编译原理研讨课 PR002 实验报告

小组成员：

杜旭蕾 2022K8009929003　王锦如 2022K8009929022

目 录

一. 实验任务

完成中间代码的生成，生成 LLVM IR。

二. 设计思路

在生成中间代码之前，需要完成语法分析，具体更改如下：

```
//消除左递归
mulExp : unaryExp ( (Star | Div | Mod) unaryExp )* ;

addExp : mulExp ( (Plus | Minus) mulExp )* ;

relExp : addExp ( (Less | Greater | LessEqual | GreaterEqual) addExp )* ;

eqExp  : relExp ( (Equal | NotEqual) relExp )* ;

lAndExp: eqExp ( AndAnd eqExp )* ;

lOrExp : lAndExp ( OrOr lAndExp )* ;
```

```
stmt :  exp [Assign exp] Semi
        ...
```

在完成语法分析对 g4 文件的修改之后，按照以下思路设计生成 IR 的代码：

1. 最基本 cact 代码生成 IR

首先针对以下代码内容设计 IR 的生成代码：

```
int main(){
     int a = 0;
     return 0;
}
```

运行实验 1 的代码，获得如下 AST：

```
(compUnit (funcDef (funcType int) main ( funcFParams ) (block { (blockItem (decl (vardecl (bType int)
(varDef a = (constInitVal (constExp (number 0)))) ;))) (blockItem (stmt return (exp (addExp (mulExp
(unaryExp (primaryExp (number 0)))))) ;)) })) <EOF>)
```

针对这个代码，我们首先设计一个简单的生成 IR 的代码。这个代码没有实现符号表和函数表，其 visit 函数只包括以上 AST 中出现的节点名。主要实现非数组变量的 vardecl 以及 stmt 模块 return 语生成 IR。运行并调试这个简单代码得到如下.ll 文件，使用 clang 将其编译为可执行代码，获取其程序退出码，值为 0，因此这个 IR 代码没有问题。

```
define i32 @main() {
entry:
  %a = alloca i32, align 4
  store i32 0, i32* %a, align 4
  ret i32 0
}
```

2. 扩展代码

逐步使用更加复杂的代码，发现当前代码运行出现问题，则为当前生成 IR 的代码逐步添加并完善其他节点的处理、加入符号表和函数表的实现、加入 6 个内置函数的声明并实现它们的运行时库。重复以上过程直到实现一个可以处理所有 cact 测试代码的程序。

三. 代码实现

本次实验代码具有如下结构：

```
cact/
|── build/
     |── build.sh          # 编译生成 compiler
     |── libcact_rt.a      # 运行时库
     |── run_code.sh # 将.ll 文件链接运行时库生成可执行文件
|── grammar/
     |── Hello.g4
|── src/
     |── io/               # 生成运行时库的文件
     |── main.cpp          # compiler 的 main 函数
     |── Analysis.cpp      # 生成 IR 的函数（主要为 visit 节点函数）
```

```
├── include/
     ├── Analysis.h  # Analysis.cpp 的头函数
     ├── SymbolTable.h    # 符号表及其相关函数实现
├── CMakeLists.txt   # 添加安装的 llvm 为 path

llvm_install/
└── clang+llvm-11.1.0-x86_64-linux-gnu-ubuntu-16.04/  # 安装的 llvm 和 clang
```

其中，本次实验生成 IR 的核心函数集中于 Analysis.cpp 文件和 SymbolTable.h 文件。

Analysis.cpp 文件中主要实现了所有直接使用 ANTLR 生成的访问器接口函数，这些接口继承自 HelloBaseVisitor。这类函数的命名规则为 "visit" + AST 节点名称，比如：visitCompUnit、visitFuncDef。每个函数对应语法树中的一个节点类型，其作用是定义访问该节点时应执行的逻辑，在语法树的遍历过程中可以实现 IR 的生成。

SymbolTable.h 文件主要实现了符号表及其相关函数的实现。

由于 Analysis.cpp 代码很长，实验报告代码实现部分主要选取这两个文件中重点部分进行分析，内容如下：

1. 符号表实现

符号表的实现全部位于 SymbolTable.h 文件中，主要实现的内容如下：

① 符号表本身的结构

符号表由作用域 scopes 和符号表内的元素 Symbol 构成。

作用域的代码如下，最外层是一个 std::vector 顺序容器存放正在使用全部的作用域，可以通过 scopes.emplace_back()操作进入新的作用域，scopes.pop_back()退出当前作用域。中间层是一个 std::unordered_map 哈希表来管理每个作用域中的符号 Symbol。最内层就是符号表内的元素 Symbol。

```
std::vector<std::unordered_map<std::string, Symbol>> scopes;
```

作用域还有一个实现的关键是在 analysis.cpp 文件中在何时进入作用域和退出作用域。我们主要遇到的问题是如果只在 visitBlock 的时候进行作用域的进入和退出会使得 if 后的语句块和 else 后的语句块在同一个作用域中，while 同理，因此需要再处理 stmt 的时候增加进入和退出作用域的操作，见 stmt 模块处理。

符号表内的元素 Symbol 具有以下结构，其主要字段内容见注释，具有一个构造函数。其中 isArrayinFunc 字段是符号是否作为函数参数出现，在 symbol 中设置这个字段主要是为了在函数声明和函数调用中可以处理指针的退化情况。

```
struct Symbol {
    std::string name;                          //符号名 name
    llvm::Type* llvmType = nullptr;            //符号的 llvm 类型
```

```
    llvm::Value* irValue = nullptr;           //符号的 llvm 值
    bool isArray = false;                     //符号是否是数组
    bool isConst = false;                     //符号是否是常量
    std::vector<int> dimensions;              //数组维度
    bool isArrayinFunc = false;               //符号是否是作为函数参数传递的数组

    Symbol() = default;
    Symbol(const std::string& name,
        llvm::Type* type,
        llvm::Value* value,
        bool isConst,
        const std::vector<int>& dimensions = {},
        bool isArrayinfunc = false)
     : name(name),
       llvmType(type),
       irValue(value),
       isConst(isConst),
       dimensions(dimensions),
       isArrayinFunc(isArrayinfunc) {}
};
```

② 符号表的插入操作

符号表的插入主要是将当前符号插入当前作用域中。主要的处理难点在于数组维度的插入。一开始我们对数组的初始化方式是自动获取数组维度不直接传入，通过 llvm 的 getNumElements 函数即可获取当前数组维度的大小，通过递归操作即可获取整个数组的维度大小。但是在处理到作为函数参数传递的数组时出现问题：因为此时需要进行指针退化，使得数组维度无法自动的获取，因此需要手动传入数组维度信息 dims（见以下代码标黄处）。具体代码如下。

```
void insert(const std::string& name, llvm::Type* type, llvm::Value* value = nullptr, bool isConst =
false,bool isArrayinFunc =false,std::vector<int> dims = {}) {
    if (scopes.empty()) return;

    auto& current = scopes.back();
    if (current.count(name)) {
        std::cerr << "[Error]: 重复定义符号: " << name << std::endl;
        exit(1);
    }

    // 自动提取数组维度信息
    std::vector<int> dimensions;
    if(isArrayinFunc){
        dimensions = dims;
    }else{
        // 检查类型是否为数组类型
        if (auto* arrType = llvm::dyn_cast<llvm::ArrayType>(type)) {
            // llvm 获取数组的维度
```

```
            int numElements = arrType->getNumElements();
            dimensions.push_back(numElements);

            // 递归处理多维数组
            llvm::Type* elementType = arrType->getElementType();
            while (auto* nestedArrType = llvm::dyn_cast<llvm::ArrayType>(elementType)) {
                numElements = nestedArrType->getNumElements();
                dimensions.push_back(numElements);
                elementType = nestedArrType->getElementType();
            }
        }
    }
    Symbol symbol(name, type, value, isConst, dimensions,isArrayinFunc);
    current[name] = symbol;
}
```

③ 符号表的查找操作

符号表查找的操作实现比较简单，重点是先在当前作用域查找，如果没有找到再去上层作用域查找，以此类推。这样就可以实现如果上层作用域和当前作用域有同名符号时，会选择当前作用域的符号。

```
    std::optional<Symbol> lookup(const std::string& name) const {
        for (auto it = scopes.rbegin(); it != scopes.rend(); ++it) {
            auto found = it->find(name);
            if (found != it->end()) {
                return found->second;
            }
        }
        return std::nullopt;
    }
```

2. 变量和常量声明与初始化

变量和常量的初始化的基本思路是相同的，但是由于 g4 中常来那个声明必须初始化，因此在这个位置二者具有差别。

① 常量声明与初始化

```
constdecl : CONST_KW bType constDef (Comma constDef)* Semi ;
constDef : Ident (LeftBracket IntConst RightBracket)* Assign constInitVal ;
constInitVal : constExp | LeftBrace ( constInitVal (Comma constInitVal)* )? RightBrace ;
```

对于常量声明和初始化，我们直接使用 visitConstDecl 函数实现以上三行的内容。

对于第一行的内容主要实现如下，遍历所有常量定义分别进行处理。

```
std::any Analysis::visitConstDecl(HelloParser::ConstdeclContext* ctx) {
    Type* type = getTypeFromBType(ctx->bType());  // 获取常量类型
    for (auto def : ctx->constDef()) {  // 遍历常量定义
    …
```

```
    }
    return nullptr;
}
```

在处理常量定义时，首先获取数组维度。

```
        std::vector<int> dims;
        for (size_t i = 0; i < def->LeftBracket().size(); ++i) {  // 处理数组维度
            dims.push_back(std::stoi(def->IntConst(i)->getText()));  // 将数组维度的字符串转换为整数
        }
```

之后就需要进行分类讨论：如果获得的数组维度 dims 为空，即 dims.empty()为真，此时常量不是数组。首先通过 processConstInitVal 函数获取等号右边常量表达式的值，这个函数的核心就是将 visit(ctx->constExp())的结果（返回的是 Value*）作为返回值。

在这里对 scopelevel 进行分类讨论。对于全局常量要通过生成一个 GlobalVariable 来生成 IR；对于非全局常量 lloca 进行内存分配，然后使用 builder.CreateStore 来存储 initVal 的值。

```
        if (dims.empty()) {
            // 处理非数组
            Value* initVal = processConstInitVal(def->constInitVal(), type); //获取初始化值
            llvm::Type* initType = initVal->getType();

            auto scopeLevel = symbolTable.getCurrentScopeLevel();
            if (scopeLevel == 1) {
                Constant* constVal = dyn_cast<Constant>(initVal);
                GlobalVariable* gv = new GlobalVariable(
                    module,
                    type,
                    true,
                    GlobalValue::ExternalLinkage,
                    constVal,
                    name
                );

                symbolTable.insert(name, type, gv,true,false);
            } else {
                // 局部作用域 -> 创建 AllocaInst
                AllocaInst* alloca = builder.CreateAlloca(type, nullptr, name);
                builder.CreateStore(initVal, alloca);  // 存初始值
                symbolTable.insert(name, type, alloca,true,false);
            }
        }
```

以上代码生成 IR 的结果如下所示：

```
const int a = 0;

int main()
{
```

```
    const int c = 1;
    return 0;
}
```

```
@a = constant i32 0                     //scopelevel == 1

define i32 @main() {
entry:
  %c = alloca i32, align 4              //scopelevel != 1
  store i32 1, i32* %c, align 4
  ret i32 0
}
```

对于多维数组常量的处理，先分全局和非全局讨论。在全局常量化中，还分初始化为空数组（填 0）和初始化数组非空来讨论。在初始化数组非空的情况中，需要通过核心函数 flattenConstInitValLinear 将数组扁平化，对缺位补 0，再将扁平化后的数组 convertFlatToMultiDim 转换回去。事实上，在不断 debug 的过程中，因为 flattenConstInitValLinear 函数写的逐渐完善，这个先扁平化后转换回去的过程可以省略，但是这样留着也没有问题，而且担心删除引入新问题，所以这个代码就放在这里了。

在这一部分重点是 flattenConstInitValLinear 函数的实现，主要思路是这个函数递归调用自身，读取数组每一层的 dims 内容，根据元素个数来相应的补 0。需要注意的是，这种处理方式要求初始化表达式中必须使用完整的大括号来明确每一层维度。比如 const int a[2][2][2] = {{{1}}, {{3}}}; 可以被正确解析并自动补 0，但像 const int a[2][2][2] = {{1}, {3}}; 这种省略了部分大括号的写法，其语义不够明确，就没有办法进行补 0。

```
    else {
        // 处理多维数组常量
        llvm::Type* arrayType = type;
        for (int i = dims.size() - 1; i >= 0; --i) {
            arrayType = llvm::ArrayType::get(arrayType, dims[i]);
        }

        bool isListInit = def->constInitVal()->LeftBrace() != nullptr;

        auto scopeLevel = symbolTable.getCurrentScopeLevel();
        if(scopeLevel == 1){
            Constant* initVal = nullptr;
            auto initVal_global = def->constInitVal();
            if (!initVal_global->constInitVal().empty()) {

                std::vector<llvm::Constant*> flatConstants;

                flattenConstInitValLinear(root, type, dims, 0, flatConstants);
                int totalElements = 1;
                for (int dim : dims) totalElements *= dim;
```

```
                // 如果 flatConstants 长度不足，补零
                while (flatConstants.size() < totalElements) {
                    flatConstants.push_back(Constant::getNullValue(type));
                }

                initVal = convertFlatToMultiDim(flatConstants, dims, type);
            } else {
                // 未初始化，填零
                initVal = ConstantAggregateZero::get(arrayType);
            }

            GlobalVariable* gv = new GlobalVariable(
                module,
                arrayType,
                true,
                GlobalValue::ExternalLinkage,
                initVal,
                name
            );

            symbolTable.insert(name, arrayType, gv,true,false);
```

flattenConstInitValLinear 函数代码如下。

```
void Analysis::flattenConstInitValLinear(
        HelloParser::ConstInitValContext* ctx,
        llvm::Type* elemType,
        const std::vector<int>& dims,
        int dimIndex,
        std::vector<llvm::Constant*>& result
) {
        if (!ctx) return;
        if (ctx->constExp()) {
            auto valAny = visitConstExp(ctx->constExp());
            llvm::Value* val = std::any_cast<llvm::Value*>(valAny);
            if (auto* constVal = llvm::dyn_cast<llvm::Constant>(val)) {
                result.push_back(constVal);
            }
            return;
        }
        int expected = dims[dimIndex];
        int count = ctx->constInitVal().size();

        for (int i = 0; i < count; ++i) {
            flattenConstInitValLinear(ctx->constInitVal(i), elemType, dims, dimIndex + 1, result);
        }

        for (int i = count; i < expected; ++i) {
            int subElements = 1;
            for (int j = dimIndex + 1; j < dims.size(); ++j) subElements *= dims[j];
            for (int k = 0; k < subElements; ++k) {
                result.push_back(llvm::Constant::getNullValue(elemType));
```

```
            }
        }
    }
```

对于非全局常量来说，其和全局常量初始化的区别只在于需要生成 alloca 指令和 GEP 指令来完成常量初始化。此时使用 initConstInitValWithValues 在递归的同时生成相应的 GEP 指令和 Store 指令。

```
 else{
                // 将数组常量插入符号表，暂时不赋初值（nullptr）
                AllocaInst* alloca = builder.CreateAlloca(arrayType, nullptr, name);
                symbolTable.insert(name, arrayType, alloca,true,false);

                auto initVal_not_global = def->constInitVal();
                // std::vector<llvm::Constant*> flatConstants;
                // flattenConstInitValLinear(initVal_not_global, type, dims, 0, flatConstants);
                std::cout<<"going to init array"<<std::endl;
                std::vector<int> indicesSoFar;
                initConstInitValWithValues(initVal_not_global,alloca,type,dims,0,indicesSoFar);
            }
```

initConstInitValWithValues 函数代码如下，其主要作用是递归地解析数组的初始化表达式，为多维数组变量构建 GEP 索引，并将值存入数组的对应位置，若未提供足够初始化值，则补零。

其主要维护了 indicesSoFar 数组，用来存放 GEP 需要使用的索引路径，每一次递归之前都要进行这样的操作:

```
indicesSoFar.push_back(i);
initConstInitValWithValues(...);
indicesSoFar.pop_back();
```

其作用就是使得递归调用的函数知道要在哪里生成 store 指令，也就是 GEP 指令的参数。具体代码如下。

```
void Analysis::initConstInitValWithValues(
    HelloParser::ConstInitValContext* ctx,
    AllocaInst* allocaPtr,
    llvm::Type* elemType,
    const std::vector<int>& dims,
    int dimIndex,
    std::vector<int>& indicesSoFar
) {
    if (!ctx) return;
    if (ctx->constExp()) {
        auto valAny = visitConstExp(ctx->constExp());
        llvm::Value* val = std::any_cast<llvm::Value*>(valAny);
        std::vector<llvm::Value*> gepIdx;
        gepIdx.push_back(builder.getInt32(0));
```

```
        for (int idx : indicesSoFar) {
            gepIdx.push_back(builder.getInt32(idx));
        }

        Value* elemPtr = builder.CreateGEP(
            allocaPtr->getType()->getPointerElementType(),
            allocaPtr,
            gepIdx
        );
        builder.CreateStore(val, elemPtr);
        return;
    }
    int expected = dims[dimIndex];
    int count = ctx->constInitVal().size();
    for (int i = 0; i < count; ++i) {
        indicesSoFar.push_back(i);
         initConstInitValWithValues(ctx->constInitVal(i), allocaPtr, elemType, dims, dimIndex + 1,
indicesSoFar);
        indicesSoFar.pop_back();
    }
    for (int i = count; i < expected; ++i) {
        indicesSoFar.push_back(i);
        int subElements = 1;
        for (int j = dimIndex + 1; j < (int)dims.size(); ++j) {
            subElements *= dims[j];
        }

        for (int k = 0; k < subElements; ++k) {
            int lastDimSize = dims[dimIndex + 1];
            int leafIdx = k % lastDimSize;
            std::vector<int> fullIdx = indicesSoFar;
            fullIdx.push_back(leafIdx);
            std::vector<llvm::Value*> gepIdx2;
            gepIdx2.push_back(builder.getInt32(0));
            for (int idx2 : fullIdx) {
                gepIdx2.push_back(builder.getInt32(idx2));
            }
            Value* elemPtr2 = builder.CreateGEP(
                allocaPtr->getType()->getPointerElementType(),
                allocaPtr,
                gepIdx2
            );
            builder.CreateStore(llvm::Constant::getNullValue(elemType), elemPtr2);
        }
        indicesSoFar.pop_back();
    }
}
```

② 变量声明与初始化

变量的声明和初始化和常量的思路完全一致，其主要区别是需要将 initConstInitValWithValues 函数改写为 initConstInitValArrayWithValues；将

flattenConstInitValLinear 函数改写为 flattenConstInitValArrayLinear 函数。其核心原因是因为以下几行 g4 代码。常量不区分数组和非数组定义，但是变量有 varDefArray 来表示数组的定义，因此相比于常量的声明所用函数，变量的声明函数主要是要不直接进行递归，而是先进行 varDefArray-> constInitValArray-> constInitVal 的遍历，然后才能进行递归。

```
constDef : Ident ( LeftBracket IntConst RightBracket )* Assign constInitVal ;
constInitVal : constExp | LeftBrace ( constInitVal ( Comma constInitVal )* )? RightBrace ;

varDefArray : Ident LeftBracket IntConst RightBracket ( LeftBracket IntConst RightBracket )* ( Assign
constInitValArray )?;
constInitValArray : LeftBrace ( constInitVal ( Comma constInitVal )* )? RightBrace ;
```

因此，在代码实现中，initConstInitValArrayWithValues 调用了 initConstInitValWithValues，函数内只进行 varDefArray-> constInitValArray-> 第一层 constInitVal 的遍历，之后直接调用 initConstInitValWithValues 函数继续执行即可，具体代码如下。

```
void Analysis::initConstInitValArrayWithValues(
    HelloParser::ConstInitValArrayContext* ctx,
    AllocaInst* allocaPtr,
    llvm::Type* elemType,
    const std::vector<int>& dims,
    int dimIndex,
    std::vector<int>& indicesSoFar
) {
    if (!ctx) return;
    int expected = dims[dimIndex];
    int count = ctx->constInitVal().size();
    for (int i = 0; i < count; ++i) {
        indicesSoFar.push_back(i);
        HelloParser::ConstInitValContext* subCtx = ctx->constInitVal(i);
        initConstInitValWithValues(
            subCtx,
            allocaPtr,
            elemType,
            dims,
            dimIndex + 1,
            indicesSoFar
        );
        indicesSoFar.pop_back();
    }
    //补0
    if (count < expected) {
        int subElements = 1;
        for (int j = dimIndex + 1; j < (int)dims.size(); ++j) {
            subElements *= dims[j];
        }
        for (int i = count; i < expected; ++i) {
            indicesSoFar.push_back(i);
```

```
            for (int k = 0; k < subElements; ++k) {
                int lastDimSize = dims[dimIndex + 1];
                int leafIdx = k % lastDimSize;
                std::vector<int> fullIdx = indicesSoFar;
                fullIdx.push_back(leafIdx);
                std::vector<llvm::Value*> gepIdx2;
                gepIdx2.reserve(fullIdx.size() + 1);
                gepIdx2.push_back(builder.getInt32(0));
                for (int idx2 : fullIdx) {
                    gepIdx2.push_back(builder.getInt32(idx2));
                }
                Value* elemPtr2 = builder.CreateGEP(
                    allocaPtr->getType()->getPointerElementType(),
                    allocaPtr,
                    gepIdx2
                );
                builder.CreateStore(llvm::Constant::getNullValue(elemType), elemPtr2);
            }
            indicesSoFar.pop_back();
        }
    }
}
```

flattenConstInitValArrayLinear 也是一样，通过调用 flattenConstInitValLinear 函数来直接完成数组的平展，最后再进行补 0 即可，具体代码如下。

```
void Analysis::flattenConstInitValArrayLinear(
        HelloParser::ConstInitValArrayContext* ctx,
        llvm::Type* elemType,
        const std::vector<int>& dims,
        int dimIndex,
        std::vector<llvm::Constant*>& result
) {
    if (!ctx) return;

    int expected = dims[dimIndex];
    int count = ctx->constInitVal().size();
    int i=0;
    for(i;i<count;i++){
        flattenConstInitValLinear(ctx->constInitVal(i), elemType, dims, dimIndex + 1, result);
    }

    //补 0
    if (count < expected) {
        int subElements = 1;
        for (int j = dimIndex + 1; j < dims.size(); ++j) {
            subElements *= dims[j];
        }
        int zerosNeeded = (expected - count) * subElements;

        llvm::Constant* zero = llvm::Constant::getNullValue(elemType);
```

```
        for (int i = 0; i < zerosNeeded; ++i) {
            result.push_back(zero);
        }
    }
}
```

3. 函数声明

① 函数表实现

函数表的实现非常简单，只需要一个数组即可。llvm::Function*类型的指针就可以提过函数的左右信息了，因此只需要函数名和这个指针即可。其插入方法就是添加一个新的表项，查找方法就是用 unordered_map 这个哈希表去查找函数名获取函数指针。

```
    std::unordered_map<std::string, llvm::Function*> functionTable;
```

② 函数参数处理方法

函数参数分为数组参数和非数组参数，函数声明时函数的第一行会有如下形式。这时候需要将函数的参数插入到符号表的新作用域中，因此首先需要进入一个新的作用域。在进入作用域之后需要分两种情况，一种是非数组，按照常规方法即可。对于非数组，我们需要现将它的组外层退化为指针，然后将其存入符号表，此时需要将函数声明中的数组这一参数 isArrayinFunc 置为 true。关键代码如下。

```
int foo(int a, int b[ ]){
```

```
        Type* paramType_insert;
        AllocaInst* alloca = nullptr;
        if (param->LeftBracket().empty()) {
            paramType_insert = paramType;
            alloca = builder.CreateAlloca(paramType_insert, nullptr, paramName);
            symbolTable.insert(paramName, paramType, alloca, false,false);
        }else{
            paramType_insert = paramType->getArrayElementType();
            paramType_insert = llvm::PointerType::get(paramType_insert, 0);
            alloca = builder.CreateAlloca(paramType_insert, nullptr, paramName);
            symbolTable.insert(paramName, paramType_insert, alloca, false,true,dims);
        }
```

此时无法使用符号表中 insert 操作实现的通用方式来自动的获取维度信息，因此需要输入 dims 作为参数的维度。因为服务器端的代码被删除现在没法跑这个代码，因此无法给出例子。

在这之后就可以将参数 alloca 到内存中，然后进行参数加载语句 CreateStore 将参数的的值从调用语句存储到 alloca 的内存中。在 store 完成之后可以访问函数体 block 了，最后在都处理完成后，void 类型的函数没有返回语句因此需要 CreateRetVoid，之后就可以退出

当前作用域了。重要代码如下。

```
        std::cout<<"函数的参数插入符号表："<<paramName<<std::endl;
        symbolTable.print();
        std::cout<<"test point "<<paramName<<std::endl;

        Value* paramValue = func->getArg(paramIdx++);
        std::cout<<"test point "<<paramName<<std::endl;
        builder.CreateStore(paramValue, alloca);

    }

    if (ctx->block()) {
        visitBlock(ctx->block());
    }
    if (returnType->isVoidTy() && !builder.GetInsertBlock()->getTerminator()) {
        builder.CreateRetVoid();
    }
    symbolTable.exitScope();
    return nullptr;
}
```

4. Stmt 模块实现

① 表达式：exp [Assign exp] Semi

表达式左边是 lhs, 右边是 rhs。Lhs 的处理方法就是先提取变量名, 然后通过 getLValuePtr 函数获取 llvm::Value*类型的指针指向这个被赋值的变量; 而 rhs 的处理方式就是直接一层一层地向下 visit，然后将最后一层一般是 primaryExp 的返回值一层一层地传回来，获得经过内存加载后的 llvm::Value*类型的指针。在获得了这两个指针之后就可以进行 CreateStore 了。重要代码如下。

```
 if (ctx->Assign()) {

            std::cout << "visiting Stmt - Assign" << std::endl;
            std::string varName = extractVarNameFromExp(ctx->exp()[0]);
            std::cout << "visiting Stmt - Assign -varname:"<< varName << std::endl;

            std::optional<Symbol> lhs_name = symbolTable.lookup(varName);
            llvm::Value* lhs = nullptr;
            try {
                //lhs = std::any_cast<llvm::Value*>(visit(ctx->exp()[0]));
                std::cout<<"lhs = getLValuePtr(ctx->exp()[0]) begin"<<std::endl;

                lhs = getLValuePtr(ctx->exp()[0]);
                std::cout<<"lhs = getLValuePtr(ctx->exp()[0]) finish"<<std::endl;
            } catch (const std::bad_any_cast& e) {
                std::cerr << "[Error] LHS is not a valid llvm::Value*: " << e.what() << std::endl;
                exit(1);
            }
```

```
            auto symbolOpt = symbolTable.lookup(varName);

            llvm::Value* rhs = nullptr;
            try {
                rhs = std::any_cast<llvm::Value*>(visit(ctx->exp()[1]));
            }
            builder.CreateStore(rhs, lhs);
        }
```

② 返回语句：RETURN_KW exp? Semi

处理 return 语句的 exp 操作和处理等式右边 rhs 的操作相似，需要使用 visit 访问下去然后将返回的 Value*指针一层一层传回来，然后就可以 CreateRet 来穿件 return 语句了。

```
        else if (ctx->RETURN_KW()) {
                size_t current = symbolTable.getCurrentScopeLevel();
                std::cerr << "处理 return 语句...scopelevel:" << current << std::endl;
                if (ctx->exp().empty()) {
                    builder.CreateRetVoid();
                } else {
                    auto returnExpr = visit(ctx->exp()[0]);
                    Value* returnValue= nullptr;
                    try {
                        returnValue = std::any_cast<Value*>(returnExpr);
                    }
                    builder.CreateRet(returnValue);
                }
```

③ Block 块：block

根据 CACT 语言规范，stmt 中可以嵌套 block，语义上表示一组语句构成的复合语句体。每当进入一个新的 block，都意味着进入一个新的作用域，因此必须在语义分析阶段创建新的作用域以便正确管理局部变量、常量的声明与查找。在我们的语义分析中，visitStmt()会检测当前语句是否为 block 类型，若是，则需调用 symbolTable.enterScope()来创建新作用域，并递归调用 visitBlock()以访问并处理该代码块中的所有子语句和定义。

完整代码实现如下：

```
    if(ctx -> block()){
        symbolTable.enterScope(); // 进入新的作用域
        return visitBlock(ctx->block()); // 访问 block 中的语句
    }
```

此外，我们在 visitBlock()内部结尾处调用 symbolTable.exitScope()，从而保证作用域管理的一致性。这种“进入时入栈、离开时出栈”的结构确保了嵌套 block 的符号解析具备良好的层次性和可控性，避免符号污染。

④ IF 语句：IF_KW LeftParen cond RightParen stmt (ELSE_KW stmt)?

根据提供的 CACT 语言规范，if 语句有两种形式：可以包含 else 分支，也可以不包含 else 分支。我们在实现中统一采用以下方式处理 if 和 if-else 控制流结构：

```
if ( A ) {
    B
} else {
    C
}
```

我们的设计思路如下：

条件表达式 cond 即 A 部分属于 if 语句的一部分，其求值应发生在当前基本块中，而非新的 block。if 与 else 中间的 stmt 即 B 部分应该要创建一个新的 block，即 thenBB。else 后面的 stmt 即 C 部分也应该要创建一个新的 block，即 elseBB。对于 if-else 语句结束后的部分，我们也创建了一个新的 block，即 mergeBB，作为两个分支后统一的收敛点，仅在需要时创建。then 和 else 两个分支对应不同的作用域，因此我们在进入各自的分支前调用 symbolTable.enterScope()，进入一层作用域；结束后调用 symbolTable.exitScope()，退出一层作用域。

条件表达式 cond 的跳转由 genCond(ctx->cond(), thenBB, elseBB)生成，会根据 cond 的值生成条件跳转指令，确保 true 时跳转到 thenBB，false 时跳转到 elseBB。该函数将在短路求值部分详细介绍，此处略去细节。

对于 else 分支不存在的情况，我们的处理方式是：依然创建一个 else block，并在其中直接跳转至后续 merge block。

以下代码展示了控制流基本块的创建、插入和跳转逻辑：

```
        llvm::Function* function = builder.GetInsertBlock()->getParent();
        llvm::BasicBlock* thenBB = llvm::BasicBlock::Create(context, "then", function);
        llvm::BasicBlock* elseBB = llvm::BasicBlock::Create(context, "else", function);
        llvm::BasicBlock* mergeBB = nullptr;
        bool needMerge = false;
```

关于 mergeBB 的创建时机：

最初我们在实现时选择无条件创建 mergeBB，但在调试过程中发现若 if 分支和 else 分支都已经显式终结（即都有 return 语句），则 mergeBB 实际上并不会被用到，这会导致在生成 IR 时出现 merge 空块。因此，我们对创建 mergeBB 的逻辑进行了调整：只有当至少有一个分支未终结时，才实际创建并跳转到 mergeBB。我们使用布尔变量 needMerge 来追踪是否已经创建该块，避免出现在 if 分支和 else 分支各创建了一个 mergeBB 的情况。

关于此部分的调试过程与思路将在实验报告的“Debug 过程”章节中“4. mergeBB 的创建

条件"部分进一步详细说明。

完整代码实现如下:

```
    if (ctx->IF_KW()) {

        llvm::Function* function = builder.GetInsertBlock()->getParent();

        // 创建 then、else 两个基本块
        llvm::BasicBlock* thenBB = llvm::BasicBlock::Create(context, "then", function);
        llvm::BasicBlock* elseBB = llvm::BasicBlock::Create(context, "else", function);
        //llvm::BasicBlock* mergeBB = llvm::BasicBlock::Create(context, "merge", function);
        llvm::BasicBlock* mergeBB = nullptr;
        bool needMerge = false;

        // 新增：使用短路逻辑构造的条件跳转语句，生成跳转到 then 或 else 的控制流
        genCond(ctx->cond(), thenBB, elseBB);

        // 创建条件跳转语句，根据 condValue 跳转到 then 或 else
        //builder.CreateCondBr(condValue, thenBB, elseBB);

        // 插入 thenBB 块中的语句
        builder.SetInsertPoint(thenBB);
        symbolTable.enterScope();
        std::cout << "visiting IF - thenBB"  << std::endl;
        visit(ctx->stmt(0));  // if 分支

        // 只有在没有 terminator 时才跳转到 merge 块
        if (!builder.GetInsertBlock()->getTerminator()) {
            std::cout << "visiting IF - thenBB jump to mergeBB"  << std::endl;
            if (needMerge == false) {
                mergeBB = llvm::BasicBlock::Create(context, "merge", function);
                std::cout << "visiting IF - create mergeBB"  << std::endl;
            }
            builder.CreateBr(mergeBB);
            needMerge = true;
        }
        std::cout << "visiting IF - thenBB end"  << std::endl;
        symbolTable.exitScope();

        // 插入 elseBB 块中的语句（如果有 else）
        builder.SetInsertPoint(elseBB);
        symbolTable.enterScope();
        std::cout << "visiting IF - elseBB"  << std::endl;
        if (ctx->ELSE_KW()) {
            visit(ctx->stmt(1));  // else 分支
        }

        // 只有在没有 terminator 时才跳转到 merge 块
        if (!builder.GetInsertBlock()->getTerminator()) {
            std::cout << "visiting IF - elseBB jump to mergeBB"  << std::endl;
            if (needMerge == false) {
                mergeBB = llvm::BasicBlock::Create(context, "merge", function);
```

```
            std::cout << "visiting IF - create mergeBB"  << std::endl;
        }
        builder.CreateBr(mergeBB);
        needMerge = true;
    }

    symbolTable.exitScope();

    // 如果至少有一个分支没有终结，就设置 mergeBB
    //if (!thenBB->getTerminator() || !elseBB->getTerminator()) {
    if (needMerge) {
        std::cout << "visiting IF - set insert point to mergeBB"  << std::endl;
        builder.SetInsertPoint(mergeBB);
    }

}
```

经过上述处理后，生成的 IR 成功构建了三个基本块，并正确实现了循环控制流结构。我们使用以下测试用例验证了逻辑的正确性：

测试用例	生成 IR 与运行结果

`int main(){`
`  int a, b;`
`  a = 1;`
`  b = 2;`
`  if(a > b){`
`  a = 8;`
`  } else {`
`  a = 9;`
`  }`
`  return a;`
`}`
	`; ModuleID = 'main'`
	`source_filename = "main"`
	`……`
	`define i32 @main() {`
	`entry:`
	`  %a = alloca i32, align 4`
	`  store i32 0, i32* %a, align 4`
	`  %b = alloca i32, align 4`
	`  store i32 0, i32* %b, align 4`
	`  %a_elem_ptr = getelementptr inbounds i32, i32* %a, i32 0`
	`  store i32 1, i32* %a_elem_ptr, align 4`
	`  %b_elem_ptr = getelementptr inbounds i32, i32* %b, i32 0`
	`  store i32 2, i32* %b_elem_ptr, align 4`
	`  %a_var = load i32, i32* %a, align 4`
	`  %b_var = load i32, i32* %b, align 4`
	`  %igt_tmp = icmp sgt i32 %a_var, %b_var`
	`  %cmp_tmp = icmp ne i1 %igt_tmp, false`
	`  br i1 %cmp_tmp, label %then, label %else`
	
	`then:  ; preds = %entry`
	`  %a_elem_ptr1 = getelementptr inbounds i32, i32* %a, i32 0`
	`  store i32 8, i32* %a_elem_ptr1, align 4`
	`  br label %merge`
	
	`else:  ; preds = %entry`
	`  %a_elem_ptr2 = getelementptr inbounds i32, i32* %a, i32 0`
	`  store i32 9, i32* %a_elem_ptr2, align 4`
	`  br label %merge`

```
merge:                                            ; preds = %else, %then
  %a_var3 = load i32, i32* %a, align 4
  ret i32 %a_var3
}

运行结果：
编译 output.ll 为 output.o...
链接 output.o 和 libcact_rt.a...
运行程序...
程序退出码: 9
```

⑤ WHILE 语句：WHILE_KW LeftParen cond RightParen stmt

我们采用以下控制流程设计来生成 while 语句的 IR：

```
while ( A ) {
    B
}
C
```

我们的设计思路如下：

对于条件表达式 cond 即 A 部分，由于 while 循环体执行完毕后需要重新判断条件，因此为条件表达式 cond 创建了一个单独的基本块 condBB，以便形成循环控制流。循环体 stmt 被映射为一个独立的基本块 bodyBB，用于执行循环内容。对于 while 语句结束后的部分，我们也创建了一个新的 block，即 afterBB。为了支持嵌套的 while 循环结构，我们为 condBB 和 afterBB 维护了两个栈结构，在进入每层 while 时压栈，在处理完成后出栈，以便正确处理 break 与 continue 等控制流跳转。bodyBB 部分属于新的作用域，因此我们在进入 body 块前调用 symbolTable.enterScope()，进入一层作用域；结束后调用 symbolTable.exitScope()，退出一层作用域。

条件表达式 cond 的跳转由 genCond(ctx->cond(), bodyBB, afterBB)生成，会根据 cond 的值生成条件跳转指令，确保 true 时跳转到 bodyBB，false 时跳转到 afterBB。该函数将在短路求值部分详细介绍，此处略去细节。

此外，循环体 bodyBB 中可能包含 return 等终止控制流的语句，此时当前基本块已经结束。因此需要在生成跳转回 condBB 的语句前，判断当前基本块是否已经终结，即是否存在 terminator 指令。如果有，则不应该再生成跳转到 condBB 的指令，避免生成无效跳转。

完整代码实现如下：

```
    if (ctx->WHILE_KW()) {

       std::cout << "visiting Stmt - WHILE"  << std::endl;
```

```
        // 获取当前函数作用域
        llvm::Function* function = builder.GetInsertBlock()->getParent();

        // 创建 cond、body、end 三个基本块，命名基本块的名字
        llvm::BasicBlock* condBB = llvm::BasicBlock::Create(context, "while.cond", function);
        llvm::BasicBlock* bodyBB = llvm::BasicBlock::Create(context, "while.body", function);
        llvm::BasicBlock* afterBB = llvm::BasicBlock::Create(context, "while.after", function);

        // 进入 while 之前跳转到条件块
        builder.CreateBr(condBB);

        // 条件块，插入 condBB 块中的语句
        builder.SetInsertPoint(condBB);

        // 入栈
        std::cout << "WHILE - pushing condBB and afterBB" << std::endl;
        condBBStack.push(condBB);// 将当前循环的 condBB 压入一个栈中
        afterBBStack.push(afterBB);

        // 新增：使用短路逻辑构造的条件跳转语句，生成跳转到 body 或 after 的控制流
        std::cout << "WHILE - generating conditional branch" << std::endl;
        genCond(ctx->cond(), bodyBB, afterBB);

        // 循环体块
        std::cout << "WHILE - entering body" << std::endl;
        builder.SetInsertPoint(bodyBB);
        symbolTable.enterScope();
        visit(ctx->stmt(0));  // 假设 stmt 是循环体

        // 循环体执行完回到条件判断
        std::cout << "WHILE - jumping back to condBB" << std::endl;
        //builder.CreateBr(condBB);
        // 检查当前 body 中是否已经有终止指令（terminator）
        // 若有，不应该再生成跳转到 while.cond 的指令
        if (!builder.GetInsertBlock()->getTerminator()) {
            builder.CreateBr(condBB);
        }

        // 出栈（循环结束）
        std::cout << "WHILE - popping condBB and afterBB" << std::endl;
        condBBStack.pop();
        afterBBStack.pop();
        symbolTable.exitScope();
        // 结束块
        builder.SetInsertPoint(afterBB);
    }
```

经过上述处理后，生成的 IR 成功构建了三个基本块，并正确实现了循环控制流结构。我们使用以下测试用例验证了逻辑的正确性：

测试用例	生成 IR 与运行结果

int main(){
    int a, b;
    a = 0;
    b = 2;
    while(a < b){
        a = a + 1;
    }
    return a;
}
	; ModuleID = 'main'
	source_filename = "main"
	……
	define i32 @main() {
	entry:
	  %a = alloca i32, align 4
	  store i32 0, i32* %a, align 4
	  %b = alloca i32, align 4
	  store i32 0, i32* %b, align 4
	  %a_elem_ptr = getelementptr inbounds i32, i32* %a, i32 0
	  store i32 0, i32* %a_elem_ptr, align 4
	  %b_elem_ptr = getelementptr inbounds i32, i32* %b, i32 0
	  store i32 2, i32* %b_elem_ptr, align 4
	  br label %while.cond
	
	while.cond: ; preds = %while.body, %entry
	  %a_var = load i32, i32* %a, align 4
	  %b_var = load i32, i32* %b, align 4
	  %ilt_tmp = icmp slt i32 %a_var, %b_var
	  %cmp_tmp = icmp ne i1 %ilt_tmp, false
	  br i1 %cmp_tmp, label %while.body, label %while.after
	
	while.body: ; preds = %while.cond
	  %a_elem_ptr1 = getelementptr inbounds i32, i32* %a, i32 0
	  %a_var2 = load i32, i32* %a, align 4
	  %addtmp = add i32 %a_var2, 1
	  store i32 %addtmp, i32* %a_elem_ptr1, align 4
	  br label %while.cond
	
	while.after: ; preds = %while.cond
	  %a_var3 = load i32, i32* %a, align 4
	  ret i32 %a_var3
	}
	
	运行结果:
	编译 output.ll 为 output.o...
	链接 output.o 和 libcact_rt.a...
	运行程序...
	程序退出码: 2

⑥ BREAK 语句：BREAK_KW Semi

break 语句的语义是终止当前的 while 循环，直接跳转到循环的后继基本块 afterBB。因此，在生成 IR 时，只需创建一条跳转指令至 afterBBStack.top()所表示的当前循环结束块即可。

完整代码实现如下：

```
    if (ctx->BREAK_KW()) {
        std::cout << "visiting Stmt - BREAK" << std::endl;

        // 跳转到当前循环的结束块（即 afterBB）
        std::cout << "BREAK - jumping to afterBB" << std::endl;
        builder.CreateBr(afterBBStack.top());
    }
```

⑦ CONTINUE 语句：CONTINUE_KW Semi ;

continue 语句的作用是跳过当前循环体中的剩余语句，重新进行条件判断。因此我们在 IR 中创建一条跳转指令至 condBBStack.top()，使控制流跳回条件块 condBB。

完整代码实现如下：

```
    if (ctx->CONTINUE_KW()) {
        std::cout << "visiting Stmt - CONTINUE" << std::endl;

        // 跳转到当前循环的条件判断块（即 condBB）
        std::cout << "CONTINUE - jumping to condBB" << std::endl;
        builder.CreateBr(condBBStack.top());
    }
```

由于支持 while 的嵌套结构，我们在处理 break 和 continue 时，通过 afterBBStack 和 condBBStack 分别记录每层循环的退出块和判断块，以便在任意层级中准确跳转。

为验证 break 与 continue 在嵌套 while 循环中的控制流生成逻辑是否正确，我们构造了如下测试用例：

```
int main() {
    int i = 0;
    while (i < 3) {
        int j = 0;
        while (j < 3) {
            if (j == 1) {
                j = j + 1;
                continue;
            }
            if (j == 2) {
                break;
            }
            j = j + 1;
        }
        i = i + 1;
    }
```

```
    return 0;
}
```

在该示例中，continue 语句应跳转回内层循环的条件判断基本块%while.cond1，即判断 j < 3 的位置；而 break 语句应跳转到当前循环体的结束基本块%while.after3，对应于 i = i + 1 语句的位置。我们生成的 IR 和运行结果如下：

```
; ModuleID = 'main'
source_filename = "main"
……
define i32 @main() {
entry:
  %i = alloca i32, align 4
  store i32 0, i32* %i, align 4
  br label %while.cond

while.cond:                                       ; preds = %while.after3, %entry
  %i_var = load i32, i32* %i, align 4
  %ilt_tmp = icmp slt i32 %i_var, 3
  %cmp_tmp = icmp ne i1 %ilt_tmp, false
  br i1 %cmp_tmp, label %while.body, label %while.after

while.body:                                       ; preds = %while.cond
  %j = alloca i32, align 4
  store i32 0, i32* %j, align 4
  br label %while.cond1

while.after:                                      ; preds = %while.cond
  ret i32 0

while.cond1:                                      ; preds = %merge14, %then, %while.body
  %j_var = load i32, i32* %j, align 4
  %ilt_tmp4 = icmp slt i32 %j_var, 3
  %cmp_tmp5 = icmp ne i1 %ilt_tmp4, false
  br i1 %cmp_tmp5, label %while.body2, label %while.after3

while.body2:                                      ; preds = %while.cond1
  %j_var6 = load i32, i32* %j, align 4
  %eq_tmp = icmp eq i32 %j_var6, 1
  %cmp_tmp7 = icmp ne i1 %eq_tmp, false
  br i1 %cmp_tmp7, label %then, label %else

while.after3:                                     ; preds = %then9, %while.cond1
  %i_elem_ptr = getelementptr inbounds i32, i32* %i, i32 0
  %i_var18 = load i32, i32* %i, align 4
  %addtmp19 = add i32 %i_var18, 1
  store i32 %addtmp19, i32* %i_elem_ptr, align 4
  br label %while.cond

then:                                             ; preds = %while.body2
  %j_elem_ptr = getelementptr inbounds i32, i32* %j, i32 0
```

```
  %j_var8 = load i32, i32* %j, align 4
  %addtmp = add i32 %j_var8, 1
  store i32 %addtmp, i32* %j_elem_ptr, align 4
  br label %while.cond1          ; ← continue 跳转目标

else:                                             ; preds = %while.body2
  br label %merge

merge:                                            ; preds = %else
  %j_var11 = load i32, i32* %j, align 4
  %eq_tmp12 = icmp eq i32 %j_var11, 2
  %cmp_tmp13 = icmp ne i1 %eq_tmp12, false
  br i1 %cmp_tmp13, label %then9, label %else10

then9:                                            ; preds = %merge
  br label %while.after3         ; ← break 跳转目标

else10:                                           ; preds = %merge
  br label %merge14

merge14:                                          ; preds = %else10
  %j_elem_ptr15 = getelementptr inbounds i32, i32* %j, i32 0
  %j_var16 = load i32, i32* %j, align 4
  %addtmp17 = add i32 %j_var16, 1
  store i32 %addtmp17, i32* %j_elem_ptr15, align 4
  br label %while.cond1
}

运行结果:
编译 output.ll 为 output.o...
链接 output.o 和 libcact_rt.a...
运行程序...
程序退出码: 0
```

其中，continue 对应的 IR 是“br label %while.cond1”，准确跳转到了条件表达式(j < 3)；break 对应的 IR 是 br label %while.after3，准确跳转到了”i = i + 1”语句。该示例充分验证了 break 和 continue 控制流逻辑的正确性，特别是在多层嵌套时的出栈、入栈匹配。

从控制流角度来看，每层 while 块均维护了独立的 condBB 与 afterBB，我们在语义分析时使用栈结构分别记录当前循环的条件块和结束块。每遇到一条 break 或 continue，即可正确生成相应的跳转语句，实现语义上的准确跳转。

5. 短路求值实现

在 CACT 语言中，if 语句和 while 语句中的判断条件支持逻辑与&&、逻辑或||运算。这类运算的求值需要遵循短路逻辑（short-circuit evaluation）：

• 对于 A && B && C，一旦发现 A 为假，则整体表达式恒为假，直接进入 false 对应的 block，无需判断后续的子表达式；

•对于 A || B || C，一旦发现 A 为真，则整体表达式恒为真，直接进入 true 对应的 block，无需判断后续的子表达式

若直接使用逐一求值的方式，在 visitLAndExp()函数中对所有子表达式求值并组合结果，则不符合短路求值的逻辑。因此，需要为这类表达式引入短路控制流程的生成逻辑。

为了实现短路求值，我在原先的 visitCond()、visitLAndExp()、visitLOrExp()函数的基础上，加上 genCond()、genLAndExp()、genLOrExp()函数。下面我将分别称这两类函数为 visit 系列函数和 gen 系列函数。gen 系列函数只用于短路求值，只在 if 语句和 while 语句中的条件表达式求值部分被调用。

visit 系列函数 vs gen 系列函数：

visit 系列函数用于普通表达式求值，适用于在表达式上下文中计算布尔结果，并将其返回；gen 系列函数用于带跳转的短路求值，只在 if 语句和 while 语句中被调用，通过创建基本块和条件跳转，根据子表达式的结果直接跳转到对应的基本块（trueBB/falseBB），不返回值。

保留 visit 系列函数的原因是我们使用 ANTLR 生成的访问器（visitor）结构进行语义分析与 IR 生成，它要求我们实现各语法节点对应的 visitXXX()函数来遍历 AST。很多情况下我们仍然需要对逻辑表达式进行值求解。因此需要另外构建 gen 系列函数来处理短路求值操作。

gen 系列函数的核心是不返回值，而是根据当前子表达式的结果判断跳转位置；为每一个逻辑子表达式动态创建新基本块（如 land_next，lor_next），以维持正确的控制流结构。通过 LLVM 的 CreateCondBr 精确控制跳转目标。

逻辑与（&&）的短路处理：

genLAndExp()对于 a && b && c 的处理结构如下：

```
entry:
  %a = ...          ; 计算第一个条件
  br i1 %a, land_next1, falseBB

land_next1:
  %b = ...
  br i1 %b, land_next2, falseBB

land_next2:
  %c = ...
```

```
  br i1 %c, trueBB, falseBB
```

即：只要有一个条件为假，即可立即跳转到 falseBB，否则继续判断下一个子条件，直到最后一个子条件为真时跳转到 trueBB。

逻辑或（||）的短路处理：

genLOrExp()对于 a || b || c 的处理结构如下：

```
entry:
  %a = ...          ; 计算第一个条件
  br i1 %a, trueBB, lor_next1

lor_next1:
  %b = ...
  br i1 %b, trueBB, lor_next2

lor_next2:
  %c = ...
  br i1 %c, trueBB, falseBB
```

即：只要有一个条件为真，即可立即跳转到 trueBB，否则继续判断下一个子条件，直到最后一个子条件为假时跳转到 falseBB。

genLAndExp()函数详解：

该函数用于生成短路求值逻辑的逻辑与表达式控制流。具体流程为：函数从左到右遍历所有 eqExp 子表达式。对每个子表达式，使用 visit(eq)进行值计算，再与 0 比较，以生成 i1 类型的布尔值。若当前处理的子表达式不是最后一个子表达式，则新建一个命名为"land_next"的中间基本块。若当前条件为真，则跳转到"land_next"；否则直接跳转到 falseBB，实现短路。接下来设置插入点为"land_next"，准备处理下一个子表达式。若当前处理的子表达式是最后一个子表达式，则根据其值是否为真，跳转到 trueBB 或 falseBB。

genLOrExp()函数详解：

该函数用于生成短路求值逻辑的逻辑或表达式控制流。其逻辑与 genLAndExp()对称，具体流程为：函数从左到右遍历所有 lAndExp 子表达式。若当前处理的子表达式不是最后一个子表达式，则新建一个命名为"lor_next"的中间基本块。若当前条件为假，则跳转到"lor_next"，否则直接跳转到 trueBB，实现短路。接下来调用 genLAndExp()，使当前子表达式的 true 分支跳转到最终 trueBB，false 分支跳转到"lor_next"。设置插入点为"lor_next"，处理下一个子表达式。若当前处理的子表达式是最后一个子表达式，则根据其值是否为真，

跳转到 trueBB 或 falseBB。

函数之间的关系：

以 if (cond) {……} else {……}为例，在语义分析中，visitStmt()会创建 thenBB 和 elseBB，然后调用 genCond(ctx->cond(), thenBB, elseBB), 在 genCond()中进一步调用 genLOrExp()实现控制流。genLOrExp()会按需调用 genLAndExp()。genLAndExp()又会递归调用 visitEqExp()来生成每个子表达式的值。

完整代码实现如下：

```
std::any Analysis::visitCond(HelloParser::CondContext *ctx) {
    std::cout << "visiting Cond - going to visit lOrExp"  << std::endl;
    return visit(ctx->lOrExp());
}

// 有控制参数的重载版本：不返回 Value，而是根据条件控制跳转到对应的基本块
void  Analysis::genCond(HelloParser::CondContext*  ctx,  llvm::BasicBlock*  trueBB,  llvm::BasicBlock*
falseBB) {
    std::cout << "visiting genCond - generating control flow from lOrExp" << std::endl;

    // Cond → LOrExp
    // 短路求值逻辑在 genLOrExp() 中实现：
    // 只要 LOrExp 中有一个子项为真，就跳 trueBB；否则跳 falseBB
    if (!builder.GetInsertBlock()->getTerminator()) {
        genLOrExp(ctx->lOrExp(), trueBB, falseBB);
    }
}

std::any Analysis::visitLAndExp(HelloParser::LAndExpContext *ctx) {
    std::cout << "visiting LAndExp"  << std::endl;
    llvm::Value* lhs = std::any_cast<llvm::Value*>(visit(ctx->eqExp(0)));

    // 将 lhs 转换为 i1，非零即真
    lhs = builder.CreateICmpNE(lhs, llvm::ConstantInt::get(lhs->getType(), 0), "cmp_lhs");

    for (size_t i = 1; i < ctx->eqExp().size(); ++i) {
        llvm::Value* rhs = std::any_cast<llvm::Value*>(visit(ctx->eqExp(i)));

        // 将 rhs 转换为 i1，非零即真
        rhs = builder.CreateICmpNE(rhs, llvm::ConstantInt::get(rhs->getType(), 0), "cmp_rhs");
        // 生成逻辑与：i1 and i1 -> i1
        lhs = builder.CreateAnd(lhs, rhs, "and_tmp");
    }

    return lhs; // 返回的是 i1 类型
}
```

```
// 生成一个逻辑与表达式（LAndExp）的短路逻辑代码：a && b && c ... 形式
// 参数 trueBB: 所有子表达式都为真时跳转的目标基本块
// 参数 falseBB: 一旦有一个为假就跳转的目标基本块
void Analysis::genLAndExp(HelloParser::LAndExpContext* ctx, llvm::BasicBlock* trueBB, llvm::BasicBlock*
falseBB) {
    std::cout << "visiting LAndExp - Short-circuit Evaluation"  << std::endl;
    size_t n = ctx->eqExp().size();  // n 为 EqExp 的个数（LAndExp 是多个 EqExp 用 '&&' 连接而成）

    llvm::Function* func = builder.GetInsertBlock()->getParent();  // 当前所在函数

    for (size_t i = 0; i < n; ++i) {
        std::cout << "-----entering genLAndExp for loop-----"  << std::endl;
        // 获取第 i 个子表达式
        HelloParser::EqExpContext* eq = ctx->eqExp(i);

        // 调用 visit(eq) 得到这个子表达式的结果值（通常是 i32 类型）
        llvm::Value* cond = std::any_cast<llvm::Value*>(visit(eq));

        // 将 cond 转换为 i1 类型，非零即真
        //cond = castToBool(cond);  // 转换为 i1（布尔值）
        cond = builder.CreateICmpNE(cond, llvm::ConstantInt::get(cond->getType(), 0), "cmp_tmp");

        if (i == n - 1) {
            // 最后一个子表达式，直接跳到 true 或 false
            builder.CreateCondBr(cond, trueBB, falseBB);
        } else {
            // 中间表达式：真则继续判断下一个子条件，假则直接跳 falseBB

            // 创建一个临时基本块，用于接下一个条件判断

            llvm::BasicBlock* nextBB = llvm::BasicBlock::Create(context, "land_next", func);
            std::cout << "----create a land_next----"  << std::endl;

            builder.CreateCondBr(cond, nextBB, falseBB);
            builder.SetInsertPoint(nextBB);

        }
    }
}

std::any Analysis::visitLOrExp(HelloParser::LOrExpContext *ctx) {
    std::cout << "visiting LOrExp"  << std::endl;
    llvm::Value* lhs = std::any_cast<llvm::Value*>(visit(ctx->lAndExp(0)));

    // 将 lhs 转换为 i1 类型
    lhs = builder.CreateICmpNE(lhs, llvm::ConstantInt::get(lhs->getType(), 0), "cmp_lhs");

    std::cout << "visited LOrExp - first lAndExp"  << std::endl;
    for (size_t i = 1; i < ctx->lAndExp().size(); ++i) {
        llvm::Value* rhs = std::any_cast<llvm::Value*>(visit(ctx->lAndExp(i)));
```

```
        // 将 rhs 转换为 i1 类型
        rhs = builder.CreateICmpNE(rhs, llvm::ConstantInt::get(rhs->getType(), 0), "cmp_rhs");

        lhs = builder.CreateOr(lhs, rhs, "or_tmp");
    }

    return lhs;
}

// 生成一个逻辑或表达式（LOrExp）的短路逻辑代码：a || b || c ... 形式
// 参数 trueBB: 一旦有一个为真就跳转的目标基本块
// 参数 falseBB: 所有都为假时跳转的目标基本块
void Analysis::genLOrExp(HelloParser::LOrExpContext* ctx, llvm::BasicBlock* trueBB, llvm::BasicBlock*
falseBB) {
    std::cout << "visiting LOrExp - Short-circuit Evaluation"  << std::endl;
    size_t n = ctx->lAndExp().size();  // n 为 LAndExp 的个数（LOrExp 是多个 LAndExp 用 '||' 连接而成）

    llvm::Function* func = builder.GetInsertBlock()->getParent();

    for (size_t i = 0; i < n; ++i) {
        std::cout << "-----entering genLOrExp for loop-----"  << std::endl;
        HelloParser::LAndExpContext* land = ctx->lAndExp(i);

        // 只有不是最后一个时才创建 lor_next
        llvm::BasicBlock* nextBB = nullptr;
        if (i != n - 1) {
            nextBB = llvm::BasicBlock::Create(context, "lor_next", func);
            std::cout << "----create a lor_next----" << std::endl;
        }

        genLAndExp(land, trueBB, (i == n - 1 ? falseBB : nextBB));

        if (i != n - 1) {
            builder.SetInsertPoint(nextBB);
        }
    }
}
```

我们构造了如下测试用例来验证短路求值：

```
int main() {
    int a, b, c, d, e;
    a = 0;
    b = 1;
    c = 1;
    d = 0;
    e = 0;
    if (a && b || c || d) {
        e = 4;
    } else {
        e = 5;
    }
```

```
    return e;
}
```

在该示例中，对于(a && b)部分，因为 a 为 0，所以不需要再判断 b 的值；对于(a && b || c || d)部分，因为(a && b)为假，所以需要继续判断 c 的值，而 c 的值为 1，所以不需要再判断 d 的值。我们生成的 IR 和运行结果如下：

```
; ModuleID = 'main'
source_filename = "main"
……
define i32 @main() {
entry:
  %a = alloca i32, align 4
  store i32 0, i32* %a, align 4
  %b = alloca i32, align 4
  store i32 0, i32* %b, align 4
  %c = alloca i32, align 4
  store i32 0, i32* %c, align 4
  %d = alloca i32, align 4
  store i32 0, i32* %d, align 4
  %e = alloca i32, align 4
  store i32 0, i32* %e, align 4
  %a_elem_ptr = getelementptr inbounds i32, i32* %a, i32 0
  store i32 0, i32* %a_elem_ptr, align 4
  %b_elem_ptr = getelementptr inbounds i32, i32* %b, i32 0
  store i32 1, i32* %b_elem_ptr, align 4
  %c_elem_ptr = getelementptr inbounds i32, i32* %c, i32 0
  store i32 1, i32* %c_elem_ptr, align 4
  %d_elem_ptr = getelementptr inbounds i32, i32* %d, i32 0
  store i32 0, i32* %d_elem_ptr, align 4
  %e_elem_ptr = getelementptr inbounds i32, i32* %e, i32 0
  store i32 0, i32* %e_elem_ptr, align 4
  %a_var = load i32, i32* %a, align 4
  %cmp_tmp = icmp ne i32 %a_var, 0
  br i1 %cmp_tmp, label %land_next, label %lor_next      ; 根据 a 的值，决定下一步判断 b 还是 c

then:                                             ; preds = %lor_next2, %lor_next, %land_next
  %e_elem_ptr5 = getelementptr inbounds i32, i32* %e, i32 0
  store i32 4, i32* %e_elem_ptr5, align 4
  br label %merge

else:                                             ; preds = %lor_next2
  %e_elem_ptr6 = getelementptr inbounds i32, i32* %e, i32 0
  store i32 5, i32* %e_elem_ptr6, align 4
  br label %merge

lor_next:                                         ; preds = %land_next, %entry
  %c_var = load i32, i32* %c, align 4
  %cmp_tmp3 = icmp ne i32 %c_var, 0
  br i1 %cmp_tmp3, label %then, label %lor_next2        ; 根据 c 的值，决定下一步判断 d 还是直接进入 thenBB
```

```
land_next:                                        ; preds = %entry
  %b_var = load i32, i32* %b, align 4
  %cmp_tmp1 = icmp ne i32 %b_var, 0
  br i1 %cmp_tmp1, label %then, label %lor_next          ; 根据 b 的值，决定下一步判断 c 还是直接进入 thenBB

lor_next2:                                        ; preds = %lor_next
  %d_var = load i32, i32* %d, align 4
  %cmp_tmp4 = icmp ne i32 %d_var, 0
  br i1 %cmp_tmp4, label %then, label %else              ; 根据 d 的值，决定进入 thenBB 还是 elseBB

merge:                                            ; preds = %else, %then
  %e_var = load i32, i32* %e, align 4
  ret i32 %e_var
}

运行结果：
编译 output.ll 为 output.o...
链接 output.o 和 libcact_rt.a...
运行程序...
程序退出码: 4
```

由 IR 可见，编译器确实根据表达式的左侧值判断是否继续求右侧值，避免不必要的求值与跳转，验证了短路求值的正确性。运行结果返回值为 4，符合预期逻辑，进一步说明短路控制流生成正确。

6. Lval 模块实现

Lval 的模块实现主要分成 2 中情况讨论，Lval 是常量或无下下标数组访问和是有下标的数组访问，其区分方式是按照 lval 是否含有 exp。需要区分的原因是如果是有下标的数组访问因为作为函数参数的数组退化过，所以需要多加载 1 次，这个靠符号表中 isArrayinFunc 字段来区分。

① Lval 是常量或无下下标数组访问

如果这个访问是非数组变量的访问那么只需要通过 CreateLoad 来加载它的值，如果是常量访问那么只需要返回其 irValue 即可，如果是一个数组变量，那么需要进行分类，如果 isArrayinFunc 为真即这个数组是函数的参数，那么只需要进行一次 load 操作即可获得指向这个数字的指针，如果字段不为真，那么需要获取数组第一个元素的指针，因此需要 GEP{0,0}，才能得到结果。具体代码如下。

```
    if (lValCtx->exp().empty()) {
        if(symOpt -> isArrayinFunc){
             llvm::Value* arrayPtr = builder.CreateLoad(basePtr->getType()->getPointerElementType(),
basePtr, "loaded_array_ptr");
           return arrayPtr;
```

```
        }else if(!symOpt->dimensions.empty()){
            llvm::Value* firstElemPtr = builder.CreateInBoundsGEP(
                baseTy,
                basePtr,
                { builder.getInt32(0) ,  // 先进到第 0 个数组
                    builder.getInt32(0) }, // 再进到第 0 个子数组的第 0 个元素
                baseName + "_decay"
            );
            return firstElemPtr;

        } else if (symOpt->isConst){
            std::cout<<"常量，直接返回其 Constant 值"<<std::endl;
            return symOpt->irValue;
        } else{
            std::cout<<"普通变量"<<std::endl;
            returnValue = builder.CreateLoad(symOpt->llvmType, symOpt->irValue, baseName +"_var");
            return returnValue;
        }

    }
```

② Lval 是有下标数组访问

首先需要维护一个类型为 llvm::Value*的索引数组 indices，这个数组是用来生成 GEP 指令的参数数组，在处理的过程中需要把每个维度的参数压入这个数组。

首先如果 isArrayinFunc 字段为假，那么说明数组不是函数的参数，则不需要进行多一步的 load 操作，但是需要要先想数组内压入 0 来进入这个数组。如果是函数的参数数组需要先把指针进行一次 load，相当于之前退化操作的反向，在这之后就不需要在想数组内压入 0 来进入这个数组了。

```
else {
        std::cout << "[Debug] lval is 数组"<<std::endl;
        std::vector<llvm::Value*> indices;
        Value* arrayPtr = nullptr;
        if(!symOpt->isArrayinFunc){
            //Value* arrayPtr = builder.CreateLoad(baseTy, basePtr, baseName + "_ptr");
            std::cout << "[Debug] basePtr Type: ";
            basePtr->getType()->print(llvm::outs());
            llvm::outs() << "\n";

            std::cout << "[Debug] basePtr Value: ";
            basePtr->print(llvm::outs());
            llvm::outs() << "\n";
            indices.push_back(builder.getInt32(0));
        }else{
                 arrayPtr = builder.CreateLoad(basePtr->getType()->getPointerElementType(), basePtr,
"loaded_array_ptr");
            std::cout << "[Debug] arrayPtr Type: ";
            arrayPtr->getType()->print(llvm::outs());
            llvm::outs() << "\n";
```

```
            std::cout << "[Debug] arrayPtr Value: ";
            arrayPtr->print(llvm::outs());
            llvm::outs() << "\n";
            //indices.push_back(builder.getInt32(0));
        }
```

下一步就是遍历每一个下标表达式把值压入这个数组中去，具体代码如下。

```
        // 遍历每一个下标表达式
        for (auto* expCtx : lValCtx->exp()) {
            llvm::Value* idx = std::any_cast<llvm::Value*>(visit(expCtx));
            llvm::Type* idxType = idx->getType()
            ;
            std::cout << "[Debug] Index "  <<std::endl;
            idxType->print(llvm::outs());
            llvm::outs() << "\n";

            indices.push_back(idx);
        }
        const auto& dims = symOpt->dimensions;
        std::cout<<"dims.size() is "<<dims.size()<<std::endl;
        std::cout<<"lValCtx->exp().size() is "<<lValCtx->exp().size()<<std::endl;
```

在这个操作之后还需要对不完全访问进行补0操作，比如访问二维数组a[i]而不是a[i][j]，那么要在 indices 补 0。在这之后就可以分类讨论如何 GEP 获得指针了。首先对于普通的数组变量，只需要直接进行 CreateInBoundsGEP 操作即可。对于函数参数数组来说，需要使用之前 load 出来的 arrayPtr 指针来进行 CreateInBoundsGEP。

```
        std::cout<<"going to ex llvm::Value* elemPtr = builder.CreateInBoundsGEP"<<std::endl;
        llvm::Value* elemPtr = nullptr;
        if(!symOpt->isArrayinFunc){
            if(symOpt->dimensions.size() && !(lValCtx->exp().size() == dims.size())){
                indices.push_back(builder.getInt32(0));
            }
            elemPtr = builder.CreateInBoundsGEP(
                basePtr->getType()->getPointerElementType(),
                basePtr, // 不应该是 load 出来的值，而是变量地址
                indices,
                baseName + "_elem_ptr"
            );

        }else{
            if(!(lValCtx->exp().size() == dims.size())){
                indices.push_back(builder.getInt32(0));
            }
            elemPtr = builder.CreateInBoundsGEP(
                arrayPtr->getType()->getPointerElementType(),
                arrayPtr, // 不应该是 load 出来的值，而是变量地址
                indices,
                baseName + "_elem_ptr"
```

```
            );
        }
        std::cout<<"elemPtr gepped"<<std::endl;
```

后面对 size_indices 的这些操作是为了获取准确的维度大小，这样在返回值的时候就可以进行分类讨论。返回的指针类型指向的还是数组，那么可以直接返回，如果指向的是数，那么还需要把数加载出来，返回加载出来数的指针。

```
        int size_indices = indices.size();
        if(symOpt->isArrayinFunc){
            size_indices++;
        }

        if(symOpt->dimensions.size() && !(lValCtx->exp().size() == dims.size())){
            size_indices--;
        }

        //const auto& dims = symOpt->dimensions;
        // 用正确的元素类型来创建 Load
        if (size_indices-1 < dims.size() ) {
            return elemPtr;
        } else {
            std::cout<<"返回的不是 elemPtr"<<std::endl;
            llvm::Value* returnValue =builder.CreateLoad(
                elemPtr->getType()->getPointerElementType(),
                elemPtr,
                baseName + "_val"
            );
            // 加载最终元素值
            return returnValue;
        }

    }
```

7. Number 模块实现

VisitNumber 模块我们改了好多遍，主要是因为一开始我们没有支持 char 类型，之后还有没有支持 char 类型是特殊符号的比如\n 这种，后来又改过是因为十六进制的支持不够全面。

VisitNumber 主要分为如下 3 种情况处理。首先是地府常量，在处理之前需要先考虑单引号内事 2 个字符的情况，比如'\n'。直接将 c 赋值为这两个字符组成的特殊字符，两一种情况就是正常情况，单引号内只有一个字符那么提取即可。子啊获得 c 之后就可以将其 ASCII 码进行返回。

```
std::any Analysis::visitNumber(HelloParser::NumberContext *ctx) {
    std::string text = ctx->getText();
    std::cout << "get const number: " << text << std::endl;
```

```
    // 1. 字符常量：形如 'a'（长度 3，首尾单引号）
    if (text.length() == 4 && text.front() == '\'' && text.back() == '\'') {
        char c;
        if(text[2] == 'n'){
            c = '\n';
        }else if(text[2] == 't'){
            c = '\t';
        }else if(text[2] == 'b'){
            c = '\b';
        }else if(text[2] == 'r'){
            c = '\r';
        }

        auto value = llvm::ConstantInt::get(
            llvm::Type::getInt8Ty(context),
            static_cast<uint8_t>(c),
            true  // 有符号
        );
        std::cout << "  构造出的 ConstantInt 字符值 (ASCII): " << static_cast<int>(c) << std::endl;
        return std::any(static_cast<llvm::Value*>(value));
    }

    if (text.length() == 3 && text.front() == '\'' && text.back() == '\'') {
        char c = text[1];  // 去掉两侧单引号之后的字符
        auto value = llvm::ConstantInt::get(
            llvm::Type::getInt8Ty(context),
            static_cast<uint8_t>(c),
            true  // 有符号
        );
        std::cout << "  构造出的 ConstantInt 字符值 (ASCII): " << static_cast<int>(c) << std::endl;
        return std::any(static_cast<llvm::Value*>(value));
    }

    // 2. 浮点数常量处理
    if (ctx->FloatConst()) {
        return parseFloatConstant(text);
    }

    // 3. 整数常量处理
    return parseIntConstant(text);
}
```

第二种情况是浮点数常量的处理。分为十进制和十六进制，代码比较简单，重点是需要去掉末尾的 f 或者 F，还有就是通过 0x 的开头来区分十进制和十六进制。具体代码如下。

```
llvm::Value* Analysis::parseFloatConstant(const std::string& text) {
    // 检查并处理后缀
    bool isFloat = false;
    bool isLongDouble = false;
    std::string numText = text;
```

```
    if (!text.empty()) {
        char last = text.back();
        if (last == 'f' || last == 'F') {
            isFloat = true;
            numText = text.substr(0, text.size() - 1);
        }
    }

    // 十六进制浮点数处理 (0x...p...)
    if (numText.size() > 2 && (numText[0] == '0' && (numText[1] == 'x' || numText[1] == 'X'))) {
        // 使用 strtod 处理十六进制浮点数
        char* endPtr;
        float floatVal = strtof(numText.c_str(), &endPtr);
        std::cout << "  解析为十六进制浮点数 (float): " << floatVal << std::endl;
        return llvm::ConstantFP::get(
            llvm::Type::getFloatTy(context),
            floatVal
        );
    }
    // 十进制浮点数处理
    else {
        char* endPtr;
        float floatVal = strtof(numText.c_str(), &endPtr);
        std::cout << "  解析为十进制浮点数 (float): " << floatVal << std::endl;
        return llvm::ConstantFP::get(
            llvm::Type::getFloatTy(context),
            floatVal
        );
    }
}
```

最后一种情况就是整数的处理了。首先十六进制的整数的区分方式是以"0x" 或 "0X" 开头，八进制是以“0”开头且长度大于 1，剩下的情况就都是十进制了。具体代码如下。

```
llvm::Value* Analysis::parseIntConstant(const std::string& text) {
    // 十六进制整数：以 "0x" 或 "0X" 开头
    if (text.size() > 2 && (text.rfind("0x", 0) == 0 || text.rfind("0X", 0) == 0)) {
        long long intVal = 0;
        try {
            intVal = std::stoll(text, nullptr, 16);
        } catch (...) {
            std::cerr << "[Error] 无法将 \"" << text << "\" 解析为十六进制整数! " << std::endl;
            exit(1);
        }
        auto value = llvm::ConstantInt::get(
            llvm::Type::getInt32Ty(context),
            intVal,
            true
        );
        std::cout << "  解析为十六进制整数: " << intVal << std::endl;
```

```
        return value;
    }

    // 八进制整数：以 '0' 开头且后续全部是 [0-7]
    if (text.size() > 1 && text[0] == '0') {
        bool allOctal = true;
        for (size_t i = 1; i < text.size(); ++i) {
            if (text[i] < '0' || text[i] > '7') {
                allOctal = false;
                break;
            }
        }
        if (allOctal) {
            long long intVal = 0;
            try {
                intVal = std::stoll(text, nullptr, 8);
            } catch (...) {
                std::cerr << "[Error] 无法将 \"" << text << "\" 解析为八进制整数！" << std::endl;
                exit(1);
            }
            auto value = llvm::ConstantInt::get(
                llvm::Type::getInt32Ty(context),
                intVal,
                true
            );
            std::cout << "  解析为八进制整数: " << intVal << std::endl;
            return value;
        }
    }

    // 其他情形当十进制整数处理
    long long intVal = 0;
    try {
        intVal = std::stoll(text, nullptr, 10);
    }
    auto value = llvm::ConstantInt::get(
        llvm::Type::getInt32Ty(context),
        intVal,
        true
    );
    std::cout << "  解析为十进制整数: " << intVal << std::endl;
    return value;
}
```

8. 内置函数实现方法（包括静态链接库生成、run_code.sh）

根据 CACT 语言规范，编译器需要内置以下 6 个输入/输出函数以支持常见数据的读取与打印：

- void print_int(int)
- void print_float(float)

- void print_char(char)
- int get_int()
- float get_float()
- char get_char()

这些函数应在全局作用域中声明，并在 LLVM IR 中作为外部函数链接调用。为此，我们在语义分析阶段，通过 initBuiltins()函数统一完成对这 6 个内置函数的声明，并将其注册进函数表，便于后续生成调用指令。

具体代码实现如下：

```
// 声明 CACT 内置函数
// 初始化 CACT 内置函数，在 LLVM 模块中声明它们，便于后续生成调用指令
void Analysis::initBuiltins() {
    // 定义 int、float、char、void 类型
    llvm::Type* intType = llvm::Type::getInt32Ty(context);
    llvm::Type* floatType = llvm::Type::getFloatTy(context);
    llvm::Type* charType = llvm::Type::getInt8Ty(context);
    llvm::Type* voidType = llvm::Type::getVoidTy(context);

    // void print_int(int)
    // 定义 printIntType 函数类型（返回值 void，参数 int）
    llvm::FunctionType* printIntType = llvm::FunctionType::get(voidType, {intType}, false);
    // 创建函数对象 print_int，并将其加入 LLVM 模块中
    llvm::Function* printIntFunc = llvm::Function::Create(printIntType, llvm::Function::ExternalLinkage,
"print_int", module);
    // 加入函数表，便于之后通过名字引用
    functionTable["print_int"] = printIntFunc;

    // void print_float(float)
    llvm::FunctionType* printFloatType = llvm::FunctionType::get(voidType, {floatType}, false);
                llvm::Function*     printFloatFunc     =     llvm::Function::Create(printFloatType,
llvm::Function::ExternalLinkage, "print_float", module);
    functionTable["print_float"] = printFloatFunc;

    llvm::FunctionType* printCharType = llvm::FunctionType::get(voidType, {charType}, false);
                 llvm::Function*     printCharFunc     =     llvm::Function::Create(printCharType,
llvm::Function::ExternalLinkage, "print_char", module);
    functionTable["print_char"] = printCharFunc;

    llvm::FunctionType* getIntType = llvm::FunctionType::get(intType, {}, false);// 无参数，返回 int
     llvm::Function* getIntFunc = llvm::Function::Create(getIntType, llvm::Function::ExternalLinkage,
"get_int", module);
    functionTable["get_int"] = getIntFunc;

    llvm::FunctionType* getFloatType = llvm::FunctionType::get(floatType, {}, false);
    llvm::Function* getFloatFunc = llvm::Function::Create(getFloatType, llvm::Function::ExternalLinkage,
"get_float", module);
    functionTable["get_float"] = getFloatFunc;
```

```
    llvm::FunctionType* getCharType = llvm::FunctionType::get(charType, {}, false);
    llvm::Function* getCharFunc = llvm::Function::Create(getCharType, llvm::Function::ExternalLinkage,
"get_char", module);
    functionTable["get_char"] = getCharFunc;
}
```

> *说明：这些函数被声明为 ExternalLinkage，意味着它们的实现不在当前 LLVM IR 文件中，而是由运行时静态链接库提供。*

实验中，为了使生成的 LLVM IR 能够正确调用这些输入输出函数并正常运行，我们实现了运行时支持库。它由头文件 cact_io.h 和源文件 cact_io.cpp 组成，实现上述内置函数的具体行为，并将其编译为静态链接库 libcact_rt.a，供链接使用。有了这个库，我们在测试编译器其它功能（例如数组访问、表达式计算等）时，可以调用内置函数来输出中间结果，验证 IR 的正确性。cact_io.h 文件和 cact_io.cpp 文件内容如下：

cact_io.h:

```
// cact_io.h
#ifndef CACT_IO_H
#define CACT_IO_H

#ifdef __cplusplus
extern "C" {
#endif

void print_int(int value);
void print_float(float value);
void print_char(char value);

int get_int();
float get_float();
char get_char();

#ifdef __cplusplus
}
#endif

#endif
```

功能说明：该头文件声明了 CACT 内置的输入输出函数，并通过 extern "C" 保证 C++ 实现的函数能被 LLVM 以 C 方式链接。

cact_io.cpp：

```
// cact_io.c
#include "cact_io.h"
#include <stdio.h>

void print_int(int value) {
```

```
    printf("%d", value);
}

void print_float(float value) {
    printf("%f", value);
}

void print_char(char value) {
    putchar(value);
}

int get_int() {
    int val;
    scanf("%d", &val);
    return val;
}

float get_float() {
    float val;
    scanf("%f", &val);
    return val;
}

char get_char() {
    char val;
    scanf(" %c", &val);
    return val;
}
```

功能说明：这些函数分别实现对 int、float、char 类型的打印与输出。

run_code.sh 脚本说明：

编写 run_code.sh 脚本用于自动化测试流程，即将生成的 LLVM IR 编译为目标文件，链接运行时库，并执行程序。我们的 run_code.sh 脚本具体内容如下：

```
#!/bin/bash

# 定义 LLVM 工具链路径
LLVM_BIN="/home/compiler10/llvm-install/clang+llvm-11.1.0-x86_64-linux-gnu-ubuntu-16.04/bin"

# 步骤 1: 将 LLVM IR 编译为目标文件
echo "编译 output.ll 为 output.o..."
"$LLVM_BIN"/clang -c output.ll -o output.o

# 步骤 2: 链接目标文件和运行时库，生成可执行文件
echo "链接 output.o 和 libcact_rt.a..."
"$LLVM_BIN"/clang output.o -L. -lcact_rt -o program

# 步骤 3: 运行程序并显示返回值
echo "运行程序..."
./program
```

```
EXIT_CODE=$?
echo "程序退出码: $EXIT_CODE"
```

说明：

第一步：使用 clang 将 LLVM IR(output.ll)编译为目标文件；

第二步：将目标文件与运行时静态库 libcact_rt.a 链接，生成最终可执行文件；

第三步：运行程序，输出运行结果与退出码。

四. Debug 过程

1. 函数数组形参的存储和使用（isArrayinFunc 字段）
2. 函数数组形参符号表的插入（dims 字段）

说明：由于服务器内容全部丢失无法完成前面这两点 Debug 的内容。

3. 短路求值生成 lor_next 空块问题

我们写了一个测试用例如下：

```
int foo(int d) {
    d = d + 1;
    return d;
}

int main() {
    int a, b, c, e;
    a = 1;
    b = 0;
    e = 2;
    if (a && b && foo(1)) {
        e = e + 1;
    } else {
        a = 2;
        e = e + 2;
    }
    return e;
}
```

运行后，得到的 IR 为：

```
; ModuleID = 'main'
source_filename = "main"
……
define i32 @foo(i32 %0) {
entry:
  %d = alloca i32, align 4
```

```
  store i32 %0, i32* %d, align 4
  %d_elem_ptr = getelementptr inbounds i32, i32* %d, i32 0
  %d_var = load i32, i32* %d, align 4
  %addtmp = add i32 %d_var, 1
  store i32 %addtmp, i32* %d_elem_ptr, align 4
  %d_var1 = load i32, i32* %d, align 4
  ret i32 %d_var1
}

define i32 @main() {
entry:
  %a = alloca i32, align 4
  store i32 0, i32* %a, align 4
  %b = alloca i32, align 4
  store i32 0, i32* %b, align 4
  %c = alloca i32, align 4
  store i32 0, i32* %c, align 4
  %e = alloca i32, align 4
  store i32 0, i32* %e, align 4
  %a_elem_ptr = getelementptr inbounds i32, i32* %a, i32 0
  store i32 1, i32* %a_elem_ptr, align 4
  %b_elem_ptr = getelementptr inbounds i32, i32* %b, i32 0
  store i32 0, i32* %b_elem_ptr, align 4
  %e_elem_ptr = getelementptr inbounds i32, i32* %e, i32 0
  store i32 2, i32* %e_elem_ptr, align 4
  %a_var = load i32, i32* %a, align 4
  %cmp_tmp = icmp ne i32 %a_var, 0
  br i1 %cmp_tmp, label %land_next, label %else

then:                                             ; preds = %land_next2
  %e_elem_ptr4 = getelementptr inbounds i32, i32* %e, i32 0
  %e_var = load i32, i32* %e, align 4
  %addtmp = add i32 %e_var, 1
  store i32 %addtmp, i32* %e_elem_ptr4, align 4
  br label %merge

else:                                             ; preds = %land_next2, %land_next, %entry
  %a_elem_ptr5 = getelementptr inbounds i32, i32* %a, i32 0
  store i32 2, i32* %a_elem_ptr5, align 4
  %e_elem_ptr6 = getelementptr inbounds i32, i32* %e, i32 0
  %e_var7 = load i32, i32* %e, align 4
  %addtmp8 = add i32 %e_var7, 2
  store i32 %addtmp8, i32* %e_elem_ptr6, align 4
  br label %merge

lor_next:                                         ; No predecessors!

land_next:                                        ; preds = %entry
  %b_var = load i32, i32* %b, align 4
  %cmp_tmp1 = icmp ne i32 %b_var, 0
  br i1 %cmp_tmp1, label %land_next2, label %else

land_next2:                                       ; preds = %land_next
```

```
  %calltmp = call i32 @foo(i32 1)
  %cmp_tmp3 = icmp ne i32 %calltmp, 0
  br i1 %cmp_tmp3, label %then, label %else

merge:                                            ; preds = %else, %then
  %e_var9 = load i32, i32* %e, align 4
  ret i32 %e_var9
}
```

并在执行时产生报错:

```
compiler10@teacher-PowerEdge-M640:~/cact/build$ ./run_code.sh
编译 output.ll 为 output.o...
output.ll:66:1: error: expected instruction opcode
land_next:                                        ; preds = %entry
^
1 error generated.
```

报错的原因是 lor_next 是一个空块，里面没有执行任何的操作，也没有任何跳转语句跳转到 lor_next 块，导致 LLVM IR 语法错误。仔细阅读生成的调试信息和 genLOrExp()函数的代码后，发现是因为我们在该函数中无条件创建了一个 lor_next 基本块，但是我们的测试用例中的 if 条件为"(a && b && foo(1))"，不包含任何逻辑或"||"表达式，但我们生成的 IR 中仍然创建了一个 lor_next 基本块，且该块没有任何前驱跳转或指令，导致 LLVM 报错。

为了避免这种无条件创建 lor_next 块导致 LLVM IR 语法错误的情况，我们对 genLOrExp()函数的创建 lor_next 块逻辑进行了修改：只有在当前处理的子表达式不是最后一个 LAndExp 的情况下，才创建 lor_next 块。相关代码如下：

```
        llvm::BasicBlock* nextBB = nullptr;
        if (i != n - 1) {
            nextBB = llvm::BasicBlock::Create(context, "lor_next", func);
            std::cout << "----create a lor_next----" << std::endl;
        }
```

修改后的生成的 IR 和运行结果如下：

```
; ModuleID = 'main'
source_filename = "main"
……
define i32 @foo(i32 %0) {
entry:
  %d = alloca i32, align 4
  store i32 %0, i32* %d, align 4
  %d_elem_ptr = getelementptr inbounds i32, i32* %d, i32 0
  %d_var = load i32, i32* %d, align 4
  %addtmp = add i32 %d_var, 1
  store i32 %addtmp, i32* %d_elem_ptr, align 4
  %d_var1 = load i32, i32* %d, align 4
```

```
  ret i32 %d_var1
}

define i32 @main() {
entry:
  %a = alloca i32, align 4
  store i32 0, i32* %a, align 4
  %b = alloca i32, align 4
  store i32 0, i32* %b, align 4
  %c = alloca i32, align 4
  store i32 0, i32* %c, align 4
  %e = alloca i32, align 4
  store i32 0, i32* %e, align 4
  %a_elem_ptr = getelementptr inbounds i32, i32* %a, i32 0
  store i32 1, i32* %a_elem_ptr, align 4
  %b_elem_ptr = getelementptr inbounds i32, i32* %b, i32 0
  store i32 0, i32* %b_elem_ptr, align 4
  %e_elem_ptr = getelementptr inbounds i32, i32* %e, i32 0
  store i32 2, i32* %e_elem_ptr, align 4
  %a_var = load i32, i32* %a, align 4
  %cmp_tmp = icmp ne i32 %a_var, 0
  br i1 %cmp_tmp, label %land_next, label %else

then:                                             ; preds = %land_next2
  %e_elem_ptr4 = getelementptr inbounds i32, i32* %e, i32 0
  %e_var = load i32, i32* %e, align 4
  %addtmp = add i32 %e_var, 1
  store i32 %addtmp, i32* %e_elem_ptr4, align 4
  br label %merge

else:                                             ; preds = %land_next2, %land_next, %entry
  %a_elem_ptr5 = getelementptr inbounds i32, i32* %a, i32 0
  store i32 2, i32* %a_elem_ptr5, align 4
  %e_elem_ptr6 = getelementptr inbounds i32, i32* %e, i32 0
  %e_var7 = load i32, i32* %e, align 4
  %addtmp8 = add i32 %e_var7, 2
  store i32 %addtmp8, i32* %e_elem_ptr6, align 4
  br label %merge

land_next:                                        ; preds = %entry
  %b_var = load i32, i32* %b, align 4
  %cmp_tmp1 = icmp ne i32 %b_var, 0
  br i1 %cmp_tmp1, label %land_next2, label %else

land_next2:                                       ; preds = %land_next
  %calltmp = call i32 @foo(i32 1)
  %cmp_tmp3 = icmp ne i32 %calltmp, 0
  br i1 %cmp_tmp3, label %then, label %else

merge:                                            ; preds = %else, %then
  %e_var9 = load i32, i32* %e, align 4
  ret i32 %e_var9
```

```
}
```

```
compiler10@teacher-PowerEdge-M640:~/cact/build$ ./run_code.sh
编译 output.ll 为 output.o...
链接 output.o 和 libcact_rt.a...
运行程序...
程序退出码: 4
```

可以看到，修改后生成 IR 时不再错误创建 lor_next 空块，生成的 IR 结构正确，LLVM 编译通过，程序运行结果为 4，符合预期。此问题说明，在 IR 生成阶段创建基本块时需要注意其控制流的连通性，避免生成 LLVM 无法接受的悬空块。

4. mergeBB 的创建条件（visitStmt 中的 IF_KW 部分）

我们写了一个测试用例如下：

```
int main(){
    int a, b;
    a = 1;
    b = 2;
    if( a > b ){
        return 8;
    } else {
        a = 2;
        return 9;
    }
}
```

运行后，得到的 IR 为：

```
; ModuleID = 'main'
source_filename = "main"

declare void @print_int(i32)

declare void @print_float(float)

declare void @print_char(i8)

declare i32 @get_int()

declare float @get_float()

declare i8 @get_char()

define i32 @main() {
entry:
  %a = alloca i32, align 4
  store i32 0, i32* %a, align 4
  %b = alloca i32, align 4
```

```
  store i32 0, i32* %b, align 4
  %a_elem_ptr = getelementptr inbounds i32, i32* %a, i32 0
  store i32 1, i32* %a_elem_ptr, align 4
  %b_elem_ptr = getelementptr inbounds i32, i32* %b, i32 0
  store i32 2, i32* %b_elem_ptr, align 4
  %a_var = load i32, i32* %a, align 4
  %b_var = load i32, i32* %b, align 4
  %igt_tmp = icmp sgt i32 %a_var, %b_var
  %cmp_tmp = icmp ne i1 %igt_tmp, false
  br i1 %cmp_tmp, label %then, label %else

then:                                             ; preds = %entry
  ret i32 8
  br label %merge

else:                                             ; preds = %entry
  %a_elem_ptr1 = getelementptr inbounds i32, i32* %a, i32 0
  store i32 2, i32* %a_elem_ptr1, align 4
  ret i32 9
  br label %merge

merge:                                            ; preds = %else, %then
}
```

并在执行时产生报错:

```
compiler10@teacher-PowerEdge-M640:~/cact/build$ ./run_code.sh
编译 output.ll 为 output.o...
output.ll:43:1: error: expected instruction opcode
}
^
1 error generated.
```

报错的原因是 merge 块是一个空块，里面没有执行任何操作，导致 LLVM IR 语法错误。仔细阅读 if-else 语句的代码后，发现是因为我们在代码中无条件创建了 merge 块，由于该测试用例中的 if 和 else 分支均以 return 语句终结，程序执行在条件语句后已无后续指令，因此不应再创建 merge 基本块。

未避免无操作的空 merge 块导致 LLVM IR 语法错误，我们对 mergeBB 的创建逻辑进行了修改。具体规则如下:

①若某个分支（then 或 else）未以终结指令结束，则该分支需跳转至 mergeBB;

②如果当前尚未创建 mergeBB，则在首次检测到需要跳转的分支时创建;

③使用布尔变量 needMerge 标记是否创建了 mergeBB，防止重复创建;

④所有未终结的分支都应跳转至 mergeBB;

⑤最后，若确实创建了 mergeBB，则将 IR 插入点设置至该块，供后续语句继续生成。

关键修改如下：

```
//llvm::BasicBlock* mergeBB = llvm::BasicBlock::Create(context, "merge", function);
//builder.CreateBr(mergeBB);  // 无条件跳转到 merge 块
// 只有在没有 terminator 时才跳转到 merge 块
if (!builder.GetInsertBlock()->getTerminator()) {
    std::cout << "visiting IF - thenBB jump to mergeBB"  << std::endl;
    //if (!mergeBB && needMerge == false) {
    if (needMerge == false) {
        mergeBB = llvm::BasicBlock::Create(context, "merge", function);
        std::cout << "visiting IF - create mergeBB"  << std::endl;
    }
    builder.CreateBr(mergeBB);
    needMerge = true;
}

……

// 设置插入点到 mergeBB，供后续代码插入
//builder.SetInsertPoint(mergeBB);
// 如果至少有一个分支没有终结，就设置 mergeBB
//if (needMerge && mergeBB) {
if (needMerge) {
    std::cout << "visiting IF - set insert point to mergeBB"  << std::endl;
    builder.SetInsertPoint(mergeBB);
}
```

修改后的运行结果：

有无 return	生成的 IR（只附上关键片段）	代码段

if 有
else 有
	……
	br i1 %cmp_tmp, label %then, label %else
	
	then: ; preds = %entry
	ret i32 1
	
	else: ; preds = %entry
	%a_elem_ptr1 = getelementptr inbounds i32, i32* %a, i32 0
	store i32 2, i32* %a_elem_ptr1, align 4
	ret i32 2
		if(a > b){
		return 1;
		} else {
		a = 2;
		return 2;
		}
		//return 3;

if 无
else 无
	……
	br i1 %cmp_tmp, label %then, label %else
	
	then: ; preds = %entry
	br label %merge
	
	else: ; preds = %entry
	%a_elem_ptr1 = getelementptr inbounds i32, i32* %a, i32 0
	store i32 2, i32* %a_elem_ptr1, align 4
	br label %merge
		if(a > b){
		//return 1;
		} else {
		a = 2;
		//return 2;
		}
		return 3;

	merge: ; preds = %else, %then ret i32 3	

if 有
else 无
	……
	  br i1 %cmp_tmp, label %then, label %else
	
	then: ; preds = %entry
	  ret i32 1
	
	else: ; preds = %entry
	  %a_elem_ptr1 = getelementptr inbounds i32, i32* %a, i32 0
	  store i32 2, i32* %a_elem_ptr1, align 4
	  br label %merge
	
	merge: ; preds = %else
	  ret i32 3
		if(a > b){
		  return 1;
		} else {
		  a = 2;
		  //return 2;
		}
		return 3;

if 无
else 有
	……
	  br label %merge
	
	else: ; preds = %entry
	  %a_elem_ptr1 = getelementptr inbounds i32, i32* %a, i32 0
	  store i32 2, i32* %a_elem_ptr1, align 4
	  ret i32 2
	
	merge: ; preds = %then
	  ret i32 3
		if(a > b){
		  //return 1;
		} else {
		  a = 2;
		  return 2;
		}
		return 3;

五. 实验结果

运行./build.sh，编译成功:

```
compiler10@teacher-PowerEdge-M640:~/cact/build$ ./build.sh
切换到 grammar 目录...
运行 ANTLR 生成 C++ 代码...
warning(154): Hello.g4:35:0: rule funcDef contains an optional block with at least one alternative that can match an empty string
切换到 build 目录...
运行 CMake...
-- LLVM_DIR =
/home/compiler10/llvm-install/clang+llvm-11.1.0-x86_64-linux-gnu-ubuntu-16.04/lib/cmake/llvm
-- Found LLVM 11.1.0
-- Configuring done
-- Generating done
-- Build files have been written to: /home/compiler10/cact/build
开始编译...
Consolidate compiler generated dependencies of target antlr
[ 93%] Built target antlr
```

```
Consolidate compiler generated dependencies of target compiler
[ 94%] Building CXX object CMakeFiles/compiler.dir/grammar/HelloBaseVisitor.cpp.o
[ 94%] Building CXX object CMakeFiles/compiler.dir/grammar/HelloLexer.cpp.o
[ 95%] Building CXX object CMakeFiles/compiler.dir/grammar/HelloParser.cpp.o
[ 96%] Building CXX object CMakeFiles/compiler.dir/grammar/HelloVisitor.cpp.o
[ 96%] Building CXX object CMakeFiles/compiler.dir/src/Analysis.cpp.o
[ 97%] Building CXX object CMakeFiles/compiler.dir/src/main.cpp.o
[ 98%] Linking CXX executable compiler
[100%] Built target compiler
编译完成!
```

构造测试用例 test.cact：

```
int a = 0, b = 1;

int foo(int a, int b)
{
     return a * a;
}

int main()
{
    int c;
    c = foo(a, b);
    print_int(c);
    c = c + 3;
    return c;
}
```

运行测试用例，生成 IR 并写入 output.ll 文件中。可以正确生成 IR：

```
compiler10@teacher-PowerEdge-M640:~/cact/build$ ./compiler ../test/samples_lex_and_syntax/test.cact
-------
(compUnit (decl (vardecl (bType int) (varDef a = (constInitVal (constExp (number 0)))) , (varDef b =
(constInitVal (constExp (number 1)))) ;)) (funcDef (funcType int) foo ( (funcFParams (funcFParam (bType
int) a) , (funcFParam (bType int) b)) ) (block { (blockItem (stmt return (exp (addExp (mulExp (unaryExp
(primaryExp (lVal a))) * (unaryExp (primaryExp (lVal a)))))) ;)) })) (funcDef (funcType int) main
( funcFParams ) (block { (blockItem (decl (vardecl (bType int) (varDef c) ;))) (blockItem (stmt (exp (addExp
(mulExp (unaryExp (primaryExp (lVal c)))))) = (exp (addExp (mulExp (unaryExp foo ( (funcRParams (exp (addExp
(mulExp (unaryExp (primaryExp (lVal a)))))) , (exp (addExp (mulExp (unaryExp (primaryExp (lVal
b)))))))) ))))) ;)) (blockItem (stmt return (exp (addExp (mulExp (unaryExp (primaryExp (lVal c)))))) ;)) }))
<EOF>)

……（打印的调试信息，此处省略）

IR 结果内容：; ModuleID = 'main'
source_filename = "main"

@a = global i32 0
@b = global i32 1
```

```
declare void @print_int(i32)

declare void @print_float(float)

declare void @print_char(i8)

declare i32 @get_int()

declare float @get_float()

declare i8 @get_char()

define i32 @foo(i32 %0, i32 %1) {
entry:
  %a = alloca i32, align 4
  store i32 %0, i32* %a, align 4
  %b = alloca i32, align 4
  store i32 %1, i32* %b, align 4
  %a_var = load i32, i32* %a, align 4
  %a_var1 = load i32, i32* %a, align 4
  %multmp = mul i32 %a_var, %a_var1
  ret i32 %multmp
}

define i32 @main() {
entry:
  %c = alloca i32, align 4
  store i32 0, i32* %c, align 4
  %c_elem_ptr = getelementptr inbounds i32, i32* %c, i32 0
  %a_var = load i32, i32* @a, align 4
  %b_var = load i32, i32* @b, align 4
  %calltmp = call i32 @foo(i32 %a_var, i32 %b_var)
  store i32 %calltmp, i32* %c_elem_ptr, align 4
  %c_var = load i32, i32* %c, align 4
  call void @print_int(i32 %c_var)
  %c_elem_ptr1 = getelementptr inbounds i32, i32* %c, i32 0
  %c_var2 = load i32, i32* %c, align 4
  %addtmp = add i32 %c_var2, 3
  store i32 %addtmp, i32* %c_elem_ptr1, align 4
  %c_var3 = load i32, i32* %c, align 4
  ret i32 %c_var3
}
```

运行./run_code.sh，进行编译 output.ll 并链接静态库，得到正确的运行结果：

```
compiler10@teacher-PowerEdge-M640:~/cact/build$ ./run_code.sh
编译 output.ll 为 output.o...
warning: overriding the module target triple with x86_64-unknown-linux-gnu [-Woverride-module]
1 warning generated.
链接 output.o 和 libcact_rt.a...
运行程序...
```

```
0 程序退出码: 3
```

运行结果为“0 程序退出码: 3”，其中，“0”表示调用 print_int(c)时输出的 c 的值为 0；“程序退出码: 3”表示程序最终的返回值，即 c 的值为 3。

综上所示，编译器成功完成了词法分析、语法分析、语义检查，并正确生成 LLVM IR 及可执行文件。通过运行测试用例验证，程序行为与预期一致，说明实验目标已经较好实现，编译器整体功能基本完备。

六. 实验总结

通过本次实验，我们系统掌握了中间代码生成阶段的关键流程，深入理解了从抽象语法树（AST）到 LLVM IR 的转换机制。实验过程中，我们以最基本的 CACT 程序为起点，逐步扩展支持变量和常量声明初始化、函数定义、控制流语句以及表达式求值等语义结构，完成了对大部分语言特性的中间代码生成支持。

在代码实现方面，我们设计并完善了符号表和函数表的组织结构，以支持嵌套作用域的精确管理和函数参数的正确解析。为了正确处理数组变量和函数数组参数，我们额外引入了 isArrayinFunc 等辅助字段，并在生成 GEP 指令时特别处理了多维数组的维度信息与索引展开逻辑。在控制流相关模块中，我们通过手动创建基本块并合理安排跳转指令，成功实现了 if-else、while、break、continue 等语句的控制流转换，同时在短路求值中引入 genCond 系列函数，实现了布尔表达式的精确跳转行为，有效避免了空块或冗余块的出现。

调试过程中，我们逐步修复了包括 mergeBB 创建条件、函数数组形参的存储和使用、lor_next 空块生成等在内的多项潜在问题，通过不断的样例验证与测试驱动，逐步增强了中间代码生成模块的健壮性与通用性。我们还基于 LLVM 工具链构建了完整的运行时支持系统，包括静态链接库和自动化编译脚本，从而实现了从源代码到可执行文件的完整编译流程。

总体而言，这次实验不锻炼了我们在面对复杂结构时的模块设计与调试能力。通过亲手实现从 AST 到 IR 的转换，我们更深刻地体会到中间代码在编译器架构中的桥梁作用，为后续的目标代码生成与优化奠定了坚实基础。