

Lab3：类型检查和中间代码生成

需要阅读的代码

涉及到的文件：

- `include/type.h`:语法树节点属性类的定义
- `include/tree.h`:如果需要的话，可以自行添加成员变量及函数
- `include/SysY_tree.h`:如果需要的话，可以自行添加成员变量及函数
- `ir_gen/semant.h`:类型检查辅助类的定义，在这一步骤中，我们还需要实现全局变量定义的中间代码生成，你可以根据需要自行添加成员变量及函数
- `ir_gen/semant.cc`:类型检查的主要函数定义，根据你想实现的文法实现你需要函数中的TODO()，在这一步骤中，我们还需要实现全局变量定义的中间代码生成，参考框架的实现是和类型检查一同完成，你也可以根据需要编写更多的函数
- `include/Instruction.h`:LLVMIR指令类定义
- `include/ir.h`:LLVMIR类的定义
- `include/basic_block.h`:基本块的定义
- `utils/Instruction.cc`:LLVMIR指令类辅助函数的实现
- `utils/Instruction_out.cc`:LLVMIR输出函数的实现
- `ir_gen/IRgen.h`:中间代码生成辅助类的定义
- `ir_gen/IRgen.cc`:中间代码生成的主要函数定义，根据你想实现的文法实现你需要函数中的TODO()

Codes Reading

`include/type.h`

Type类的定义：

```
class Type {
public:
    // 我们认为数组的类型为PTR
    enum ty { VOID = 0, INT = 1, FLOAT = 2, BOOL = 3, PTR = 4, DOUBLE = 5 } type;
    std::string GetTypeInfo();
    Type() { type = VOID; }
};
```

定义了type，也就是节点的类型

```
enum ty { VOID = 0, INT = 1, FLOAT = 2, BOOL = 3, PTR = 4, DOUBLE = 5 } type;
```

这是一个枚举类，后续**类型转换**的时候会使用到（llvm转type、type转llvm）

常量值的定义：

```

class ConstValue {
public:
    bool ConstTag;
    union ConstVal {
        bool BoolVal;
        int IntVal;
        float FloatVal;
        double DoubleVal;
    } val;
    std::string GetConstValueInfo(Type ty);
    ConstValue() {
        val.IntVal = 0;
        ConstTag = false;
    }
};

```

- 有一个ConstTag表示是否为常量
- ConstVal表示值的类型，有bool、int、float和double
- 注意的是，初始化的时候，值为0，ConstTag为false

变量的属性类：

```

// 变量的属性
class VarAttribute {
public:
    Type::ty type;
    bool ConstTag = 0;
    std::vector<int> dims{};    // 存储数组类型的相关信息
    // 对于数组的初始化值，我们将高维数组看作一维后再存储 eg. ([3 x [4 x i32]]) => [12 x i32]
    std::vector<int> IntInitVals{};
    std::vector<float> FloatInitVals{};

    // TODO(): 也许你需要添加更多变量
    VarAttribute() {
        type = Type::VOID;
        ConstTag = false;
    }
};

```

变量的属性：

- 初始化为VOID类型和非常量
- IntInitVals和FloatInitVals用于数组初始化
- dims表示数组的维度

节点——用于类型检查

```

// 语法树节点的属性
class NodeAttribute {

```

```

public:
    int line_number = -1;
    Type T;
    ConstValue V;
    std::string GetAttributeInfo();

    //定义运算符的对应值
    enum opcode {
        ADD = 0,      // +
        SUB = 1,      // -
        MUL = 2,      // *
        DIV = 3,      // /
        MOD = 4,      // %
        GEQ = 5,      // >=
        GT = 6,       // >
        LEQ = 7,      // <=
        LT = 8,       // <
        EQ = 9,       // ==
        NE = 10,      // !=
        OR = 11,      // ||
        AND = 12,     // &&
        NOT = 13,     // !
    };
};

```

- 行号为-1，初始值
- 类型Type T
- 常量V
- 新定义一个枚举类opcode，列举了一些操作码

include/tree.h

```

class tree_node {
protected:
    int line_number;

public:
    NodeAttribute attribute;    // 在类型检查阶段需要阅读该类的代码，语法分析阶段可以先忽略该变量
    int GetLineNumber() { return line_number; }
    void SetLineNumber(int t) { line_number = t; }

    virtual void codeIR() = 0;    // 中间代码生成
    virtual void printAST(std::ostream &s, int pad) = 0;    // 打印语法树
    virtual void TypeCheck() = 0;    // 类型检查
};

```

和上次差不多，TypeCheck为类型检查，codeIR为中间代码生成

include/SysY_tree.h

之前的语法分析的实验已经分析过了，就不再赘述

最顶层是Program，一层一层递归下来，最后到expression表达式

看一下几个特殊的：

左值的类

```
class Lval : public __Expression {
public:
    Symbol name;
    std::vector<Expression> *dims;
    bool is_left = true;    // left value or right value
    int scope = -1;
    Operand ptr;    // use IRgen, determine the ptr of this lval
    Lval(Symbol n, std::vector<Expression> *d) : name(n), dims(d) {}
    void codeIR();
    void TypeCheck();
    void printAST(std::ostream &s, int pad);
};
```

- dims为数组维度
- is_left为是否为左值，就是能不能被赋值的意思
- 左值一般来说，操作数都是用的ptr，因为可以代表指针
- 作用域初始化为-1

函数参数列表：

```
//{Exp,Exp,Exp,Exp}
class FuncRParams : public __Expression {
public:
    std::vector<Expression> *params{};
    FuncRParams(std::vector<Expression> *p) : params(p) {}
    void codeIR();
    void TypeCheck();
    void printAST(std::ostream &s, int pad);
};
```

成员params，代表参数列表中的参数

函数调用：

```
// name(FuncRParams)
class Func_call : public __Expression {
public:
    Symbol name;
    Expression funcr_params;
    Func_call(Symbol n, Expression f) : name(n), funcr_params(f) {}
    void codeIR();
    void TypeCheck();
    void printAST(std::ostream &s, int pad);
};
```

name为函数名, funcr_params为参数列表中的参数, 为表达式

while_stmt:

```
class while_stmt : public __Stmt {
public:
    Expression Cond;
    Stmt body;
    // construction
    while_stmt(Expression c, Stmt b) : Cond(c), body(b) {}
    void codeIR();
    void TypeCheck();
    void printAST(std::ostream &s, int pad);
};
```

Cond为控制条件; body为while循环体

include/Instruction.h

```
// @instruction
class BasicInstruction {
public:
    // @Instruction types
    enum LLVMIROpcode {
        OTHER = 0,
        LOAD = 1,
        STORE = 2,
        ADD = 3,
        SUB = 4,
        ICMP = 5,
        PHI = 6,
        ALLOCA = 7,
        MUL = 8,
        DIV = 9,
        BR_COND = 10,
        BR_UNCOND = 11,
        FADD = 12,
        FSUB = 13,
        FMUL = 14,
        FDIV = 15,
```

```

    FCMP = 16,
    MOD = 17,
    BITXOR = 18,
    RET = 19,
    ZEXT = 20,
    SHL = 21,
    FPTOSI = 24,
    GETELEMENTPTR = 25,
    CALL = 26,
    SITOFPPTR = 27,
    GLOBAL_VAR = 28,
    GLOBAL_STR = 29,
    FPEXT = 30,
};

// @Operand datatypes
enum LLVMType { I32 = 1, FLOAT32 = 2, PTR = 3, VOID = 4, I8 = 5, I1 = 6, I64
= 7, DOUBLE = 8 };

// @ <cond> in icmp Instruction
enum IcmpCond {
    eq = 1,      //: equal
    ne = 2,      //: not equal
    ugt = 3,     //: unsigned greater than
    uge = 4,     //: unsigned greater or equal
    ult = 5,     //: unsigned less than
    ule = 6,     //: unsigned less or equal
    sgt = 7,     //: signed greater than
    sge = 8,     //: signed greater or equal
    slt = 9,     //: signed less than
    sle = 10     //: signed less or equal
};

enum FcmpCond {
    FALSE = 1,   //: no comparison, always returns false
    OEQ = 2,     // ordered and equal
    OGT = 3,     //: ordered and greater than
    OGE = 4,     //: ordered and greater than or equal
    OLT = 5,     //: ordered and less than
    OLE = 6,     //: ordered and less than or equal
    ONE = 7,     //: ordered and not equal
    ORD = 8,     //: ordered (no nans)
    UEQ = 9,     //: unordered or equal
    UGT = 10,    //: unordered or greater than
    UGE = 11,    //: unordered or greater than or equal
    ULT = 12,    //: unordered or less than
    ULE = 13,    //: unordered or less than or equal
    UNE = 14,    //: unordered or not equal
    UNO = 15,    //: unordered (either nans)
    TRUE = 16    //: no comparison, always returns true
};

private:

protected:
    LLVMIROpcode opcode;

```

```

public:
    int GetOpcode() { return opcode; }    // one solution: convert to pointer of subclasses

    virtual void PrintIR(std::ostream &s) = 0;
};

```

- LLVMIROpcode: 指令类型
- LLVMType: 操作数的数据类型
- lcmpCond: 整形比较
- fcmpCond: 浮点型比较

include/ir.h

```

class LLVMIIR {
public:
    // 全局变量定义指令
    std::vector<Instruction> global_def{};
    // 函数声明指令
    std::vector<Instruction> function_declare{};
    // key为函数定义指令(需要保证函数定义指令与函数一一对应), value为函数对应的cfg
    // 在中间代码生成中, 你可以暂时忽略该成员变量, 你可以在代码优化时再考虑该变量
    std::map<FuncDefInstruction, CFG *> llvm_cfg{};
    // key为函数定义指令, value为函数对应的所有基本块, 该成员变量会在输出中使用, 所以你必须对该变量正确赋值
    // 你必须保证函数的入口基本块为0号基本块, 否则在后端会出现错误。
    std::map<FuncDefInstruction, std::map<int, LLVMBlock>> function_block_map;
    // 我们用函数定义指令来对应一个函数
    // 在LLVMIIR中新建一个函数
    void NewFunction(FuncDefInstruction I) { function_block_map[I] = {}; }
    // 获取一个在函数I中编号为now_label的基本块, 该基本块必须已存在
    LLVMBlock GetBlock(FuncDefInstruction I, int now_label) { return
function_block_map[I][now_label]; }
    // 在函数I中新建一个新的编号为x的基本块, 该编号不能与已有的重复
    LLVMBlock NewBlock(FuncDefInstruction I, int &x) {
        ++x;
        function_block_map[I][x] = new BasicBlock(x);
        return GetBlock(I, x);
    }
    void printIR(std::ostream &s);

    void CFGInit();
    void BuildCFG();
};

```

- global_def: 全局变量定义指令
- function_block_map: 我们用函数定义指令来对应一个函数
- GetBlock: 寻找当前存在的块
- NewBlock: 新建一个块, 会将label的值加一来更新块号

include/basic_block.h

```
class BasicBlock {
public:
    std::string comment;    // used for debug
    int block_id = 0;
    std::deque<Instruction> Instruction_list{};
    void InsertInstruction(int pos, Instruction Ins);
    void printIR(std::ostream &s);
    BasicBlock(int id) : block_id(id) {}
};
```

block_id就是对应的块号

InsertInstruction就是插入一条指令

utils/Instruction.cc

```
static std::unordered_map<int, RegOperand *> RegOperandMap;

RegOperand *GetNewRegOperand(int RegNo) {
    auto it = RegOperandMap.find(RegNo);
    if (it == RegOperandMap.end()) {
        auto R = new RegOperand(RegNo);
        RegOperandMap[RegNo] = R;
        return R;
    } else {
        return it->second;
    }
}
```

获取寄存器的编号

首先在 `RegOperandMap` 中查找是否有与给定 `RegNo` 关联的 `RegOperand`。如果 `RegNo` 存在于映射中，`it` 会指向相应的键值对，否则 `it` 会指向 `end()`。

- 如果 `RegNo` **不在**映射中（即 `it == RegOperandMap.end()`），则创建一个新的 `RegOperand` 对象，并将其添加到 `RegOperandMap` 中
 - 创建了一个新的 `RegOperand` 对象，使用给定的 `RegNo` 进行初始化，然后将其存储到映射中，并返回这个新创建的对象
- 如果 `RegNo` **在**映射中，直接返回已经存在的 `RegOperand` 对象

其他的getnew:

- `GetNewLabelOperand`: 获取一个新的标签号
- `GetNewGlobalOperand`: 获取一个新的全局变量名称的编号

一些指令：

```
void IRgenArithmeticI32(LLVMBlock B, BasicInstruction::LLVMIROpcode opcode, int
reg1, int reg2, int result_reg) {
    B->InsertInstruction(1, new ArithmeticInstruction(opcode,
BasicInstruction::LLVMType::I32, GetNewRegOperand(reg1), GetNewRegOperand(reg2),
GetNewRegOperand(result_reg)));
}
```

在块后面插入一条指令。opcode就是指定操作的运算符。

调用 `GetNewRegOperand` 函数来获取寄存器编号 `reg1` 和 `reg2` 的操作数对象。通过缓存机制，函数会返回已有的寄存器操作数对象或创建新的对象。

一些用到的指令函数：

```
// 一些指令，用于在基本块B末尾生成一条新指令
// 生成整数类型的算术指令（如加法、减法等），操作数来自两个寄存器
void IRgenArithmeticI32(LLVMBlock B, BasicInstruction::LLVMIROpcode opcode, int
reg1, int reg2, int result_reg);
// 生成浮点类型的算术指令（如加法、减法等），操作数来自两个寄存器
void IRgenArithmeticF32(LLVMBlock B, BasicInstruction::LLVMIROpcode opcode, int
reg1, int reg2, int result_reg);
// 生成整数类型的算术指令，左操作数为立即数，右操作数为寄存器
void IRgenArithmeticI32ImmLeft(LLVMBlock B, BasicInstruction::LLVMIROpcode
opcode, int val1, int reg2, int result_reg);
// 生成浮点类型的算术指令，左操作数为立即数，右操作数为寄存器
void IRgenArithmeticF32ImmLeft(LLVMBlock B, BasicInstruction::LLVMIROpcode
opcode, float val1, int reg2, int result_reg);
// 生成整数类型的算术指令，两个操作数均为立即数
void IRgenArithmeticI32ImmAll(LLVMBlock B, BasicInstruction::LLVMIROpcode opcode,
int val1, int val2, int result_reg);
// 生成浮点类型的算术指令，两个操作数均为立即数
void IRgenArithmeticF32ImmAll(LLVMBlock B, BasicInstruction::LLVMIROpcode opcode,
float val1, float val2, int result_reg);
// 生成整数类型的比较指令（如等于、大于等于等），操作数来自两个寄存器
void IRgenIcmp(LLVMBlock B, BasicInstruction::IcmpCond cmp_op, int reg1, int
reg2, int result_reg);
// 生成浮点类型的比较指令（如等于、大于等于等），操作数来自两个寄存器
void IRgenFcmp(LLVMBlock B, BasicInstruction::FcmpCond cmp_op, int reg1, int
reg2, int result_reg);
// 生成整数类型的比较指令，右操作数为立即数
void IRgenIcmpImmRight(LLVMBlock B, BasicInstruction::IcmpCond cmp_op, int reg1,
int val2, int result_reg);
// 生成浮点类型的比较指令，右操作数为立即数
void IRgenFcmpImmRight(LLVMBlock B, BasicInstruction::FcmpCond cmp_op, int reg1,
float val2, int result_reg);
// 生成浮点数到整数的类型转换指令
void IRgenFptosi(LLVMBlock B, int src, int dst);
// 生成整数到浮点数的类型转换指令
void IRgenSitofp(LLVMBlock B, int src, int dst);
// 将布尔值（i1）扩展为 32 位整数（i32）
void IRgenZextI1toI32(LLVMBlock B, int src, int dst);
```

```

// 根据索引生成数组元素地址（整数索引，指针类型为 i32）
void IRGenGetElementptrIndexI32(LLVMBlock B, BasicInstruction::LLVMType type, int
result_reg, Operand ptr,
                                std::vector<int> dims, std::vector<Operand>
index);
// 根据索引生成数组元素地址（整数索引，指针类型为 i64）
void IRGenGetElementptrIndexI64(LLVMBlock B, BasicInstruction::LLVMType type, int
result_reg, Operand ptr,
                                std::vector<int> dims, std::vector<Operand>
index);
// 从内存地址加载值到目标寄存器
void IRGenLoad(LLVMBlock B, BasicInstruction::LLVMType type, int result_reg,
Operand ptr);
// 将寄存器值存储到指定内存地址
void IRGenStore(LLVMBlock B, BasicInstruction::LLVMType type, int value_reg,
Operand ptr);
// 将立即数存储到指定内存地址
void IRGenStore(LLVMBlock B, BasicInstruction::LLVMType type, Operand value,
Operand ptr);
// 调用函数（有返回值、有参数）
void IRGenCall(LLVMBlock B, BasicInstruction::LLVMType type, int result_reg,
               std::vector<std::pair<enum BasicInstruction::LLVMType, Operand>>
args, std::string name);
// 调用函数（无返回值、有参数）
void IRGenCallVoid(LLVMBlock B, BasicInstruction::LLVMType type,
                  std::vector<std::pair<enum BasicInstruction::LLVMType,
Operand>> args, std::string name);
// 调用函数（有返回值、无参数）
void IRGenCallNoArgs(LLVMBlock B, BasicInstruction::LLVMType type, int
result_reg, std::string name);
// 调用函数（无返回值、无参数）
void IRGenCallVoidNoArgs(LLVMBlock B, BasicInstruction::LLVMType type,
std::string name);
// 返回寄存器值
void IRGenRetReg(LLVMBlock B, BasicInstruction::LLVMType type, int reg);
// 返回立即数（整数类型）
void IRGenRetImmInt(LLVMBlock B, BasicInstruction::LLVMType type, int val);
// 返回立即数（浮点类型）
void IRGenRetImmFloat(LLVMBlock B, BasicInstruction::LLVMType type, float val);
// 返回 void 类型
void IRGenRetVoid(LLVMBlock B);
// 无条件跳转到目标基本块
void IRGenBRUnCond(LLVMBlock B, int dst_label);
// 条件跳转，根据条件寄存器决定跳转到 `true_label` 或 `false_label`
void IRGenBrCond(LLVMBlock B, int cond_reg, int true_label, int false_label);
// 在栈上为指定类型分配空间
void IRGenAlloca(LLVMBlock B, BasicInstruction::LLVMType type, int reg);
// 在栈上为数组分配空间
void IRGenAllocaArray(LLVMBlock B, BasicInstruction::LLVMType type, int reg,
std::vector<int> dims);
// 根据索引生成数组元素地址
void IRGenGetElementptr(LLVMBlock B, BasicInstruction::LLVMType type, int
result_reg, Operand ptr, std::vector<int> dims, std::vector<Operand> index);
// 根据寄存器编号创建一个新的寄存器操作数
RegOperand *GetNewRegOperand(int RegNo);

```

utils/Instruction_out.cc

输出llvm的指令

```
std::ostream &operator<<(std::ostream &s, BasicInstruction::LLVMType type) {
    switch (type) {
        case BasicInstruction::I32:
            s << "i32";
            break;
        case BasicInstruction::FLOAT32:
            s << "float";
            break;
        case BasicInstruction::PTR:
            s << "ptr";
            break;
        case BasicInstruction::VOID:
            s << "void";
            break;
        case BasicInstruction::I8:
            s << "i8";
            break;
        case BasicInstruction::I1:
            s << "i1";
            break;
        case BasicInstruction::I64:
            s << "i64";
            break;
        case BasicInstruction::DOUBLE:
            s << "double";
            break;
    }
    return s;
}
```

这个就是，根据不同的BasicInstruction中的类，输出不同的字符，相当于重载

定义函数的指令：

```
void FunctionDefineInstruction::PrintIR(std::ostream &s) {
    // define void @FunctionName
    s << "define " << return_type << " @" << Func_name;

    // Print Parameter List
    // example:(i32 %0,f32 %1)
    s << "(";
    for (uint32_t i = 0; i < formals.size(); ++i) {
        s << formals[i] << " " << formals_reg[i];
        if (i + 1 < formals.size()) {
            s << ",";
        }
    }
    s << ")\n";
}
```

```
}
```

对应的输出情况: `define void @FunctionName (i32 %0,f32 %1)`

基本块的输出:

```
void BasicBlock::printIR(std::ostream &s) {
    s << "L" << block_id << ": "; s << comment << "\n";

    for (Instruction ins : Instruction_list) {
        s << "    ";
        ins->PrintIR(s);    // Auto "\n" In Instruction::printIR()
    }
}
```

输出L+块的编号, 然后分别做中间代码生成

最顶层:

```
void LLVMIR::printIR(std::ostream &s) {
    // output lib func decl
    for (Instruction lib_func_decl : function_declare) {
        lib_func_decl->PrintIR(s);
    }

    // output global
    for (Instruction global_decl_ins : global_def) {
        global_decl_ins->PrintIR(s);
    }

    // output Functions
    for (auto Func_Block_item : function_block_map) {    //<function,<id,block> >
        FuncDefInstruction f = Func_Block_item.first;
        current_CFG = llvm_cfg[f];
        // output function Syntax
        f->PrintIR(s);

        // output Blocks in functions
        s << "{\n";
        for (auto block : Func_Block_item.second) {
            block.second->printIR(s);
        }
        s << "}\n";
    }
}
```

先打印库函数, 然后打印全局定义, 然后再打印其他的函数

ir_gen/semant.h

```
class SemantTable {
public:
    // 如果你对已有的成员变量不满意，可以删除它们并添加自己想要的

    // key的含义是函数名，FuncDef为该函数名对应的语法树
    // 可以用于函数的错误检查(例如函数调用实参与形参是否匹配)
    std::map<Symbol, FuncDef> FunctionTable;

    // 存储局部变量名与该局部变量的信息(初始值，类型等)
    SymbolTable symbol_table;

    // key的含义是全局变量名，value的含义是该全局变量的信息(初始值，类型等)
    // 我们可以在semant阶段就在llvmIR中生成全局变量定义指令
    std::map<Symbol, VarAttribute> GlobalTable;

    std::map<Symbol, int> GlobalStrTable; // 添加全局变量的存储表，没多一个全局变量的字符串，我们就加一，为其分配唯一的编号

    SemantTable() {
        // 添加库函数到函数表中，我们不要求实现putf这个库函数
        // 可以非常方便地检查对库函数的调用是否符合定义
        Symbol getint = id_table.add_id("getint");
        FunctionTable[getint] = new __FuncDef(Type::INT, getint, new
std::vector<FuncFParam>{}, nullptr);

        Symbol getch = id_table.add_id("getch");
        FunctionTable[getch] = new __FuncDef(Type::INT, getch, new
std::vector<FuncFParam>{}, nullptr);

        Symbol getfloat = id_table.add_id("getfloat");
        FunctionTable[getfloat] = new __FuncDef(Type::FLOAT, getfloat, new
std::vector<FuncFParam>{}, nullptr);

        Symbol getarray = id_table.add_id("getarray");
        FunctionTable[getarray] = new __FuncDef(
Type::INT, getarray,
new std::vector<FuncFParam>{new __FuncFParam(Type::INT, new
std::vector<Expression>(1, nullptr))}, nullptr);

        Symbol getfarray = id_table.add_id("getfarray");
        FunctionTable[getfarray] = new __FuncDef(
Type::INT, getfarray,
new std::vector<FuncFParam>{new __FuncFParam(Type::FLOAT, new
std::vector<Expression>(1, nullptr))}, nullptr);

        Symbol putint = id_table.add_id("putint");
        FunctionTable[putint] =
new __FuncDef(Type::VOID, putint, new std::vector<FuncFParam>{new
__FuncFParam(Type::INT)}, nullptr);

        Symbol putch = id_table.add_id("putch");
        FunctionTable[putch] =
```

```

    new __FuncDef(Type::VOID, putch, new std::vector<FuncFParam>{new
__FuncFParam(Type::INT)}, nullptr);

    Symbol putfloat = id_table.add_id("putfloat");
    FunctionTable[putfloat] =
    new __FuncDef(Type::VOID, putfloat, new std::vector<FuncFParam>{new
__FuncFParam(Type::FLOAT)}, nullptr);

    Symbol putarray = id_table.add_id("putarray");
    FunctionTable[putarray] =
    new __FuncDef(Type::VOID, putarray,
        new std::vector<FuncFParam>{new __FuncFParam(Type::INT),
        new __FuncFParam(Type::INT, new
std::vector<Expression>(1, nullptr))},
        nullptr);

    Symbol putfarray = id_table.add_id("putfarray");
    FunctionTable[putfarray] = new __FuncDef(
    Type::VOID, putfarray,
    new std::vector<FuncFParam>{new __FuncFParam(Type::INT),
        new __FuncFParam(Type::FLOAT, new
std::vector<Expression>(1, nullptr))},
    nullptr);

    Symbol starttime = id_table.add_id("_sysy_starttime");
    FunctionTable[starttime] =
    new __FuncDef(Type::VOID, starttime, new std::vector<FuncFParam>{new
__FuncFParam(Type::INT)}, nullptr);

    Symbol stoptime = id_table.add_id("_sysy_stoptime");
    FunctionTable[stoptime] =
    new __FuncDef(Type::VOID, stoptime, new std::vector<FuncFParam>{new
__FuncFParam(Type::INT)}, nullptr);

    // 这里你可能还需要对这些语法树上的节点进行类型的标注，进而检查对库函数的调用是否符合形
参
    // 即正确填写NodeAttribute或者使用其他方法来完成检查
    for (auto entry : FunctionTable) {
        entry.second->TypeCheck();
    }
}
};

```

ir_gen/IRgen.h

```
class IRgenTable {
public:
    // 如果你无从下手,推荐先阅读LLVMIR类的printIR函数,了解我们是如何输出中间代码的
    // 然后你可以尝试往和输出相关联的变量中随便添加一些函数定义指令,新建一些基本块或添加几条指令看看输出是怎么变化的
    // 弄懂LLVMIR类是如何存储中间代码后,剩余的就是理解中间代码生成算法了

    // TODO():添加更多你需要的成员变量和成员函数
    SymbolRegTable symbol_reg_table;//symbol->int(查看一个符号的寄存器类型)
    std::map<int, VarAttribute> Reg_VarA_Table;//如果symbol->varA, 容易出现作用域、重命名的问题
    operand current_putin_ptr = nullptr;
    std::map<int, int> Formal_Array_Table;//映射表,键是寄存器编号,值是数组的元信息
    IRgenTable() {}
};
```

- symbol_reg_table: symbol->int(查看一个符号的寄存器类型)
- Reg_VarA_Table: 如果symbol->varA, 容易出现作用域、重命名的问题, 所以是int->varA
- Formal_Array_Table: 映射表, 键是寄存器编号, 值是数组的元信息

ir_gen/semant.cc

写了注释

ir_gen/IRgen.cc

写了注释

Questions

1.请以 `int a[5][4][3] = {{{2,3},6,7},7,8,11};` 为例来说明你是如何处理全局变量数组的初始化语法的。

(这部分是在写Semant时候完成的,参考了助教的代码,在写IRGEN的时候,局部的里面是按照C的写的,过了样例就没再动)

```
//调用流程为CompUnit_Decl->Solve_Init_Val->Semant_Array_Init
//(这里var有一个初始化的列表,如果是数组里面就会有好几个数,不然就只会会有一个数也就是[0]处)
```

global在semant.cc里Semant_Array_Init的完成:

```
//递归地初始化多维数组,根据提供的初始化值init和变量属性val,将数据存储到val对应的位置,并处理不同的数据类型或嵌套的初始化列表。
/*
参数说明:
init:表示当前的初始化值,可以是一个表达式(单一值)或嵌套的初始化列表。
val:变量的属性,在type.h中有定义
```

```

begpos:需要初始化的数组的范围, 开始地址
endpos:需要初始化的数组的范围, 结束地址
dimsIdx:表示当前正在处理的维度索引, 用于多维数组递归初始化
*/
void Semant_Array_Init(InitVal init, VarAttribute &val, int begPos, int endPos,
int dimsIdx) {
    int pos = begPos;//首先, 给pos赋值为开始地址, 表示初始化开始

    if (init->IsExp()) {// 如果是表达式expression类型的初始化值
        if (init->attribute.T.type == Type::VOID) {//如果初始化表达式的类型为void的
            话, 相当于不合法, 抛出error
            error_msgs.push_back("Expression cannot be void in initval in line "
+ std::to_string(init->GetLineNumber()) + "\n");
        }
        if (pos <= endPos) {//如果不为空的话, 就遍历我们的pos直到末尾, 进行处理。
            switch (val.type) {//匹配val的类型, 有int和float, 其他都是error
                case Type::INT://如果是int类型的话, 给val变量的pos位置的值赋值, 如果为
float类型就进行转换, 如果是int类型就直接赋值
                    val.IntInitVals[pos] = init->attribute.T.type == Type::FLOAT
                        ? static_cast<int>(init-
>attribute.V.val.FloatVal)
                        : init->attribute.V.val.IntVal;
                    break;

                case Type::FLOAT://float也是一样的道理
                    val.FloatInitVals[pos] = init->attribute.T.type == Type::INT
                        ? static_cast<float>(init-
>attribute.V.val.IntVal)
                        : init-
>attribute.V.val.FloatVal;
                    break;

                default://如果不是这两种类型, 就报错
                    error_msgs.push_back("Unsupported type in initval in line " +
std::to_string(init->GetLineNumber()) + "\n");
            }
            pos++;//继续往后进行处理
        }
    } else {
        // 如果不是表达式, 说明需要递归。进行处理嵌套的初始化
        auto initList = init->GetList();//首先获取到init的列表
        int blockSize = 1;//初始化当前块的size大小为1
        for (int i = dimsIdx + 1; i < val.dims.size(); ++i) {//这一块完成的是, 根据我
们给定的数组维度计算出我们应该需要的块大小
            blockSize *= val.dims[i];
        }

        for (auto &subInit : *initList) {
            if (pos > endPos)
            {
                error_msgs.push_back("too many values that more than dims " +
std::to_string(init->GetLineNumber()) + "\n");
                break;
            }
            Semant_Array_Init(subInit, val, pos, pos + blockSize - 1, dimsIdx +
1);//递归调用函数, 来处理每一个块
        }
    }
}

```



```

        pos += blockSize;//处理完之后，移动到下一个块的起始位置进行处理
    }
}
}
//a[5][4][3] 如果不是表达式，就会一直递归，如果是表达式，就会给当前的位置赋值

```

`int a[5][4][3] = {{2,3},6,7},7,8,11};` 最终应该生成（看 `recursive_print` 处，如果不是全零就会一直往下递归打印）（无缩进和回车）：

```

//a[5][4][3]
//60 0-59
// 0-11      12-23 ..
// /  |      /  |
//0-3 4-7 ... 12-15 16-19      56-59
//0-0 1-1 2-2 3-3
// [i32 1, i32 2 ]
@a = global [5 x [4 x [3 x i32]]]
[
    [4 x [3 x i32]]
    [
        [3 x i32] [i32 2,i32 3,i32 0],
        [3 x i32] [i32 6,i32 0,i32 0],
        [3 x i32] [i32 7,i32 0,i32 0],
        [3 x i32] [i32 0,i32 0,i32 0]
    ],
    [4 x [3 x i32]]
    [
        [3 x i32] [i32 7,i32 0,i32 0],
        [3 x i32] [i32 0,i32 0,i32 0],
        [3 x i32] [i32 0,i32 0,i32 0],
        [3 x i32] [i32 0,i32 0,i32 0]
    ],
    [4 x [3 x i32]]
    [
        [3 x i32] [i32 8,i32 0,i32 0],
        [3 x i32] [i32 0,i32 0,i32 0],
        [3 x i32] [i32 0,i32 0,i32 0],
        [3 x i32] [i32 0,i32 0,i32 0]],
        [4 x [3 x i32]
    ]
    [
        [3 x i32] [i32 11,i32 0,i32 0],
        [3 x i32] [i32 0,i32 0,i32 0],
        [3 x i32] [i32 0,i32 0,i32 0],
        [3 x i32] [i32 0,i32 0,i32 0]
    ],
    [4 x [3 x i32]]
    [
        [3 x i32] [i32 0,i32 0,i32 0],
        [3 x i32] [i32 0,i32 0,i32 0],
        [3 x i32] [i32 0,i32 0,i32 0],
        [3 x i32] [i32 0,i32 0,i32 0]
    ]
]

```

```
]
]
```

也就是说，根据括号的层数进行逐级的调用和输出，在out文件里有对应的输出语法

2. `int a[5][4][3] = {{{2,3},6,7,5,4,3,2,11,2,4,5},{7,8,11}};` 该初始化是否合法，请说明理由。

不合法，因为初始化{2,3},6,7,5,4,3,2,11,2,4,5时，会将{2,3}识别为一个更低纬度的整个块的初始化，也就是 `a[5][4]` 的初始化，变为{2,3,0}，最后变为2 3 0 6 7 5 4 3 2 11 2 4 5，一共13个，超出了12个的限制，所以不合法

(对应到irgen中的数组初始化的操作)

3.请描述框架中的符号表是如何设计的，使用到了什么数据结构。如果你没使用框架中的符号表，你是如何设计一个可以支持作用域的符号表的。

```
class SymbolTable {
private:
    // 当前作用域
    int current_scope = -1;
    std::vector<std::map<Symbol, VarAttribute>> symbol_table;
}
```

其中

```
class Entry {
private:
    std::string name;
};
typedef Entry *Symbol;

class VarAttribute {
public:
    Type::ty type;
    bool ConstTag = 0;
    std::vector<int> dims{};    // 存储数组类型的相关信息
    // 对于数组的初始化值，我们将高维数组看作一维后再存储 eg. ([3 x [4 x i32]] => [12 x i32])
    std::vector<int> IntInitVals{};
    std::vector<float> FloatInitVals{};
};
```

4.请说明你是如何处理int/float/bool类型的隐式转换的，简单说明 `int a = 5; int b = a + 3.5 + !a` 的语法树结构，并说明各节点上的类型。

处理方式：

int、float和bool

遇到float先往float转，避免丢失精度

没有float的情况下，遇到int就往int转

只有全是bool的情况，才是输出bool值

总体流程为：

根据左值，先处理a和3.5，将a转化为float类型

然后加起来，为float类型

然后将la转化为int，再转为float，加起来 为float类型

然后根据assign_stmt

完成类型转换，变为int型

见下

```
//处理二元运算，暴力枚举，通过枚举左右节点的类型，去进行不同的转换
void AddExp_plus::codeIR(){
    LLVMBlock B = llvmlr.GetBlock(function_now, curr_block_label); //首先定位到当前的block块

    //如果是两个int类型的话，就直接对子节点进行ir生成，然后分配寄存器的编号
    if(addexp->attribute.T.type == Type::INT && mulexp->attribute.T.type == Type::INT){
        addexp->codeIR();
        int reg1 = max_reg;
        mulexp->codeIR();
        int reg2 = max_reg;
        IRgenArithmeticI32(B, BasicInstruction::LLVMIROpcode::ADD, reg1, reg2, ++max_reg);
    }

    else if(addexp->attribute.T.type == Type::INT && mulexp->attribute.T.type == Type::FLOAT){
        addexp->codeIR();
        int reg1 = max_reg;
        mulexp->codeIR();
        int reg2 = max_reg;
        IRgenSitoFP(B, reg1, ++max_reg); // 将int型转换为float型
        reg1 = max_reg;
        IRgenArithmeticF32(B, BasicInstruction::LLVMIROpcode::FADD, reg1, reg2, ++max_reg);
    }

    else if(addexp->attribute.T.type == Type::INT && mulexp->attribute.T.type == Type::BOOL){
        addexp->codeIR();
        int reg1 = max_reg;
        mulexp->codeIR();
    }
```

```

        int reg2 = max_reg;
        IRGenZextI1toI32(B, reg2, ++max_reg);    // bool转int
        reg2 = max_reg;
        IRGenArithmeticI32(B, BasicInstruction::LLVMIROpcode::ADD, reg1, reg2,
++max_reg);

    }

    else if(addexp->attribute.T.type == Type::FLOAT && mulexp->attribute.T.type
== Type::INT){
        addexp->codeIR();
        int reg1 = max_reg;
        mulexp->codeIR();
        int reg2 = max_reg;
        IRGenSitofp(B, reg2, ++max_reg);    // int转float
        reg2 = max_reg;
        IRGenArithmeticF32(B, BasicInstruction::LLVMIROpcode::FADD, reg1, reg2,
++max_reg);

    }

    else if(addexp->attribute.T.type == Type::FLOAT && mulexp->attribute.T.type
== Type::FLOAT){
        addexp->codeIR();
        int reg1 = max_reg;
        mulexp->codeIR();
        int reg2 = max_reg;
        IRGenArithmeticF32(B, BasicInstruction::LLVMIROpcode::FADD, reg1, reg2,
++max_reg);

    }

    else if(addexp->attribute.T.type == Type::FLOAT && mulexp->attribute.T.type
== Type::BOOL){
        addexp->codeIR();
        int reg1 = max_reg;
        mulexp->codeIR();
        int reg2 = max_reg;
        reg2 = max_reg;
        IRGenSitofp(B, reg2, ++max_reg);    // int -> float
        reg2 = max_reg;
        IRGenArithmeticF32(B, BasicInstruction::LLVMIROpcode::FADD, reg1, reg2,
++max_reg);

    }

    else if(addexp->attribute.T.type == Type::BOOL && mulexp->attribute.T.type ==
Type::INT){
        addexp->codeIR();
        int reg1 = max_reg;
        mulexp->codeIR();
        int reg2 = max_reg;
        IRGenZextI1toI32(B, reg1, ++max_reg);    // bool -> int
        reg1 = max_reg;
        IRGenArithmeticI32(B, BasicInstruction::LLVMIROpcode::ADD, reg1, reg2,
++max_reg);

```

```

    }

    else if(addexp->attribute.T.type == Type::BOOL && mulexp->attribute.T.type ==
Type::FLOAT){
        addexp->codeIR();
        int reg1 = max_reg;
        mulexp->codeIR();
        int reg2 = max_reg;
        IRgenZextI1toI32(B, reg1, ++max_reg);    // bool -> int
        reg1 = max_reg;

        IRgenSitoFP(B, reg1, ++max_reg);    // int -> float
        reg1 = max_reg;
        IRgenArithmeticF32(B, BasicInstruction::LLVMIROpcode::FADD, reg1, reg2,
++max_reg);

    }

    else if(addexp->attribute.T.type == Type::BOOL && mulexp->attribute.T.type ==
Type::BOOL){
        addexp->codeIR();
        int reg1 = max_reg;
        mulexp->codeIR();
        int reg2 = max_reg;
        IRgenZextI1toI32(B, reg1, ++max_reg);    // bool -> int
        reg1 = max_reg;

        IRgenZextI1toI32(B, reg2, ++max_reg);    // bool -> int
        reg2 = max_reg;
        IRgenArithmeticI32(B, BasicInstruction::LLVMIROpcode::ADD, reg1, reg2,
++max_reg);

    }
    else{
        assert(false); //停止
    }
}
}

```

例子:

输入:

```

int main(){
    int a =5;
    int b = a + 3.5 + !a;
    return b;
}

```

语法树结构:

AddExp:=AddExp '+' MulExp:=AddExp '+' UnaryExp:=AddExp '+' '!'UnaryExp

```

VarDecls  Type: Int
  VarDef  name:a  scope:-1
    init:
      VarInitVal_exp
        Intconst  val:5  Type: Void
VarDecls  Type: Int
  VarDef  name:b  scope:-1
    init:
      VarInitVal_exp
        AddExp_plus: (+)  Type: Void
          AddExp_plus: (+)  Type: Void
            Lval  Type: Void      name:a  scope:-1
              Floatconst  val:3.5  Type: Void
            UnaryExp_not: (!)  Type: Void
              Lval  Type: Void      name:a  scope:-1

```

输出llvm:

```

L0:  ;
    %r2 = alloca i32
    %r0 = alloca i32
    br label %L1
L1:  ;
    %r1 = add i32 5,0
    store i32 %r1, ptr %r0
    %r3 = load i32, ptr %r0
    %r4 = fadd float 0x400c000000000000,0x0
    %r5 = sitofp i32 %r3 to float
    %r6 = fadd float %r5,%r4
    %r7 = load i32, ptr %r0
    %r8 = icmp eq i32 %r7,0
    %r9 = zext i1 %r8 to i32
    %r10 = sitofp i32 %r9 to float
    %r11 = fadd float %r6,%r10
    %r12 = fptosi float %r11 to i32
    store i32 %r12, ptr %r2
    ret i32 0

```

分析一下过程:

int a =5; 对应了llvm的 %r1 = add i32 5,0; store i32 %r1, ptr %r0

%r3存储了int型的值

%r4存储着float值, 用了FADD

然后int型就需要转化为float型, 用了

```

%r5 = sitofp i32 %r3 to float
%r6 = fadd float %r5,%r4

```

然后是bool的转换

```

    %r7 = load i32, ptr %r0
    %r8 = icmp eq i32 %r7, 0
    %r9 = zext i1 %r8 to i32
    %r10 = sitofp i32 %r9 to float
    %r11 = fadd float %r6, %r10

```

这样就加完了，但是现在是float类型

```

    %r12 = fptosi float %r11 to i32
    store i32 %r12, ptr %r2
    ret i32 0

```

转int，然后store，ret即可

5.请说明对于局部变量 `int a[5][4][3] = {{{2,3},1}};` 应当如何生成 llvm-ir。

内存分配：

首先需要**变量声明**，分配了3*4*5个i32空间

```

%a = alloca [5 x [4 x [3 x i32]]], align 16

```

数组初始化：

需要对该数组做初始化操作

在 VarDecl 或 ConstDecl 中先申请内存，设为全0，再对特别值赋值。

通过代码

```

CallInstruction *memsetCall = new CallInstruction(BasicInstruction::VOID,
    nullptr, std::string("llvm.memset.p0.i32"));
memsetCall->push_back_Parameter(BasicInstruction::PTR,
    GetNewRegOperand(alloca_reg));    // array address
memsetCall->push_back_Parameter(BasicInstruction::I8, new ImmI32Operand(0));
    // value to set (0)
memsetCall->push_back_Parameter(BasicInstruction::I32, new ImmI32Operand(array_sz
    * sizeof(int))); // size in bytes
memsetCall->push_back_Parameter(BasicInstruction::I1, new ImmI32Operand(0));
    // alignment
llvmIR.function_block_map[function_now][curr_block_label]->InsertInstruction(1,
    memsetCall);

```

将整个内存空间都填满0

然后开始填充我们声明的数组位置的值

通过 `Init_Array(B_now, type_decl, max_reg, val.dims, init, 0, 0, array_sz, array_sz);` 来填充我们的数组

这里的逻辑是，如果迭代的内容不是一个列表，就往后正常线性移动，否则就要移动一整个下一个数组维度的。

这里如果为迭代{}的式子，应该在整维度倍数处，如 `int a[5][4][3] = {{1,2,3,{4,5}},4}` 可以，如 `int a[5][4][3] = {{1,2,{4,5}},4}` 不可以

生成的对应的llvm:

```
define i32 @main()
{
L0:  ;
    %r0 = alloca [5 x [4 x [3 x i32]]]
    br label %L1
L1:  ;
    call void @llvm.memset.p0.i32(ptr %r0,i8 0,i32 240,i1 0)
    %r1 = add i32 2,0
    %r2 = getelementptr [5 x [4 x [3 x i32]]], ptr %r0
    store i32 %r1, ptr %r2
    %r3 = add i32 3,0
    %r4 = getelementptr [5 x [4 x [3 x i32]]], ptr %r0, i32 0, i32 0, i32 0, i32
1
    ;0*60+0*12+0*3+1*1
    store i32 %r3, ptr %r4
    %r5 = add i32 1,0
    %r6 = getelementptr [5 x [4 x [3 x i32]]], ptr %r0, i32 0, i32 0, i32 1
    ;0*60+0*12+1*3
    store i32 %r5, ptr %r6
    %r7 = add i32 0,0
    ret i32 %r7
}
```

```
int a[5][4][3] = {{{2,3},1}};
a[0]:
[
    [2,3],
    [1],
    [],
    []
]
a[1]:
[
    [0],
    [],
    [],
    []
]
...
```

基本上就是，找到对应的数组的指针地址，然后使用store指令进行赋值即可

具体llvm:

6.请说明编译器是如何处理除数为0的情况的。

调用MulExp_div::TypeCheck()

```
void MulExp_div::TypeCheck() {
    mul_exp->TypeCheck();
    unary_exp->TypeCheck();

    attribute = Perform_Binary_Operation(mul_exp->attribute, unary_exp->attribute,
NodeAttribute::DIV);
}
```

调用Perform_Binary_Operation, 参数为DIV

```
NodeAttribute Perform_Binary_Operation(NodeAttribute a, NodeAttribute b,
NodeAttribute::opcode opcode) {
    NodeAttribute result;

    if (a.T.type == Type::INT && b.T.type == Type::INT) {
        result = Semant_Int_Int(a, b, opcode);
    }

    else if (a.T.type == Type::INT && b.T.type == Type::FLOAT) {
        result = Semant_Int_Float(a, b, opcode);
    }

    else if (a.T.type == Type::INT && b.T.type == Type::BOOL) {
        result = Semant_Int_Bool(a, b, opcode);
    }

    else if (a.T.type == Type::FLOAT && b.T.type == Type::INT) {
        result = Semant_Float_Int(a, b, opcode);
    }

    else if (a.T.type == Type::FLOAT && b.T.type == Type::FLOAT) {
        result = Semant_Float_Float(a, b, opcode);
    }

    else if (a.T.type == Type::FLOAT && b.T.type == Type::BOOL) {
        result = Semant_Float_Bool(a, b, opcode);
    }

    else if (a.T.type == Type::BOOL && b.T.type == Type::INT) {
        result = Semant_Bool_Int(a, b, opcode);
    }

    else if (a.T.type == Type::BOOL && b.T.type == Type::FLOAT) {
        result = Semant_Bool_Float(a, b, opcode);
    }

    else if (a.T.type == Type::BOOL && b.T.type == Type::BOOL) {
        result = Semant_Bool_Bool(a, b, opcode);
    }

    else {
```

```

        result = Semant_ERROR(a, b, opcode);
    }

    return result;
}

```

根据左右类型确定调用函数，以int和int为例：

```

NodeAttribute Semant_Int_Int(NodeAttribute a,NodeAttribute
b,NodeAttribute::opcode opcode){
    return Calculate_Binary_Int(opcode , a , b);//如果是两个int型运算的话，就不需要进行转换
}

```

调用Calculate_Binary_Int

```

NodeAttribute Calculate_Binary_Int(NodeAttribute::opcode op, NodeAttribute a,
NodeAttribute b) {
    switch (op) {
        case NodeAttribute::ADD:
            return Binary_Add_Int(a, b);
        case NodeAttribute::SUB:
            return Binary_Sub_Int(a, b);
        case NodeAttribute::MUL:
            return Binary_Mul_Int(a, b);
        case NodeAttribute::DIV:
            return Binary_Div_Int(a, b);
        case NodeAttribute::MOD:
            return Binary_Mod_Int(a, b);
        case NodeAttribute::GEQ:
            return Binary_Geq_Int(a, b);
        case NodeAttribute::GT:
            return Binary_Gt_Int(a, b);
        case NodeAttribute::LEQ:
            return Binary_Leq_Int(a, b);
        case NodeAttribute::LT:
            return Binary_Lt_Int(a, b);
        case NodeAttribute::EQ:
            return Binary_Eq_Int(a, b);
        case NodeAttribute::NE:
            return Binary_Ne_Int(a, b);
        case NodeAttribute::OR:
            return Binary_Or_Int(a, b);
        case NodeAttribute::AND:
            return Binary_And_Int(a, b);
        default:
            NodeAttribute error_result;
            error_result.T.type = Type::VOID;
            error_result.V.ConstTag = false;
            error_msgs.push_back("the opcode in the Binary Int Calculate is
wrong!!\n");
            return error_result;
    }
}

```

```
}
```

调用Binary_Div_Int

```
//处理双目的除法运算，考虑除以0的情况
NodeAttribute Binary_Div_Int(NodeAttribute a, NodeAttribute b) {
    NodeAttribute div_result;
    div_result.T.type = Type::INT;

    //判断父节点的类型，如果两个子节点都是常量的话就给上面的父节点赋值为常量
    if(a.V.ConstTag == true && b.V.ConstTag == true){
        div_result.V.ConstTag = true;
    }
    else{
        div_result.V.ConstTag = false;
    }
    //如果被除数是0的话，就抛出除0错误
    if((b.V.ConstTag==true && b.V.val.IntVal == 0)){
        div_result.T.type = Type::VOID;
        div_result.V.ConstTag = false;

        error_msgs.push_back("cannot div zero!!!Error in line " +
std::to_string(b.line_number)
        + " a.Attr:"+ a.GetAttributeInfo() + " b.Attr:"+ b.GetAttributeInfo()+
"\n");
    }

    if(div_result.V.ConstTag == true){
        div_result.V.val.IntVal = a.V.val.IntVal / b.V.val.IntVal;
    }
    return div_result;
}
```

除零处理：判断 `b.V.ConstTag==true && b.V.val.IntVal == 0`

如果符合的话，说明被除数为0，我们抛出错误，然后把结果节点的type赋值为VOID，把ConstTag赋值为false

7.说明你是如何在语义分析步骤中检查SysY库函数调用是否合法的。

检查对应的参数个数和类型

个数就是去检查形参和实参的个数是否对应

类型的话，由于sysy库函数基本都完成了隐式转换，所以可以不考虑类型的合法性

```
void FuncRParams::TypeCheck(){

}

void Func_call::TypeCheck() {
    int funcr_params_number = 0; //表示实参的数量
```

```

    if (funcr_params != nullptr) { //说明有参数, 是实参
        auto params = static_cast<FuncRParams *>(funcr_params)->params; //首先, 提取
参数列表
        funcr_params_number = params->size();
        for (auto param : *params) {
            param->TypeCheck(); //遍历检查
            if (param->attribute.T.type == Type::ty::VOID) { //实参的类型不能为空!
                error_msgs.push_back("FuncRParam is void in line " +
std::to_string(line_number) + "\n");
            }
        }
    }
    //检查函数是否已经定义
    auto func_it = semant_table.FunctionTable.find(name); //首先在我们的语义分析函数表
中查找该name
    if (func_it == semant_table.FunctionTable.end()) { //如果没有的话, 说明未定义,
error
        error_msgs.push_back("Function is undefined in line " +
std::to_string(line_number) + "\n");
        return;
    }

    FuncDef funcdef = func_it->second; //形参
    if (funcdef->formals->size() != funcr_params_number) { //如果形参和实参的数量不对
应的话, 报错
        error_msgs.push_back("Function FuncFParams and FuncRParams are not
matched in line " +
                                std::to_string(line_number) + "\n");
    }

    attribute.T.type = funcdef->return_type; //提取返回值类型
    attribute.V.ConstTag = false; //函数调用的结果基本不为常量
}

```

8.说明你是如何实现短路求值的控制流翻译的。

与: 处理左边的时候, 如果false 就全false

或: 左边true就true

```

//处理二元短路逻辑 与
void LAndExp_and::codeIR() {
    assert((true_label != -1) && (false_label != -1)); //确保 true_label 和
false_label 已经被设置
    //int and_begin_label = curr_block_label; //定义初始的label数
    int start_label = curr_block_label; //start_label表示当前的label

    LLVMBlock t = llvmIR.NewBlock(function_now, max_block_label); //由于要跳转, new
一个块
    int left_label = t->block_id; //然后把label赋值给left_label

    landex->true_label = left_label; //表示左操作数为真的情况下需要跳转到的标签
    landex->false_label = this->false_label; //如果false的话就直接跳转到整体的假分支

```

```

    landexp->codeIR(); //对左值进行中间代码的生成
    LLVMBlock B1 = llvmIR.GetBlock(function_now, curr_block_label); //获取当前的块
    IRgenTypeConverse(B1, landexp->attribute.T.type, Type::BOOL, max_reg); //为了符合
    条件跳转，我们进行类型转换，转换为bool型
    IRgenBrCond(B1, max_reg, landexp->true_label, landexp->>false_label); //设置条件
    跳转指令

    curr_block_label = left_label;
    //接下来开始判断右值的true和false
    eqexp->true_label = this->true_label;
    eqexp->>false_label = this->>false_label;

    eqexp->codeIR(); //对右值进行中间代码生成

    LLVMBlock B2 = llvmIR.GetBlock(function_now, curr_block_label);

    Type::ty temp = eqexp->attribute.T.type;
    IRgenTypeConverse(B2, temp, Type::BOOL, max_reg);

}

// 处理二元短路逻辑 或
void LORExp_or::codeIR() { //跟前面的整体逻辑是一样的，只不过变成了，只要有true的就赋值为
true
    assert((true_label != -1) && (false_label != -1));
    //int and_begin_label = curr_block_label; //定义初始的label数
    LLVMBlock B0 = llvmIR.NewBlock(function_now, max_block_label); //新开一个块
    lorexp->true_label = this->true_label; //左边只要true了就全true了
    lorexp->>false_label = B0->block_id; //false的话就赋值为右边的块
    lorexp->codeIR(); //对左值进行中间代码的生成

    LLVMBlock B1 = llvmIR.GetBlock(function_now, curr_block_label);
    IRgenTypeConverse(B1, lorexp->attribute.T.type, Type::BOOL, max_reg);
    IRgenBrCond(B1, max_reg, this->true_label, B0->block_id);

    curr_block_label = B0->block_id;
    //接下来开始判断右值的true和false
    landexp->true_label = this->true_label;
    landexp->>false_label = this->>false_label;
    landexp->codeIR(); //做右值的中间代码的生成

    LLVMBlock B2 = llvmIR.GetBlock(function_now, curr_block_label);

    Type::ty temp = landexp->attribute.T.type;
    IRgenTypeConverse(B2, temp, Type::BOOL, max_reg);

}

```

分为与和或的逻辑

在 `&&` 操作中，如果左侧操作数为 `false`，则整个表达式结果为 `false`，无需评估右侧操作数。
 在 `||` 操作中，如果左侧操作数为 `true`，则整个表达式结果为 `true`，无需评估右侧操作数。

```
IRgen.cc  SysY_tree.h  Instruction.cc  parser_test.sy M  sen ...  example.out.ll M
parser_test.sy
> cond-> Aa ab_* 无结果  ↑ ↓ ≡ ×
1
2 int main(){
3     int a1=5;
4     int a2=6;
5     int b=1;
6     if(a1 && a2)
7     ...{
8     ... b=2;
9     ...}
10    return 0;
11 }
12
13
example.out.ll
24 br label %L1
25 L1: ;
26 %r1 = add i32 5,0
27 store i32 %r1, ptr %r0
28 %r3 = add i32 6,0
29 store i32 %r3, ptr %r2
30 %r5 = add i32 1,0
31 store i32 %r5, ptr %r4
32
33 %r6 = load i32, ptr %r0
34 %r7 = icmp ne i32 %r6,0
35 br i1 %r7, label %L4, label %L3
36 L2: ;if_stmt
37 %r10 = add i32 2,0
38 store i32 %r10, ptr %r4
39 br label %L3
40 L3: ;then_stmt
41 %r11 = add i32 0,0
42 ret i32 %r11
43 L4: ;
44 %r8 = load i32, ptr %r2
45 %r9 = icmp ne i32 %r8,0
46
47 br i1 %r9, label %L2, label %L3
48 }
```

相当于，给if开一个block为L2，然后给then开一个block为L3，然后左边的第一个式子a1，如果为真的话就跳转到L4，如果为假的话就直接跳转到L3

L4之后，判断右边的，如果为真的话就跳转到L2，如果为假的话就跳转到L3