编译原理研讨课实验PR002实验报告

编译原理研讨课实验PR002实验报告

任务说明

本次实验需要完成对CACT语言的语义分析部分。主要涉及的实验步骤包括

1. 设计符号表，完成符号表相应的操作
2. 改写g4文件
3. 编写部分语义动作，在语法树的遍历中实现对语义的分析，当出现语义错误的时候，实现编译过程中的报错
4. 修改脚本，通过测试文件

成员组成

王畅路 2019K8009907018
杜政坤 2019K8009909005
王意晨 2019K8009929016

实验设计

设计思路

本次实验需要设计符号表的数据结构和语义动作，符号表用来存储语义分析中的各个作用域的终结符，语义动作用来完善语法树遍历中对程序语义的分析。

实验实现

符号表的设计

符号表应该包括的信息

程序的构成是由不同的函数和声明构成的，因此作用域可以分成全局作用域和函数作用域，全局变量（或全局常量）的声明在全局作用域中进行。在函数作用域中不同的块（block）会在函数中创建新的子作用域。

可以在符号表类（SymbolTable）中使用全局变量表(global_symbols)和函数表(func_symbols)记录全局作用域中的函数和全局变量，用函数表记录函数作用域中的块信息和变量信息。

由于符号表相关函数的实现方式，需要使用cur_func记录当前正在处理的函数，使用block_s数组记录当前处理函数各个层次块的符号表。

temp_var表示程序产生的临时值，比如返回值的值。

```
class SymbolTable{
public:
    std::map<std::string, Var> global_symbols;  // symbols for global variable
    std::map<std::string, Func> func_symbols;   // symbols for functions
    std::vector<BlockTable*> block_s;           // stack for blocks
    std::string cur_func;                       // current function
    int temp_var;
public:
    //成员函数
};

```

对于每个变量，每个函数，每个块应该记录的信息如下：

```
struct Var{
    int cls;                // 变量的基本类型
    int type;               // 是否为const或array
```

```
    int length;             // 对于数组变量表示长度，对于函数参数表示第几个。
    int line;               // 行号
    int global = 0;;        // 全局变量标志
    std::string name;       // 变量的名字
};
struct BlockTable{
    int line;                                       // 行号
    std::map<int, struct BlockTable*> sub_blocks;   // 子块
    std::map<std::string, Var> local_symbols;       // 函数块的符号信息
};
struct Func{
    int cls;                // 返回值类型
    int param_num;          // 参数个数
    int line;               // 行号
    struct BlockTable func_block;//记录函数符号信息和子块
    std::string name;       //函数名
};
```

符号表应支持的操作

```
class SymbolTable{
public:
    ...//类成员
public:
    SymbolTable(){
        temp_var = 0;
        cur_func = "$";
        block_s.push_back(nullptr);
    }
    Var* lookup_var(const std::string & name);          // look up variable
    Func* lookup_func(const std::string & func_name);   // look up funtion
    Var* lookup_param(Func& func, int para_index);      // look up parameter in function declaration

```

```
14      void addSymbol(std::string name, int cls, int type, int
    length, int line);
15      void addFunc(std::string name, int return_class, int
    param_num, int line);
16      void addBlock(int line);
17
18      std::string gen_temp_var(int lc, int cls, int type =
    TP_VAR);
19      std::string gen_temp_array(int lc, int cls, int size);
20  };
```

- `SymbolTable` 作为构造函数，初始化 `tempvar` 为0；使用 `$` 符号表示当前处于全局作用域，不在函数内部。使用默认析构函数即可。
- `lookup_var` 函数通过变量名在符号表中查找变量信息。查找过程如下：首先在 `block_s` 中查找变量，如果没有找到继续向上一层栈查找，如果没有查找到，查找全局变量符号表。找到的话返回，如果没有找到返回空指针。
- `lookup_func` 函数通过函数名在 `func_symbols` 中找到相应的表项，没有找到返回空指针。
- `lookup_param` 函数通过函数名和参数索引在该函数的本地符号表中找到参数信息。
- `addSymbol` 函数将变量信息加入符号表。如果当前没有处理函数(即 `cur_func = $`)那么将该表项加入全局变量符号表，否则将该表项加入此函数的符号表的相应子块（即当前 `block_s` 顶端指向的块）

> addSymbol函数同时检查加入的新变量是否在当前符号表有同名情况，如果有，抛出 multiply declaration 异常

- `addFunc` 函数将函数信息加入符号表 `func_symbols`

> addFunc函数同时检查加入的新函数是否有同名情况，如果有，抛出 multiply declaration 异常

- `addBlock` 主要用于在函数中添加子块。当要为当前函数添加子块的时候，需要在 `Func->func_block->sub_blocks` 添加新的块，同时在 `block_s` 中加入指向该块的指针。
- `gen_temp_var` 会产生一个临时变量。该临时变量一般为返回值或者表达式的结果。
- `gen_temp_array` 与 `gen_temp_array` 函数类似

.g4文件修改

为了对不同的规则附加不同的语义动作，本次实验我们给部分产生式添加了标签，这样就可以使用标签来进行语义分析。

```
varDef
    : Ident
        # varDefVal
    | Ident '[' IntConst ']'
       # varDefArray
    | Ident '=' constExp
       # varDefInitVal
    | Ident '[' IntConst ']' '=' '{' (constExp (',' constExp)*)? '}'    # varDefInitArray
    ;
```

同时，我们为部分规则加上了综合属性，这样可以使得这些属性传递更加方便，不需要额外得数据结构保存这些信息

```
lVal
    locals[
        std::string name,
        int cls,
        bool elemwise
    ]
    :Ident ('[' exp ']')?
    ;
```

语义分析

当在g4文件中编写完规则之后，antlr会在baselistener文件中自动生成代表每条规则和每个标签的语义动作的虚函数，当遍历语法分析树的时候，每次经过或者退出一个节点，都会调用相应的enter和exit函数，我们要做的就是编写这些语义动作，使得程序的语义正确。

接下来将会介绍我们在每个节点中都实现了怎样的语义动作。

compUnit编译单元

- 进入compUnit的时候，我们将会加载一些库函数，将这些库函数加到符号表中。这些函数为实验所需要的 `print_int` 等函数

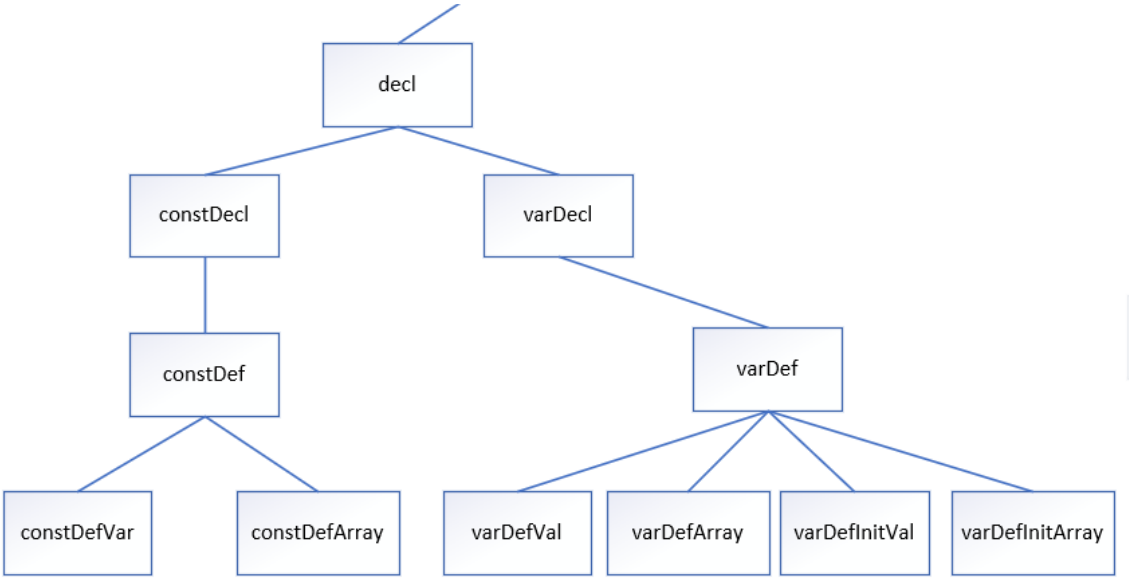
```
    sym_table.addFunc("print_int", CLS_VOID, 1, 0);
    sym_table.addSymbol("_int_", CLS_INT, TP_PARAM, 0, 0);
    sym_table.block_s.pop_back();//因为addfunc会有把函数块入栈的操作，因此需要显示把该块弹出
    sym_table.cur_func = "$";
```

- 退出时我们需要检查程序里有没有 `main` 函数，如果没有的话将会抛出异常。

根据产生式 `compUnit->(decl | funcDef)+ EOF` 进行后续分析

decl声明

声明部分的简化语法树大致如下



进入退出 `decl` 的语义动作分成两中情况，分别在 `constdecl` 和 `vardecl` 中定义

constDecl 和 varDecl

- 在进入constDecl和varDecl的时候，我们需要获得bType的类型，并保存在栈中，接下来在分析类型是否匹配的时候就可以用到。
- 退出的时候，我们把保存的类型弹出

constDef

- 进入时
  - 对比在栈中记录的类型和 `=` 号右边的类型是否匹配，如果不匹配将抛出 `invalid init value` 异常。
  - 判断该类型是否是未知类型（不是预定义的四类类型），如果是，抛出 `unknown type name` 异常。
  - 将该 `Ident` 加入到符号表中。
- 退出时无操作

constDefArray

- 进入时
  - 获取 `IntConst` 的值，如果该值是负数，将会抛出 `array negative` 异常。
  - 判断栈顶类型是否是未知类型，如果是的话那么抛出 `unknown type name` 异常。
  - 遍历每个 `constExp` 如果出现类型不匹配的情况，就抛出 `invalid init value` 异常。
  - 在遍历过程中同时记录 `constExp` 的数目，如果数量大于 `IntConst` （即数组元素个数大于数组长度），那么抛出 `excess elements in array init value` 异常。
  - 加入符号表。
- 退出时无操作

varDef

和constDef类似，在进入时需要获得btype的类型保存在栈里用来进行类型匹配，之后退出时弹出栈。

varDefVal

该标签代表进行类似 `int a;` 的定义，不初始化。

- 进入时
  - 检查栈顶类型是否是未知类型，如果是抛出异常
  - 加入符号表。
- 退出时无操作

VarDefArray

该标签代表进行类似 `int a[5]` 的定义，不初始化。

- 进入时
  - 检查是否是未知类型
  - 检查数组索引是否是正常值
  - 加入符号表
- 退出时无操作

VarDefInitVal

该标签代表进行类似 `int a = 5;` 的定义

- 进入时
  - 检查类型是否匹配
  - 检查类型是否是未知类型
  - 加入符号表
- 退出时无操作

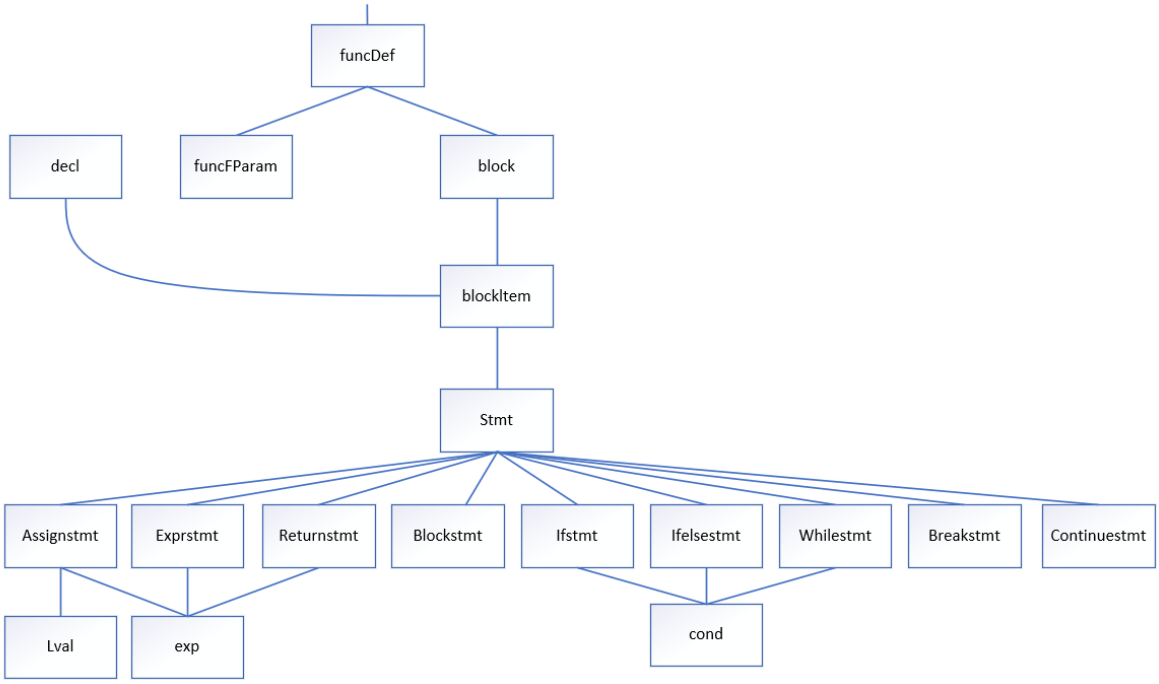
VarDefInitArray

该标签代表进行类似 `int a[5] = {1,2,3};` 的定义

- 进入时
  - 检查数组索引是否正常
  - 检查是否是未知类型
  - 检查每个元素是否匹配
  - 检查元素个数是否超过数组长度
  - 加入符号表
- 退出时无操作

funcDecl

函数定义部分简化语法树如下



- 进入时
  - 检查返回值的类型，如果是未知类型抛出异常
  - 如果该函数名为 `main` ，那么
    - 检查参数列表，如果不是0个参数，抛出 `too many arguments for function 'main'` 异常
    - 检查返回值类型，如果不是int类型，抛出 `function 'main' returns error` 异常
  - 将函数加入符号表，同时会在栈中创建指向该函数符号表的指针
  - 更新 `cur_func` 为该函数
  - 检查每个参数（ `funcFparam` ）的类型，如果是未知类型抛出异常
  - 将每个参数加入符号表
- 退出时
  - 将该函数的符号表从栈中弹出
  - 将 `cur_func` 更新为 `$`

block

- 进入时为当前函数的符号表创建一个subblock
- 退出时弹出该subblock

stmt

当我们要对stmt进行分析的时候，在进入的时候，我们并不能执行很多操作，因为这个时候任何词法单元的属性都是未知的，如Lval和Exp的类型和名称，都要分别进入相应的子树分析之后才能获得，因此只有当退出该节点的时候我们才能对得到的结果进行处理。这里注意，我们对Lval，Exp等产生式添加了综合属性，因此在从stmt节点退出的时候，可以获得这些综合属性的值。

```
        std::string name,
        int cls,
```

我们将stmt划分成了9个子标签，分别处理不同的语句。以下只涉及退出节点的操作。

assignstmt

- 检查当左值为数组的时候

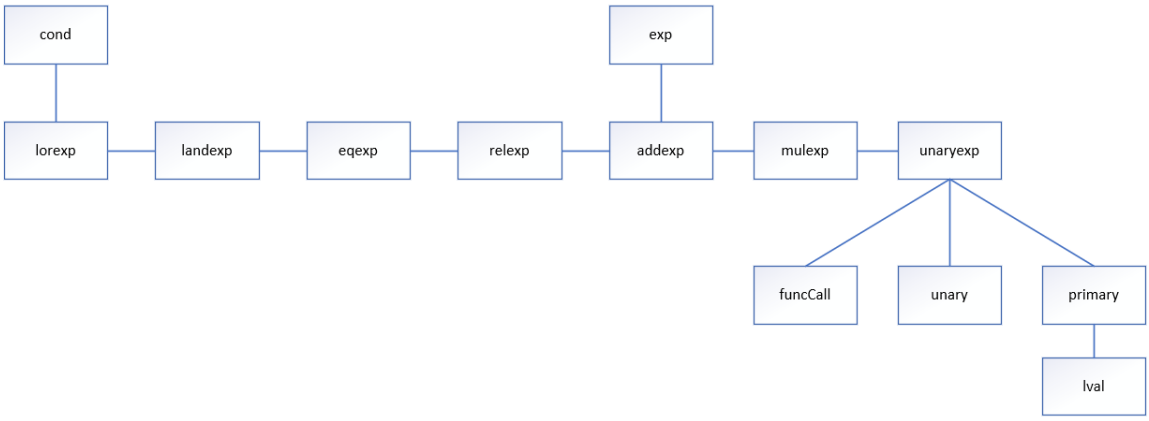
  ```
  if (ctx->lVal()->exp() != nullptr)
  ```

  检查lVal节点数组索引是否正常
- 查找符号表，检查左值是否已经声明，没有的话抛出 `undeclared` 异常
- 查找符号表，检查表达式的temp值是否已经声明，没有的话抛出 `undeclared` 异常
- 由于CACT不支持类型转换，因此需要检查等号左右两边类型是否匹配，不匹配的话抛出 `incompatible` 异常
- 检查左值是否为const类型，如果是的话抛出异常 `assignment of constant variable`
- 检查左值是否为void类型，如果是的话抛出异常 `void value not ignored as it ought to be` 异常

returnstmt

- 如果return没有返回值，查找符号表找到该函数，如果该函数的返回值不是void，抛出 `incompatible` 异常
- 如果return有返回值（`ctx->exp() != nullptr`），获取该返回值表达式的类型，和函数的返回值对比，如果不同，抛出 `incompatible` 异常

exp



exp部分涉及的简化语法树如图所示，其中涉及的运算表达式的产生方式依据优先级。

退出节点exp时，填入元素的综合属性 `cls` 和 `name` 。节点exp的属性由子节点得到。

lval

任何左值都必须是声明过可修改的变量

- lval不是数组时 (ctx->exp() == nullptr)
  - 检查该左值是否已经声明，没有则抛出异常
  - 填入综合属性
- lval是数组时
  - 检查数组索引是否正常
  - 该数组是否已经声明
  - 该数组在符号表中的表项记录是否为 `const` 数组
  - 填入综合属性，其中由于name属性没有可用定义（如a[5]在符号表中并无定义，也不是终结符），因此我们生成一个临时name代表该左值加入符号表，代码如下。

```
1         std::string arrayelem =
sym_table.gen_temp_var(ctx->getStart()->getLine(), ctx-
>cls, sym_table, array_info->type);
2         ctx->name = arrayelem;
```

cond

- 获得lorexp综合属性
- 检查cond的类型是否为bool，如果不是，抛出异常

primary

- primary是exp时，直接填入其综合属性
- primary是lVal时，直接填入其综合属性
- primary是number时，根据number的类型，在 `cls` 填入number的类型，在 `name` 填入产生的临时name。

funcCall

- 进入时
  - 根据func_name在符号表中查找表项，没找到抛出异常 `undefined function`
  - 根据funcRParams->size()获得参数个数，和得到的表项中的记录对比，如果较少，抛出异常 `too few arguments to function` ，如果多出，抛出异常 `too many arguments to function`
- 退出时
  - 检查funcRParams的每个参数类型和记录是否相同，不同则抛出异常
  - 为返回值创建临时name,cls填入返回值的类型

unary

- unaryop是 `!` ，检查unaryexp的类型是否为 `bool` ，如果不是，抛出 `wrong type argument to unary exclamation mark` 异常
- unaryop是 `+` 或 `-` ,检查unaryexp的类型，如果是 `bool` ，抛出 `wrong type argument to unary exclamation mark` 异常
- 填入unaryexp的综合属性

mulexp

- 如果mulexp是unaryexp，且其cls为void（这是可能的，因为unaryexp可以产生funcCall），此时综合属性 `name` 填入NULL。
- 如果mulexp是乘法运算
  - 检查两端exp类型是否相同，如果不相同，抛出异常
    - 如果两端都是数组类型，检查两者长度是否相同，如果不相同，抛出 `wrong size to element wise operation` 异常

        - 如果两端都是bool类型，抛出异常 `invalid operands in mulExp`
    - 为运算结果创建临时name,cls填入两端表达式的类型

addexp

- 如果addexp是mulexp，填入其综合属性
- 如果addexp是加法运算
    - 检查两端exp类型是否相同，如果不相同，抛出异常
        - 如果两端都是数组类型，检查两者长度是否相同，如果不相同，抛出 `wrong size to element wise operation` 异常
        - 如果两端都是bool类型，抛出异常 `invalid operands in mulExp`
    - 为运算结果创建临时name,cls填入两端表达式的类型

relexp

- 如果是boolconst,为其创建临时name，填入其综合属性
- 如果是关系运算
    - 检查两端类型是否相同，如果不同，抛出异常
    - 为运算结果创建临时name,cls填入两端表达式的类型

eqexp

- 如果是relexp,填入其综合属性
- 如果是等价运算
    - 检查两端类型是否相同，如果不同，抛出异常
    - 为运算结果创建临时name,cls填入两端表达式的类型

landexp

- 如果是eqexp,填入其综合属性
- 如果是与运算
    - 检查两端类型是否相同，如果不同，抛出异常
    - 为运算结果创建临时name,cls填入两端表达式的类型

lorexp

- 如果是landexp,填入其综合属性
- 如果是或运算
  - 检查两端类型是否相同，如果不同，抛出异常
  - 为运算结果创建临时name,cls填入两端表达式的类型

测试

代码成功跑过48个测试用例

其它

- 代码中还有很多冗余代码，可能在部分地方做了重复的检错。同时可以简化代码，比如把未查找到表项的异常放到lookup_symbol中可以节省代码量
- 代码中部分地方可以使用综合属性或者继承属性代替，比如类型栈 `tp_stack`
- 代码部分ifelse逻辑有冗余

总结

实验结果总结

本次实验主要实现了符号表以及语义分析的设计。符号表在分析阶段起着至关重要的作用，通过本次实验加深了对符号表的构造机制与管理方法的理解。语义分析是编译过程的重要阶段，对语义分析的实现过程加深了我们对编译过程的理解。

分成员总结

王畅路

通过本次实验，我明白了antlr对g4文件的处理方式，也明白了如何通过调用antlr的接口实现对语义动作的编写。通过设计符号表，我对CACT语言的结构有了更清晰的认识，设计符号表的时候，应该注意如何设计好全局变量，函数列表，函数块的结构，以及每个表项应该保存什么信息。通过使用综合属性，我对综合属性在语法树中的传递有了更清晰的认识。设计语义动作的细节太多，并且繁琐，需要考虑周全。本次实验加深了我对“综合属性”、“继承属性”以及类型检查等内容的理解。

杜政坤

在这次的实验中，我对符号表的设计和C++提供的一些工具有了更深的理解，关于符号表的设计，感受到一个合适的数据结构对于完成工作很重要，比如这次的func中包含一个block，block中包含一个储存本地符号的local_symbols的map等结构，结构的合理性会给任务带来很大帮助。关于C++提供的工具，主要是感受到方便，比如直接用std::map即可存储string和struct的映射，再用map.insert函数即可以插入，又比如vector本身有一些很好的性质很方便使用。除此之外，我对语法制导翻译的理解也更深，层层递进的enter和exit将整个符号表的生成和类型检查的过程维护得很好，对综合属性、继承属性的使用让我对理论知识的理解也得到了加深。

王意晨

通过本次实验，我对符号表的结构有了更加清晰的认识，相比于理论课上学到的符号表的结构，我们还需要注意更多的细节以及思考怎么构造出一个符号表（比如说在符号表中要设置那些元素，其中要包括全局变量表、函数表、目前函数等元素），其中的重点是为函数、变量等选择适合的数据结构，以及维护不同block之间的符号表嵌套关系。除此之外，我还对语义分析有了更深的理解，本次实验设计语义动作的情况很多，一般来说就是声明类的在进入的时候做一定的处理，运算类的在进入的时候先传递继承属性，然后在结束的时候进行具体的处理。本次实验让我了解了符号表的构建以及语法分析的流程，从而让我对语法制导翻译有了更深一步的理解。