编译原理研讨课CACT实验PR002报告

成员组成

第16小组：

彭睿思 2020K8009915018

严哲虞 2020K8009907030

游昆霖 2020K8009926006

赵若雯 2020K8009929031

任务说明

本实验的主要内容是在pr001的基础上增加对程序的语义分析，主要内容包括：

1. 根据所给的CACT文法和语义规范，补充完善.g4文件，为各个节点添加属性和进出节点的语义动作
2. 通过antlr工具生成访问语法树的接口，对输入的CACT测试程序进行语义分析
3. 当测试程序符合语义规范时，返回值为0，否则返回非0值

实验过程

设计思路

1. 符号表的设计

   函数和同一作用域中的变量是唯一的，并且不允许出现重名，这与C++中map数据结构的特点相符。另一方面，由于没有规定函数和变量在符号表中出现的顺序，因此为了提高速度，我们采用了基于hash的unordered_map。同时也考虑到函数有形参，变量有嵌套作用域等因素，我们将分开实现函数和变量的符号表。

2. 嵌套作用域的设计

   根据CACT规范，程序中的作用域呈树状，全局作用域为树的根节点，函数和block是它的子节点。而每进入一个block，就相当于进入一级新的子节点。因此我们可以使用多叉树的数据结构来记录各个嵌套作用域。与此同时，为了实现对变量进行跨作用域的查找，每个节点需要包含一个指向父节点的指针，从而实现向上级作用域查找。我们在语义检查的过程中还需要两个指针，分别标记根作用域和当前作用域。

3. 语义分析

   根据实验讲义上的要求，通过antlr生成的访问语法树的接口，用enter和exit方法定义进出节点时需要进行的语义动作，即对各个节点属性的操作，以此检查测试程序是否符合语义规则。在语义检查的过程中，将函数和变量添加到符号表，并收集类型信息，为代码生成提供支持。如果发现语义错误，应该输出错误信息并返回非0值。

代码实现

1. **cact_types.h文件：**

   `cact_types.h`主要定义了各种基本类型，以供后续代码文件使用。

   `cact_basety_t`定义了变量的基本类型：

```
typedef enum {
    BTY_UNKNOWN = 0,
    BTY_VOID,
    BTY_INT,
    BTY_BOOL,
    BTY_FLOAT,
    BTY_DOUBLE,
} cact_basety_t;
```

   在`cact_type_t`中进一步定义了变量的类型，包含了变量是否为数组，变量是否为常量以及变量的基本类型的信息。采用`vector`容器记录数组的各个维度，而对于非数组变量，该容器的size为0。

```
typedef std::vector<uint32_t> arrdims_t;
typedef struct {
    bool            is_const;
    cact_basety_t   basety;
    arrdims_t       arrdims;
} cact_type_t;
```

   而`cact_expr_t`定义了表达式的结构。在该结构体中，op表示一元表达式或二元表达式中的操作类型，如+，-，*，/。basety表示经此次操作表达式返回值的类型，为基本类型其中之一。此外，根据CACT规范，合法的子表达式不能为数组，所以只用basety就可以表达其类型。

   为了方便赋值操作中对操作对象的检查，我们将数组左值`lval`和终结符`number`也当作表达式，其中lVal如果对应的是数组（多维数组的情况），其op为OP_ARRAY，如果对应的是元素（一维数组的情况），其op为OP_ITEM，同时指向子表达式的指针数组为空。对于普通变量，其op也为OP_ITEM，指向子表达式的指针数组为空。对于常量，其op为OP_BASE，指向子表达式的数组为空。

   subexprs是指向子表达式的数组。由于我们将表达式规定成一元或者二元，因此一个长表达式可以被分成多个子表达式。例如长表达式`a + b * c`的子表达式为`a`和`b * c`，它们之间的操作为+；同理在`b * c`当中，`b`和`a`为子表达式，它们之间的操作符为*。

   当一个表达式的op为OP_ARRAY时，表明此时的子表达式是左值子数组，用arrdims标记该数组的维度。int_result用来表示直接由IntConst推出的表达式的值。只有当表达式类型为OP_BASE，且basety为int时才有效。该值用于检查维度前几维的index是否在规定范围内，同时也实现了数组越界检查（实验手册未做要求）

```
struct cact_expr;
typedef std::shared_ptr<cact_expr> cact_expr_ptr;
//指向子表达式的指针数组
typedef std::vector<cact_expr_ptr> subexprs_t;
typedef struct cact_expr{
    cact_op_t       op;
    cact_basety_t   basety;
    subexprs_t      subexprs;
    //只有当op为OP_BASE且basety为int时才会使用
    int             int_result;
    //只有当op为OP_ARRAY时使用
    arrdims_t       arrdims;
}cact_expr_t;
```

为了简化语义分析过程的代码，我们在`cact_types.h`文件中还定义了`TypeUtils`类，用于实现类型的符号，字符串和所占字节大小的转换，因此后续文件代码在进行转换时，可直接调用该类，从而使代码看起来更简洁。

2. **符号表设计（SymbolTable.h文件）**

`class SymbolTable`为本次实验定义的符号表类，该类的主要作用是记录各标识符的信息并处理作用域的范围。其提供的主要方法是对符号表的查找。由于变量有嵌套作用域，函数有形参，因此我们将变量和函数的符号表分开实现如下：

- 变量表设计

map数据结构具有key不重复性的特征，且变量在声明时不允许重名，所以适合用作表的检查。此外，由于不关注变量的声明顺序，使用基于hash的unordermap可以提高查找的性能，符号表表项数据结构的定义如下，

```
typedef std::unordered_map <name_scope_t, var_symbol_item_t, hash_utils> \
        var_table_t;
```

由于不同作用域内可能出现同名变量，如果只使用变量名作为map的key，可能会导致不同作用域的同名变量不能插入。而使用multimap数据结构虽然可以解决上述问题，但在极端情况下，当每个作用域都有同名变量时，查找的过程就退化成了线性查找比对作用域，从而影响性能。因此我们以变量名和它所在的作用域所构成的二元组作为key，二元组`name_scope_t`数据结构的定义如下：

```
typedef struct name_scope{
    std::string name;
    scope_t *scope_ptr;

    bool operator==(const name_scope &other) const{
        return (other.scope_ptr == scope_ptr) && (other.name == name);
    }
} name_scope_t;
```

指针`scope_ptr`指向该变量从属的作用域。按照CACT规范，程序中的作用域呈树状，也就是以全局作用域为根节点，函数和全局block块作为一级子节点，而它们的子节点又组成了二级子节点……所以我们可以使用多叉树来组织嵌套作用域的结构，树的每一个节点便表示一个作用域。与此同时，为了实现变量在嵌套作用域中的查找，每个子节点还包含一个指向其父节点的指针，用于向上

级作用域查找变量。

由于map在插入时会和已有的表项进行比较，默认的比较方式是指针比较，即具有相同name和scope的两个变量，如果存储地址不同，则两者也不相同，这与我们期待的比较结果不符。因此对于非基本类型，需要重新定义比较的方法，可以利用重载运算符==来实现，即`operator==`

`var_symbol_item_t`表示变量表中的表项，同时也是map中存储的value值。其中`cact_type_t`的定义位于`cact_types.h`头文件中，该类型包含了常量和数组的信息。

```
typedef struct var_symbol_item{
    cact_type_t type;
} var_symbol_item_t;
```

同时，由于作为unorder_map中键值的`name_scope`不是基本类型，而是自定义的结构体，因此需要重载unordered_map中使用的hash函数，即定义下述`hash_utils`结构体，使用`operator()`运算符来重载。

```
struct hash_utils{
    size_t operator()(const name_scope_t &tmp) const {
        return (std::hash<std::string>()(tmp.name) ^ std::hash<scope_t *>()
(tmp.scope_ptr));
    }
};
```

- 函数表设计

函数表的设计与变量表的设计类似。由于函数内不允许嵌套函数，函数只能在全局作用域里声明和定义，因此unordered_map中只需用函数名作为key。而函数名的类型属于string基本类型，因此可以直接使用其对应的hash方法，不必像变量表那样额外定义`hash_utils`。函数表表项数据结构的定义如下：

```
typedef std::unordered_map <std::string, func_symbol_item_t> \
        func_table_t;
```

`func_symbol_item_t`表示函数表表项，是unordered_map中存储的value值。函数的返回值类型只能为基本类型，具体定义为：

```
typedef struct func_symbol_item{
    cact_basety_t ret_type;
    fparam_list_t fparam_list;
}func_symbol_item_t;
```

上述的`fparam_list`表示函数形参列表，列表的每一项由形参名和存储形参信息的`fparam_item_t`类型条目构成，每个条目包含了形参的基本类型type和引用的顺序order。由于`fparam_item_t`是一个map，而map在排序时默认按照字典序排，而我们要求形参应该按照在引用中出现的先后顺序进行排列，所以使用`operator<`重载运算符<来指定排序方式。

```
typedef std::map<std::string, fparam_item_t> fparam_list_t;
typedef struct fparam_item{
    cact_type_t type;
    int order;

    bool operator<(fparam_item const &item) const{
        return order < item.order;
    }
} fparam_item_t;
```

- 符号表查找

  在`SymbolTable`对象中定义对符号表的查找操作，即deepfind方法。查找从当前作用域开始，如果在本作用域未找到该变量的声明，则通过scope_ptr指针向上级作用域查找，以此类推，如果始终未找到该变量，则返回符号表末端end_flag，便于判断查找结果。其中，可以使用find在var_table中查找特定作用域的变量，如果查找成功，返回指向该变量的指针，否则返回符号表末端end_flag。

```
//find只能查找特定作用域的var，用于声明检查
//deepfind将向上搜索
var_table_t::iterator deepfind(std::string name, scope_t *scope_ptr){
    auto end_flag = var_table.end();
    while (scope_ptr != NULL){
        auto iter = var_table.find((name_scope){.name=name,
.scope_ptr=scope_ptr});
        if (iter != end_flag){
            //找到该变量
            return iter;
        }
        else
            scope_ptr = scope_ptr->parent;
    }
    //如果最终未找到，返回end_flag用于判断
    return end_flag;
}
```

3. **语义分析头文件设计（SemanticAnalysis.h）**

   该头文件主要定义了`SemanticAnalysis`类，它从CACTListener父类继承。在该类中，我们分别声明`root_scope`和`cur_scope`，用来表示根作用域（全局作用域）和当前所处的作用域。同时我们也在该类中声明各种语义动作：

- 添加内联函数。由于测试用例中直接调用`print_double`等内联函数且未声明，因此我们需要将这些内联函数自行填入符号表中，否则测试样例在调用这些函数时会出现报错。
- 添加进入和退出.g4文件中定义的每个非终结符节点的行为，主要是对各节点综合属性和继承属性进行语义操作。在代码实现上，需要获得对应RuleContext指针，再调用enter和exit方法进行定义。
- 进行语义检查，检查的内容主要可以分为函数、表达式&初值和作用域三个方面。需要检查函数中的函数注册、参数校验、递归和返回值，也需要检查表达式中的多维数组的赋值与定义，变量类型和子表达式间的操作，同时还要创建和维护嵌套作用域。

4. **各节点属性值的定义（CACT.g4）**

   使用antlr提供的locals指令可为各结点添加属性，由于在添加属性时会用到之前自定义的结构类型，因此要在头文件中加入`# include "cact_types.h"`。在各节点定义并相互传递属性时语义分析的要求。

- 常量和变量的声明

  在常量声明中，根据语法规则`constDecl : CONST bType constDef (COMMA constDef)* SEMICOLON ;`可知，被声明的常量类型定义在bType中，因此与类型相关的属性需要传递到constDef节点中。此外，constDef节点可能有多个，这些节点都需要来自bType节点的类型属性。所以，bType和constDef节点中的属性定义如下所示：

```
bType
    locals[
        cact_basety_t basety,
        std::vector<cact_basety_t*> passTo,
    ]
    : INT
    | BOOL
    | FLOAT
    | DOUBLE
    ;
constDef
    locals [
        //由同级btype填充
        cact_basety_t basety,
        std::vector<uint32_t> arraydims,
        std::string name,
        cact_type_t type,
    ]
    : Ident arrayDims ASSIGN constInitVal
    ;
```

  bType节点和constDef节点都添加了表示被声明变量类型的属性basety，其中constDef中的basety是由bType中的basety继承而来。bType中还定义了passTo属性，它是一个指向后面各个constDef节点中basety的指针数组，使用vector容器来实现。容器中的指针数量即为后面constDef的数量，可以在进入父节点（constDecl）时得到。遍历语法树，当离开bType节点，即将进入constDef时，语义动作定义成用bType节点获得的basety填充其vector容器中所指向的后续constDef节点basety属性。通过这种方式，我们可以在进入constDef节点时提前获得同一层级的btype节点提供的类型信息，避免在退出上层节点时再判断constDef变量声明的合法性，从而减少子节点的重复遍历，提升性能。

  此外，在constDef节点中还定义了维度（数组），常量名和常量的type，在离开constDef节点时要根据这些属性把新声明的常量加入到符号表中适当的位置中。

- 多维数组初始化

  然后继续考虑语法规则`constDef : Ident arrayDims ASSIGN constInitVal;`，constInitVal节点中的basety属性从constDef处继承，继承方式类似于bType和consDef之间的继承方式，区别是constDef表达式中只存在一个constInitVal。同时，向constInitVal节点中添加的属性，其语义动作要能够检查多维数组的初始化。

```
constInitVal
    locals[
        cact_basety_t basety,
        //维度数组指针，值依次是从最外层到最内层的维度
        std::vector<uint32_t> *dims_ptr,
        //维度索引，最内层为0
        uint16_t dim_index,
        //表征是否最顶层，只有index为1，且为最顶层才允许以平铺列表初始化
        bool top,
    ]
    : constExp
    | LeftBrace (constInitVal (COMMA constInitVal)*)? RightBrace
    ;
```

考虑到constInitVal推导出的是多维数组初始化的情形，定义属性dims_ptr，它是一个数组指针，指向存储被声明数组各个维度的数组（用vector实现），该vector所记录的各个维度的值在进入constDef节点时由arraydims传递到constInitVal中。

为了区分多维数组初始化中的嵌套定义和平铺列表式定义，引入dim_index和top两个自顶向下的集成属性，前者表示嵌套的`{}`的层数，可以根据层数判断多维数组初始化的定义方式，后者表示此时constInitVal节点推出的`{}`是否为最顶层的，仅当该节点由constDef推出时top为True。如果top为TRUE且dim_index为1，则认为可能是平铺列表初始化，与维度数组各维长度的乘积进行比较，否则只能与维度数组的特定维度比较

通过上述几种属性的传递，可以逐层退出constInitVal节点时自底向上的判断多维数组各维度的合法性。特别的，由于自底向上的维度检查不易确定终点，在离开constDef节点时还需要检查constInitVal最顶层数组的dim_index并与声明的多维数组进行比较，从而保证数组初始化的正确性。

非常量的变量声明和常量声明类似，因此在语义分析时的动作函数实现可以借助模板函数，即`template`来同时实现这两种情况。如在SemanticAnalysis.h中声明的：

```
template <typename T1,typename T2>
void enterConst_Var_Decl(T1 *ctx,std::vector<T2*> def_list);

template <typename T1>
void enterConst_Var_Def(T1 *ctx);

//is_const区分constDef和varDef添加到变量表的不同类型
template <typename T1>
void exitConst_Var_Def(T1 *ctx, bool is_const);
```

- 函数的声明

  考虑语法规则`funcDef : funcType Ident LeftParen funcFParam? (COMMA funcFParam)* RightParen block;`，funcType中定义了函数返回值类型，funcFParam中定义了函数形参，其数量>=0。由于形参列表需要从各个funcFparam中获得，并且还要传递到block中，以便检查函数作用域中是否出现和形参重名的变量声明，因此在进入funcDef节点时需要把该节点中的fparam_list属性传递到它的各个funcFParam子节点中，由它们进行插入，而funcDef只负责管理它的fparam_list属性。由此又可以得出每个funcFParam中的属性需要包含指向fparam_list的指针，即fparam_list_ptr，同时也要记录该节点所生成形参的在列表中的顺序，用order进行标记。

对于block，由于block节点需要从其父节点funcDef中继承形参列表，因此也需要向其添加fparam_list_ptr的属性。为了检查函数返回值的类型与函数定义时的返回值类型是否相同，block节点还需定义一个ret_type属性，用来和funcType中的basety比较。在退出funcDef节点时，已经知晓funcType和block中的bastety和ret_type，就可以进行函数返回值类型是否和定义相同的语义检查。

5. **SemanticAnalysis.cpp**

- 表达式&初值
  - 对表达式操作的检查（OperandCheck函数）

    此函数主要用来检查表达式中的操作是否合法，其中包括对操作对象类型的检查和对操作符使用的检查，在后面介绍的函数中会被多次调用。

    当表达式为一元表达式时，即subexprs的size为1时，需要检查操作对象的类型，如果类型为数组的话则报错，因为CACT规定子表达式的类型不能为数组。若子表达式不是数组类型，则继续检查操作对象的类型和运算符是否匹配。出现在一元表达式中的运算符只能为+，-和非运算，而+和-只能作用在整型和浮点型变量，非运算只能作用在bool型变量，若不满足上述匹配关系则报错

    当表达式为二元表达式时，即subexprs的size为2时，同样地，任一子表达式的类型不可以为数组。同时，被操作的两个子表达式类型必须相同，否则报错。此外，还需要检查被操作对象的类型与二元运算符是否匹配，若不匹配则报错。

  - enterExp/exitExp

    用于检查表达式。`exitExp`在退出表达式规则时，根据具体情况处理，这里需要单独考虑布尔常量：

    - 如果表达式是布尔常量（`BoolConst`），则创建一个对应的结构体，通过`reset`方法让共享指针`ctx->self`指向它。
    - 否则，将表达式的返回值设置为`addExp`的返回值。

  - enterConstExp/exitConstExp

    检查常量类型，分别考虑常量是数字和布尔常量两种情形，如果常量的基本类型与声明不一致，则报错。对于是数字的情形，前面在`ConstInitVal`中继承下来的基本类型`basety`在这里发挥作用，用于检查。

  - enterCond/exitCond

    用于处理条件表达式。返回值必须是布尔类型且不是数组，否则报错。

  - enterLVal/exitLVal

    对左值表达式规则进行语义分析。`exitLVal`首先通过符号表中定义的向根追溯查找的方式，从变量表中检查和提取变量，并判断变量是否存在，如果不存在则报错。

    如果存在，则继续进行维度检查和类型继承。如果是数组，先检查维度列表的长度是否超过维度个数，随后检查每个维度的类型是否是终结符int常量。这里利用了直接从intconst推出来的数值常量的`op`是`OP_BASE`、`basety`是`int`的特性。

    ```
    auto expr_ptr = exp_list[i]->self;
    if(expr_ptr->op != OP_BASE || expr_ptr->basety != BTY_INT){
        std::cout << "Err: LVal: expr for dims must be base_int" <<
    std::endl;
        exit(Semantic_ERR);
    }
    ```

最后，检查索引是否越界。虽然本课程中所有样例都不存在数组访问越界的情形，但我们额外实现了这一功能。如果通过了所有的检查，则继承后续维度。

- enterPrimaryExp/exitPrimaryExp

  在退出主表达式时，根据具体的子表达式类型，自下而上传递指向子表达式结构体的指针。

- enterNumber/exitNumber

  用于处理数字表达式。在退出数字表达式时，根据数字类型是整数、单精度浮点、双精度浮点三种情况，分别创建对应的表达式结构体，并让`ctx->self`指向它。需要注意的是，对于整数的情形需要指定`int_result`。

- enterUnaryExp和exitUnaryExp

  用于处理一元表达式。在退出`UnaryExp`时，根据具体的子表达式处理：

  - 如果是`primaryExp`，则自下而上传递指向子表达式结构体的指针。
  - 如果是一元表达式，构造self指向的结构体，并调用`OperandCheck`来针对操作符的操作对象检查。
  - 如果是函数调用，先检查函数表项是否存在，然后根据返回值的类型构造self指向的结构体。由于它后续的用法和变量一样，因此这里我们直接将其看做变量处理，op设置为`OP_ITEM`。

    ```
    //构造self指向的结构体，类型同变量
    cact_op_t op = OP_ITEM;
    cact_basety_t basety = (iter->second).ret_type;
    ctx->self.reset(new cact_expr_t{.op=op, .basety=basety});
    ```

    然后检查参数列表。如果实参列表是空，则形参列表也应该是空。如果实参列表不为空，则分别检查列表长度是否相同（即参数个数是否匹配）、基本类型是否相同、是否数组，如果是数组还要检查每维度的长度。特别地，对于首维，如果数组在形参中是隐式声明（比如`a[]`），则无需检查维度的长度。这里对于隐式声明的判断依赖在`exitFuncParam`填入的作为标记的0.

    ```
    //逐维度检查长度
    for(int i=0;i<dim_size;i++){
        if(i==0 && fparam_type.arrdims[i]==0){
            //首维为0表示是隐式声明，不用检查
            continue;
        }
        if(expr_ptr->arrdims[i]!=fparam_type.arrdims[i]){
            std::cout << "Err: UnaryExp: param dim_len mismatched" <<
    std::endl;
            exit(Semantic_ERR);
        }
    }
    ```

- 作用域
  - enterCompUnit函数

当进入到CompUnit时，创建一个新的`scope_node_t`对象，赋值给`root_scope`，用来表示全局作用域。根作用域节点的父节点设为空，并把当前作用域设为`root_scope`。此后，每进入一个block就新建一个`scope_t`节点，将当前作用域设为这个新建的作用域节点，并把父指针指向`cur_scope`的旧值。同时调用addBuiltinFunc将内联函数，如`print_int`，`print_double`和`get_int`添加到函数表中，防止在测试用例直接调用时报错。

- enterBlock/exitBlock

  `enterBlock`在进入块规则时，创建一个新的作用域，并将其设置为当前作用域。如果块规则关联着函数参数列表，也就是函数声明作用域的情形，则将函数参数逐一添加到作用域的变量表中。

  `exitBlock`在退出块规则时，根据实验指导中的简化约定，如果块中有`return`语句或包含`return`的stmt，则一定在最后一句。由于这个返回类型的获取与位置有关，可以考虑使用综合属性，按自底向上的方式传递。这种设计也方便了对if-else的分支结构返回类型的检查。

  在该函数中，我们根据指针判断最后一个子节点是否为stmt来确定是否存在`return`语句。如果存在，则将块的返回类型设置为最后一个语句的返回类型；否则，将块的返回类型设置为`BTY_VOID`（空类型）。然后退出当前作用域，将作用域切换回父作用域。

  ```
  void SemanticAnalysis::exitBlock(CACTParser::BlockContext *ctx){
      //根据简化，若有return，一定在最后一个语句
      //根据指针判断最后一个子节点是stmt
      if(ctx->stmt().size()!=0 && (void*)(ctx->children[ctx-
  >children.size()-2]) == (void*)(*ctx->stmt().rbegin())){
          ctx->ret_type = (*ctx->stmt().rbegin())->ret_type;
      }
      else{
          ctx->ret_type = BTY_VOID;
      }
      //退出作用域
      cur_scope = cur_scope->parent;
  }
  ```

  这里在使用`(void*)(ctx->children[ctx->children.size()-2]) == (void*)(*ctx->stmt().rbegin())`时，需要特别注意第一个`ctx`那里没有*，**实验指导有误**。此外，应用该条件时需要增加检查`ctx->stmt().size()!=0`，否则当block中没有stmt时，会触发内存泄漏。

- 语句

  本部分的函数与.g4中设计的分支对应：

  ```
  stmt
      locals[
          cact_basety_t ret_type,
      ]
      : lVal ASSIGN exp SEMICOLON                          #stmt_assign
      | (exp)? SEMICOLON                                   #stmt_exp
      | block                                              #stmt_block
      | IF LeftParen cond RightParen stmt (ELSE stmt)?     #stmt_if
      | WHILE LeftParen cond RightParen stmt               #stmt_while
      | (BREAK | CONTINUE | RETURN exp?) SEMICOLON         #stmt_bcr
      ;
  ```

- enterStmt_assign/exitStmt_assign

  `enterStmt_assign`和`exitStmt_assign`对赋值语句规则进行语义分析。在进入赋值语句时无操作，而在退出赋值语句规则时，首先进行左值检查，确保左值不是常量。左值是否存在于变量表的检查会在exitLVal中完成，这里无需检查。然后调用`OperandCheck`函数进行类型检查。最后，将赋值语句的返回类型设置为`BTY_VOID`。

- enterStmt_exp/exitStmt_exp

  在进入和退出表达式语句时被调用。`exitStmt_exp`在退出表达式语句规则时，将表达式语句的返回类型设置为`BTY_VOID`。

- enterStmt_block/exitStmt_block

  用于处理block套block的情形。

  `enterStmt_block`：在进入块语句规则时，将该块的函数参数列表指针设置为`nullptr`。

  `exitStmt_block`：在退出块语句规则时，自下而上传递子块的返回类型。

- enterStmt_if/exitStmt_if

  `enterStmt_if`无操作。

  `exitStmt_if`：在退出条件语句规则时，根据简化的约定，如果条件语句的`if-else`分支中的语句包含`return`语句，则该`return`语句必定是块的最后一句。并且，我们将没有`else`分支的情形视为有`else`分支，但不执行任何操作，其返回类型为`void`。根据这一约定，进行以下错误检查：

  - 如果只有`if`分支且其stmt的返回类型不是`BTY_VOID`，则报错。
  - 如果有`if-else`分支且stmt的返回类型不一致，则报错。

  最后，将条件语句的返回类型设置为第一个分支的语句的返回类型。

- enterStmt_while/exitStmt_while

  `enterStmt_while`无操作。

  `exitStmt_while`：这里认为跳出循环后的分支是`void`。和前面提到的简化约定一样，如果while语句的语句部分包含`return`语句，则该语句必须在块的最后。随后检查异常的情形，确保stmt的返回类型是`BTY_VOID`。最后，将`while`语句的返回类型设置为stmt的返回类型。这里的处理实际上类似前面`if-else`中只有`if`分支的情形。

- enterStmt_bcr/exitStmt_bcr

  对分支控制语句进行语义分析。`exitStmt_bcr`在退出分支控制语句时，根据具体情况处理返回类型：

  - 如果语句是`return exp`形式，则返回类型由`exp`的类型决定，需要检查`exp`是否是数组，因为`return`不允许返回数组类型，只允许返回基本类型。如果不是数组，则可以把`exp`的类型赋给`ctx->ret_type`。
  - 如果语句不是`return exp`形式，即对应`break`、`continue`或`return`的情形，则返回类型为`BTY_VOID`。

- 函数

  - exitCompUnit函数

    在退出CompUnit时需要检查main函数相关的属性。首先在函数表中查询是否存在main函数，如果查询失败，则出现语义错误。如果查询成功，则取函数表中记录函数信息的表项（即value值），继续检查main函数的形参列表和返回值类型。正常情况下main函数的形参列表为空并且返回值为整型，如果不满足这两个条件则报错。

- enterFuncDef/exitFuncDef

  用于在进入和退出函数定义时对其进行语义分析。

  locals定义了属性`fparam_list`，表示形参列表。这里`enterFuncDef`函数会把形参列表的指针传给形参节点和和`block`。传给形参节点是为了让它后面负责填写该列表，而传给`block`则是为了让其可以提前获取形参列表以加入作用域，避免重复遍历。

  ```
  void SemanticAnalysis::enterFuncDef(CACTParser::FuncDefContext *ctx){
      //由子节点填写形参列表，指定形参顺序
      int order = 0;
      for(auto fparam: ctx->funcFParam()){
          fparam->fparam_list_ptr = &(ctx->fparam_list);
          fparam->order = order;
          order ++;
      }

      //对于block，由fparam提供形参列表
      ctx->block()->fparam_list_ptr = &(ctx->fparam_list);
  }
  ```

  `exitFuncDef`函数在退出函数定义规则时，获取函数的名称和返回类型，检查函数是否已经在符号表中定义，如果未定义则创建函数符号表项并添加到符号表中，最后再检查函数声明的返回类型与函数体的返回类型是否一致。

  - enterFuncType/exitFuncType

    `enterFuncType`无操作，而`exitFuncType`在退出函数类型时，将函数类型填入函数类型规则的基本类型字段`basety`。这里直接获得的是函数类型的字符串形式，需要借助我们在`cact_types.h`中定义的map转化为对应的枚举类型。

  - enterFuncFParam/exitFuncFParam

    在进入和退出函数参数规则时被调用。`exitFuncParam`首先获取函数参数的名称、基本类型和维度信息，根据左括号的数量确定数组的总层数，并将维度长度依次添加到数组维度列表中。对于隐式声明的情形，向数组维度列表中填入0，这样后续可以通过是否为0判断隐式声明。这里使用0是因为样例中不会出现0，正好可以用来作为标记。

    随后创建函数参数的类型对象，并填入前面获取的内容，然后检查函数参数是否已经在参数列表中定义，如果未定义则创建函数参数项并添加到参数列表中。这里的参数列表正是前面由`enterFuncDef`函数传过来的列表。

- 变/常量声明

  由于变量和常量的检查高度相似，因此我们使用模板函数`enter/exitConst_Var_Decl`实现相关功能，在需要使用时调用即可。这里仅以常量为例进行说明，对于变量的部分不再赘述。

  - enterDecl/exitDecl函数

    当在CACTParser解析过程中遇到声明语句时，ANTLR会调用这两个函数，用于进入和退出声明（包括变量声明和函数声明），但在我们的实现中没有定义具体的操作。由于我们在`enterConst_Var_Decl`函数中的特殊设计，这里无需在`exitDecl`时重复遍历Def再来进行类型检查，而是在前面第一次自动遍历`Def`时就已完成。

  - enterConstDecl/exitConstDecl函数

    这两个函数在进入和退出常量声明时调用。

在进入常量声明时，通过调用`enterConst_Var_Decl`函数来对常量定义进行语义分析，将后续Def的基本类型传递向btype。该函数在定义时使用了模板`template <typename T1,typename T2>`，在本次调用中，`T1`被替换成`CACTParser::ConstDeclContext`，而`T2`被替换成指向`CACTParser::ConstDefContext`的指针。`constDef()`方法会获取`ctx`中所有匹配`constDef`规则的语法结构，并返回指向它们的指针。

```
template <typename T1,typename T2>
void SemanticAnalysis::enterConst_Var_Decl(T1 *ctx,std::vector<T2*>
def_list){
    //将后续Def的基本类型传递向btype，&def->type.basety非法
    for (auto def: def_list){
        ctx->bType()->passTo.push_back(&(def->basety));
    }
}
```

- enterBType/exitBType函数

  用于进入和退出基本类型（`BType`）。在退出时，将基本类型值赋值给`ctx->basety`，并且填写`ctx->passTo`指向的所有节点的`BType`，从而实现基本类型的继承传递，这样后面的函数才能检查类型是否匹配。

- 多维数组
  - enterArrayDims/exitArrayDims函数

    用于在进入和退出数组维度`ArrayDims`时进行语义分析。其中`ArrayDims`是为了方便在进入`constDef`和`varDef`时为`constinitval`获取维度数组而设计的。

    `exitArrayDims`函数负责解析数组的维度。它会遍历`ctx->IntConst()`中匹配的所有整数常量，对于每个整数常量，将其文本表示转换为无符号整数并存储在`len`中，这样就得到了维度的长度。随后它将这些维度长度填入`dims_ptr`所指向的vector。

  - enter/exitConstInitVal函数

    用于检查常量初始值的合法性，包括基本类型的继承、维度的传递和维度索引的检查。

    具体而言，`enterConstInitVal`中会自顶向下继承数组的基本类型`basety`，在最内层节点(`constExp`)完成检查。随后，它将`basety`和维度指针`dims_ptr`传递给子数组或子元素的初始值。

    而在`exitConstInitVal`函数中，实现了对列举和嵌套两种形式初始值的支持，并且不允许二者混合。该函数会自底向上检查维度，直接推出`constExp`的`dim_index`为0，推出{}为1，这两种情况都不需要检查维度。对于其他情况，先检查所有子数组的index是否一致，然后分情况考虑。对最外层数组且无嵌套的情形，应当以列举方式初始化初始化，并且列举的值的个数应该等于数组的元素个数。而对于嵌套形式定义，则从最内层维度开始检查：对于初始值维度大于声明的情况，在该节点报错；对于初始值维度小于声明的情况，则由`exitDef`负责检查。

6. **main.cpp**

`main.cpp`中的逻辑与上一次实验大致相同。由于本次实验新增的功能，我们需要先实例化符号表`SymbolTable`和用于语义分析的`SemanticAnalysis`。随后，程序会打开一个文件流，并将其传递给`ANTLRInputStream`以获取输入。然后，它创建`CACTLexer`和`CACTParser`对象，用来解析输入，并创建语法分析树。由于我们实现的是listener模式，因此需要使用`walk`方法遍历树，并调

用`SemanticAnalysis`中的函数来执行语义分析，并输出结果。最后，程序返回0表示正常退出。

```
 tree::ParseTree *tree = parser.compUnit();

 std::cout << "----------------------------Print AST:-----------------------
-----" << std::endl;
 std::cout << tree->toStringTree(&parser) << std::endl;

 tree::ParseTreeWalker::DEFAULT.walk(&semantic_analysis, tree);
```

过程思考

在编写代码的过程中主要遇到了下面几个问题：

1. shared_ptr的使用问题

   我们在构建语义分析的代码中为每个exp，number和lVal都设置了指向表达式结构体的shared_ptr。考虑语义规则推导`exp -> addExp -> mulExp -> unaryExp -> primaryExp -> number`，在数值从number节点向上传递至exp节点的过程中，只通过每个exp的self属性将数值逐级进行传递，没有其他额外的操作。如果不使用共享指针，在向上传递表达式结构体时会不断进行新结构体的创建和赋值，从而影响了性能。而在使用共享指针时，被指向结构体的生存周期是各个共享指针的并集，因此在传递时会避免在各个exp节点中创建的临时结构体在退出该节点时被释放，从而产生野指针的问题。

   但需要注意的是，在创建指向一个新结构体的共享指针时，应该直接用共享指针进行创建，而不是先创建一个新结构体，再把该结构体的地址传到共享指针中。以下述`exitRelExp`函数为例，正确的写法是`ctx->self.reset(new cact_expr_t{.op=op,.basety=BTY_BOOL})`，而不是先要`rel_exp = new cact_expr_t`，再把`&rel_exp`应用到`ctx->self.reset`中。后者的写法会导致内存的泄露，从而报错double free or corruption。

   ```
   void SemanticAnalysis::exitRelExp(CACTParser::RelExpContext *ctx){
       if(ctx->BoolConst()!=nullptr){
           //创建self所指结构体
           cact_op_t op = OP_BASE;

           ctx->self.reset(new cact_expr_t{.op=op, .basety=BTY_BOOL});
       }
       else{
           exitBinaryExp(ctx, ctx->relExp(), ctx->relOp(), ctx->addExp(),
   true);
       }
   }
   ```

2. stmt().rbegin()的使用

   当block中含有return语句时，需要判断退出block时返回值的类型，其语义动作定义在exitBlock函数中，相关语法规则为`block : LeftBrace (decl|stmt)* RightBrace;`。由于CACT规范中规定return语句必须在函数的最后，因此在判断函数block里最后一句是否为return语句时，需要查看最后的语句是否可由`stmt`推导而来。在代码中使用rbegin逆迭代器来定位block所有stmt子节点的最后一个，以便后续的检查。

但要注意的是，在使用stmt.rbegin前需要先判断block的子节点中是否含有stmt节点，若直接使用可能会出现stmt.rbegin是空指针的情况，会导致内存泄露。

```
if(ctx->stmt().size()!=0 && (void*)(ctx->children[ctx->children.size()-2])
== (void*)(*ctx->stmt().rbegin()))
{
   ctx->ret_type = (*ctx->stmt().rbegin())->ret_type;
 }
```

3. 在定义exitBinaryExp动作时未考虑到布尔表达式的结果类型和子表达式可能不一致

exitBinaryExp定义了离开二元表达式节点的语义动作，由于addExp、mulExp、relExp等都属于二元表达式，它们的语义动作都是相似的，所以可以直接用模板函数来定义：

```
template <typename T1,typename T2, typename T3, typename T4>
void SemanticAnalysis::exitBinaryExp(T1 *ctx, T2 *aExp, T3 *op_ptr, T4
*bExp, bool is_boolexp)
{
    if(aExp == nullptr){
        ctx->self = bExp->self;
    }
    else{
        cact_op_t op = (typeutils.str_to_op)[op_ptr->getText()];
        cact_basety_t basety;
        if(is_boolexp){
            basety = BTY_BOOL;
        }
        else{
            //任取一个子表达式类型，如果两个子表达式类型不一致，会在OperandCheck中报错
            basety = bExp->self->basety;
        }
        subexprs_t subexprs;
        subexprs.push_back(aExp->self);
        subexprs.push_back(bExp->self);
        //作为维度的intConst和子数组不允许操作，不需指定int_result和arrdims
        ctx->self.reset(new cact_expr_t{.op=op, .basety=basety,
.subexprs=subexprs});
        //类型检查
        OperandCheck(ctx->self);
}
```

原本错误的代码在定义exitBinaryExp时没有加入is_boolexp形参，导致结果出错。之所以添加is_boopexp参数，是因为在布尔表达式（relExp、eqExp、lOrExp、lAndExp）中，表达式的返回类型被确定为bool，与子表达式类型未必一致，不能简单地使用指向子表达式的指针来获取表达式类型，而是需要单独考虑。我们添加一个新的参数is_boolexp来标识该表达式为二元表达式的情况。

4. 检查多维数组初始化与维度的匹配出错

考虑一个多维数组嵌套定义的例子`a[3][2][2] = {{1, 2}, {2, 3}}`，在执行`exitConstInitVal`语义动作时，根据等号右边的嵌套 定义，只会检查dim_index为1和2的情形。在该例子中，ConstInitVal节点的dim_ptr指向的数组元素为{3,2,2}，根据下述代码的逻辑，index的值只能得到1和2两种，也就是只检查了该多维数组的后两维没有出现越界，然后便结束语义检查。但实际上由于维度与嵌套定义不符，该例子应该报错，而错误代码没有报错。

```
//嵌套形式定义
//维度索引相对应的数组索引
int index = (*ctx->dims_ptr).size() - ctx->dim_index;
if(index<0){
    std::cout << "Err: constInitVal: nested: layer greater than dims_size"
<< std::endl;
    exit(Semantic_ERR);
}
size_t dim_len = (*ctx->dims_ptr)[index];
if(len > dim_len){
    std::cout << "Err: constInitVal: nested: len greater than dim_len" <<
std::endl;
    exit(Semantic_ERR);
}
```

这个错误的原因是，当我们在exitConstInitVal方法中自底向上的判断维度合法性时，不易确定检查的终点，就可能出现只检查初始值部分维度的情况。解决该bug的办法是在离开ConstDef节点时增加对数组维度和嵌套定义维度的检查，保证初始值检查的层次与预期一致，即在exitConstDef中增加下面的代码：

```
if(ctx->constInitVal()!=nullptr){
    //同时考虑非数组左右都应当为0
    if(ctx->arraydims.size() != ctx->constInitVal()->dim_index){
        //只有左边为数组，右侧为元素列表才允许不等
        if((ctx->arraydims.size() != 0) && (ctx->constInitVal()-
>dim_index==1)){
            ;
        }
        else{
            std::cout << "Err: Const_Var_Def: Def and Initval dims
mismatched"<<std::endl;
            exit(Semantic_ERR);
        }
    }
}
```

总结

实验结果总结

在本实验中，我们使用locals指令为上个实验中构建的抽象语法树增加了属性，并在遍历节点的过程中计算属性。我们需要检查这些属性是否符合语义规范，这一部分的功能依赖我们在遍历过程中构建的符号表和树状作用域。至此，我们的设计已经具有了词法、语法、语义分析的功能，再添加一个中间代码生成器就可以成为一个完整的前端。

分成员总结

游昆霖：

在本次实验中我与小组成员充分讨论，在理解实验手册的基础上完善了整体逻辑框架，并进行了代码实现。在本次实验中，我对如何提升代码的逻辑、性能和可读性产生了较为深刻的体会。简洁、易读、高效的代码对我们提出了几点要求：

1. 更加清晰易读的代码框架：通过高效简洁的类型设计和声明和更加有条理的注释可以大大提高代码的逻辑性和易读性，模板函数的使用也能为相似逻辑进行封装，避免代码的重复冗杂，为开发和调试提供便利。
2. 更加精简高效的功能设计：在本次实验的逻辑设计和代码实现中，我特别关注了代码性能的提高。通过属性的设计和对其传递方向的梳理，我们可以有效的避免节点的重复遍历，提升语义分析性能。清晰、精简的属性传递也同时便利了错误的发现和调试。

彭睿思：

在本次实验中我主要负责对程序功能性和正确性的调试。由于语义分析整体框架较为复杂，同时对C++各种语法的用法并不熟悉，我们小组在开始调试时也面临了段错误、内存泄漏等多种问题。通过对代码的逻辑梳理和print、gdb等方式的错误定位，我们让程序得以正确运行。而在对程序功能和正确性的验证中，我们主要对出错样例进行了删减和改写，从而精准的定位出错原因，并对可能存在的类似逻辑错误进行了调试。总体而言，在本次实验的过程中，我对CACT语言规范和实验中所应用到的C++、ANTLR4等工具有了更深的认识，获益匪浅。

赵若雯：

本次实验的主要难点是代码框架的搭建比较复杂，需要细致地考虑语义分析当中出现的每一种情况，这就需要对CACT语言规范进行熟练掌握，同时还需要使各个非终结符属性和符号表数据结构设计与语义检查代码的编写相互配合起来。在编写代码时，还需要根据所要实现的功能，结合C++语言的用法，设计出合适的数据结构。因此我在整个过程中又学习到了许多C++语言的用法。在本次实验中，与同组成员相互讨论代码编写的思路和原理也使我更好地理解了语义分析的过程。

严哲虞：

本实验代码量较大，需要理清属性与语义动作之间的关系，仔细考虑所有需要检查的地方，才能保证语义分析的完备性；同时，也要认真考虑在进入和退出每个节点时实现的功能，从而在正确实现功能的基础上提升性能。通过本次实验，我对ANTLR接口的使用更加熟悉，也加深了对编译过程的认识。在实验的过程中，对C++和面向对象思想不够熟悉是一大困难，但这也正好能成为学习相关知识的契机。