实验二：LLVM IR 中间代码

组别：12 组
姓名：吴亚晨 祝田田 全荟霖

1、 实验内容

实验目标：

1. 实现从语法树到 LLVM IR 的转换逻辑
2. 支持基本语言特性：
   变量声明与初始化
   算术运算（+、-、*、/、%）
   控制流语句（if-else、while 循环）
   函数定义与调用
   返回语句
3. 实现符号表管理机制
4. 生成符合 LLVM 规范的 IR 代码
5. 处理 SSA（静态单赋值）形式的中间表示

核心实现要求：

1. 变量存储模型：使用 **alloca/load/store** 实现变量存储
2. 控制流结构：使用基本块和标签实现分支和循环
3. 函数调用约定：遵循标准函数调用规范
4. 临时变量管理：自动生成 SSA 形式的临时变量

2、 实验过程

1. 核心数据结构与初始化

实现分析：使用 stringstream 累积生成的 IR 代码，符号表采用 unordered_map 实现变量名到内存地址的映射，newTemp()自动生成SSA形式的临时变量（如%1, %2），newLabel()生成唯一标签用于控制流。

```
class IRGeneratorVisitor : public HelloBaseVisitor {
private:
    std::stringstream ir;  // IR代码输出流
    int tempId;            // 临时变量计数器
    int labelId;           // 标签计数器
    std::unordered_map<std::string, std::string> symbolTable;  // 符号表

public:
    IRGeneratorVisitor() : tempId(1), labelId(0) {}

    // 生成新的临时变量（SSA形式）
    std::string newTemp() {
        return "%" + std::to_string(tempId++);
    }

    // 生成新的标签
    std::string newLabel() {
        return "label" + std::to_string(labelId++);
    }
};
```

2. 函数定义处理

1．函数签名生成：define dso_local i32 @funcName(...)
2. 参数处理：为每个参数创建栈空间
3. 内存模型：alloca 在栈上分配空间，store 将参数值存入分配的地址
4. 符号表管理：记录变量名→内存地址的映射

```cpp
std::any visitFuncDef(HelloParser::FuncDefContext* ctx) {
    std::string funcName = ctx->Ident()->getText();
    // 处理参数列表
    std::vector<std::string> paramNames;
    if (ctx->funcFParams()) {
        for (auto param : ctx->funcFParams()->funcFParam()) {
            paramNames.push_back(param->Ident()->getText());
        }
    }

    // 生成函数头
    ir << "define dso_local i32 @" << funcName << "(";
    for (size_t i = 0; i < paramNames.size(); ++i) {
        if (i > 0) ir << ", ";
        ir << "i32 %" << paramNames[i];  // 所有参数均为i32类型
    }
    ir << ") #0 {\nentry:\n";

    // 为参数分配栈空间
    for (const auto& name : paramNames) {
        std::string varPtr = newTemp();
        symbolTable[name] = varPtr;  // 注册到符号表
        ir << "  " << varPtr << " = alloca i32, align 4\n";
        ir << "  store i32 %" << name << ", i32* " << varPtr << "\n";
    }

    // 处理函数体
    visit(ctx->block());

    // 默认返回值
    ir << "  ret i32 0\n}\n\n";
    return nullptr;
}
```

3. **算术表达式处理**

   递归下降遍历表达式树，为每个操作生成新的临时变量（SSA）

   支持的操作：加法：**add** 减法：**sub** 乘法：**mul** 除法：**sdiv** 取模：**srem**

```
// 加法/减法表达式
std::any visitAddExp(HelloParser::AddExpContext* ctx) {
    if (ctx->addExp()) {
        std::string left = std::any_cast<std::string>(visit(ctx->addExp()));
        std::string right = std::any_cast<std::string>(visit(ctx->mulExp()));
        std::string result = newTemp();  // 新的SSA临时变量

        // 选择操作符
        std::string op = ctx->PLUS() ? "add" : "sub";
        ir << "  " << result << " = " << op << " i32 " << left << ", " << right << "\n";
        return result;
    }
    return visit(ctx->mulExp());
}

// 乘法/除法表达式
std::any visitMulExp(HelloParser::MulExpContext* ctx) {
    if (ctx->mulExp()) {
        std::string left = std::any_cast<std::string>(visit(ctx->mulExp()));
        std::string right = std::any_cast<std::string>(visit(ctx->unaryExp()));
        std::string result = newTemp();

        // 选择操作符
        std::string op = ctx->MUL() ? "mul" : (ctx->DIV() ? "sdiv" : "srem");
        ir << "  " << result << " = " << op << " i32 " << left << ", " << right << "\n";
        return result;
    }
    return visit(ctx->unaryExp());
}
```

4. 控制流语句实现

基本块结构：条件块、真分支块、假分支块、结束块

跳转指令：条件跳转：br i1 条件, label %真分支, label %假分支无条件跳转：br label %目标块

循环实现：使用条件块→循环体→回跳的结构，避免使用 phi 节点（通过内存存储实现变量更新）

```
// if-else语句
if (ctx->IF()) {
    std::string condVal = std::any_cast<std::string>(visit(ctx->cond()));
    std::string trueLabel = newLabel();
    std::string falseLabel = newLabel();
    std::string endLabel = newLabel();

    // 条件跳转
    ir << "  br i1 " << condVal << ", label %" << trueLabel << ", label %" << falseLabel <<
"\n";

    // true分支
    ir << trueLabel << ":\n";
    visit(ctx->stmt(0));  // if块
    ir << "  br label %" << endLabel << "\n";  // 跳转到结束

    // false分支
    ir << falseLabel << ":\n";
    if (ctx->ELSE()) visit(ctx->stmt(1));  // else块
    ir << "  br label %" << endLabel << "\n";  // 跳转到结束

    // 结束标签
    ir << endLabel << ":\n";
}
```

```
// while循环
else if (ctx->WHILE()) {
    std::string condLabel = newLabel();
    std::string bodyLabel = newLabel();
    std::string endLabel = newLabel();

    // 初始跳转到条件判断
    ir << "  br label %" << condLabel << "\n";

    // 条件判断块
    ir << condLabel << ":\n";
    std::string condVal = std::any_cast<std::string>(visit(ctx->cond()));
    ir << "  br i1 " << condVal << ", label %" << bodyLabel << ", label %" << endLabel << "\n";

    // 循环体
    ir << bodyLabel << ":\n";
    visit(ctx->stmt(0));  // while体
    ir << "  br label %" << condLabel << "\n";  // 跳回条件判断

    // 结束标签
    ir << endLabel << ":\n";
}
```

5. 函数调用与返回

1. 函数调用：**call i32 @函数名(i32 参数 1, i32 参数 2)**
2. 参数传递：所有参数按值传递
3. 返回指令：**ret i32 值**
4. 统一返回类型：所有函数返回 i32 类型

```
// 函数调用
if (ctx->Ident()) {
    std::string funcName = ctx->Ident()->getText();
    std::vector<std::string> args;
    if (ctx->funcRParams()) {
        for (auto expCtx : ctx->funcRParams()->exp()) {
            args.push_back(std::any_cast<std::string>(visit(expCtx)));
        }
    }
    std::string result = newTemp();
    ir << "  " << result << " = call i32 @" << funcName << "(";
    for (size_t i = 0; i < args.size(); ++i) {
        if (i > 0) ir << ", ";
        ir << "i32 " << args[i];  // 所有参数为i32
    }
    ir << ")\n";
    return result;
}

// 返回语句
if (ctx->RETURN()) {
    std::string val = std::any_cast<std::string>(visit(ctx->exp()));
    ir << "  ret i32 " << val << "\n";
}
```

6. 变量访问与加载

```
std::any visitLVal(HelloParser::LValContext* ctx) {
    std::string varName = ctx->Ident()->getText();
    std::string varPtr = symbolTable[varName];  // 从符号表获取地址
    std::string result = newTemp();

    // 加载变量值
    ir << "  " << result << " = load i32, i32* " << varPtr << "\n";
    return result;
}
```

3、 实验总结

通过本次 LLVM IR 中间代码生成的实验，我深刻理解了编译器前端到后端的关键转换过程。在实现过程中，我掌握了如何将高级语言结构映射为 LLVM IR 的基本方法，特别是变量存储模型的设计和 SSA 形式的处理让我受益匪浅。实验中最具挑战性的是控制流语句的翻译，需要精确管理基本块和跳转指令，这让我对程序执行流程有了更深入的认识。