Lab 3: 中间代码生成

1. 中间代码生成简介

在本次实验中，我们将根据已经构建好的抽象语法树（AST）生成中间代码。中间代码生成是编译器实现过程中的重要步骤，它将源代码转换为一种中间表示形式，为后续的代码优化和目标代码生成打下基础。本次实验的核心在于实现`run_ir_constructor`函数，通过遍历语法树生成对应的中间代码指令。

2. 功能实现

2.1 总体结构

`run_ir_constructor`函数是本实验的主函数，调用初始化和生成中间代码的辅助函数，最终将生成的中间代码输出到指定文件。

```
void run_ir_constructor(Node* node, std::ofstream &fout) {
    init_Runtime();
    construct_ir(node, nullptr, nullptr);
    auto first_bb = module->getFunction("main")->begin();
    auto first_inst = first_bb->begin();
    for (auto iter = InitBB->begin(); iter != InitBB->end(); ++iter) {
        iter->insertBefore(*first_bb, first_inst);
    }
    module->print(fout, true);
}
```

2.2 实现逻辑及关键代码片段

2.2.1 初始化运行时环境

首先，`init_Runtime`函数负责初始化运行时所需的函数和类型，例如`getch`、`putint`等标准库函数。这些函数在中间代码生成过程中可能会被调用，因此需要提前定义。我们通过创建多个函数对象，并将它们存储在一个全局的运行时函数表中，以便在生成中间代码时可以方便地调用这些预定义的函数。

```
void init_Runtime() {
    ...
    Function* getch = Function::Create(nintegerty, false, "getch", nullptr);
    ...
    RuntimeFunc["getch"] = getch;
    ...
}
```

在这段代码中，我们定义了多个运行时函数，并将它们存储在一个`RuntimeFunc`映射表中。这样做是为了在生成中间代码时，可以方便地调用和检查这些预定义的函数。

2.2.2 生成中间代码

`construct_ir`函数通过递归遍历语法树的每个节点，根据节点类型生成相应的中间代码指令，遵循`Module-Function-BasicBlock-Instruction`的`Accipit-IR`层级，以下是几个典型的`if`分支的详细说明：

变量声明

对于变量声明节点，我们需要根据当前是否在全局作用域来生成相应的全局变量或局部变量声明指令，我们通过使用`cur_func`来记录变量声明时所处的函数位置，如果在初始的`Invalid`函数中，则说明是全局变量，需要将相应的指令插入到所有函数的开始，我们在最后其余函数都构建完成后再将这部分`Invalid`中的指令插入到`main`函数的开头即可，在以下是处理变量声明的代码片段：

```
else if (node->node_type == "IntegerVarDef") {
    auto temp_node = static_cast<NonTerminalNode*>(node);
    std::string name = static_cast<IdentLiteral*>(temp_node->child_list[0])->value;

    if (cur_func->getName() == "Invalid") { // 全局变量
        ...
    } else { // 局部变量
        ...
    }
}
```

函数定义

对于函数定义节点，我们需要处理函数的参数，并生成相应的函数头和基本块，然后递归生成函数体的中间代码。以下是处理函数定义的代码片段：

```
else if (node->node_type == "FuncDef") {
    auto temp_node = static_cast<NonTerminalNode*>(node);
    std::vector<Type *> params;
    std::string funcName = static_cast<IdentLiteral*>(temp_node->child_list[1])->value;

    if (temp_node->child_list.size() == 3) { // 没有参数
        ...
    } else { // 有参数
        Params.clear();
        findParams(temp_node->child_list[2]);
        ...
        construct_ir(temp_node->child_list[3], tb, fb);
    }
    ...
}
```

解释：
在处理函数定义时，我们首先处理函数的参数列表，通过调用`findParams`函数收集参数信息，并根据参数类型创建相应的`Type`对象。如果函数没有参数，我们直接生成函数头和基本块；如果函数有参数，我们处理参数传递，创建相应的局部变量，并生成存储指令。接着，递归处理函数体的节点，最后恢复初始函数和基本块。

条件语句

对于`if-else`语句节点，我们需要生成条件判断和分支跳转指令。以下是处理条件语句的代码片段：

```
else if (node->node_type == "IfElseStmt") {
```

```
    auto temp_node = static_cast<NonTerminalNode*>(node);
    auto ret_bb = cur_bb;
    cur_bb = BasicBlock::Create(cur_func);
    JumpInst::Create(cur_bb, ret_bb);  // 跳转到新基本块
    ret_bb = cur_bb;
    auto temp = BasicBlock::Create(cur_func, tb);  // 创建临时基本块
    cur_bb = BasicBlock::Create(cur_func, temp);  // 创建新的基本块并链接
    tb = cur_bb;

    // 处理if语句体
    construct_ir(temp_node->child_list[1], tb, fb);
    ...
}
```

解释:
在处理`if-else`语句时，我们首先创建新的基本块用于处理`if`和`else`语句体。通过调用`JumpInst::Create`生成跳转指令，使得控制流能够根据条件判断在`if`分支和`else`分支之间跳转，如此就实现了短路的要求。接着，我们递归处理`if`语句体，生成相应的中间代码。如果存在`else`分支，我们也递归处理`else`语句体，生成相应的中间代码。最后，将条件判断指令插入到相应的位置。

2.2.3 处理函数参数

在函数定义中，我们使用`findParams`函数来处理函数参数。`findParams`函数遍历参数列表节点，并为每个参数创建相应的`Param`对象，存储参数的类型和名称，这里同样是遍历了抽象语法树的节点，但是由于复用较多，因此单独抽出来作为一个函数。

```
int findParams(Node* node) {
    int result = 0;
    if (node->is_terminal()) {
        return 0;
    }
    auto temp_node = static_cast<NonTerminalNode*>(node);
    if (node->node_type == "FuncFParams") {...}
    else {
        if (temp_node->node_type == "IntegerFuncFParam") {...}
        else if (temp_node->node_type == "BracketFuncFParam") {...}
        else {std::cout << "Error: findParams" << std::endl;}
        result = 1;
    }
    return result;
}
```

解释:
在处理函数参数时，`findParams`函数遍历参数列表节点，并根据参数的类型创建相应的`Param`对象。如果参数是普通变量，我们设置`isArray`为`false`；如果参数是数组，我们存储数组的维度信息。所有的参数信息都存储在`Params`向量中，以便后续处理。

2.2.4 左值处理

由于`lab1-2`中并没有关于左值的严格检验，因此在构建抽象语法树和进行语法分析时，我们直接将从左值到最终结果之间的几点压缩了，这导致了`lab3`中无法从语法树中获得左值在赋值语句中的位置这一信息，进而使得中间代码会在使用左值表达式时多一条`LOAD`指令，导致使用的是左值的指针，因此我对语法分析构建语法树的方法进行改造，增加了中继节点`PrimaryExp`，以此来区分左值在左和在右。

```
PrimaryExp
    : LPAREN Exp RPAREN {
        auto r = new NonTerminalNode("PrimaryExp");
        r->add_child_back($2);
        $$ = static_cast<Node*>(r);
    }
    | LVal {
        auto r = new NonTerminalNode("PrimaryExp");
        r->add_child_back($1);
        $$ = static_cast<Node*>(r);
    }
    | Number {
        auto r = new NonTerminalNode("PrimaryExp");
        r->add_child_back($1);
        $$ = static_cast<Node*>(r);
    }
```

```
else if (node->node_type == "PrimaryExp") {
    auto temp_node = static_cast<NonTerminalNode*>(node);
    if (temp_node->child_list.size() == 1) {
        if (temp_node->child_list[0]->node_type == "LVal") {
            construct_ir(temp_node->child_list[0], tb, fb);
            ret->value = cur_inst;
            return ret;
        } else {...}
    } else {...}
}
```

3. 编译运行

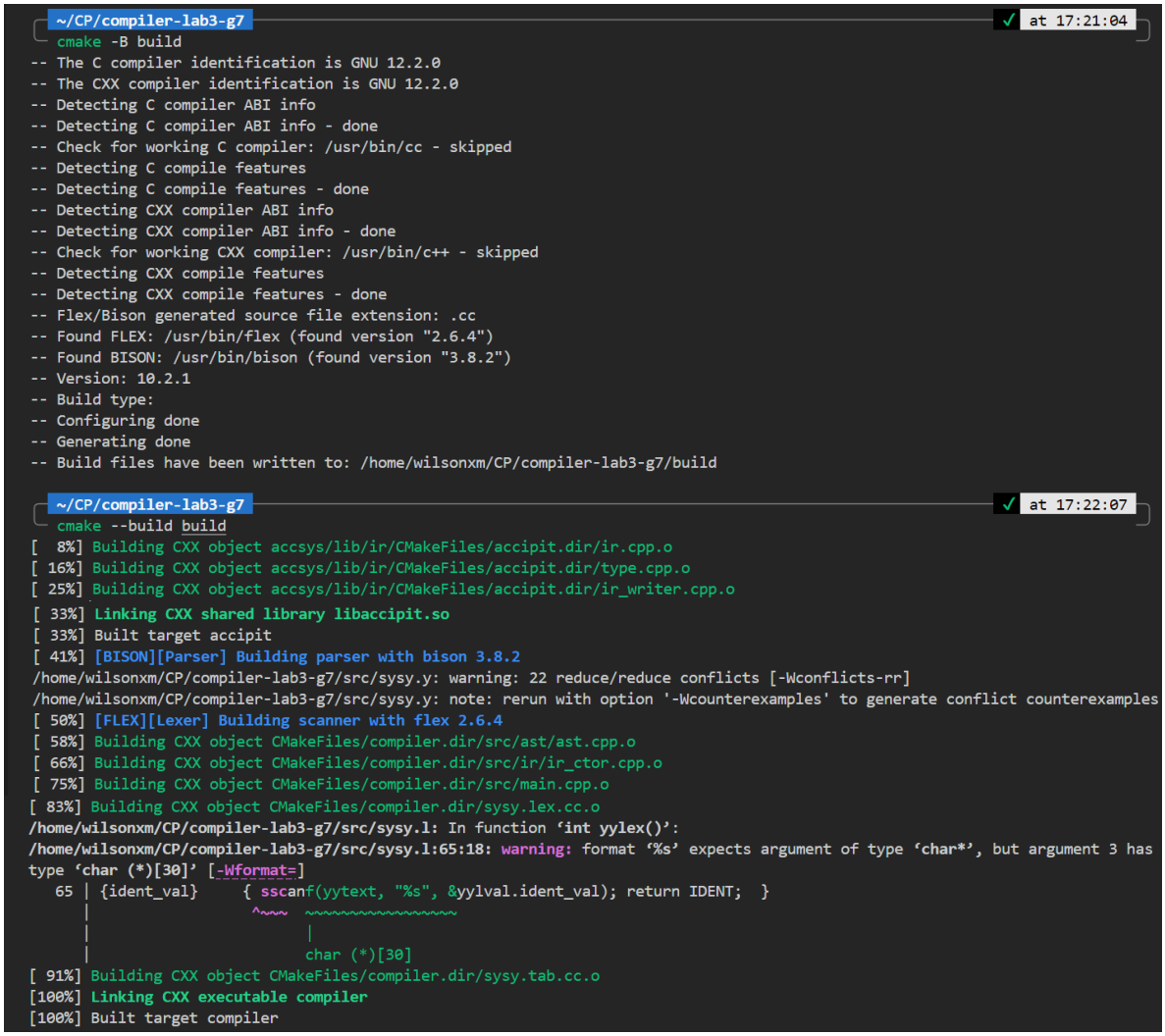
由于修改了部分`IR`相关的模板代码，因此请使用压缩包中提供的`accsys`

Build

进入Lab文件夹，执行下述命令：

```
cmake -B build
cmake --build build
```

正常情况下，终端会输出如下信息：



Run

整体测试：进入`tests`文件夹下，执行下述命令

```
python3 ./test.py ../build/compiler lab3
```

单文件测试：执行下述命令，生成单个`.sy`文件对应的中间代码`.acc`文件

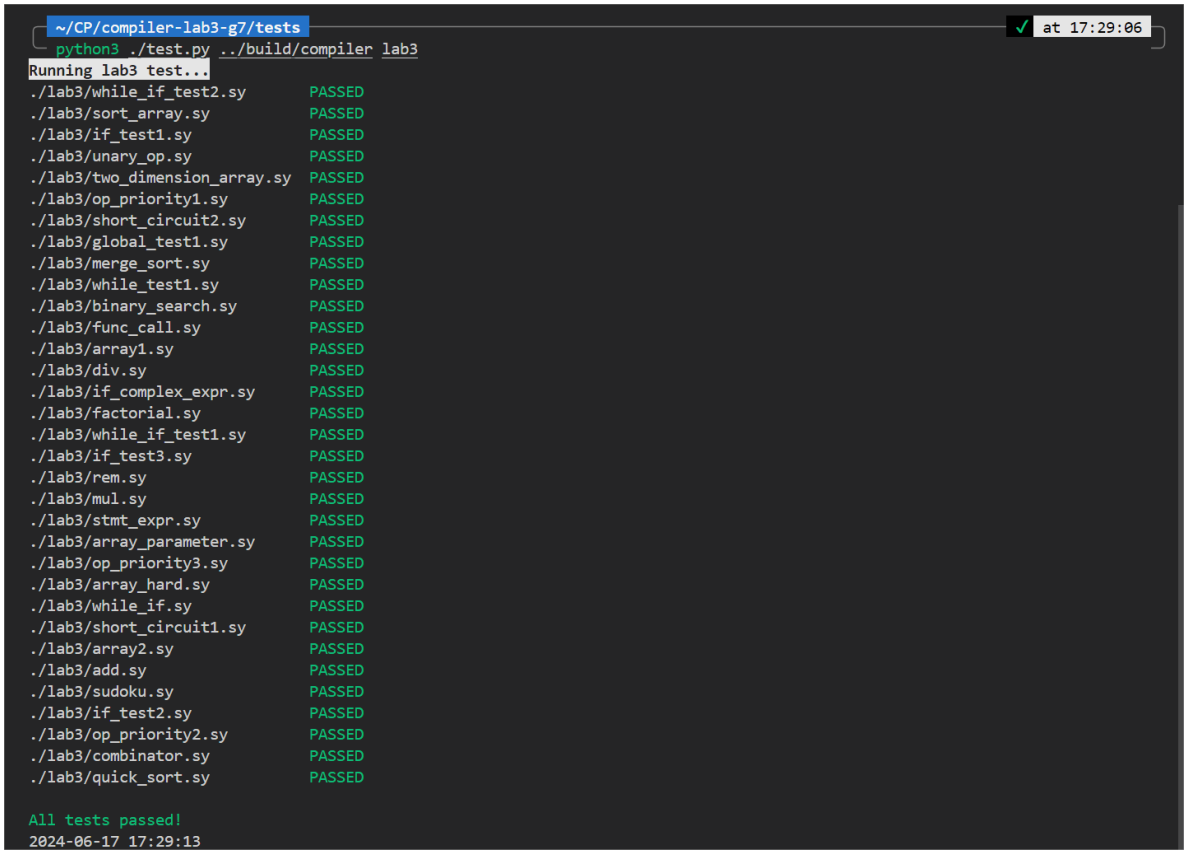
```
path/to/the/compiler <input_file> <output_file>
# eg. 如果在Lab文件夹下，则可以运行：
#     ./build/compiler ./tests/lab3/sudoku.sy ./tests/lab3/sudoku.acc
```

然后可以执行下述命令，用提供的解释器执行该`.acc`文件来判断中间代码的正确性

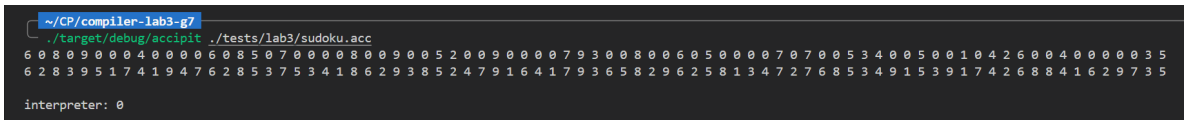
```
path/to/the/accipit <input_file>
# eg. 如果在Lab文件夹下，则可以运行：
#     ./target/debug/accipit ./tests/lab3/sudoku.acc
```

测试结果

整体测试：全部通过



单文件测试：以 `sudoku.sy` 为例，通过



4. 总结

本次实验通过实现 `run_ir_constructor` 函数完成了中间代码的生成。我们采用模块化的设计思想，将不同类型节点的中间代码生成逻辑进行了封装，确保了代码的可读性和可维护性。通过多个测试用例的验证，我们确认了中间代码生成的正确性和完整性。本实验为后续的代码优化和目标代码生成打下了坚实的基础。