编译原理研讨课实验PR002实验报告

任务说明

本次实验的内容主要是对 CACT 源程序进行语义分析：

1. 根据给出的 CACT 文法和语义规范，补充完善 .g4 文件，添加节点属性和进出产生式的语义动作；
2. 通过 AntLR 生成访问语法树的接口，对输入的 CACT 语言源代码进行语义分析；
3. 遇到语义错误时，进行正确的处理

成员组成

代瀚堃（2019K8009929051）：语义分析框架的搭建，实现符号表、作用域的数据结构，关于 main 函数的检查，检查重名、检查变量定义，实现嵌套作用域，完善组员代码

孙彬（2019K8009906001）：实现函数声明和返回值的检查，以及实参与形参匹配

郑天羽（2019K8009907006）：实现运算数类型是否匹配的检查，以及数组运算相关的检查

三位组员一起讨论了符号表的设计和作用域的表示

实验设计

设计思路

1. 符号表的设计

函数和同一作用域内的变量都是唯一的，不允许重名，这正好与 C++ 中 map 数据结构的特点相符，另一方面，由于我们不需要强调函数和变量在符号表中的顺序，为了提高速度，我们采用基于 hash 的 unordered_map，map中的每一个元素是一个符号表项，记录类型、函数/变量名、是否是数组等信息。

2. 嵌套作用域

按照 CACT 规范，程序中的作用域是树状的，也就是全局作用域为根节点，函数和全局 block 为其子节点，而它们的子节点又为二级子节点……于是我们可以组织一个树状的数据结构，树的每个节点表示一个作用域，节点内包含一个 vector 结构，每个 vector 元素指向一个子节点，同时节点内还包含一个指向父节点的指针，用于向上级查找变量，变量符号表项中添加一个成员指向树中的节点，表示该变量归属于某一作用域。最后再用一个变量指向当前作用域。

3. 语义分析

使用 AntLR 生成的访问语法树的接口，定义进出规则节点时需要进行的语义动作，在此过程中检查源程序是否符合语义规则，将函数和变量添加进符号表，并收集类型信息，为代码生成提供支持。如果发现语义错误，应当输出错误提示并返回非零值。

实验实现

1. 符号表中记录的信息

   考虑到函数有形参，变量有嵌套作用域等因素，我们将函数和变量的符号表分开实现，下面是函数符号表中的一个表项：

   ```
   typedef struct func_symbol_item {
       std::string func_name;
       val_type_t ret_type;
       std::map <std::string, func_fparam_item_t> fparam_list;
   } func_symbol_item_t;
   ```

   其中，`func_name`为函数名，`ret_type`为返回值类型(枚举类型`val_type_t`，分为`void_type, int_type, float_type, double_type, bool_type`)，而`fparam_list`为函数的形参列表，其中的表项定义如下：

   ```
   typedef struct func_fparam_item {
       std::string fparam_name;
       val_type_t fparam_type;
       int which_fparam; // order
       bool is_array;

       // overload for compare operator to guarantee the order of fparams
       bool operator < (func_fparam_item const &item) const {
           return which_fparam < item.which_fparam;
       }
   } func_fparam_item_t;
   ```

   `fparam_name`为形参的名字，用于检查形参是否重名，`fparam_type`为形参的类型，用于检查形参与实参的类型是否匹配，`which_fparam`用于记录形参在函数声明中的顺序(因为 map 默认按字典序排，我们要求按照形参在声明中出现的先后排，并重载小于运算符指定排序方式)，`is_array`表示该形参是否为数组(是则为`true`)，没有实现为 vector 是因为检查参数重名时需要频繁比较，而 unordered_map 的 find 方法则能高效地完成

   于是函数的符号表定义如下，以函数名为索引

   ```
   std::unordered_map <std::string, func_symbol_item_t> func_symbol_table;
   ```

   变量符号表中的一个表项如下：

```
typedef struct var_symbol_item {
    std::string var_name;
    val_type_t var_type;
    bool is_array;
    bool is_const;
    size_t num_items;
    scope_node_t * scope;
} var_symbol_item_t;
```

与函数符号表项类似，`is_array` 表示变量是否为数组，`num_items` 表示是数组时含有的元素个数，`is_const` 表示该变量是否为常量(从而只能在定义时赋值)，`scope` 指向该变量所归属的那一个作用域节点，于是变量的符号表定义如下：

```
std::unordered_map <name_in_scope, var_symbol_item_t, pair_hash_func> \
var_symbol_table;
```

由于同名变量可以存在于不同的作用域中，所以我们以变量名和它所在的作用域组成的二元组为该表的索引，二元组 `name_in_scope` 的定义如下：

```
typedef struct name_scope_pair {
    std::string var_name;
    scope_node_t * scope_ptr;

    // overload for compare operator to compare two pairs
    bool operator == (const name_scope_pair &pair) const {
        return (pair.scope_ptr == scope_ptr) && (pair.var_name == var_name);
    }
} name_in_scope;
```

`scope_ptr` 指向变量所在的作用域节点，重载相等运算符，要求只有当变量名和作用域都相同时，两个 pair 才相同。另一方面，我们还需要定义变量符号表所使用的 hash 函数：

```
// override for hash function
// will not create a new object, but return its hash value
struct pair_hash_func {
    size_t operator()(const name_scope_pair &pair) const {
        return (std::hash<std::string>()(pair.var_name)) ^
(std::hash<scope_node_t *>()(pair.scope_ptr) << 1);
    }
};
```

2. 检查函数、同一作用域中的变量重名、实现嵌套作用域

   由于 CACT 规范要求函数应先定义再使用，且函数内不允许嵌套函数，相当于函数都只位于全局作用域内，其重名检查可以调用 map 结构的 find 方法完成。

   而同名变量允许存在于不同作用域中，于是我们需要将变量与作用域关联，这里我们选择将所有变量实现在一个符号表内，并在每个表项中用一个指针指向它归属的作用域，同名变量通过该指针来区分。

   一个作用域表示为 n 叉树中的一个节点，其中 n 是它的子作用域个数，并用一个指针指向父节点，在引用变量时，如果当前作用域内未定义过，需要通过该指针向上级作用域查找，直至到根节点，即全局作用域。于是节点的数据结构定义如下：

```
// Scope tree for variable
typedef struct scope_node {
    scope_node * father;
    std::vector <scope_node *> children;
} scope_node_t;
```

并定义根节点的指针 `scope_root` 和当前作用域的指针 `current_scope`：

```
// Global scope
scope_node_t * scope_root;
scope_node_t * current_scope;
```

当进入到一个 CompUnit 时，创建一个新的 `scope_node_t` 对象，赋给 `scope_root`，表示全局作用域，并将当前作用域指向全局作用域。此后，每进入一个新的作用域（即进入一个 block），就新建一个节点，通过 push_back 添加到作用域树中，在退出 block 时恢复到原来的作用域。

特别地，对于函数体的 block，由于形参和局部变量不能同名，于是我们为 block 添加属性 `from_func` 和 `which_func`，表示该 block 是否属于函数体，以及它属于的函数名，当进入到一个 block 时，如果 `from_func` 为真，则需要根据 `which_func` 所指的函数名，到函数的符号表中找到形参，将形参添加到新的作用域内。

3. 检查运算的两个操作数是否同类型、赋值语句的左值和右值是否同类型

在 CACT.g4 中为运算相关的非终结符设计了一系列的 val_type 属性，在退出相应节点时，向父节点传送此属性，若遇到两个运算数属于不同类型，则报错并退出。例如在函数

```
void
SemanticAnalysis::exitMulExp_mul_unary(CACTParser::MulExp_mul_unaryContext *
ctx)
```

中有如下程序片段：

```
 // Check same type
 val_type_t type_1 = (val_type_t)(ctx -> mulExp() -> mul_exp_val_type);
 val_type_t type_2 = (val_type_t)(ctx -> unaryExp() -> unary_exp_val_type);
 if(type_1 != type_2){
     std::cout << "Type mismatch: " << type_1 << " and " << type_2 <<
std::endl;
     exit(ERROR);
 }

 // Type pass
 ctx -> mul_exp_val_type = (int)type_2;
```

4. 检查数组运算是否合法

·检查运算的数组长度是否相等

与检查操作数类型一致地，定义属性 `is_array` 和 `num_items` 并将其与 val_type 类型一起向上传递，其中 `is_array` 是 bool 类型，代表变量是否是数组，`num_items` 是 int 类型，表示数组的长度（非数组变量的此属性设置为 1 ）。在退出相应运算节点时，向父节点传递此二属性，并且在 `is_array` 都为 true 时检查两数组的长度是否相等。

例如在函数

```
void
SemanticAnalysis::exitMulExp_mul_unary(CACTParser::MulExp_mul_unaryContext *
ctx)
```

中的如下片段：

```
  // Check same type
  bool is_array_1   = (bool)(ctx -> mulExp() -> is_array);
  int  num_items_1  = (int)(ctx -> mulExp() -> num_items);
  bool is_array_2   = (bool)(ctx -> unaryExp() -> is_array);
  int  num_items_2  = (int)(ctx -> unaryExp() -> num_items);

  // check array & length
  if(is_array_1 != is_array_2){
      std::cout << "Operation between array and variable." << std::endl;
      exit(ERROR);
  }

  if(is_array_1 && is_array_2 && num_items_1!=num_items_2){
      std::cout << "Operation between arrays with different length." <<
std::endl;
      exit(ERROR);
  }

  // Type pass
  ctx -> is_array = (bool)is_array_2;
  ctx -> num_items = (int)num_items_2;
```

·检查数组是否进行了逻辑运算

在 CACT 规范中，数组仅能进行 '+' , '-' , '*' , '/' , '%' 这五种运算，在退出逻辑运算的相应节点时进行检查，若存在一操作数的`is_array`属性为 true ， 则报错并退出。

·检查进行算术运算的数组是否为bool类型

在退出算术运算的相应节点时进行检查，若存在一操作数的 val_type 类型为 bool，且其 ```is_array``` 属性为 true，则报错并退出。
例如如下代码段：

``````C++
// bool array in arithmetic operation
if(is_array_1 && (type_1 == bool_type)|| is_array_2 && (type_2 == bool_type))){
    std::cout << "Bool array in arithmetic operation." << std::endl;
    exit(ERROR);
}
``````

5. 检查函数调用是否合法

· 检查函数在调用前是否被声明

先使用`map`的`find()`方法，根据函数名在函数符号表中进行搜索，若函数在调用前没有被声明，则会返回其末项。

例如在函数

```
void SemanticAnalysis::exitUnaryExp_call(CACTParser::UnaryExp_callContext *
ctx)
```

中的如下片段：

```
std::string func_name = ctx -> Ident() -> getText();
auto func_iter = func_symbol_table.find(func_name);

// Check undefined function
if(func_iter == func_symbol_table.end()){
    std::cout << "Undefined function name: " << func_name << std::endl;
    exit(ERROR);
}
```

· 检查函数的返回值是否合法

该功能大部分已经在先前类型匹配的时候实现，这里谈一谈没有声明返回函数时的处理方法：
没有声明返回类型（返回类型为`void`）的函数，只能作为一个语句（`Stmt`）被调用，即其不能被当作一个表达式（`Exp`）来使用（即赋值以及返回语句当中）

而在具体处理时，需要注意的是，该类函数可以作为一个`{Exp;}`形式的语句存在，所以剩下需要考虑的就是赋值语句、返回语句以及函数的调用语句：赋值和返回语句已经在类型匹配中实现，比如在函数

```
void
SemanticAnalysis::exitMulExp_mul_unary(CACTParser::MulExp_mul_unaryContext *
ctx)
```

中

```
std::string operand = ctx -> mulOp() -> getText();
if((operand == "*" || operand == "/") && (type_2 != int_type) && (type_2 !=
float_type) && (type_2 != double_type)){
    std::cout << "Operator * and / can only be used with int, float or
double." << std::endl;
    exit(ERROR);
}else if((operand == "\%") && (type_2 != int_type)){
    std::cout << "Operator \% can only be used with int." << std::endl;
    exit(ERROR);
}
```

如果函数返回值为空的话，在类型判断时已经报错

而对于函数的调用语句，会具体在下面的函数实参形参对比中实现。

· 检查函数实参形参是否匹配

先考虑函数没有实参的情况，如果形参列表也为空的话就匹配，否则报错；
再考虑参数数量的匹配，直接比较即可；
然后是参数类型的匹配，对实参进行遍历，形参进行迭代器遍历，一一比较类型以及数组，需要注意的是，如果实参是常量数组的话，也需要报错：

```
// Check function parameter
auto param_list = func_iter -> second.fparam_list;
```

```
 if (ctx -> funcRParams()){
     // Check if the number of parameters matches
     if (param_list.size() != ctx -> funcRParams() -> exp().size()){
         std::cout << "Number of parameters mismatched for function name: "
<< func_name << std::endl;
         exit(ERROR);
     }
     auto exp_list = ctx -> funcRParams() -> exp();
     size_t num_exps = exp_list.size();
     auto fparam_iter = param_list.begin();

     // Check if the types of parameters matches
     for(size_t i = 0; i < num_exps; i++, fparam_iter++){
         if (exp_list[i] -> exp_val_type != fparam_iter ->
second.fparam_type || exp_list[i] -> is_array != fparam_iter ->
second.is_array){
             std::cout << "Type of parameters mismatched for function name:
" << func_name << std::endl;
             exit(ERROR);
         }
         if (exp_list[i] -> is_const && exp_list[i] -> is_from_symtab){
             std::cout << "Const can't be used for parameter in function
name " << func_name << std::endl;
             exit(ERROR);
         }
     }
 }else{
     // the function has no parameter
     if(param_list.empty()){
         std::cout << "Number of parameters mismatched for function name: "
<< func_name << std::endl;
         exit(ERROR);
     }
 }
```

6. 控制流语句的使用

`continue`和`break`只能在循环体中使用，而`return`只能在函数体中使用，我们为 block 添加继承属性`allow_break`，`allow_continue`和`allow_return`，表示该 block 内是否允许出现这三种语句，并依次由 blockItem, stmt 继承，并为不同的 stmt 分支定义各自的语义动作：

```
stmt
    locals[
        bool allow_break = false,
        bool allow_continue = false,
        bool allow_return = false,

        int stmt_type,          // Seems unnecessary?
        int stmt_val_type,

        bool actual_has_return = false, //synthetic attributes
        int actual_return_type
    ]
    : lVal '=' exp ';'                      #stmt_assign
```

```
    | (exp)? ';'                              #stmt_exp
    | block                                   #stmt_block
    | 'if' '('cond')' stmt ('else' stmt)?     #stmt_if
    | 'while' '('cond')' stmt                 #stmt_while
    | 'break' ';'                             #stmt_break
    | 'continue' ';'                          #stmt_continue
    | 'return' (exp)? ';'                     #stmt_return
    ;
```

在`exit_Stmt_break/continue/return`中也继承上述 allow 属性，于是，如果在`exit_Stmt_break`中如果发现`allow_break`为假，即不允许使用`break`，则应当报错，其余两个语句类似。

7. main 函数有关的检查和 I/O 函数的使用

   按照 CACT 规范，我们在进入 compUnit 时将这组 I/O 库函数添加到函数的符号表中；

   多个 main 函数的错误和普通的函数重名是一样的；其余检查则在退出 compUnit 前完成，即检测符号表中是否存在 main 函数的表项、检查 main 函数的形参和返回类型，其实现如下：

```
void SemanticAnalysis::checkMainFunc()
{
    // Check if there is MAIN function
    auto main_sym_item_iter = func_symbol_table.find(MAIN_FUNC_NAME);
    if(main_sym_item_iter == func_symbol_table.end()){
        std::cout << "Missing main function." << std::endl;
        exit(ERROR);
    }

    // Check type for main
    auto main_sym_item = main_sym_item_iter -> second; // first is key,
second is value
    val_type_t ret_type = main_sym_item.ret_type;
    if(ret_type != int_type){
        std::cout << "Wrong type for main, should be defined as int, but get
" << ret_type << std::endl;
        exit(ERROR);
    }

    // Check fparam list for main
    size_t num_args = main_sym_item.fparam_list.size();
    if(num_args != 0){
        std::cout << "No arguments is supported for main function, but get "
<< num_args << " argument(s)." << std::endl;
        exit(ERROR);
    }
}
```

其它

我们在设计符号表时，曾经想过用栈的方式来存储符号表，当进入一个作用域时压栈，出作用域时弹栈，这一方案在目前的应用场景下是可行的，因为我们离开一个作用域后便不会再使用一个变量，然而考虑到代码生成的需要，以及实际编译器中符号表是能够完整保留的，我们放弃了这种方案，而将作用域组织成树状结构。

总结

实验结果总结

这次实验中，我们由于对 C++ 语法不是很熟悉，遇到了不小的阻力，通过查阅资料等方式，我们通力合作，最终克服了困难，顺利完成实验。

分成员总结

1、代瀚堃

这次实验中，由于对 C++ 语法和面向对象思想不了解，又没能弄明白语义分析的整个流程，组员们有些畏难，后来逐渐上手尝试，组员间相互讨论，我们都掌握了编写 .g4 文件并调用访问语法树的接口完成语义分析的过程。另一方面，由于 AntLR 生成的接口使用起来十分方便，而 C++ 相比 C 语言在一定程度上也更为完善，所以我在调试阶段并没有遇到很大的困难，我在这次实验中主要把时间花在搭建语义分析的框架和完善组员编写的代码上。

2、孙彬

本次实验难度主要在于框架的搭建以及细节的处理，代码既杂又多，尤其是在类型匹配的处理上比较磨人。同时也较好地锻炼了团队协作能力，大家既有分工，在遇到困难时也会互相支持，在这里要感谢代瀚堃同学，他完成了实际上大部分的实验内容，实现了实验的本地操作，同时也不厌其烦地帮助我们了解实验内容以及方法。

3、郑天羽

本次实验的难点主要在于要细致地考虑每一种可能出现的情况，首先需要对 CACT 规范中的文法规则非常熟悉，其次需要把非终结符属性和符号表数据结构的设计与检查代码的编写相互配合起来。在本次实验中，与同组成员的交流帮助我更好地理解了编译的过程。