C–编译器源码结构和功能详细说明

编译器构造报告

专业：软件工程

年级：2023 级

班级：软工 2 班

组员：陈添硕 (3023244225)

李昊蓬 (3023244162)

许英帅 (3023244221)

谢雨航 (3023244222)

目录

目录

1 词法分析器

1.1 算法描述

1.1.1 基于 NFA→DFA 转换的词法分析器 (SLRLexer)

算法流程：

1. 构建 **NFA** (buildDFA() 函数)：
   - 为关键字、标识符、整数、浮点数、运算符、界符分别构建 NFA
   - 使用 epsilon 转换合并所有 NFA 为一个组合 NFA
   - 通过子集构造法（Subset Construction）将 NFA 转换为 DFA
   - 使用 DFA 最小化算法优化 DFA
2. 最长匹配算法 (analyze() 函数)：

```
初始化: currentState = dfa.start, startPos = pos

WHILE pos < source.length():
   跳过空白字符和注释

   currentState = dfa.start
   startPos = pos
   lastAcceptState = nullptr
   lastAcceptPos = -1
   currentPos = pos

   // 最长匹配
   WHILE currentPos < source.length():
      char c = sourceCode[currentPos]
      IF currentState有c的转换:
         currentState = currentState.transitions[c]
         currentPos++
         IF currentState是接受状态:
            lastAcceptState = currentState
            lastAcceptPos = currentPos
      ELSE:
         BREAK
   END WHILE

   IF lastAcceptState != nullptr AND lastAcceptPos > startPos:
      创建token: Token(lastAcceptState->acceptType,
                  sourceCode.substr(startPos, lastAcceptPos - startPos))
      pos = lastAcceptPos
   ELSE:
      创建ERROR token
```

```
        pos++
    END WHILE

    添加EOF token
```

1.2 输出格式说明

Token 输出格式：`[单词符号] TAB <[类型],[属性]>`
类型分类：

- **KW** (关键字): int(1), void(2), return(3), const(4), main(5), float(6), if(7), else(8)
- **OP** (运算符): +(9), -(10), *(11), /(12), %(13), =(14), >(15), <(16), ==(17), <=(18), >=(19), !=(20), &&(21), ||(22), !(29)
- **SE** (界符): ((23),)(24), {(25), }(26), ;(27), ,(28)
- **IDN** (标识符): `<IDN, 标识符名称 >`
- **INT** (整数): `<INT, 整数值 >`
- **FLOAT** (浮点数): `<FLOAT, 浮点数值 >`
- **EOF** (文件结束): `$`

示例输出：

```
int <KW,1>
main <IDN,main>
( <SE,23>
) <SE,24>
{ <SE,25>
return <KW,3>
10 <INT,10>
; <SE,27>
} <SE,26>
```

1.3 代码实现细节

1.3.1 Token 结构定义

Token 结构体：

```
struct Token {
   TokenType type; // Token类型
   std::string value; // Token值
   int line; // 行号
```

```
    int column; // 列号

    // 方法：获取类型字符串表示、输出格式化Token等
    bool shouldOutput() const; // 判断Token是否应该输出到结果文件
    std::string getTypeString() const; // 获取类型的字符串表示
    int getTypeCode() const; // 获取类型编号
    std::string toString() const; // 输出格式化的Token
};
```

1.3.2 关键字表初始化

功能：初始化关键字映射表，将关键字字符串映射到对应的 TokenType。

函数声明：

```
void Lexer::initKeywords();
```

功能说明：建立关键字（int, void, return, const, float, if, else）到 TokenType 的映射关系，关键字不区分大小写。注意：`main` 不在关键字表中，作为普通标识符（IDN）扫描，但在输出时会特殊处理为 `<KW,5>`。

1.3.3 标识符扫描实现

功能：扫描标识符或关键字。

函数声明：

```
Token Lexer::scanIdentifier();
```

功能说明：扫描字母、数字和下划线组成的标识符，然后检查是否是关键字（不区分大小写），如果是关键字则返回对应的关键字 Token，否则返回标识符 Token。

1.3.4 数字扫描实现

功能：扫描整数或浮点数。

函数声明：

```
Token Lexer::scanNumber();
```

功能说明：扫描数字序列，如果遇到小数点且后面有数字，则识别为浮点数，否则识别为整数。

1.3.5 注释处理实现

功能：跳过行注释（//）或块注释（/* */）。

函数声明：

```
bool Lexer::skipComment();
```

功能说明：检测并跳过注释。行注释跳过到行尾，块注释跳过到结束标记 `*/`。如果块注释未闭合则报错。

1.4 源程序编译步骤

使用词法分析器的编译步骤：

1. 加载源文件：

```
Lexer lexer;
lexer.loadFromFile("source.sy");
```

2. 执行词法分析：

```
lexer.tokenize();
```

3. 获取 **Token** 序列：

```
const auto& tokens = lexer.getTokens();
```

4. 输出结果：

```
lexer.printTokens(); // 打印到标准输出
// 或
std::string tokenStr = lexer.getTokensString(); // 获取字符串
```

命令行使用：

```
./compiler -l source.sy    # 仅词法分析
```

2 语法分析器

2.1 算法描述

本项目实现了两种语法分析器：

2.1.1 方法一：SLR 语法分析器 (SLRParser)

算法流程：

1. 初始化文法 (initGrammar())：
   - 定义 81 个产生式规则
   - 识别终结符和非终结符集合
2. 计算 **FIRST** 集 (computeFirst())：

```
FOR 每个终结符t:
   FIRST(t) = {t}

WHILE FIRST集发生变化:
   FOR 每个产生式 A -> α:
      IF α = ε:
         FIRST(A) += {ε}
      ELSE:
         FOR 每个符号X in α:
            FIRST(A) += FIRST(X) - {ε}
            IF ε NOT IN FIRST(X):
               BREAK
            IF 到达α的末尾:
               FIRST(A) += {ε}
   END WHILE
```

3. 计算 **FOLLOW** 集 (computeFollow())：

```
FOLLOW(开始符号) = {$}

WHILE FOLLOW集发生变化:
   FOR 每个产生式 A -> αBβ:
      IF β存在:
         FOLLOW(B) += FIRST(β) - {ε}
         IF ε IN FIRST(β):
            FOLLOW(B) += FOLLOW(A)
      ELSE:
         FOLLOW(B) += FOLLOW(A)
   END WHILE
```

4. 构建 **LR(0)** 项目集族 (buildCollection()):

```
I0 = closure({S' -> ·Program})
canonicalCollection = {I0}

WHILE 有新的项目集产生:
    FOR 每个项目集I:
        FOR 每个符号X (终结符或非终结符):
            J = goto(I, X)
            IF J非空 AND J不在canonicalCollection中:
                将J添加到canonicalCollection
                记录转换: (I, X) -> J
    END WHILE
```

5. 构建 **SLR** 分析表 (buildTable()):

```
FOR 每个项目集Ii:
    FOR 每个项目 [A -> α·aβ] in Ii:
        J = goto(Ii, a)
        action[i, a] = shift j

    FOR 每个项目 [A -> α· ] in Ii:
        IF A != S':
            FOR 每个a IN FOLLOW(A):
                action[i, a] = reduce A -> α
        ELSE:
            action[i, $] = accept

    FOR 每个非终结符A:
        IF goto(Ii, A) = Ij:
            goto[i, A] = j
```

6. **SLR** 分析算法 (parse()):

```
初始化: stateStack = [0], valueStack = []
ip = 0 // 输入指针

WHILE true:
    s = stateStack.top()
    a = tokens[ip] // 当前输入符号

    IF action[s, a] = shift t:
        stateStack.push(t)
        valueStack.push(语义值)
        ip++
```

```
        ELSE IF action[s, a] = reduce A -> β:
            从栈中弹出|β|个状态和语义值
            t = stateStack.top()
            stateStack.push(goto[t, A])
            valueStack.push(reduce(产生式ID, 语义值列表))

        ELSE IF action[s, a] = accept:
            RETURN success

        ELSE:
            RETURN error
    END WHILE
```

2.1.2 方法二：递归下降语法分析器 (Parser)

算法特点：

- 为每个非终结符编写一个递归函数
- 通过向前看（lookahead）解决选择冲突
- 自顶向下构建 AST

主要分析函数：

- `parseCompUnit()`: 解析编译单元
- `parseConstDecl()`: 解析常量声明
- `parseVarDecl()`: 解析变量声明
- `parseFuncDef()`: 解析函数定义
- `parseStmt()`: 解析语句
- `parseExp()`: 解析表达式（通过 parseAddExp）
- `parseCond()`: 解析条件表达式

算法流程示例（**parseAddExp**）：

```
parseAddExp():
    left = parseMulExp()

    WHILE 当前token是 + 或 -:
        保存运算符op
        前进token
        right = parseMulExp()
        创建新的AddExpNode，left作为左子树，right作为右子树
```

```
    left = 新节点
  END WHILE

  RETURN left
```

2.2 SLR 分析表结构

分析表包含两部分：

1. **ACTION** 表：`actionTable`[state, terminal] → Action
   - `SHIFT n`: 移进，跳转到状态 n
   - `REDUCE m`: 归约，使用产生式 m
   - `ACC`: 接受
   - `ERR`: 错误
2. **GOTO** 表：`gotoTable`[state, nonTerminal] → state
   - 记录状态转换

冲突解决：

- 移进-归约冲突：优先移进（shift-reduce conflict resolution）

示例分析表片段：

```
State 0:
  action[0, "int"] = shift 5
  action[0, "const"] = shift 6
  goto[0, "Program"] = 1
  goto[0, "compUnit"] = 2

State 5:
  action[5, "Ident"] = shift 10
  goto[5, "bType"] = 7

State 10:
  action[10, "="] = shift 15
  action[10, ";"] = reduce 19  // varDef -> Ident
  ...
```

2.3 输出格式说明

递归下降分析器输出格式：

步骤号 TAB Token值 TAB 动作

```
1 int move
2 main move
3 ( move
4 ) move
5 { move
6 return move
7 10 move
8 ; move
9 Stmt reduction
10 Block reduction
11 FuncDef reduction
12 compUnit reduction
13 $#$ accept
```

动作类型：

- move: 移进（匹配终结符）
- reduction: 归约（应用产生式）
- accept: 接受
- error: 错误

2.4 代码实现细节

2.4.1 递归下降分析器核心实现

Token 匹配和前进：

```
Token Parser::currentToken() const; // 获取当前Token
bool Parser::match(TokenType type); // 匹配并前进Token
bool Parser::expect(TokenType type, const std::string& msg); // 期望匹配Token，失败则报错
```

功能说明：

- currentToken(): 返回当前待处理的 Token，如果已到文件末尾则返回 EOF Token
- match(): 如果当前 Token 类型匹配，则前进到下一个 Token 并返回 true，否则返回 false
- expect(): 期望匹配指定类型的 Token，如果失败则输出错误信息

编译单元解析：

```
std::shared_ptr<CompUnitNode> Parser::parseCompUnit();
```

功能说明：解析编译单元，循环处理全局声明和函数定义。通过向前看（lookahead）区分函数定义和变量声明。

表达式解析（左结合）：

```
std::shared_ptr<AddExpNode> Parser::parseAddExp();
```

功能说明：解析加法表达式，处理左结合的 + 和-运算符。先解析第一个 mulExp 作为左操作数，然后循环处理后续的运算符和右操作数，构建左结合的 AST。

2.4.2 AST 节点创建示例

函数定义节点创建：

```
std::shared_ptr<FuncDefNode> Parser::parseFuncDef();
```

功能说明：解析函数定义，包括返回类型、函数名、参数列表和函数体。返回类型可以是 int、float 或 void。参数列表可能为空。

2.5 源程序编译步骤

使用语法分析器的编译步骤：

1. 词法分析：

   ```
   SLRLexer lexer;
   auto tokens = lexer.analyze(sourceCode);
   ```

2. 语法分析：

   ```
   SLRParser parser;
   bool success = parser.parse(tokens);
   ```

3. 获取 **AST**：

   ```
   auto ast = parser.getAST();
   ```

命令行使用：

```
./compiler -p source.sy    # 词法+语法分析
```

3 语法树（AST）

3.1 语法树存储结构

AST 节点层次结构：

```
CompUnitNode (根节点)
├── decls: vector<DeclNode*>          // 全局声明列表
│   ├── ConstDeclNode
│   │   ├── bType: BType
│   │   └── constDefs: vector<ConstDefNode*>
│   │       └── ConstDefNode
│   │           ├── ident: string
│   │           └── initVal: ExpNode*
│   └── VarDeclNode
│       ├── bType: BType
│       └── varDefs: vector<VarDefNode*>
│           └── VarDefNode
│               ├── ident: string
│               └── initVal: ExpNode* (可选)
│
└── funcDefs: vector<FuncDefNode*>     // 函数定义列表
    └── FuncDefNode
        ├── returnType: BType
        ├── ident: string
        ├── params: vector<FuncFParamNode*>
        └── block: BlockNode*
            └── items: vector<BlockItemNode*>
                ├── decl: DeclNode* (可选)
                └── stmt: StmtNode* (可选)
                    ├── ASSIGN: lVal + exp
                    ├── EXP: exp (可选)
                    ├── BLOCK: block
                    ├── IF: cond + thenStmt + elseStmt (可选)
                    └── RETURN: exp (可选)
```

表达式节点层次：

```
ExpNode (基类)
├── AddExpNode
│   ├── left: AddExpNode* (可选)
│   ├── op: BinaryOp (ADD/SUB)
```

```
│   └── right: MulExpNode*
│
├── MulExpNode
│   ├── left: MulExpNode* (可选)
│   ├── op: BinaryOp (MUL/DIV/MOD)
│   └── right: UnaryExpNode*
│
├── UnaryExpNode
│   ├── PRIMARY: primaryExp
│   ├── FUNC_CALL: funcName + args
│   └── UNARY_OP: unaryOp + unaryExp
│
├── PrimaryExpNode
│   ├── PAREN_EXP: exp
│   ├── LVAL: lVal
│   └── NUMBER: number
│
├── LValNode
│   └── ident: string
│
└── NumberNode
    ├── isFloat: bool
    ├── intVal: int
    └── floatVal: float
```

存储方式：

- 使用 std::shared_ptr 智能指针管理节点生命周期
- 树形结构通过指针链接
- 列表使用 std::vector<std::shared_ptr<Node>>

3.1.1 AST 节点类定义

AST 节点基类：

```
class ASTNode {
public:
    virtual ~ASTNode() = default;
    std::string type; // 节点类型名称

    ASTNode() : type("ASTNode") {}
    explicit ASTNode(const std::string& t) : type(t) {}
```

```
};
```

编译单元节点：

```
class CompUnitNode : public ASTNode {
public:
    std::vector<std::shared_ptr<DeclNode>> decls; // 全局声明
    std::vector<std::shared_ptr<FuncDefNode>> funcDefs; // 函数定义

    CompUnitNode() : ASTNode("CompUnit") {}
};
```

声明节点：

```
class DeclNode : public ASTNode {
public:
    bool isConst = false;
    DeclNode() : ASTNode("Decl") {}
};

class ConstDeclNode : public DeclNode {
public:
    BType bType;
    std::vector<std::shared_ptr<ConstDefNode>> constDefs;
    ConstDeclNode() : DeclNode("ConstDecl") { isConst = true; }
};

class VarDeclNode : public DeclNode {
public:
    BType bType;
    std::vector<std::shared_ptr<VarDefNode>> varDefs;
    VarDeclNode() : DeclNode("VarDecl") { isConst = false; }
};
```

函数定义节点：

```
class FuncDefNode : public ASTNode {
public:
    BType returnType;
    std::string ident;
    std::vector<std::shared_ptr<FuncFParamNode>> params;
    std::shared_ptr<BlockNode> block;

    FuncDefNode() : ASTNode("FuncDef") {}
};
```

表达式节点层次：

```
// 表达式基类
```

```
class ExpNode : public ASTNode {
public:
    ExpNode() : ASTNode("Exp") {}
};

// 加法表达式（左结合）
class AddExpNode : public ExpNode {
public:
    std::shared_ptr<AddExpNode> left; // 左操作数（可选）
    BinaryOp op; // 运算符 (ADD, SUB)
    std::shared_ptr<MulExpNode> right; // 右操作数

    AddExpNode() : ExpNode("AddExp"), op(BinaryOp::ADD) {}
};

// 乘法表达式（左结合）
class MulExpNode : public ExpNode {
public:
    std::shared_ptr<MulExpNode> left; // 左操作数（可选）
    BinaryOp op; // 运算符 (MUL, DIV, MOD)
    std::shared_ptr<UnaryExpNode> right; // 右操作数

    MulExpNode() : ExpNode("MulExp"), op(BinaryOp::MUL) {}
};

// 一元表达式
class UnaryExpNode : public ExpNode {
public:
    enum class UnaryType {
        PRIMARY, // primaryExp
        FUNC_CALL, // Ident '(' funcRParams? ')'
        UNARY_OP // unaryOp unaryExp
    };

    UnaryType unaryType;
    std::shared_ptr<PrimaryExpNode> primaryExp; // PRIMARY
    std::string funcName; // FUNC_CALL
    std::vector<std::shared_ptr<ExpNode>> args; // FUNC_CALL
    UnaryOp unaryOp; // UNARY_OP
    std::shared_ptr<UnaryExpNode> unaryExp; // UNARY_OP

    UnaryExpNode() : ExpNode("UnaryExp"), unaryType(UnaryType::PRIMARY) {}
};

// 逻辑与表达式（短路求值）
```

```
class LAndExpNode : public ExpNode {
public:
    std::shared_ptr<LAndExpNode> left; // 左操作数（可选）
    std::shared_ptr<EqExpNode> right; // 右操作数

    LAndExpNode() : ExpNode("LAndExp") {}
};

// 逻辑或表达式（短路求值）
class LOrExpNode : public ExpNode {
public:
    std::shared_ptr<LOrExpNode> left; // 左操作数（可选）
    std::shared_ptr<LAndExpNode> right; // 右操作数

    LOrExpNode() : ExpNode("LOrExp") {}
};
```

语句节点：

```
enum class StmtType {
    ASSIGN, // lVal '=' exp ';'
    EXP, // (exp)? ';'
    BLOCK, // block
    IF, // 'if' '(' cond ')' stmt ('else' stmt)?
    RETURN // 'return' (exp)? ';'
};

class StmtNode : public ASTNode {
public:
    StmtType stmtType;

    // ASSIGN类型
    std::shared_ptr<LValNode> lVal;
    std::shared_ptr<ExpNode> exp;

    // BLOCK类型
    std::shared_ptr<BlockNode> block;

    // IF类型
    std::shared_ptr<CondNode> cond;
    std::shared_ptr<StmtNode> thenStmt;
    std::shared_ptr<StmtNode> elseStmt; // 可选

    // RETURN类型 - 使用exp字段
    StmtNode() : ASTNode("Stmt"), stmtType(StmtType::EXP) {}
};
```

4 中间代码生成（LLVM IR）

4.1 IR 生成器整体架构

设计模式：访问者模式（**Visitor Pattern**） IR 生成器采用访问者模式遍历 AST 并生成 LLVM IR：

- 为每种 AST 节点类型定义专门的访问函数（如 `visitCompUnit`、`visitFuncDef` 等）
- 通过动态分发（`dynamic_cast`）识别节点类型并调用相应的访问函数
- 递归遍历 AST 树，自顶向下生成 IR

核心组件：

- `IRGenerator`: IR 生成器主类
- `IRBuilder`: IR 指令构建器，负责创建各种 IR 指令
- `SymbolTable`: 符号表，管理作用域和变量映射

访问者模式遍历 **AST**：

```
算法: visitCompUnit(node)
输入: CompUnitNode* node
输出: 无

BEGIN
    // 先处理全局声明
    FOR EACH decl IN node->decls:
        visitDecl(decl)
    END FOR

    // 再处理函数定义
    FOR EACH funcDef IN node->funcDefs:
        visitFuncDef(funcDef)
    END FOR
END

算法: visitDecl(node)
输入: DeclNode* node
输出: 无

BEGIN
    IF node是ConstDeclNode:
        visitConstDecl(node)
    ELSE IF node是VarDeclNode:
        visitVarDecl(node)
    END IF
END
```

```
算法: visitFuncDef(node)
输入: FuncDefNode* node
输出: 无

BEGIN
    // 创建函数
    retType = bTypeToLLVMType(node->returnType)
    paramTypes = []
    FOR EACH param IN node->params:
        paramTypes.append(bTypeToLLVMType(param->bType))
    END FOR

    func = Function::create(FunctionType::get(retType, paramTypes),
                            node->ident, module)

    // 创建入口基本块
    entryBB = BasicBlock::create(module, node->ident + "_ENTRY", func)
    builder->set_insert_point(entryBB)

    // 进入函数作用域
    symbolTable.enterScope()

    // 处理参数
    FOR EACH param IN node->params:
        alloca = createLocalVariable(param->ident, paramType)
        builder->create_store(arg, alloca)
        symbolTable.insert(param->ident, alloca, paramType, false)
    END FOR

    // 处理函数体
    visitBlock(node->block)

    // 如果基本块没有终结指令，添加默认返回
    IF currentBB没有终结指令:
        IF retType是void:
            builder->create_void_ret()
        ELSE:
            builder->create_ret(ConstantInt::get(0, module))
        END IF
    END IF

    // 退出函数作用域
    symbolTable.exitScope()
END
```

```
算法: visitStmt(node)
输入: StmtNode* node
输出: 无

BEGIN
   SWITCH node->stmtType:
      CASE ASSIGN:
         addr = visitLVal(node->lVal, false) // 获取地址
         val = visitExp(node->exp) // 计算值
         builder->create_store(val, addr) // 存储
      CASE EXP:
         IF node->exp存在:
            visitExp(node->exp) // 计算表达式（丢弃结果）
         END IF
      CASE BLOCK:
         visitBlock(node->block)
      CASE IF:
         visitIfStmt(node)
      CASE RETURN:
         visitReturnStmt(node)
   END SWITCH
END

算法: visitExp(node)
输入: ExpNode* node
输出: Value* (IR值)

BEGIN
   IF node是AddExpNode:
      RETURN visitAddExp(node)
   ELSE IF node是MulExpNode:
      RETURN visitMulExp(node)
   ELSE IF node是UnaryExpNode:
      RETURN visitUnaryExp(node)
   ELSE IF node是PrimaryExpNode:
      RETURN visitPrimaryExp(node)
   ELSE IF node是LValNode:
      RETURN visitLVal(node, true) // 加载值
   ELSE IF node是NumberNode:
      RETURN visitNumber(node)
   END IF
END

算法: visitAddExp(node)
```

```
输入: AddExpNode* node
输出: Value*

BEGIN
    IF node->left == nullptr:
        // 只有右操作数
        RETURN visitMulExp(node->right)
    END IF

    // 递归计算左操作数
    left = visitAddExp(node->left)
    right = visitMulExp(node->right)

    // 类型提升
    IF left是float类型 OR right是float类型:
        IF left是int类型:
            left = builder->create_sitofp(left, floatType)
        END IF
        IF right是int类型:
            right = builder->create_sitofp(right, floatType)
        END IF
    END IF

    // 生成IR指令
    IF node->op == ADD:
        IF 是float类型:
            RETURN builder->create_fadd(left, right)
        ELSE:
            RETURN builder->create_iadd(left, right)
        END IF
    ELSE IF node->op == SUB:
        IF 是float类型:
            RETURN builder->create_fsub(left, right)
        ELSE:
            RETURN builder->create_isub(left, right)
        END IF
    END IF
END

算法: visitLVal(node, load)
输入: LValNode* node, bool load
输出: Value*

BEGIN
    info = symbolTable.lookup(node->ident)
```

```
    IF info == nullptr:
        报错: 未定义的变量
        RETURN nullptr
    END IF

    addr = info->value

    IF load == true:
        IF addr是Argument类型:
            RETURN addr // 参数直接返回值
        END IF
        // 加载值
        loadType = addr->get_type()->get_pointer_element_type()
        RETURN builder->create_load(loadType, addr)
    ELSE:
        RETURN addr // 返回地址
    END IF
END

算法: visitIfStmt(node)
输入: StmtNode* node
输出: 无

BEGIN
    // 创建基本块
    thenBB = BasicBlock::create(module, "", currentFunction)
    IF node->elseStmt存在:
        elseBB = BasicBlock::create(module, "", currentFunction)
    ELSE:
        elseBB = nullptr
    END IF
    mergeBB = BasicBlock::create(module, "", currentFunction)

    // 计算条件
    condVal = visitCond(node->cond)
    condVal = ensureInt1(condVal) // 转换为i1类型

    // 条件跳转
    IF elseBB存在:
        builder->create_cond_br(condVal, thenBB, elseBB)
    ELSE:
        builder->create_cond_br(condVal, thenBB, mergeBB)
    END IF

    // 生成then分支
```

```
    currentBB = thenBB
    builder->set_insert_point(thenBB)
    visitStmt(node->thenStmt)
    IF currentBB没有终结指令:
        builder->create_br(mergeBB)
    END IF

    // 生成else分支
    IF elseBB存在 AND node->elseStmt存在:
        currentBB = elseBB
        builder->set_insert_point(elseBB)
        visitStmt(node->elseStmt)
        IF currentBB没有终结指令:
            builder->create_br(mergeBB)
        END IF
    END IF

    // 设置当前基本块为merge
    currentBB = mergeBB
    builder->set_insert_point(mergeBB)
END
```

4.2 LLVM IR 指令类型

本编译器生成的 IR 指令类型包括：

1. 终结指令（**Terminator Instructions**）：

- `ret`: 返回指令
    - `ret i32 %val`: 返回值
    - `ret void`: 无返回值返回
- `br`: 跳转指令
    - `br label %bb`: 无条件跳转
    - `br i1 %cond, label %if_true, label %if_false`: 条件跳转

2. 二元运算指令（**Binary Instructions**）：

- 整数运算：
    - `add`: 加法 (`%result = add i32 %a, %b`)
    - `sub`: 减法 (`%result = sub i32 %a, %b`)
    - `mul`: 乘法 (`%result = mul i32 %a, %b`)
    - `sdiv`: 有符号除法 (`%result = sdiv i32 %a, %b`)

    - srem: 有符号取模 (%result = srem i32 %a, %b)

- 浮点运算：

    - fadd: 浮点加法 (%result = fadd float %a, %b)
    - fsub: 浮点减法 (%result = fsub float %a, %b)
    - fmul: 浮点乘法 (%result = fmul float %a, %b)
    - fdiv: 浮点除法 (%result = fdiv float %a, %b)

3. 内存操作指令（**Memory Instructions**）：

- alloca: 栈分配 (%ptr = alloca i32)
- load: 加载值 (%val = load i32, i32* %ptr)
- store: 存储值 (store i32 %val, i32* %ptr)

4. 比较指令（**Comparison Instructions**）：

- 整数比较（返回 i1 类型）：

    - icmp eq: 相等 (%cmp = icmp eq i32 %a, %b)
    - icmp ne: 不等 (%cmp = icmp ne i32 %a, %b)
    - icmp sgt: 有符号大于 (%cmp = icmp sgt i32 %a, %b)
    - icmp sge: 有符号大于等于 (%cmp = icmp sge i32 %a, %b)
    - icmp slt: 有符号小于 (%cmp = icmp slt i32 %a, %b)
    - icmp sle: 有符号小于等于 (%cmp = icmp sle i32 %a, %b)

- 浮点比较（返回 i1 类型）：

    - fcmp oeq: 有序相等 (%cmp = fcmp oeq float %a, %b)
    - fcmp one: 有序不等 (%cmp = fcmp one float %a, %b)
    - fcmp ogt: 有序大于 (%cmp = fcmp ogt float %a, %b)
    - fcmp oge: 有序大于等于 (%cmp = fcmp oge float %a, %b)
    - fcmp olt: 有序小于 (%cmp = fcmp olt float %a, %b)
    - fcmp ole: 有序小于等于 (%cmp = fcmp ole float %a, %b)

5. 类型转换指令（**Conversion Instructions**）：

- zext: 零扩展 (%ext = zext i1 %val to i32)
- sitofp: 有符号整数转浮点 (%fp = sitofp i32 %val to float)
- fptosi: 浮点转有符号整数 (%int = fptosi float %val to i32)

6. 函数调用指令（**Call Instruction**）：

- call: 函数调用 (%result = call i32 @func(i32 %arg))

7. Phi 指令（**Phi Instruction**）：

- phi: SSA 形式的 Phi 节点 (%val = phi i32 [%a, %bb1], [%b, %bb2])

4.3 基本块和控制流图（CFG）

基本块（**Basic Block**）概念：

- 基本块是最大化的线性指令序列，只有一个入口点和一个出口点

- 基本块以终结指令（ret 或 br）结束

- 基本块之间通过跳转指令连接，形成控制流图（CFG）

基本块管理算法：
算法：创建基本块
输入：函数**Function*** func，基本块名称string name
输出：BasicBlock*

```
BEGIN
   bb = BasicBlock::create(module, name, func)
   func->add_basic_block(bb)
   RETURN bb
END
```

算法：设置当前基本块
输入：BasicBlock* bb
输出：无

```
BEGIN
   currentBB = bb
   builder->set_insert_point(bb) // 设置IRBuilder的插入点
END
```

控制流处理示例（**if** 语句）：

IF语句的CFG结构：

```
    entryBB (入口基本块)
        |
        | (条件跳转)
        v
    +-------+-------+
    |               |
```

```
    v               v
thenBB          elseBB (可选)
    |               |
    +-------+-------+
            |
            v
        mergeBB (合并基本块)
```

4.4 符号表和作用域管理

符号表结构：

```
SymbolTable
├── scopes: vector<Scope*>  // 作用域栈
|   └── Scope
|       └── symbols: map<string, SymbolInfo>
|           └── SymbolInfo
|               ├── value: Value*      // IR中的Value指针
|               ├── type: Type*        // 类型
|               ├── isConst: bool      // 是否为常量
|               └── isGlobal: bool     // 是否为全局变量
```

作用域管理算法：

```
算法: enterScope()
输出: 无

BEGIN
   scopes.push_back(make_shared<Scope>())
END

算法: exitScope()
输出: 无

BEGIN
   IF scopes.size() > 1: // 保留全局作用域
      scopes.pop_back()
   END IF
END

算法: lookup(name)
输入: string name
输出: SymbolInfo*

BEGIN
```

```
    // 从内层作用域向外查找
    FOR i = scopes.size() - 1 TO 0:
        info = scopes[i]->lookup(name)
        IF info != nullptr:
            RETURN info
        END IF
    END FOR
    RETURN nullptr
END
```

```
算法: insert(name, value, type, isConst)
输入: string name, Value* value, Type* type, bool isConst
输出: bool (成功返回true)

BEGIN
    IF scopes.empty():
        RETURN false
    END IF

    info = SymbolInfo(value, type, isConst, isGlobalScope())
    RETURN scopes.back()->insert(name, info)
END
```

4.5 全局变量和局部变量处理

全局变量生成:

```
算法: visitConstDef (全局常量)
输入: ConstDefNode* node, BType bType
输出: 无

BEGIN
    IF symbolTable.isGlobalScope():
        type = bTypeToLLVMType(bType)

        // 计算常量初始值
        IF bType == INT:
            initVal = evalConstInt(node->initVal)
            init = ConstantInt::get(initVal, module)
        ELSE:
            initVal = evalConstFloat(node->initVal)
            init = ConstantFP::get(initVal, module)
        END IF

        // 创建全局变量
        gv = GlobalVariable::create(node->ident, module, type, true, init)
```

```
        symbolTable.insert(node->ident, gv, type, true)
    END IF
END
```

局部变量生成（**SSA** 形式）：
算法：visitVarDef (局部变量)
输入：VarDefNode* node, BType bType
输出：无

```
BEGIN
    IF NOT symbolTable.isGlobalScope():
        type = bTypeToLLVMType(bType)

        // 创建alloca指令（在栈上分配空间）
        alloca = builder->create_alloca(type)

        // 如果有初始化值
        IF node->initVal存在:
            initVal = visitExp(node->initVal)
            builder->create_store(initVal, alloca)
        END IF

        symbolTable.insert(node->ident, alloca, type, false)
    END IF
END
```

变量访问：
算法：visitLVal(node, load)
输入：LValNode* node, bool load
输出：**Value***

```
BEGIN
    info = symbolTable.lookup(node->ident)
    IF info == nullptr:
        报错：未定义的变量
        RETURN nullptr
    END IF

    addr = info->value

    IF load == true:
        // 需要加载值
        IF addr是Argument类型:
            RETURN addr // 函数参数直接返回值
        ELSE:
```

```
            // 生成load指令
            loadType = addr->get_type()->get_pointer_element_type()
            RETURN builder->create_load(loadType, addr)
        END IF
    ELSE:
        // 需要地址（用于赋值）
        RETURN addr
    END IF
END
```

4.6 函数定义和调用

函数定义生成：

算法：visitFuncDef(node)

输入：FuncDefNode* node

输出：无

```
BEGIN
    // 1. 构建函数类型
    retType = bTypeToLLVMType(node->returnType)
    paramTypes = []
    FOR EACH param IN node->params:
        paramTypes.append(bTypeToLLVMType(param->bType))
    END FOR

    funcType = FunctionType::get(retType, paramTypes)

    // 2. 创建函数
    func = Function::create(funcType, node->ident, module)
    currentFunction = func
    builder->set_curFunc(func)

    // 3. 创建入口基本块
    entryBB = BasicBlock::create(module, node->ident + "_ENTRY", func)
    currentBB = entryBB
    builder->set_insert_point(entryBB)

    // 4. 进入函数作用域
    symbolTable.enterScope()

    // 5. 处理参数（为每个参数创建alloca并存储）
    argIt = func->arg_begin()
    FOR EACH param IN node->params:
        paramType = bTypeToLLVMType(param->bType)
        alloca = createLocalVariable(param->ident, paramType)
```

```
        builder->create_store(*argIt, alloca)
        symbolTable.insert(param->ident, alloca, paramType, false)
        argIt++
    END FOR

    // 6. 处理函数体
    visitBlock(node->block)

    // 7. 如果基本块没有终结指令，添加默认返回
    IF currentBB没有终结指令:
        IF retType是void:
            builder->create_void_ret()
        ELSE:
            builder->create_ret(ConstantInt::get(0, module))
        END IF
    END IF

    // 8. 退出函数作用域
    symbolTable.exitScope()
    currentFunction = nullptr
END
```

函数调用生成：

```
算法: visitUnaryExp (函数调用)
输入: UnaryExpNode* node (unaryType == FUNC_CALL)
输出: Value*

BEGIN
    // 1. 查找函数
    funcVal = symbolTable.getValue(node->funcName)
    IF funcVal == nullptr:
        报错: 未定义的函数
        RETURN nullptr
    END IF

    // 2. 准备参数
    args = []
    FOR EACH arg IN node->args:
        argVal = visitExp(arg)
        args.append(argVal)
    END FOR

    // 3. 创建调用指令
    RETURN builder->create_call(funcVal, args)
END
```

4.7 类型系统和类型提升

类型映射：

```
BType -> LLVM Type:
    INT -> i32 (32位有符号整数)
    FLOAT -> float (32位浮点数)
    VOID -> void (无类型)
```

类型提升规则：

```
算法: 类型提升（二元运算）
输入: Value* left, Value* right
输出: Value* left, Value* right (提升后的值)

BEGIN
   isFloat = left->get_type()->is_float_type() OR
           right->get_type()->is_float_type()

   IF isFloat:
      IF left是int32类型:
         left = builder->create_sitofp(left, floatType)
      END IF
      IF right是int32类型:
         right = builder->create_sitofp(right, floatType)
      END IF
   END IF

   RETURN left, right
END
```

类型转换指令生成：

- sitofp: 整数转浮点（用于混合类型运算）
- fptosi: 浮点转整数（用于赋值给整数变量）
- zext: 零扩展（i1 转 i32，用于条件判断）

4.8 常量折叠优化

常量求值算法：

```
算法: evalConstInt(node)
输入: ExpNode* node
输出: int

BEGIN
   IF node是NumberNode:
```

```
    RETURN node->intVal

ELSE IF node是AddExpNode:
    IF node->left == nullptr:
        RETURN evalConstInt(node->right)
    ELSE:
        left = evalConstInt(node->left)
        right = evalConstInt(node->right)
        IF node->op == ADD:
            RETURN left + right
        ELSE IF node->op == SUB:
            RETURN left - right
        END IF
    END IF

ELSE IF node是MulExpNode:
    IF node->left == nullptr:
        RETURN evalConstInt(node->right)
    ELSE:
        left = evalConstInt(node->left)
        right = evalConstInt(node->right)
        IF node->op == MUL:
            RETURN left * right
        ELSE IF node->op == DIV:
            RETURN left / right
        ELSE IF node->op == MOD:
            RETURN left % right
        END IF
    END IF

ELSE IF node是UnaryExpNode:
    IF node->unaryType == PRIMARY:
        RETURN evalConstInt(node->primaryExp)
    ELSE IF node->unaryType == UNARY_OP:
        val = evalConstInt(node->unaryExp)
        IF node->unaryOp == PLUS:
            RETURN val
        ELSE IF node->unaryOp == MINUS:
            RETURN -val
        ELSE IF node->unaryOp == NOT:
            RETURN (val == 0 ? 1 : 0)
        END IF
    END IF

ELSE IF node是PrimaryExpNode:
```

```
        IF node->primaryType == NUMBER:
            RETURN node->number->intVal
        ELSE IF node->primaryType == PAREN_EXP:
            RETURN evalConstInt(node->exp)
        END IF

    RETURN 0
END
```

常量折叠应用：

- 全局变量初始化时，如果初始值是常量表达式，直接计算并生成常量
- 避免生成不必要的运行时计算指令

4.9 短路求值实现

逻辑与（&&）短路求值：

算法：visitLAndExp(node)

输入：LAndExpNode* node

输出：**Value*** (i1类型)

```
BEGIN
    IF node->left == nullptr:
        // 只有右操作数（递归终止条件）
        RETURN visitEqExp(node->right)
    END IF

    // 创建基本块
    rhsBB = BasicBlock::create(module, "", currentFunction)
    mergeBB = BasicBlock::create(module, "", currentFunction)

    // 递归计算左操作数（左操作数也是LAndExp类型）
    leftVal = visitLAndExp(node->left)
    leftVal = ensureInt1(leftVal) // 转换为i1类型
    leftBB = currentBB

    // 短路：左边为真则计算右边，否则跳到merge（整体为假）
    builder->create_cond_br(leftVal, rhsBB, mergeBB)

    // 计算右操作数（右操作数是EqExp类型）
    currentBB = rhsBB
    builder->set_insert_point(rhsBB)
    rightVal = visitEqExp(node->right)
    rightVal = ensureInt1(rightVal)
    builder->create_br(mergeBB)
```

```
    rightEndBB = currentBB

    // merge基本块
    currentBB = mergeBB
    builder->set_insert_point(mergeBB)

    // 创建phi节点：左边为假则整体为假，否则等于右边
    phi = PhiInst::create_phi(module->get_int1_type(), mergeBB)
    phi->add_phi_pair_operand(ConstantInt::get(false, module), leftBB)
    phi->add_phi_pair_operand(rightVal, rightEndBB)

    RETURN phi
END
```

逻辑或（||）短路求值：

算法：visitLOrExp(node)

输入：LOrExpNode* node

输出：**Value*** (i1类型)

```
BEGIN
    IF node->left == nullptr:
        // 只有右操作数（递归终止条件）
        RETURN visitLAndExp(node->right)
    END IF

    // 创建基本块
    rhsBB = BasicBlock::create(module, "", currentFunction)
    mergeBB = BasicBlock::create(module, "", currentFunction)

    // 递归计算左操作数（左操作数也是LOrExp类型）
    leftVal = visitLOrExp(node->left)
    leftVal = ensureInt1(leftVal)
    leftBB = currentBB

    // 短路：左边为真则跳到merge（整体为真），否则计算右边
    builder->create_cond_br(leftVal, mergeBB, rhsBB)

    // 计算右操作数（右操作数是LAndExp类型）
    currentBB = rhsBB
    builder->set_insert_point(rhsBB)
    rightVal = visitLAndExp(node->right)
    rightVal = ensureInt1(rightVal)
    builder->create_br(mergeBB)
    rightEndBB = currentBB
```

```
    // merge基本块
    currentBB = mergeBB
    builder->set_insert_point(mergeBB)

    // 创建phi节点：左边为真则整体为真，否则等于右边
    phi = PhiInst::create_phi(module->get_int1_type(), mergeBB)
    phi->add_phi_pair_operand(ConstantInt::get(true, module), leftBB)
    phi->add_phi_pair_operand(rightVal, rightEndBB)

    RETURN phi
END
```

4.10 Phi 节点和 SSA 形式

Phi 节点的作用：

- Phi 节点用于 SSA（Static Single Assignment）形式
- 当多个控制流路径汇合时，使用 Phi 节点合并不同路径的值
- 格式：`%val = phi i32 [%a, %bb1], [%b, %bb2]`
    - 如果从`%bb1` 到达，值为`%a`
    - 如果从`%bb2` 到达，值为`%b`

Phi 节点生成示例（短路求值）：

```
; 逻辑与的CFG
entry:
    %left = icmp ne i32 %x, 0
    br i1 %left, label %rhs, label %merge

rhs:
    %right = icmp ne i32 %y, 0
    br label %merge

merge:
    %result = phi i1 [false, %entry], [%right, %rhs]
    ; 如果从entry到达（left为false），result为false
    ; 如果从rhs到达（left为true），result为right的值
```

4.11 IR 生成完整流程

主生成流程：

```
算法: generate(ast)
输入: CompUnitNode* ast
输出: 无

BEGIN
    // 1. 声明运行时库函数
    declareRuntimeFunctions()

    // 2. 遍历AST，生成IR
    visitCompUnit(ast)

    // 3. 设置打印名称（为所有Value分配名称）
    module->set_print_name()
END

算法: visitCompUnit(node)
输入: CompUnitNode* node
输出: 无

BEGIN
    // 先处理全局声明（全局变量和常量）
    FOR EACH decl IN node->decls:
        visitDecl(decl)
    END FOR

    // 再处理函数定义
    FOR EACH funcDef IN node->funcDefs:
        visitFuncDef(funcDef)
    END FOR
END
```

运行时库函数声明：

```
void IRGenerator::declareRuntimeFunctions() {
    std::vector<Type*> emptyParams;
    std::vector<Type*> intParam = {module->get_int32_type()};
    std::vector<Type*> intPtrParam = {module->get_int32_ptr_type()};
    std::vector<Type*> putArrayParams = {
        module->get_int32_type(),
        module->get_int32_ptr_type()
    };

    // int getint()
    FunctionType* getintType = FunctionType::get(
        module->get_int32_type(), emptyParams);
```

```
    Function::create(getintType, "getint", module);

    // int getch()
    FunctionType* getchType = FunctionType::get(
        module->get_int32_type(), emptyParams);
    Function::create(getchType, "getch", module);

    // int getarray(int*)
    FunctionType* getarrayType = FunctionType::get(
        module->get_int32_type(), intPtrParam);
    Function::create(getarrayType, "getarray", module);

    // void putint(int)
    FunctionType* putintType = FunctionType::get(
        module->get_void_type(), intParam);
    Function::create(putintType, "putint", module);

    // void putch(int)
    FunctionType* putchType = FunctionType::get(
        module->get_void_type(), intParam);
    Function::create(putchType, "putch", module);

    // void putarray(int, int*)
    FunctionType* putarrayType = FunctionType::get(
        module->get_void_type(), putArrayParams);
    Function::create(putarrayType, "putarray", module);

    // void starttime()
    FunctionType* starttimeType = FunctionType::get(
        module->get_void_type(), emptyParams);
    Function::create(starttimeType, "starttime", module);

    // void stoptime()
    FunctionType* stoptimeType = FunctionType::get(
        module->get_void_type(), emptyParams);
    Function::create(stoptimeType, "stoptime", module);
}
```

运行时库函数说明：

- getint(): 从标准输入读取一个整数
- getch(): 从标准输入读取一个字符
- getarray(int*): 从标准输入读取一个整数数组

- putint(int): 向标准输出写入一个整数
- putch(int): 向标准输出写入一个字符
- putarray(int, int*): 向标准输出写入一个整数数组
- starttime(): 开始计时
- stoptime(): 停止计时

4.12 IR 生成示例

输入源代码：

```
int a = 10;

int main() {
   int b = 20;
   if (a > 5) {
      b = a + 10;
   }
   return b;
}
```

生成的 **LLVM IR**：

```
@a = global i32 10

define i32 @main() {
entry:
    %0 = alloca i32
    %1 = alloca i32
    store i32 20, i32* %1
    %2 = load i32, i32* @a
    %3 = icmp sgt i32 %2, 5
    br i1 %3, label %then, label %merge

then:
    %4 = load i32, i32* @a
    %5 = add i32 %4, 10
    store i32 %5, i32* %1
    br label %merge

merge:
    %6 = load i32, i32* %1
```

```
    ret i32 %6
}
```

IR 说明：

1. `@a = global i32 10`: 全局变量定义
2. `%0 = alloca i32`: 为局部变量分配栈空间（未使用）
3. `%1 = alloca i32`: 为变量 b 分配栈空间
4. `store i32 20, i32* %1`: 初始化 b = 20
5. `%2 = load i32, i32* @a`: 加载全局变量 a
6. `%3 = icmp sgt i32 %2, 5`: 比较 a > 5
7. `br i1 %3, label %then, label %merge`: 条件跳转
8. then 基本块：计算 b = a + 10
9. merge 基本块：加载 b 并返回

4.13 代码实现细节

IRGenerator 类核心成员：

```
class IRGenerator {
private:
    Module* module; // 当前模块
    IRBuilder* builder; // IR构建器
    SymbolTable symbolTable; // 符号表
    Function* currentFunction; // 当前函数
    BasicBlock* currentBB; // 当前基本块

    // 临时存储常量值（用于常量折叠）
    int tmpIntVal;
    float tmpFloatVal;
    bool tmpIsFloat;
    bool isConstExpr;

    // 用于短路求值的基本块
    BasicBlock* trueBB;
    BasicBlock* falseBB;
};
```

构造函数和初始化：

```
IRGenerator::IRGenerator(const std::string& sourceFileName);
```

功能说明：创建 IR 模块和 IRBuilder，初始化成员变量，声明运行时库函数。

类型转换辅助函数：

```
Type* IRGenerator::bTypeToLLVMType(BType bType); // BType转换为LLVM Type
Value* IRGenerator::ensureInt1(Value* val); // 确保值为i1类型（布尔值）
```

功能说明：

- `bTypeToLLVMType()`: 将 BType（INT/FLOAT/VOID）映射到对应的 LLVM 类型（i32/float/void）

- `ensureInt1()`: 如果值是 i32 类型，则与 0 比较转换为 i1 类型；如果已经是 i1 类型则直接返回

常量求值：

```
int IRGenerator::evalConstInt(std::shared_ptr<ExpNode> node);
```

功能说明：递归计算常量表达式的整数值，支持数字字面量、加法表达式、乘法表达式和一元表达式。用于常量折叠优化。

表达式访问（类型提升）：

```
Value* IRGenerator::visitAddExp(std::shared_ptr<AddExpNode> node);
```

功能说明：访问加法表达式节点，递归处理左操作数，处理类型提升（如果有一个操作数是 float，则将 int 转换为 float），生成对应的 IR 指令（fadd/iadd 或 fsub/isub）。

左值访问（加载/地址）：

```
Value* IRGenerator::visitLVal(std::shared_ptr<LValNode> node, bool load);
```

功能说明：访问左值节点。如果 `load=true`，则从符号表查找变量并生成 load 指令加载值（函数参数直接返回值）；如果 `load=false`，则返回变量的地址（用于赋值）。

if 语句实现（控制流）：

```
void IRGenerator::visitIfStmt(std::shared_ptr<StmtNode> node);
```

功能说明：生成 if 语句的 IR。创建 thenBB、elseBB（如果存在）和 mergeBB 基本块，计算条件值并生成条件跳转，分别生成 then 和 else 分支的 IR，最后设置当前基本块为 mergeBB。

4.14 设计思想总结

核心设计原则：

1. 访问者模式：为每种 AST 节点类型定义专门的访问函数，实现关注点分离

2. **SSA** 形式：使用 alloca/store/load 模式实现局部变量，符合 LLVM IR 的 SSA 要求

3. 基本块管理：每个函数有入口基本块，控制流语句创建新基本块，使用 Phi 节点合并控制流

4. 符号表作用域：使用作用域栈管理变量可见性，支持嵌套作用域

5. 类型系统：统一的类型映射和类型提升规则，确保类型安全

6. 常量折叠：编译时计算常量表达式，减少运行时开销

7. 短路求值：逻辑运算符使用控制流实现短路，提高效率

8. 错误处理：在 IR 生成过程中检查未定义变量、类型不匹配等错误

IR 生成示例： 输入 AST：

```
CompUnit
├── VarDecl (int a = 10)
└── FuncDef (int main() { return 0; })
```

生成的 LLVM IR：

```
@a = global i32 10

define i32 @main() {
entry:
    ret i32 0
}
```

4.15 符号表实现细节

4.15.1 符号表数据结构

SymbolInfo 结构：

```
struct SymbolInfo {
   Value* value; // IR中的Value指针
   Type* type; // 类型
   bool isConst; // 是否为常量
   bool isGlobal; // 是否为全局变量

   SymbolInfo() : value(nullptr), type(nullptr),
               isConst(false), isGlobal(false) {}
   SymbolInfo(Value* v, Type* t, bool c, bool g)
      : value(v), type(t), isConst(c), isGlobal(g) {}
};
```

Scope 类实现：

```
class Scope {
public:
   std::map<std::string, SymbolInfo> symbols;

   SymbolInfo* lookup(const std::string& name); // 在当前作用域中查找符号
   bool insert(const std::string& name, const SymbolInfo& info); // 在当前作用域中插入符号
};
```

功能说明：

- `lookup()`: 在当前作用域中查找符号，找到返回 SymbolInfo 指针，否则返回 nullptr
- `insert()`: 在当前作用域中插入符号，如果符号已存在则返回 false（重复定义），否则返回 true

SymbolTable 类实现：

```
class SymbolTable {
private:
    std::vector<std::shared_ptr<Scope>> scopes; // 作用域栈

public:
    SymbolTable(); // 构造函数，创建全局作用域
    void enterScope(); // 进入新的作用域
    void exitScope(); // 退出当前作用域（保留全局作用域）
    bool isGlobalScope() const; // 判断当前是否在全局作用域
    SymbolInfo* lookup(const std::string& name); // 从内层作用域向外查找符号
    bool insert(const std::string& name, Value* value, Type* type, bool isConst); // 在当前作用域
        中插入符号
};
```

功能说明：

- enterScope(): 创建新的作用域并压入作用域栈
- exitScope(): 弹出当前作用域（但保留全局作用域）
- isGlobalScope(): 判断当前作用域栈大小是否为 1（即全局作用域）
- lookup(): 从内层作用域向外层查找符号，实现作用域嵌套查找
- insert(): 在当前作用域（栈顶）中插入符号

4.16 IRBuilder 实现细节

4.16.1 IRBuilder 核心接口

IRBuilder 类结构：

```
class IRBuilder {
private:
    BasicBlock *BB_; // 当前基本块
    Module *m_; // 模块
    Function *curfunc; // 当前函数

public:
    IRBuilder(BasicBlock *bb, Module *m);

    // 设置插入点
    void set_insert_point(BasicBlock *bb);

    // 整数运算指令
    BinaryInst* create_iadd(Value *lhs, Value *rhs);
    BinaryInst* create_isub(Value *lhs, Value *rhs);
    BinaryInst* create_imul(Value *lhs, Value *rhs);
```

```
    BinaryInst* create_isdiv(Value *lhs, Value *rhs);

    // 浮点运算指令
    BinaryInst* create_fadd(Value *lhs, Value *rhs);
    BinaryInst* create_fsub(Value *lhs, Value *rhs);
    BinaryInst* create_fmul(Value *lhs, Value *rhs);
    BinaryInst* create_fdiv(Value *lhs, Value *rhs);

    // 比较指令
    CmpInst* create_icmp_eq(Value *lhs, Value *rhs);
    CmpInst* create_icmp_ne(Value *lhs, Value *rhs);
    CmpInst* create_icmp_sgt(Value *lhs, Value *rhs);
    CmpInst* create_icmp_sge(Value *lhs, Value *rhs);
    CmpInst* create_icmp_slt(Value *lhs, Value *rhs);
    CmpInst* create_icmp_sle(Value *lhs, Value *rhs);
    CmpInst* create_fcmp_oeq(Value *lhs, Value *rhs);
    CmpInst* create_fcmp_ogt(Value *lhs, Value *rhs);
    // ... 其他比较指令

    // 内存操作指令
    AllocaInst* create_alloca(Type *ty);
    LoadInst* create_load(Type *ty, Value *ptr);
    StoreInst* create_store(Value *val, Value *ptr);

    // 控制流指令
    BranchInst* create_br(BasicBlock *if_true);
    BranchInst* create_cond_br(Value *cond, BasicBlock *if_true, BasicBlock *if_false);
    ReturnInst* create_ret(Value *val);
    ReturnInst* create_void_ret();

    // 类型转换指令
    ZextInst* create_zext(Value *val, Type *ty);
    SiToFpInst* create_sitofp(Value *val, Type *ty);
    FpToSiInst* create_fptosi(Value *val, Type *ty);

    // 函数调用指令
    CallInst* create_call(Value *func, std::vector<Value *> args);
};
```

功能说明：IRBuilder 提供创建各种 LLVM IR 指令的便捷接口，所有方法都在当前基本块（BB_）中创建指令：

- 整数运算：create_iadd, create_isub, create_imul, create_isdiv 等
- 浮点运算：create_fadd, create_fsub, create_fmul, create_fdiv 等

- 比较指令：create_icmp_*（整数比较），create_fcmp_*（浮点比较）
- 内存操作：create_alloca（栈分配），create_load（加载），create_store（存储）
- 控制流：create_br（无条件跳转），create_cond_br（条件跳转），create_ret（返回）
- 类型转换：create_zext（零扩展），create_sitofp（整数转浮点），create_fptosi（浮点转整数）
- 函数调用：create_call（函数调用）

4.17 源程序编译步骤（完整流程）

完整编译流程：

1. 词法分析：

```
SLRLexer lexer;
auto tokens = lexer.analyze(sourceCode);
```

2. 语法分析：

```
SLRParser parser;
bool success = parser.parse(tokens);
auto ast = parser.getAST();
```

3. **IR** 生成：

```
IRGenerator generator("source.sy");
generator.generate(ast);
std::string ir = generator.print();
```

命令行使用：

```
./compiler -i source.sy    # 完整编译（生成LLVM IR）
```

5 总结

本编译器实现了完整的编译流程：

1. 词法分析：将源代码转换为 Token 序列
2. 语法分析：将 Token 序列解析为 AST
3. **IR** 生成：遍历 AST 生成 LLVM IR 中间代码

关键特性：

- 支持两种词法分析器（DFA 状态机、NFA→DFA）
- 支持两种语法分析器（递归下降、SLR）
- 完整的 AST 节点体系
- 符号表作用域管理
- 类型系统和类型提升
- 常量折叠优化
- 短路求值
- 控制流图生成

6 小组成员分工与心得体会

6.1 成员一：陈添硕 （学号：**3023244225**）

主要分工： 语法分析器设计与实现

心得体会： 在实现语法分析器的过程中，深入理解了自顶向下和自底向上两种分析方法的区别与联系。通过实现递归下降分析器，掌握了如何将语法规则转化为递归函数，理解了向前看（lookahead）在解决选择冲突中的重要作用。在构建 SLR 分析表时，通过计算 FIRST 集和 FOLLOW 集，深刻体会到了形式化方法在编译器设计中的严谨性。语法分析器的实现让我认识到，良好的设计需要充分考虑语法的特点和实际应用场景。

6.2 成员二：李昊蓬 （学号：**3023244162**）

主要分工： IR 生成器设计和实现

心得体会： IR 生成是编译器中最核心也最复杂的部分。通过实现访问者模式遍历 AST 并生成 LLVM IR，深入理解了中间代码生成的过程。在处理类型提升、常量折叠、短路求值等特性时，深刻体会到了编译器优化的复杂性。符号表的作用域管理是一个重要挑战，通过实现作用域栈成功解决了变量可见性问题。控制流图的生成和 Phi 节点的使用让我理解了 SSA 形式的重要性。这次经历让我对编译器的后端有了更深入的认识。

6.3 成员三：许英帅 （学号：**3023244221**）

主要分工： 词法分析器设计和实现

心得体会： 通过实现词法分析器，深入理解了词法分析的基本原理。在实现 DFA 状态机的过程中，掌握了如何将正则表达式转化为确定有限自动机，理解了状态转换表的设计方法。关键字识别、标识符扫描、数字解析和注释处理等细节的实现让我认识到词法分析的复杂性。通过实现 NFA 到 DFA 的转换，进一步理解了形式语言理论在实际应用中的价值。词法分析器作为编译器的第一道关卡，其正确性对整个编译过程至关重要。

6.4 成员四：谢雨航 （学号：**3023244222**）

主要分工： 测试与调试，文档编写

心得体会： 在测试与调试过程中，学会了如何系统地验证编译器的正确性。通过设计测试用例覆盖各种边界情况，发现了许多隐藏的 bug。调试过程让我深刻理解了编译器各个模块之间的依赖关系，学会了如何定位问题所在。文档编写工作让我对整个编译器的架构有了全面的认识，通过整理和总结，不仅帮助了团队协作，也加深了自己对编译器原理的理解。这次经历让我认识到，测试和文档是软件开发中不可或缺的重要环节。