << to\_string(indexV(data, quad[i].s2) \* 4 + 4) + "]" << endl;  if (quad[i].s0 == "+" || quad[i].s0 == "-")  ofile << tab + asmMap[quad[i].s0] + " eax,ebx" << endl;  else if (quad[i].s0 == "\*")  ofile << tab + "mul ebx" << endl;  else if (quad[i].s0 == "/")  ofile << tab + "div ebx" << endl;  else if (quad[i].s0 == "%") {  ofile << tab + "div ebx" << endl;  ofile << tab + "mov eax,edx" << endl;  } else if (quad[i].s0 == "^")  ofile << tab + "times " + quad[i].s2 + " mul eax" << endl;  else if (quad[i].s0 == "&&" || quad[i].s0 == "||")  ofile << tab + asmMap[quad[i].s0] + " eax,ebx" << endl;  else if (quad[i].s0 == "!")  ofile << tab + asmMap[quad[i].s0] + " eax" << endl;  else if (isInVector(judge, quad[i].s0))  ;  else  ofile << tab + "----运算符未定义----" << endl;  data[indexV(data, emptystr)] = quad[i].s3;  ofile << tab + "mov " + inx  << to\_string(indexV(data, quad[i].s3) \* 4 + 4) + "],eax" << endl;  }  }  // ~4 写入固定内容[2]，关闭文件  ofile << endl  << "section .data" << endl  << tab + "out\_format: db \"%#d\", 10, 0" << endl  << tab + "in\_format: db \"%d\", 0" << endl  << "section .bss" << endl  << tab + "number resb 4" << endl;  ofile.close();  return 0;  } |

某代码的汇编代码的生成如下，证明上述代码能够生成允许条件/非条件跳转的汇编代码。



1. 实验中遇到的问题及解决方法
2. 问题一：在设计文法时，产生了很多移入-归约冲突和归约-归约冲突。

移入-归约冲突指的是在语法分析过程中，在同一个状态下，有多个产生式的产生符号的下一个符号相同，编译器不知道应该使用哪个产生式来处理输入的符号。归约-归约冲突指的是在语法分析过程中，在同一个状态下，有多个产生式的产生符号的下一个符号相同，并且可以使用归约来消除这些符号。

解决方式：我们测试了以下三种方式来解决：

在编译命令中加入-v参数，在生成的output文件中查看文法具有的冲突；

修改文法；

设置优先级以避免冲突。

问题二：语法树结构冗余，无法很好的用于生成中间代码。

解决方式：可以通过优化语法树结构来减少冗余（删除结点、合并结点或者重新排列结点等）。

我们选择将代码重构，将语法分析树改为抽象语法树来解决这个问题。

问题三：无法确定好变量的定义域和复位义问题，且链表符号表性能不佳。

解决方式：原始方案采用了链表符号表，使用链表来存储键值对，其支持插入、删除和查找操作，时间复杂度为O(n)。我们选用的链式哈希表使用哈希表来存储键值对，每个结点都保存着一个键值对，并使用指针将结点连接起来。链式哈希表的时间复杂度为O(1)。

链式哈希表在查找性能上有明显的优势。因此我们将链表符号表改为链式哈希表，可以解决这个问题。

问题四：将中间代码Innercode转化为汇编语言需要大量数据结构的支持，且条件跳转、无条件跳转等逻辑较为复杂。

解决方式：相比Python，C++的执行效率要更高，且C++也具有完整的STL库，其中的集合和字典等类型非常适合完成中间代码Innercode转化汇编语言的动作。查询相关资料和汇编语言语法，发现条件跳转、无条件跳转可以通过在.asm文件中增加标签完成（类似C语言中的goto）。将条件跳转后执行的一段代码块统一用序号记录，并接在主语句后，就可以完成汇编语言中的跳转功能。

1. 总结

本次实践中，我们使用lex与yacc开发了一个简单的C语言编译器，通过lex与yacc解析输入的代码并生成抽象语法树，实现词法分析、语法分析和语义分析等编译器功能。

在使用lex与yacc的过程中，我们遇到了语法分析中的冲突、语法树处理和汇编代码转换等问题，通过查阅资料得以解决。在实践中，我们巩固了有关《编译原理》课程的知识，如正则表达式、词法/语法分析、抽象语法树的构建等。因此，我们更加深入地理解编译器的工作原理和流程，认识到了《编译原理》课程的理论指导和实践意义。

实验中也存在一些可以改进的地方，如实现的变量类型较少、未实现函数的读取和编译、汇编代码的生成难以做到优化。如果进一步进行编译器的开发，可以添加数组、指针类型和尝试中间代码优化。