**《编译原理》实验报告**

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| **姓名** | |  | | | **年级** |  |
| 学号 | |  | | | **专业、班级、** |  |
| **实验名称** | **实验三.语义分析** | | | | | |
| **实验时间** | **2019/5/20** | | **实验地点** | **主教404** | | |
| **实验成绩** | **9** | | **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：9  评价教师签名：张敏 | | | | | | |
| 一、实验目的  在词法及语法分析程序的基础上，编写一个程序对使用类C语言书写的源代码进行语法制导翻译，语义分析及类型检查，并打印分析结果。 | | | | | | |
| 二、实验项目内容   1. 程序要能够检查源代码中是否符合以下语义要求： 2. 最低要求3.1： 3. 能够实现对整型（int）和浮点型（float）变量的类型检查，两类变量不能相互赋值及运算；仅整型及浮点型变量才能参与算术运算； 4. 实现整数计算器的语义动作定义及语法制导翻译，并绘制出相应抽象语法树； 5. 为表达式“（2+3”生成错误信息“丢失右括号”； 6. 为表达式“2 3”生成错误信息“丢失运算符”； 7. 为表达式“（2+）”生成错误信息“丢失操作数”； 8. 其他要求3.2: 9. 能判断源代码是否符合以下语义假设并给出相应错误具体位置： 10. 函数仅能定义一次、程序中所有变量均不能重名、函数不可嵌套定义 11. 定位源代码中的错误位置 12. 其他要求3.3: 13. 能检查结构体中域是否与变量重名，不同结构体中域是否重名； 14. 编写一个FreeSymbolTable函数，在main函数的最后调用此函数时，可以讲整个符号表的内存释放，避免内存泄露 | | | | | | |
| 三、实验过程或算法（源程序）  **1.语义分析的理解**  通过阅读相关的资料，我了解到语义分析是编译程序的一个重要程序。由语义分析程序可以检查名字的定义和引用是否合法从而使程序能够合理的组织在一起，之后的中间代码生成需要通过语义分析的检测。  方法：语义分析通过将变量的定义与变量的引用联系起来，对源程序的含义进行检查即检查每个语法成分是否具有正确的语义。通常为编译程序设计一个称作**符号表**的数据结构来保存上下文有关的信息。当分析声明的语句时候，收集所声明标识符的有关信息并记录在符号表中。在每次对标识符进行检索或者插入的操作。  **2.语义分析程序的设计**  2.1语义分析的位置  由于在第二个实验中bison对源程序进行分析之后生成了一个分析树，所以语义分析可以基于分析树上，自顶向下的递归的分析，然后自底向上的回收符号表。  2.2 符号表的设计  符号表的结点设计：  首先对于一个标识符一定有自己的名字。 这里用char的数组存放：  char name[MAX\_NAME\_LEN];  标识符需要有自己的类型，这里设计了类型的结构体Type\_ 如下：  struct Type\_  {      enum {BASIC,ARRAY,STRUCTURE} kind;      union      {          int basic;  //0:int     1:float          struct {Type elem;int size;} array;          FieldList structure;      }u;};  在类型中需要记录是普通变量还是数组还是结构体变量。如果是普通变量0代表int型，1代表float类型。如果是数组，需要记录数组的元素类型和个数。如果是结构体需要记录结构体是哪一种。  对于结构体的记录，这里又需要用结构体来存储每个自定义结构的域的名字，类型，用tail指向下一个域。如下：  struct FieldList\_  {      char\* name;      Type type;      FieldList tail;  };  这里对于标识符，我将其分成了两种，变量或者函数标识符。变量可以用过name和type记录所有的信息，函数还需要记录函数参数的个数和是否定义。所以符号表的结点如下：  struct Symbol\_  {      char name[MAX\_NAME\_LEN];      Type type;      enum {VARIABLE,FUNCTION} kind;      int numOfArgument;      Type\* argument;      int ifDefined;          //0:not defined     1:defined  };  2.3符号表的组织  在设计完符号表的内容之后，需要将每个符号表的结点组织起来以便于查询或者插入等操作。组织符号表可以有很多的方法，比如利用栈的方式或者利用哈希列表的方式。为了便于快速的查询，这里我使用了哈希的方式组织符号表，每个散列表的结点内容包括符号表，冲突指针，指向下一个的指针。这里还有一个num，代表符号表的层级。来区分不同的定义域，如何将层级赋值将在后面解释。  struct HashNode\_  {      Symbol sym;      unsigned int num;      HashNode left;      //open hashing      HashNode next;      //in one compst  };  设计的具体思想见<<编译原理与技术>>的P163页.有设计两个数组,一个是散列表,一个是栈式符号表.  HashNode stack[20];  HashNode symbolTable[16384];  C:\Users\HuJun\Documents\Tencent Files\1319579758\FileRecv\MobileFile\IMG_20190525_183148.jpg  因为符号表的空间比较大所以哈希函数的设计是通过字符数组name获得不同的哈希值的.name越大哈希值越大.name不同哈希值也不同.如下  //哈希函数  int unsignedinthash\_pjw(char\* name) {  unsigned int val = 0, i;  for (; \*name; ++name)  {  val = (val << 2) + \*name;  if (i = val & ~0x3fff)  val = (val ^ (i >> 12)) & 0x3fff;  }  return val;  }  另外这里还设计了一个类型指针链表记录定义的结构体的类型.  FieldList typeHead;    **2.3 符号表的操作**  符号表的主要操作就是插入和检索.  插入符号表:首先通过name获取哈希值,然后在插入之前需要检测是否发生冲突.如果发生了冲突需要进行判断:作用域是否相同,如果作用域相同,会判断然后返回并报错.具体判断如下:  if (symbolTable[n]->num == layer)  {  if (symbolTable[n]->sym->kind == VARIABLE && sym->kind == VARIABLE)  return -1;  else if (symbolTable[n]->sym->kind == FUNCTION && sym->kind == FUNCTION)  {  if(symbolTable[n]->sym->ifDefined == 1 && sym->ifDefined == 1)  return -1;  else if(symbolTable[n]->sym->ifDefined == 0 && sym->ifDefined == 1)  {  if(func\_equal\_func(symbolTable[n]->sym,sym) == 0)  return -2;  else  {  symbolTable[n]->sym->ifDefined = 1;  return 1;  }  }  else  {  if(func\_equal\_func(symbolTable[n]->sym,sym) == 0)  return -2;  else  return 0;  }  }  }  如果没有冲突会生成一个符号表然后插入进去.  检索符号表:通过标识符的名字和类型检测符号表并返回符号表指针.  Symbol search\_symbol(char\* name,int kind){  unsigned int n = unsignedinthash\_pjw(name);  HashNode node = symbolTable[n];  while(node != NULL)  {  if(kind == 0 && node->sym->kind == VARIABLE)  return node->sym;  else if(kind == 1 && node->sym->kind == FUNCTION)  return node->sym;  node = node->left;  }  return NULL;  }  **3.语义分析过程**  整个语义分析的过程的思想:根据建树的过程一致,在符号表建立的同时进行语义分析.  先介绍几个主要的函数的作用：  Debug（）：语义分析的主函数，递归的分析分析树，每次遇到不同的带有语义的产生式的时候，将进行不同的语义分析。  get\_type\_of\_specifier（）：识别传入节点的类型，返回一个type类型的指针。  add\_type（）：传入由定义的结构体增加的specifier，然后添加到结构体定义的栈中。  Type get\_type(treeNode\* exp)：根据传入的表达式返回表达式结果的类型。  void add\_symbol\_s\_v(treeNode\* specifier,treeNode\* vardec)：对于一个声明变量的产生式将，变量的类型和标识符传入，然后构造符号表结点并添加到符号表中。  void de\_function(treeNode\* specifier,treeNode\* fundec,int ifDef)：将函数添加到符号表中，并且记录函数的参数内容和个数。ifDef代表当前的产生式是声明函数的产生式还是定义函数的产生式。  有了上述的函数作为辅助的作用，然后就可以根据产生式对分析树进行分析了。  接下来对分析过程解释，整个的过程是根据产生式的从前到后的顺序分析的。也就是树的从顶到底的顺序。  对于：  ExtDef : Specifier SEMI 需要添加一个类型。如下  //定义一个结构体：增加一个类型  if (t->children[2] == NULL)  {  compst\_or\_struct = 1;  add\_type(t->children[0]);  layer++;  }  ExtDefList : ExtDef ExtDefList  定义变量或者数组的时候需要添加进符号表中。  else if (strcmp(t->children[1]->name,"ExtDecList") == 0)  {  treeNode\* extDecList = t->children[1];  while(extDecList != NULL)  {  add\_symbol\_s\_v(t->children[0],extDecList->children[0]);  extDecList = extDecList->children[2];  }  }  Specifier FunDec SEMI 函数的声明的时候只需要将函数的名字和返回值通过de\_function()函数记录到符号表中。  Specifier FunDec CompSt函数定义的时候，需要对于函数的参数记录下来。并且进入一个函数的时候需要将层级也就是layer增加，在分析完之后再减1.  //函数的定义：          else  {  de\_function(t->children[0],t->children[1],1);  layer++;              //如果定义的函数有参数  if(t->children[1]->children[3] != NULL)  {  treeNode\* varlist = t->children[1]->children[2];  while(varlist != NULL)  {                      //将函数的形参加入到符号表中  add\_symbol\_s\_v(varlist->children[0]->children[0],varlist->children[0]->children[1]);  varlist = varlist->children[2];  }  }  layer--;  }  Def : Specifier DecList SEM  如果是变量的声明：需要将每个变量加入到符号表中去，如果声明的时候赋值则类型检查。使用add\_symbol\_s\_v（）函数，就可以直接添加。如图：  else if(strcmp(t->name,"Def") == 0){  Type type = get\_type\_of\_specifier(t->children[0]);  treeNode\* declist = t->children[1];  while(declist != NULL)  {  add\_symbol\_s\_v(t->children[0],declist->children[0]->children[0]);  if(declist->children[0]->children[1] != NULL)  if(type\_equal\_type(type,get\_type(declist->children[0]->children[2])) == 0)  printf("Error type 5 at Line %d:Type mismatched for assignment.\n",t->line);  declist = declist->children[2];  }  }  c程序是块结构语言。程序中所有的块由CompSt标识，每个块就对于作用域进行了分割。程序中通过layer的大小表示层级。  else if(strcmp(t->name,"CompSt") == 0) {  compst\_or\_struct = 0;  layer++;  }  还有一个主要的分析对象是表达式EXP的分析：表达式的产生式的形式最多。  对于右边只是一个标识符的EXp来说，只用检索符号表有没有定义就可以了。  Exp： Exp DOT ID 对于结构体的引用，如struct st.s这种表达式来说需要检测结构体中有没有定义这个域。如下：  else if(strcmp(t->children[1]->name,"DOT") == 0)  {  Type type = get\_type(t->children[0]);  if(type != NULL)  { if(type->kind != STRUCTURE)  printf("Error type 13 at Line %d:Illegal use of \".\"\n",t->line);  else  {  FieldList stru = type->u.structure;  while(stru != NULL)  {  if(strcmp(stru->name,t->children[2]->content) == 0)  break;  stru = stru->tail;  }  if(stru == NULL)  printf("Error type 14 at Line %d:Non-existent field \"%s\"\n",t->children[0]->line,t->children[2]->content); }  }  }  Exp： Exp LB Exp RB 对于数组的应用需要做两个事情，第一查看左边的是不是一个数组，第二查看中括号里面的那个表达式是不是整数。这个可以通过get\_type函数直接获取。  else if(strcmp(t->children[1]->name,"LB") == 0)  {  Type type = get\_type(t->children[0]);  if(type != NULL)  {  if(type->kind != ARRAY)  {  printf("Error type 10 at Line %d:\"",t->children[0]->line);  print\_node(t->children[0]);  printf("\" is not an array.\n");  }  }  type = get\_type(t->children[2]);  if(type != NULL)  {  if(!(type->kind == BASIC && type->u.basic == 0))  { printf("Error type 12 at Line %d:\"",t->children[2]->line);  print\_node(t->children[2]);  printf("\" is not an integer.\n");  }  }  }  Exp : ID LP Args RP函数的引用需要知道函数有没有被申明有没有被定义。如果不满足这两个条件也会有语义错误。可以通过search\_symbol()函数和ifDef标识来确定。如下：  else if(strcmp(t->children[1]->name,"LP") == 0)  {  Symbol sym = search\_symbol(t->children[0]->content,1);  if(sym == NULL)  {  if(search\_symbol(t->children[0]->content,0) == NULL)  printf("Error type 2 at Line %d:Undefined function \"%s\"\n",t->line,t->children[0]->content);  else  printf("Error type 11 at Line %d:\"%s\" is not a function\n",t->line,t->children[0]->content);  }  else  {  if(sym->ifDefined == 0)  printf("Error at Line %d:Function \"%s\" has not been defined.\n",t->line,t->children[0]->content);  if(t->children[3] != NULL && args\_func\_consistent(t->children[2],sym) == 0)  {  printf("Error type 9 at Line %d: Function ",t->line);  print\_func\_args\_type(sym,1);  printf("is not applicable for arguments ");  print\_func\_args\_type(t->children[2],2);  printf("\n");  }  }  }  函数中还有一个需要考虑的是，返回值的类型是不是和当前函数的类型一致。如果不一致也有语义错误。如下：  else if(strcmp(t->name,"Stmt") == 0 && strcmp(t->children[0]->name,"RETURN") == 0){  Type type = get\_type(t->children[1]);  if(type\_equal\_type(type,typeOfFunction) == 0)  printf("Error type 8 at Line %d:Type mismatched for return.\n",t->line);  typeOfFunction = NULL;  }  最后要考虑的是类型检测，有了上面的一系列的辅助函数类型检测就很容易实现了，只需要将表达式左右两边都通过get\_type获取类型再进行比较就可以了。如下：  else if(t->children[2] != NULL && strcmp(t->children[2]->name,"Exp") == 0)  {  Type t1 = get\_type(t->children[0]);  Type t2 = get\_type(t->children[2]);  if(t1 != NULL && t2 != NULL && type\_equal\_type(t1,t2) == 0)  {  if(strcmp(t->children[1]->name,"ASSIGNOP") == 0)  printf("Error type 5 at Line %d:Type mismatched for assignment.\n",t->line);  else  printf("Error type 7 at Line %d:Type mismatched for operands.\n",t->line);  }  if(strcmp(t->children[1]->name,"ASSIGNOP") == 0)  if(can\_be\_assigned(t->children[0]) == 0)  printf("Error type 6 at Line %d:The left-hand side of an assignment must be a variable.\n",t->line);  }  **4.符号表释放和退栈的过程**  由于语义分析是通过递归的方式分析的，所以根据产生是在分析完一个块即Compst的时候需要，将块中的局部变量释放并且将层级layer自减。代码如下：  HashNode head = stack[layer];  stack[layer] = NULL;  while(head != NULL)  { if(head->sym->kind == VARIABLE)  { unsigned int n = unsignedinthash\_pjw(head->sym->name);  symbolTable[n] = head->left;  HashNode temp = head;  head = head->next;  free(temp);  }  else  head = head->next;  }  layer--  **5.抽象语法树的绘制生成。**  由于实验要求是整数计算器的语义动作定义的相对的抽象语法树，而计算器中主要表达式是Exp的产生式，所以在这里就只对于Exp的产生式进行处理。如下：  void opExp(treeNode\* node){  if(node->children[1]==NULL){  assignNode(node,node->children[0]);  node->children[0] = NULL;  }  if(node->children[1]!=NULL){  // assignNode(node,node->children[1]);  strcpy(node->name,node->children[1]->name);  node->children[1]=NULL;  }  }  void createAST(treeNode\* node){  if(node==NULL) return;  if(strcmp(node->name,"Exp")==0)  opExp(node);  int i=0;  for(;i<7;i++){  createAST(node->children[i]);}}  方法也是从上到下的递归。如果Exp的右部只有一个标识符或者一个常量的时候，就将其替换成他的子结点。如果右边是一个表达式，则将其替换成操作符，然后将其子节点替换成两个操作数。然后一直递归操作下去。 | | | | | | |
| 四、实验结果及分析和（或）源程序调试过程  **1.运行步骤：**  直接点击程序文件中的.bat运行，然后结果会在output上面显示。（所有的运行，我已经在批处理中写好了）。  **2.运行结果：**  这次的语义分析的实验，我写的语义分析可以检查19种错误，接下来先展示实验要求的错误，再展示其他错误。所有出错的事例我在test.c文件中写出来了，而且在后面有标识会出现第几种类型的错误，所有的错误的说明在说明的文档中。先贴出所有报错的信息，避免重复帖图。    图4.2.1  2.1  首先是要求3.1.1 的类型检查：    图4.2.2  如图，定义了整数c和浮点数g。然后将g赋值给c的时候报错。即两类变量不能相互赋值及运算；  3.1.2 接着是整数计算器的抽象语法树。  对于这个要求，我重新写了一个简单的测试文件calTest.c更简单看出抽象语法分析树与原来的语法分析树的不同，测试结果在calOutput中。如图  测试文件  int main(){  int res;  res = 1+2-3\*4/5;    }  结果：    图4.2.3  可以看出抽象语法分析树（右边）比原来的分析树（左边）简单很多。  3.1.3以及3.1.4 3.1.5的测试文件在上面以及给出  结果如下：    图4.2.4  对于这个要求，是在语法分析的时候做的（可以在parser.y文件中看到）。对于要求的三个例子都报错了，但是还是错误恢复了。继续生成分析树，对下一行不影响。  3.2测试结果  测试文件如下：    图4.2.5  在第九行重复定义了函数fun1，然后在报错图片4.2.1中可以看到错误类型4：重复定义函数。并且定位了错误位置。  3.3测试结果  如图4.2.5 在第20行的结构体的域名重复定义了，所以会产生第15类型的错误。域名重复。  第23行，定义了相同的结构体，产生错误16.重复定义结构体的错误。  **3.扩展的语义分析功能**  程序其他的语义分析的功能如下，我将其列举出来了，并且在test.c中写了对应的错误案例，可以对比图4.2.1和4.2.2；  还有值得提的一点就是对于错误类型的报错，程序中不仅定位了错误在那一行。而且还会说明错误的内容。我是通过print\_node（）函数实现的，此函数递归的查询到数组的名字并显示。对于函数的报错还可以将函数应该有的类型显示出来并且与调用的时候出错的类型进行对比，这里通过print\_func\_args\_type（）函数来实现，效果如下  Error type 9 at Line 46: Function "fun1(int,int)"is not applicable for arguments "(int,float)"  最后附上错误类型说明。  语法分析错误类型：  type：1  变量未定义  type：2  函数未定义  type：3  变量重定义  type：4  函数重定义  type：5  不同类型相互操作  type：6  被赋值得左边不是变量  type：7  运算符两边类型不同  type：8  函数返回值得类型不匹配  type：9  函数传入得参数类型不对  type：10  引用未定义得数组  type：11  引用了为定义的函数  type：12  数组的下标不是整数  type：13  非法的使用 .  type：14  引用不存在的结构体的域  type：15  结构体中域重名  type：16  结构体定义重名  type：17  使用了没有定义的结构体类型  type：18  使用了没有实现的函数  type：19  函数定义与声明不一致 | | | | | | |
| **五．实验分工与心得**   |  |  |  | | --- | --- | --- | | 组员 | 工作 | 相对工作量 | | 葛胡军 | 设计，函数功能与符号表的框架，实现语义分析的data\_structure.h文件，实现debug函数模块的实现，最终代码综和调试 | 40% | | 朱春雪 | 实现功能性函数的代码部分，如：操作符号表的两个函数，获取类型函数，增加类型函数，识别符号类型函数等。 | 30% | | 颜无双 | 实现交互型函数的编写与测试代码的编写，对于十几种的错误类型报错的信息提示，和两个错误打印代码，和test文件。 | 30% |   心得： 一个特别明显也非常显著的规律：当你一步步向前走的时候，偶尔回头看看就会发现曾经让你觉得特别艰难的事情，也不过如此。在写实验二的时候觉得实验二已经挺复杂的了，可是当看到实验三要求的时候第一反应竟然是有点想逃避，但在队友的鼓励下，先自己去复习了课本上的知识点，大概理解原理之后也开始尝试着去网上学习别人已经写好的代码。先理解每一部分的作用然后理解各个函数之间的关系。参照网上已经写好的代码框架，把我们自己的东西替换过去，似乎听起来很容易的一件事，但是在真正的实现起来的时候，我们大部分的时间都在调bug，一开始最现实的问题在于实验二的语法分析树，当时自己写的时候没有考虑到那么多，输出的语法分析树是没有办法直接用在实验三里面的，所以我们需要对实验二中的语法分析树进行一个优化去掉其冗余的部分。接着就是漫长的根据需要添加函数，实现函数，最终编码测试的过程了。然后在整个实验过程中也再一次让我感受到了团队合作的重要性。希望自己在下一次的实验中仍然不要畏惧，和队友一起尽自己最大的努力去解决问题，锻炼coding的能力，最终圆满完成任务。 | | | | | | |