CACT-PR002 语义分析

任务说明

本次实验需要对CACT源程序进行语义分析和类型检查：当测试程序符合语义规范时，编译器程序的返回值应为0，否则应返回非0值。

本次实验设计的重点在于，对于符号表和类型表示的设计、语义分析。需要我们思考并解决的问题包括但不限于：

1. 符号表采用何种数据结构？
2. 符号表中存储什么信息？
3. 如何新建符号表？
4. 如何查询符号表？
5. 符合语义规范的类型有哪些？
6. 如何判断两个表达式的类型是否等价？
7. 如何从变量的类型推断表达式的类型？

成员组成

第7组成员：

陈飞羽 2018K8009929031

彭思叡 2018K8009908040

袁欣怡 2018K8009929021

实验设计

设计思路

1. 符号表设计

`class SymbolTable`为本次实验定义的符号表类。该类主要的作用为：记录标识符信息、处理作用域范围。为实现这些功能，该类维护的主要变量如下：

- 全局符号表`global_table`：记录当前所有声明过的全局变量
- 函数记录表`func_table`：记录当前所有声明过的函数
- 符号表栈`block_stack`：栈顶为当前正在分析的作用域的符号表，从栈顶到栈底，第n层为第n-1层的父作用域。
- 当前函数名`cur_func_name`：记录当前正在分析的函数名字。

同时维持全局符号表和函数记录表是因为在程序的任意位置，都至少有两个有效的作用域：全局作用域和函数作用域，而且变量/常量允许与函数同名，这样的设计方便了标识符查询。而符号表栈和当前函数名的作用在于确定标识符的优先级。

提供的主要方法如下：

- 查找符号：从当前作用域到它的父作用域，逐级查找标识符，保证了局部作用域中定义的标识符可以隐藏掉该局部作用域外的同名标识符。
- 查找函数：查找指定名字的函数。
- 添加标识符：向当前作用域的符号表中添加一个标识符。
- 添加函数：在函数记录表中记录函数信息。
- 创建符号表：创建当前作用域的符号表。

2. 语义分析设计

   符号表的创建、维护、更新及类型检查等动作都在语义分析的过程中完成。

   本实验中，我们实现的语义分析过程借助了ANTLR4 提供的`listener`机制提供的`enter()`,`exit()`方法。同时我们还添加了一些结点的继承属性、综合属性进行传递。

实验实现

符号表设计

我们为符号表设计的数据结构为`class SymbolTable`，涵盖了设计思路中提到的内容，具体代码实现为（`src /SymbolTable.h`）：

```
class SymbolTable{
public:
    std::map<std::string,VarInfo> global_table;
    std::map<std::string,FuncInfo> func_table;
    std::vector<BlockTable *> block_record;
    std::vector<BlockTable *> block_stack;
    std::string cur_func_name = "nullptr";
public:
    SymbolTable(){
        block_record.push_back(nullptr);
        block_stack.push_back(nullptr);
    }
    VarInfo *lookup_symbol(const std::string &name);
    FuncInfo *lookup_func(const std::string &func_name);
    void add_symbol(std::string name, int cls, int type, int value, int def_line);
    void add_func(std::string name, int cls, int params, int def_line);
    void add_block(int line);
    void pop_block();
    int StringToInt(std::string rt_type);
};
```

CACT支持四种基本类型：int、float、double、bool，以及函数返回值可以为空，所以我们定义了五种基本类型（`src /SymbolTable.h`）：

```
enum{
    CLS_INT,
    CLS_DOUBLE,
    CLS_FLOAT,
    CLS_BOOL,
    CLS_VOID
  };
```

CACT支持四种基本类型及它们数组的变量、常量声明，所以我们我们定义了六种基本种类（`src /SymbolTable.h`）：

```
 enum{
    TYPE_CONST,
    TYPE_VAR,
    TYPE_ARRAY,
    TYPE_CONST_ARRAY
  };
```

我们还定义了三个结构体`struct VarInfo`,`struct FuncInfo`,`struct BlockTable`分别用于记录变量信息、函数信息和作用域的符号表（`src /SymbolTable.h`）：

```
struct VarInfo {
    std::string name;
    int cls;        // class: int double float bool void
    int type;       // type: constvar var constarray array
    int def_line;   // def position
    unsigned long long value;
    int global = 0;
    int offset = -1;
  };

  struct BlockTable{
    int line; // start position
    std::map<std::string,VarInfo> symbol_table;
  };

  struct FuncInfo{
    std::string name;
    int cls; //return type
    int params; //number of paramters
    int def_line; //def position
    BlockTable base_block;
  };
```

符号表类的提供的方法如下（`src /SymbolTable.cpp`）：

- `VarInfo *lookup_symbol(const std::string &name)`：在符号表栈`block_record`中从栈顶到栈底寻找名字为`name`的变量。

```
VarInfo *SymbolTable::lookup_symbol(const std::string &name){
    int count = this->block_stack.size();
    int i;
    BlockTable * block;
    for(i=count;i>1;i--){
      //从栈顶到栈底寻找标识符
        block = block_stack[i-1];
        if(block->symbol_table.count(name)>0)
            return &block->symbol_table[name];
    }
  //如果所有局部作用域中均无该标识符，则查找全局作用域。
    if (this->global_table.count(name) == 1)
        return &this->global_table[name];
      else
          return nullptr;
```

```
}
```

- `FuncInfo *lookup_func(const std::string &func_name)`：在函数记录表`func_table`中查找名字为`func_name`的函数。

```
FuncInfo *SymbolTable::lookup_func(const std::string &func_name)
{
    if (this->func_table.count(func_name) == 1) // exists
        return &this->func_table[func_name];
    else
        return nullptr;
}
```

- `void add_symbol(std::string name, int cls, int type, int value, int def_line)`：向当前作用域的符号表添加一个变量的信息。由于在我们的设计中，函数的形参和局部变量不在同一个作用域中，所以我们需要一些特殊处理。

```
void SymbolTable::add_symbol(std::string name, int cls, int type, int value,
int def_line){
    int count = this->block_stack.size();
    VarInfo tmp;
    tmp.name = name;
    tmp.cls = cls;
    tmp.type = type;
    tmp.value = value;
    tmp.def_line = def_line;
    tmp.global = 0;

    if(this->cur_func_name == "nullptr"){ // 如果当前不在函数作用域内，则把符号
加到全局符号表
        if(this->global_table.count(name) == 0){
            tmp.global = 1;
            this->global_table.insert(std::pair<std::string, VarInfo>(name,
tmp));
        }
    else
            throw std::runtime_error("\nMultiple definition!\n");
    }
    else{ // 如果在某个函数的作用域中
        BlockTable *top = this->block_stack[count - 1];
        FuncInfo *func = lookup_func(this->cur_func_name);

        if(top->symbol_table.count(name) == 0 && func-
>base_block.symbol_table.count(name) == 0) // 检查是否已有局部变量或形参的定义
            top->symbol_table.insert(std::pair<std::string, VarInfo>(name,
tmp));
```

```
        else
            throw std::runtime_error("\nMultiple definition!\n");
    }
}
```

- `void add_func(std::string name, int cls, int params, int def_line)`：向函数记录表`func_table`添加一个函数的信息。

```
void SymbolTable::add_func(std::string name, int cls, int params, int def_line)
{
    FuncInfo tmp;
    tmp.name = name;
    tmp.cls = cls;
    tmp.params = params;
    tmp.def_line = def_line;

    if(this->func_table.count(name) == 0){
        this->func_table.insert(std::pair<std::string,FuncInfo>(name,tmp));
        block_stack.push_back(&func_table[name].base_block);
        block_record.push_back(&func_table[name].base_block);
    }
    else
        throw std::runtime_error("\nMultiple definition!\n");
}
```

- `void add_block(int line)`：在符号表栈`block_stack`栈顶添加一个符号表。

```
void SymbolTable::add_block(int line){
    BlockTable *block = new BlockTable;

    block->line = line;
    block_record.push_back(block);
    block_stack.push_back(block);
}
```

- `void pop_block()`：将符号表栈栈顶的符号表弹出。

```
void SymbolTable::pop_block(){
    block_stack.pop_back();
}
```

- `int StringToInt(std::string rt_type)`：将string类型的rt_type转化成int类型。

```
int SymbolTable::StringToInt(std::string rt_type){
    int rt_type_int;
```

```
    if (rt_type == "int")
        rt_type_int = CLS_INT;
    else if (rt_type == "float")
        rt_type_int = CLS_FLOAT;
    else if (rt_type == "double")
        rt_type_int = CLS_DOUBLE;
    else if (rt_type == "bool")
        rt_type_int = CLS_BOOL;
    else if (rt_type == "void")
        rt_type_int = CLS_VOID;
    else{
        rt_type_int = -1;
        std::cout<< rt_type << std::endl;
    }

    return rt_type_int;
}
```

语义分析实现

我们为语义分析器设计的类为`class SemanticAnalysis`，该类的主要功能为：维护一个符号表 Symbol Table，并且在分析树中添加语义动作，判断程序是否符合语义规范。添加的语义动作包括：`enterConstDecl`，`exitConstDecl`，`enterFuncDef`，`exitFuncDef`，`enterUnaryExp`，`exitUnaryExp`等等。这些函数的命名均为“`enter/exit + 非终结符`”，其中“`enter + 非终结符`”表示遍历语法分析树时读到这个非终结符时进行的语义动作，“`exit + 非终结符`”表示遍历语法分析树时离开这个非终结符时进行的语义动作。

因为需要实现的函数很多而且总体思路相同，所以下面我们举几个函数为例说明我们设计的总体思路（`src /semanticAnalysis.cpp`）：

1. 开始分析时，调用函数`void SemanticAnalysis::enterCompUnit(CACTParser::CompUnitContext * ctx)`，将实现基本I/O功能的库函数添加到符号表中，具体代码如下：

```
void SemanticAnalysis::enterCompUnit(CACTParser::CompUnitContext * ctx) {
    symbol_table.add_func("print_int", CLS_VOID, 1, 0);
    symbol_table.add_func("print_bool", CLS_VOID, 1, 0);
    symbol_table.add_func("print_float", CLS_VOID, 1, 0);
    symbol_table.add_func("print_double", CLS_VOID, 1, 0);
    symbol_table.add_func("get_int", CLS_INT, 0, 0);
    symbol_table.add_func("get_float", CLS_FLOAT, 0, 0);
    symbol_table.add_func("get_double", CLS_DOUBLE, 0, 0);
}
```

2. 与之对应的是函数 `void SemanticAnalysis::exitCompUnit(CACTParser::CompUnitContext * ctx)`，在结束分析时判断 `.cact` 程序中是否有 `main` 函数，如果没有则报错。代码如下：

```
void SemanticAnalysis::exitCompUnit(CACTParser::CompUnitContext * ctx) {
    if(this->symbol_table.lookup_func("main") == nullptr){
      std::cout<<"line"<<std::to_string(ctx->getStart()->getLine())<<":"
<<std::endl;
      throw std::runtime_error("\nDid't define a main function!\n");
    }
}
```

声明处理

在遇到常量声明时，调用函数 `void SemanticAnalysis::exitConstDecl(CACTParser::ConstDeclContext * ctx)` 进行处理，检查类型是否匹配。需要注意的是，`cact` 语言中允许声明常量数组，在这里要注意对数组的处理。为了降低代码编写难度和提高可读性，使用 `auto` 来简化变量初始化（参考资料：c++ auto基本用法）。实现的代码如下：

```
void SemanticAnalysis::exitConstDecl(CACTParser::ConstDeclContext * ctx)
{
    std::string name;
    std::string cls_string;
    std::string value_string;
    int type = TYPE_CONST;
    int init_type = TYPE_CONST;
    int length = 0;
    unsigned long long value=0;
    //std::cout << "const variable define: " << std::endl;
    for(const auto & const_def : ctx->constDef())
    {
        if(const_def->Ident() == nullptr)
            return;
        name = const_def->Ident()->getText().c_str();
        cls_string = ctx->bType()->getText().c_str();
        //检查常量类型是否为常量数组
        if(const_def -> arraySymbol() != nullptr){
            type = TYPE_CONST_ARRAY;
            std::string length_str = const_def -> arraySymbol()->IntConst()-
>getText();
            length = std::stoi(length_str, 0, getIntStringBase(length_str));
        }
        //检查初始化类型是否为常量数组
        if(const_def -> constInitVal()->constInitArray()!=nullptr)
```

```
            init_type = TYPE_CONST_ARRAY;
        else if(const_def -> constInitVal()->constExp() != nullptr)
            value_string = const_def ->constInitVal()-> constExp()->getText();

        if(type != init_type){
            std::cout<<"line"<<std::to_string(ctx->getStart()->getLine())<<":"
<<std::endl;
            throw std::runtime_error("\nUnmatched initial value!\n");
        }
        int cls = symbol_table.StringToInt(cls_string);
        int def_line = ctx->getStart()->getLine();
        if(type == TYPE_CONST_ARRAY)
            value = length;
        else if(cls == CLS_INT)
            value = std::stoi(value_string, 0, getIntStringBase(value_string));
        else if(cls == CLS_BOOL)
            value = (value_string == "true") ? 1:0;
        else if(cls == CLS_DOUBLE)
            value = std::stod(value_string);
        else
            value =  std::stof(value_string);
        symbol_table.add_symbol(name,cls,type,value,def_line);
    }

}
```

类似地，在遇到变量声明时也需要检查类型是否匹配。如果变量类型和初始化类型不匹配，则需要报错。具体代码如下：

```
void SemanticAnalysis::exitVarDecl(CACTParser::VarDeclContext * ctx)
{
    std::string name;
    std::string cls_string;
    std::string init_cls_string;
    std::string value_string;
    int type = TYPE_VAR;
    int init_type = TYPE_VAR;
    int length=0;
    int inited=0;
    unsigned long long value=0;
    //对于同一条语句中声明的多个变量进行以下操作
    for(const auto & var_def : ctx->varDef())
    {
        if(var_def->Ident()==NULL)
            return;
```

```
        name = var_def->Ident()->getText().c_str();
        cls_string = ctx->bType()->getText().c_str();
        //检查变量是否为数组
        if(var_def -> arraySymbol() != nullptr){
            type = TYPE_ARRAY;
            std::string length_str = var_def -> arraySymbol()->IntConst()-
>getText();
            length = std::stoi(length_str, 0, getIntStringBase(length_str));
        }
            //如果有初始化
        if(var_def -> constInitVal()){
            inited = 1;
            //检查初始化类型是否为数组
            if(var_def -> constInitVal()->constInitArray()!=nullptr)
                init_type = TYPE_ARRAY;
            //记录初始化类型
            else if(var_def -> constInitVal()->constExp() != nullptr){
                value_string = var_def ->constInitVal()-> constExp()->getText();
                if(var_def -> constInitVal()->constExp()->BoolConst()!=nullptr)
                    init_cls_string = "bool";
                else if(var_def -> constInitVal()->constExp()->number()-
>IntConst()!=nullptr)
                    init_cls_string = "int";
                else if(var_def -> constInitVal()->constExp()->number()-
>DoubleConst()!=nullptr)
                    init_cls_string = "double";
                else if(var_def -> constInitVal()->constExp()->number()-
>FloatConst()!=nullptr)
                    init_cls_string = "float";
            }
            //如果变量类型与初始化类型不匹配
            if(cls_string != init_cls_string && var_def -> constInitVal()-
>constExp() != nullptr){
                std::cout<<"line"<<std::to_string(ctx->getStart()->getLine())<<":"
<<std::endl;
                throw std::runtime_error("\nUnmatched initial value!\n");
            }
            if(type != init_type){
                std::cout<<"line"<<std::to_string(ctx->getStart()->getLine())<<":"
<<std::endl;
                throw std::runtime_error("\nUnmatched initial value!\n");
            }
        }
        int cls = symbol_table.StringToInt(cls_string);
        int def_line = ctx->getStart()->getLine();
```

```
        //std::cout<<value_string<<"\n"<<std::endl;
        if(type != TYPE_ARRAY && inited){
            if(cls == CLS_INT)
                value = std::stoi(value_string, 0, getIntStringBase(value_string));
            else if(cls == CLS_BOOL)
                value = (value_string == "true") ? 1:0;
            else if(cls == CLS_DOUBLE)
                value = std::stod(value_string);
            else
                value =  std::stof(value_string);
        }else
            value = length;
        symbol_table.add_symbol(name,cls,type,value,def_line);
    }
}
```

作用域的语义分析

在程序的任意位置，都至少有两个有效的作用域：全局作用域和函数作用域。全局作用域包括顶层变量/常量、定义的函数。函数作用域包括全局变量/常量和局部变量/常量，以及函数定义中的形参。其余的局部作用域由代码中跟在if、while之后的Block（语句块）创建。函数作用域中定义的标识符可以隐藏掉全局作用域中的同名标识符。同理，局部作用域中定义的标识符可以隐藏掉该局部作用域外的同名标识符。局部作用域中的变量的生存期在该函数或语句块内。

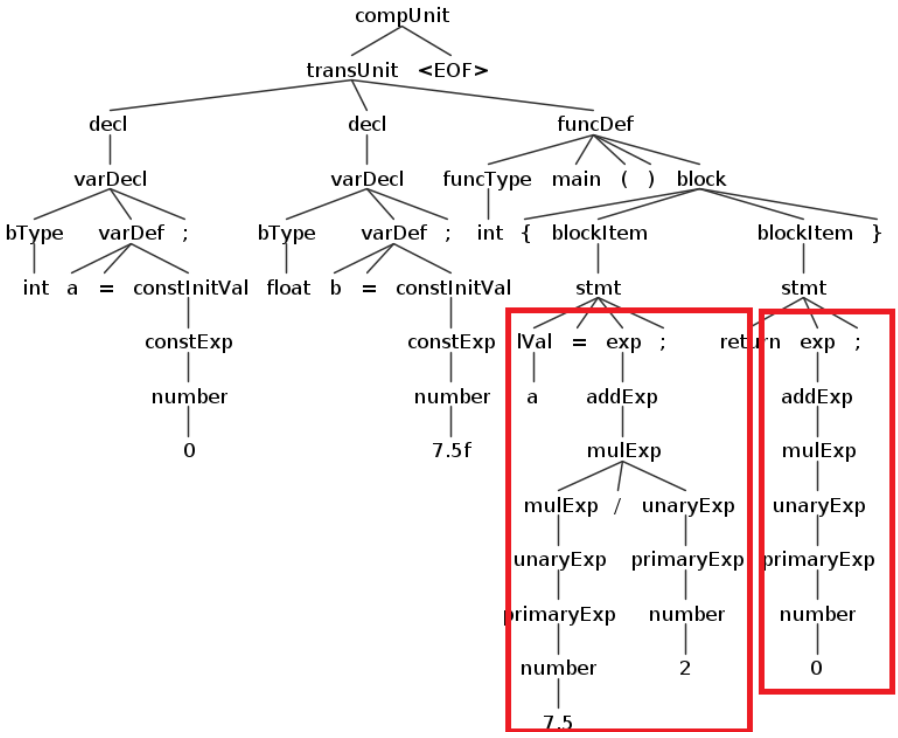
实现同名标识符的隐藏作用主要依赖以下两个方法：

```
void SemanticAnalysis::enterBlock(CACTParser::BlockContext * ctx){
    int line = ctx->getStart()->getLine();
    symbol_table.add_block(line);//添加当前Block的符号表并压入block_stack
}
void SemanticAnalysis::exitBlock(CACTParser::BlockContext * ctx){
    symbol_table.pop_block();//将当前block的符号表弹出block_stack
}
```

而在上文提到的实现变量/常量搜索功能的方法`lookup_symbol()`从`block_stack`的栈顶向栈底搜索，保证了搜到的变量/常量一定是同名标识符中优先级最高的。

表达式的语义分析：

下图右下角的两个红框为两处非终结符`exp`推导得到表达式的过程。从图中可以看出，如果利用CACTparser提供的`context`类中的方法来确定一个表达式的值及类型，是非常繁琐。以右框为例，enterExp()中确定`exp`的值需要起码6级间接索引，而且`addExp`，`mulExp`，`unaryExp`等非终结符均有多条推导规则，这会使得判断逻辑非常复杂。为了解决这些问题，我们在语法规则文件`CACT.g4`中引入了一些SDD来方便表达式类型及值的确定。



下面给出表达式的一些属性规则文法作为示例：

下表中`cls`属性为枚举类型，表示四种基本类型；`val`属性为8字节无符号量，用于存储值，`elementwise`属性为布尔值，用于表示该表达式是否为数组的向量操作。

PRODUCTION	SEMANTIC RULES
$exp \to addExp$	$exp.val = addExp.val, exp.cls = addExp.cls, exp.elementwise = addExp.elementwise$
$addExp \to mulExp$	$addExp.val = mulExp.val, addExp.cls = mulExp.cls, addExp.elementwise = mulExp.elementwise$
$addExp \to addExp_1 + mulExp$	$if(addExp_1.cls! = mulExp.cls \|\| addEXP_1.elementwise! = mulExp.elementwise)$ $throw\ runtime\_error(semantic\ error);$, $addExp.cls = addExp_1.cls, addExp.elementwise = addExp_1.elementwise, addExp.val = addExp_1.val + mulExp.val$
$mulExp \to unaryExp$	$mulExp.val = unaryExp.val, mulExp.cls = unaryExp.cls mulExp.elementwise = unaryExp.elementwise$
$mulExp \to mulExp_1 / unaryExp$	$if(mulExp_1.cls! = unaryExp.cls \|\| mulEXP_1.elementwise! = unaryExp.elementwise)$ $throw\ runtime\_error(semantic\ error);$, $mulExp.cls = mulExp_1.cls, mulExp.elementwise = mulExp_1.elementwise, mulExp.val = mulExp_1.val / unaryExp.val$
$unaryExp \to primaryExp$	$unaryExp.val = primaryExp.val, unaryExp.cls = primaryExp.cls, unaryExp.elementwise = primaryExp.elementwise$
$primaryExp \to number$	$primaryExp.val = number.val, primaryExp.cls = number.cls, primaryExp.elementwise = false$
$number \to IntConst$	$number.cls = CLS\_INT, number.val = IntConst.val$
$number \to DoubleConst$	$number.cls = CLS\_DOUBLE, number.val = DoubleConst.val$
$number \to FloatConst$	$number.cls = CLS\_Float, number.val = FloatConst.val$

借助这三个及其他综合属性的计算，我们可以比较容易地完成表达式类型及值的确定和表达式相关的语义检查，如变量与初始化值是否匹配、运算符两侧是否匹配、数组的逐元素计算等。

下面给出赋值语句借助综合属性进行类型检查的代码作为示例：

```
void SemanticAnalysis::exitAssignStmt(CACTParser::AssignStmtContext * ctx){//在
exit()方法中才能利用综合属性。
    std::string name = ctx->lVal()->Ident()->getText();
    auto *var = symbol_table.lookup_symbol(name);
    //检查赋值语句左侧的标识符是否存在
    if(var == nullptr){
        std::cout<<"line"<<std::to_string(ctx->getStart()->getLine())<<":"
<<std::endl;
        throw std::runtime_error("Undefined variable\n");
    }
    //检查赋值语句左侧是否为变量
```

```
    int var_type = var->type;
    if(var_type == TYPE_CONST || var_type == TYPE_CONST_ARRAY){
        std::cout<<"line"<<std::to_string(ctx->getStart()->getLine())<<":"
<<std::endl;
        throw std::runtime_error("\nIllegal assignment to constant\n");
    }
    //检查赋值语句左侧与右侧类型是否相同
    int rval_cls = ctx->exp()->cls;
    if(var->cls != rval_cls){
        std::cout<<"line"<<std::to_string(ctx->getStart()->getLine())<<":"
<<std::endl;
        throw std::runtime_error("\nIllegal assignment between unmatched type\n");
    }
    //检查赋值语句左侧与右侧是否为匹配的数组
    if(var->type != TYPE_ARRAY && ctx->exp()->elementwise){
        std::cout<<"line"<<std::to_string(ctx->getStart()->getLine())<<":"
<<std::endl;
        throw std::runtime_error("\nIllegal assignment between unmatched type\n");
    }else if(ctx->exp()->elementwise){
        auto *array1= symbol_table.lookup_symbol(ctx->exp()->tmp);
        if(array1->value != var->value){
            std::cout<<"line"<<std::to_string(ctx->getStart()->getLine())<<":"
<<std::endl;
            throw std::runtime_error("\nIllegal elementwise assignment because of
unmatched length\n");
        }
    }
}
```

其它

- 报错机制统一使用`throw std::runtime_error(...)`，保证发现语义错误后`compiler`程序仍然正常运行。
- 使用`batch_test_pr002.sh`脚本进行批量测试，提高测试效率。

总结

实验结果总结

- 本次实验的47个测试全部通过。
- 到目前为止，再添加一个中间代码生成器，本设计就将成为一个完整的前端。

分成员总结

陈飞羽

在本次实验中，理论课上的知识对于我进行初期设计的帮助很大。以下是我完成的任务：

- 设计并实现符号表的数据结构。
- 了解`antlr4`的`listener`模式，使用简单高效的`enter()`和`exit()`方法，添加结点的综合属性进行传递。
- 学习使用`grun -gui`，将语法分析树以图的形式展现，便于理解树的遍历过程。
- 实现语义分析，添加、修改符号表，并进行类型检查。

袁欣怡

本次实验中主要负责书写实验报告。本次实验中需要添加语义动作，需要在思考后确定每个语义动作插入的位置。个人认为困难主要在于涉及的代码量较大，理清每个函数需要实现的功能需要花费较长的时间。在实验过程中需要有条理地一步一步来，明确每个步骤实现了什么，以及下一步如何进展。

彭思叡

本次实验主要工作由其他两名组员完成，我负责协调工作，包括gitlab分支管理和code review（compare主分支`master`和`PSR_dev`，并且提交`issue`）。