实验报告

任务说明

- 完成中间代码的生成

成员组成

艾华春，李霄宇，王淼

设计中间代码生成的目录结构

```
src/
├── Analysis.cpp        # 负责遍历语法树并生成IR
├── Analysis.h          # 定义语法树遍历和语义分析的逻辑
├── IRGenerator.cpp     # 实现 LLVM IR 的生成逻辑
├── IRGenerator.h       # 定义生成 LLVM IR 的相关数据结构和方法
├── SymbolTable.cpp     # 实现符号表的相关操作，包括符号的定义和查找
├── SymbolTable.h       # 定义符号表及相关符号信息的管理，负责变量、函数等符号的作用域管理
├── Types.h             # 定义了编译器中与类型相关的核心数据结构
└── main.cpp            # 编译器的入口文件，负责解析输入文件并生成中间代码
```

实验实现

中间代码生成的核心实现在Analysis.cpp文件中，它通过访问语法树节点生成类似LLVM IR的中间代码。

入口函数

在函数visitCompUnit()中，创建全局符号表并遍历所有声明和函数定义，最后检查main函数是否存在。

```
    currentSymbolTable = new SymbolTable(nullptr);
    isGlobal = true;
    addBuiltinFunc();
    visitChildren(context);
    // check if the main function is defined
    if (!currentSymbolTable->lookupInCurrentScope("main", true /*is function*/)) {
        std::cerr << "Error: main function not defined!" << std::endl;
        exit(EXIT_FAILURE);
    }
    return 0;
```

变量声明处理

1.常量声明

在函数visitConstDef()中，分别针对全局常量和局部常量两种情况做声名处理。
获取初始值后做类型检查，生成符号后全局常量直接生成IR指令。

```
        // get the initial value
        auto initval = std::any_cast<LLVMValue>(visitConstInitVal(context->constInitVal()));
        if (initval.type.baseType != currentBType) {
            std::cerr << "Error: Type mismatch in constant declaration! Expected " << TypeToLLVM(currentT) << ", got " << TypeToLLVM(initval.type) << std::endl;
            exit(EXIT_FAILURE);
        }
        // add to symbol table
        std::string ssa = "@" + newSSA(ident);
        currentSymbolTable->define(Symbol(ident, currentT, ssa, initval.name));
        // gen llvm code
        LLVMGlobalVar globalVar(ssa, currentT, initval.name, true /* is Const*/);
        llvmmodule.addGlobalVar(globalVar);
```

处理局部常量时需要显示的内存分配，值存储在栈上。

```
    // get the initial value
        auto initval = std::any_cast<LLVMValue> (visitConstInitVal(context->constInitVal()));
        if (initval.type.baseType != currentBType) {
            std::cerr << "Error: Type mismatch in constant declaration! Expected " << TypeToLLVM(currentT) << ", got " << TypeToLLVM(initval.type) << std::endl;
            exit(EXIT_FAILURE);
        }
        // add to symbol table
        std::string ssa = "%" + newSSA(ident);
        currentSymbolTable->define(Symbol(ident, currentT, ssa, initval.name));
        // alocate memory
        std::stringstream ss;
        ss << ssa << " = alloca " << TypeToLLVM(currentT);
        currentBlock->addInstruction(ss.str());
        ss.str("");
        if (!dimSize.empty()) {
            std::string globalid = "@" + newSSA("const_" + ident);
            LLVMGlobalVar globalVar(globalid, currentT, initval.name, true /* is Const*/);
            llvmmodule.addGlobalVar(globalVar);
            std::string identcast = "%" + newSSA("cast_i8_" + ident);
            ss << identcast << " = bitcast " << TypeToLLVM(currentT) << "* " << ssa << " to i8*";
            currentBlock->addInstruction(ss.str());
            ss << "";
            std::string globalcast = "%" + newSSA("cast_i8_global");
            ss << globalcast << " = bitcast " << TypeToLLVM(currentT) << "* " << globalid << " to i8*";
            currentBlock->addInstruction(ss.str());
            ss << "";
```

```
            ss << "call void @llvm.memcpy.p0i8.p0i8.i32(i8* " << identcast
<< ", i8* " << globalcast << ", i32 " << currentT.getArraySize() << ", i1
false)";
            currentBlock->addInstruction(ss.str());
        } else {
            ss << "store " << TypeToLLVM(initval.type) << " " <<
initval.name << ", " << TypeToLLVM(currentBType) << "* " << ssa;
            currentBlock->addInstruction(ss.str());
        }
```

2.变量声明

处理变量的声明在函数visitVarDef()，同样分为全局变量和局部变量两种情况，思想类似于处理常量。

3.未初始化变量

对于未初始化的变量，在函数visitConstInitVal()中用zeroinitializer做处理。

```
std::vector<int> dimSize = { 0 };
return LLVMValue("zeroinitializer", VarType(currentBType, false /*is Const*/,
false /*not function*/, dimSize));
```

函数定义处理

在函数visitFuncDef()中，访问函数体，先处理参数，再生成函数体IR。

```
    // visit the function body
    LLVMBasicBlock *block = new LLVMBasicBlock("entry");
    function->addBasicBlock(block);
    currentBlock = block;
    isGlobal = false;
    currentSymbolTable = new SymbolTable(currentSymbolTable);
    /* load params */
    for (size_t i = 0; i < parameters.size(); ++i) {
        if (parameters[i].type.isArray()) {
            currentSymbolTable->define(Symbol(parameters[i].name,
parameters[i].type, "%" + parameters[i].name));
        } else {
            std::stringstream ss;
            std::string ssa = "%" + newSSA(parameters[i].name + "_local");
            ss << ssa << " = alloca " << TypeToLLVM(parameters[i].type);
            currentBlock->addInstruction(ss.str());
            ss.str("");
            ss << "store " << TypeToLLVM(parameters[i].type) << " %" <<
parameters[i].name << ", " << TypeToLLVM(parameters[i].type) << "* " << ssa;
            currentBlock->addInstruction(ss.str());
            currentSymbolTable->define(Symbol(parameters[i].name,
parameters[i].type, ssa));
        }
    }
    visitBlock(context->block());
    if (currentBlock->instructions.empty() || currentBlock-
>instructions.back().find("ret") == std::string::npos) {
        // if the function does not return, add a return instruction
        if (retBT == BaseType::VOID) {
```

```
            currentBlock->addInstruction("ret void");
        } else {
            currentBlock->addInstruction("ret " + BTypeToLLVM(retBT) + "
0");
        }
    }
    // visit the body end
    isGlobal = true;
    currentSymbolTable = currentSymbolTable->getParent();
    llvmmodule.addFunction(*function);
```

控制流语句

控制流语句的处理在函数visitStmt()当中。

1.条件语句

处理IF类条件语句时，先创建基本快标签。

```
        std::string thenLabel = newLabel("then");
        std::string elseifLabel = context->elseIFStmt().empty() ? "" :
newLabel("elseif");
        std::string elseLabel = context->elseStmt() ? newLabel("else") : "";
        std::string endLabel = newLabel("ifend");
```

再处理条件跳转，并分别生成then块、elseif then块、else块的代码。

```
         currentBlock->addInstruction(ss.str());
         // then
         LLVMBasicBlock *thenBlock = new LLVMBasicBlock(thenLabel);
         currentBlock = thenBlock;
         visitStmt(context->stmt());
         currentBlock->addInstruction("br label %" + endLabel);
         currentFunction->addBasicBlock(thenBlock);

         // else if
         for (int i = 0; i < context->elseIFStmt().size(); i++) {
             std::string nextLabel;
             if (i < context->elseIFStmt().size() - 1) {
                 nextLabel = newLabel("elseif_next");
             } else if (context->elseStmt()) {
                 nextLabel = elseLabel;
             } else {
                 nextLabel = endLabel;
             }

             LLVMBasicBlock *elseifBlock = new LLVMBasicBlock(elseifLabel);
             currentBlock = elseifBlock;
             currentFunction->addBasicBlock(elseifBlock);
             std::string elseifCond = std::any_cast<std::string>
(visitCond(context->elseIFStmt(i)->cond()));
             std::stringstream elseifSS;
             std::string elsethenLabel = newLabel("elseif_then");
             elseifSS << "br i1 " << elseifCond << ", label %" <<
elsethenLabel << ", label %" << nextLabel;
```

```
            currentBlock->addInstruction(elseifSS.str());
            // elseif then
            LLVMBasicBlock *elseifThenBlock = new
LLVMBasicBlock(elsethenLabel);
            currentBlock = elseifThenBlock;
            visitStmt(context->elseIFStmt(i)->stmt());
            currentBlock->addInstruction("br label %" + endLabel);
            currentFunction->addBasicBlock(elseifThenBlock);
            // update elseifLabel for next iteration
            if (i < context->elseIFStmt().size() - 1) {
                elseifLabel = newLabel("elseif");
            }
        }

        // else
        if (context->elseStmt()) {
            LLVMBasicBlock *elseBlock = new LLVMBasicBlock(elseLabel);
            currentBlock = elseBlock;
            currentFunction->addBasicBlock(elseBlock);
            visitStmt(context->elseStmt()->stmt());
            currentBlock->addInstruction("br label %" + endLabel);
        }

        // end
        LLVMBasicBlock *endBlock = new LLVMBasicBlock(endLabel);
        currentBlock = endBlock;
        currentFunction->addBasicBlock(endBlock);
```

2.循环语句

处理WHLIE类循环语句时，在判断处、循环体处、结尾处各添加标签，再利用这些标签做处理。

```
        // cond
        LLVMBasicBlock *condBlock = new LLVMBasicBlock(condLabel);
        currentFunction->addBasicBlock(condBlock);
        currentBlock->addInstruction("br label %" + condLabel);
        currentBlock = condBlock;
        std::string cond = std::any_cast<std::string>(visitCond(context-
>cond()));
        std::stringstream ss;
        ss << "br i1 " << cond << ", label %" << bodyLabel << ", label %" <<
endLabel;
        currentBlock->addInstruction(ss.str());
        // body
        LLVMBasicBlock *bodyBlock = new LLVMBasicBlock(bodyLabel);
        currentFunction->addBasicBlock(bodyBlock);
        currentBlock = bodyBlock;
        visitStmt(context->stmt());
        currentBlock->addInstruction("br label %" + condLabel);
        // end
        LLVMBasicBlock *endBlock = new LLVMBasicBlock(endLabel);
        currentFunction->addBasicBlock(endBlock);
        currentBlock = endBlock;
        curEndLabel = "";
        curCondLabel = "";
```

表达式处理

以函数visitAddExp()为例，如果表达式只有一个操作数则直接返回，否则遍历后续操作数，同时做类型检查确保左右操作数类型一致，再根据运算符类型生成IR，最后将当前结果作为下一次操作的左操作数。

```
    for (int i = 1; i < context->mulExp().size(); i++) {
        std::stringstream ss;
        auto right = std::any_cast<LLVMValue>(visitMulExp(context->mulExp(i)));
        if (left.type.baseType != right.type.baseType) {
            std::cerr << "Error: Type mismatch in addition! Expected " << TypeToLLVM(left.type) << ", got " << TypeToLLVM(right.type) << std::endl;
            exit(EXIT_FAILURE);
        }
        sum.name = "%" + newSSA("sum");
        if (context->addOp(i - 1)->PLUS()) {
            if (left.type.baseType == BaseType::FLOAT || left.type.baseType == BaseType::DOUBLE) {
                ss << sum.name << " = fadd " << TypeToLLVM(left.type) << " " << left.name << ", " << right.name;
            } else {
                ss << sum.name << " = add " << TypeToLLVM(left.type) << " " << left.name << ", " << right.name;
            }
        } else if (context->addOp(i - 1)->MINUS()) {
            if (left.type.baseType == BaseType::FLOAT || left.type.baseType == BaseType::DOUBLE) {
                ss << sum.name << " = fsub " << TypeToLLVM(left.type) << " " << left.name << ", " << right.name;
            } else {
                ss << sum.name << " = sub " << TypeToLLVM(left.type) << " " << left.name << ", " << right.name;
            }
        }
        currentBlock->addInstruction(ss.str());
        left = sum;
    }
```

数组访问处理

先查找标识符，检查常量数组的写操作违规，并检查索引类型。
再使用getelementptr指令生成数组元素的偏移地址。对于多维数组，按索引逐层计算偏移量。如果数组的第一维是动态大小（dimSizes[0] == -1），需要额外处理基地址。

```
        std::string ptr = "%" + newSSA("ptr");
        std::stringstream ss;
        if (s->type.dimSizes[0] == -1) {
            std::string type = TypeToLLVM(s->type);
            type.pop_back(); // delete *
            ss << ptr << " = getelementptr inbounds " << type << ", " << "ptr " << identssa;
            for (const auto &idx : index) {
                ss << ", i32 " << idx; // add index
```

```
            }
        } else {
            ss << ptr << " = getelementptr inbounds " << TypeToLLVM(s->type)
<< ", " << TypeToLLVM(s->type) << "* " << identssa;
            ss << ", i32 0"; // base address
            for (const auto &idx : index) {
                ss << ", i32 " << idx; // add index
            }
        }
        currentBlock->addInstruction(ss.str());
```

函数调用

在函数visitUnaryExp()中，根据函数返回类型生成IR。

```
        if (s->type.baseType == BaseType::VOID) {
            funcret = "";
            ss << "call void @" << ident << "(";
        } else {
            funcret = "%" + newSSA("ret");
            ss << funcret << " = call " << TypeToLLVM(s->type) << " @" <<
ident << "(";
        }
```

符号表

SymbolTable.h 和 SymbolTable.cpp (符号表)

- 目的：管理程序中的标识符（变量、函数、常量）及其属性（类型、作用域、SSA名称等）。
- 核心数据结构：
  - Symbol 结构体：存储单个符号的详细信息。
    - name (string): 符号名称。
    - type (VarType): 符号类型（包括基本类型、数组维度、是否常量等）。
    - ssa (string): 符号的SSA（静态单赋值）形式名称，用于LLVM IR。
    - constvalue (string): 如果是常量，存储其值。
    - params (vector<LLVMValue*>): 如果是函数，存储其参数列表。
  - SymbolTable 类：实现符号表的查找、定义、作用域管理。
    - var_table (unordered_map): 存储变量和常量。
    - func_table (unordered_map): 存储函数。
    - parent (SymbolTable*): 指向父作用域的指针，用于实现嵌套作用域。
- 主要功能：
  - define(const Symbol &symbol): 在当前作用域定义一个新符号。
  - lookup(const std::string &name, bool isFunction): 查找符号，会沿着作用域链向上查找。
  - lookupInCurrentScope(const std::string &name, bool isFunction): 仅在当前作用域查找符号。
  - getParent(): 获取父作用域。
- 设计特点：

    - 分层作用域：通过 parent 指针实现嵌套作用域，符合C语言的作用域规则。
    - SSA支持：ssa 字段直接关联到LLVM IR的变量命名。
    - 类型系统集成：VarType 紧密集成，用于类型检查和IR生成

IR生成器

IRGenerator.h 和 IRGenerator.cpp (LLVM IR 生成器)

- 目的：将在遍历语法分析树时生成的LLVM IR按层次管理, 通过to_string()方法可以生成完整CACT程序的LLVM IR。
- 核心数据结构：
    - LLVMValue 结构体 (应在 Types.h 或类似文件中定义)：表示LLVM IR中的一个值（变量、常量、指令结果）。
        - name (string): 值的名称（如 %1, @gvar）。
        - type (VarType): 值的类型。
    - LLVMBasicBlock 类：表示LLVM IR中的一个基本块。
        - name (string): 基本块的标签名称。
        - instructions (vector): 基本块内的指令列表。
    - LLVMFunction 类：表示LLVM IR中的一个函数。
        - name (string): 函数名。
        - returnType (string): 返回类型的LLVM表示。
        - parameters (vector): 参数列表。
        - basicblocks (vector<LLVMBasicBlock*>): 函数内的基本块列表。
    - LLVMModule 类：表示一个LLVM模块（编译单元）。
        - functions (vector<LLVMFunction*>): 模块内的函数列表。
        - globalVars (vector): 全局变量列表。
- 主要功能：
    - 基本块管理：创建、添加和组织基本块。
    - 函数管理：构建函数定义，包括参数、返回类型和基本块。
    - 模块管理：将函数、全局变量等组合成一个完整的LLVM模块。
- 设计特点：
    - 面向对象：将LLVM的各个概念（模块、函数、基本块、值）封装成类，结构清晰。