HW5-实验报告

Q1.1: treep.hh中有许多tigerirp的 class，他们分别起到了什么作用?

1. 程序结构类

- **tree::Program**
  表示整个程序，包含所有函数声明（FuncDecl）的列表
- **tree::FuncDecl**
  表示函数声明，存储函数名、参数列表、基本块（Block）、返回类型等信息，是代码生成的起点
- **tree::Block**
  表示基本块（一组顺序执行的语句），包含入口标签、语句列表和出口标签，用于构建控制流图（CFG）

2. 语句类（Stm）

- **tree::Seq**
  将多个语句组合成顺序执行的语句列表（类似 { stmt1; stmt2; }）
- **tree::LabelStm**
  在代码中插入标签（如 L100:），作为跳转目标
- **tree::Jump**
  无条件跳转（goto L100），直接跳转到指定标签
- **tree::Cjump**
  条件跳转（if (x > y) goto L1; else goto L2;），根据比较结果选择跳转目标
- **tree::Move**
  赋值语句（t1 = t2），将源表达式（src）的值赋给目标（dst）
- **tree::Phi**
  SSA 形式的 Phi 函数，用于合并不同控制流路径的变量值（如 t1 = φ(t2@B1, t3@B2)）
- **tree::ExpStm**
  将表达式转换为语句，忽略返回值（如 a + b; 仅执行副作用）
- **tree::Return**
  函数返回语句（return x），携带返回值表达式

3. 表达式类（Exp）

- **tree::Binop**
  二元运算（如 a + b、x < y），包含操作符和左右操作数
- **tree::Mem**
  内存访问（如 *p），表示从地址加载值或向地址存储值
- **tree::TempExp**
  将临时变量（Temp）转换为表达式（如 t1 可直接参与运算）
- **tree::Eseq**
  组合语句和表达式（如 { stmt; return exp; }），先执行语句再求值表达式
- **tree::Name**
  将标签（Label）转换为地址表达式（如 &L100），用于函数调用或跳转
- **tree::Const**
  常量值（如 42），直接嵌入到表达式中
- **tree::Call**
  函数调用（如 obj.f(a, b)），包含调用目标、对象和参数列表
- **tree::ExtCall**
  外部函数调用（如 print("hello")），类似 Call 但目标为外部函数名

4. 辅助类

- **tree::Label**
  表示代码标签（如 L100:），用于控制流跳转（goto、if 等）
- **tree::Temp**
  表示临时变量（如 t100），用于寄存器分配或中间值存储
- **tree::Type**
  表示变量或表达式的类型（INT 或 PTR）

Q1.2: 相对于虎书中的Tiger IR，我们的Tiger IR+多了哪些内容，为什么需要多的这些内容?

- tree::Block（基本块）
  作用:表示一组顺序执行的语句，包含入口标签、语句列表。
  便于构建控制流图（CFG），支持后续的优化和分析。
- tree::Return（显式返回语句）
  作用:将返回值作为显式语句（而非原始 Tiger IR 中隐含的表达式）。
  更清晰地表示函数返回逻辑，便于优化和调试。
- tree::ExtCall（外部函数调用）
  作用:显式区分外部函数调用（如系统库函数 print）和普通函数调用。
  外部函数可能具有特殊调用约定（如寄存器传参、副作用处理），需要单独处理。
- tree::Phi（Phi 函数）
  作用:支持 SSA（静态单赋值）形式，用于合并不同控制流路径的变量值。
  原始 Tiger IR 没有显式的 SSA 支持，而现代编译器优化（如常量传播、死代码消除）依赖 SSA。
  例如，循环优化中需要处理 $x = \varphi(x_1, x_2)$ 来合并不同分支的变量值。

Q2: 在不带class的翻译情况下，需要关注运算（算数运算、比较运算、逻辑运算……）、赋值、条件（if、while）等成分的翻译，你是如何完成它们的翻译的?

1. 算数运算

算数运算（如 `+`、`-`、`*`、`/`）通过 `tree::Binop` 表示。
每个算数运算节点会递归访问左右操作数，并将它们转换为 `tree::Exp`，然后构造一个 `tree::Binop` 节点。

```
void ASTToTreeVisitor::visit(fdmj::BinaryOp* node) {
    node->left->accept(*this);
    auto left = newExp->unEx(&temp_map)->exp;

    node->right->accept(*this);
    auto right = newExp->unEx(&temp_map)->exp;

    newExp = new Tr_ex(new tree::Binop(tree::Type::INT, node->op->op, left, right));
}
```

2. 比较运算

比较运算（如 `<`、`>`、`==` 等）通过 `tree::Cjump` 表示。
在翻译时，会生成一个条件跳转语句，包含两个目标标签（true 和 false）。
这些标签会被后续的控制流逻辑填补。

```
void ASTToTreeVisitor::visit(fdmj::BinaryOp* node) {
    if (is_comparison_op(node->op->op)) {
        node->left->accept(*this);
        auto left = newExp->unEx(&temp_map)->exp;

        node->right->accept(*this);
        auto right = newExp->unEx(&temp_map)->exp;

        Label* t = temp_map.newlabel();
        Label* f = temp_map.newlabel();
        newExp = new Tr_cx(new Patch_list(t), new Patch_list(f),
                           new tree::Cjump(node->op->op, left, right, t, f));
    }
}
```

3. 逻辑运算

逻辑运算（如 `&&` 和 `||`）通过短路求值实现。

- 对于 `&&`，先翻译左操作数，如果为 false，则直接跳转到 false 标签；否则继续翻译右操作数。
- 对于 `||`，先翻译左操作数，如果为 true，则直接跳转到 true 标签；否则继续翻译右操作数。

```
void ASTToTreeVisitor::visit(fdmj::BinaryOp* node) {
    if (node->op->op == "&&") {
        auto left_cx = left_tr->unCx(&temp_map);
        auto right_cx = right_tr->unCx(&temp_map);

        auto L1 = left_cx->true_list;
        auto L2 = left_cx->false_list;
        auto L3 = right_cx->true_list;
        auto L4 = right_cx->false_list;

        L1->patch(temp_map.newlabel());
        L2->add(L4);

        newExp = new Tr_cx(L3, L2, new tree::Seq({left_cx->stm, right_cx->stm}));
    }
}
```

4. 赋值

赋值语句通过 `tree::Move` 表示。
左值和右值分别被翻译为 `tree::Exp`，然后构造一个 `tree::Move` 节点。

```
void ASTToTreeVisitor::visit(fdmj::Assign* node) {
    node->left->accept(*this);
    auto left = newExp->unEx(&temp_map)->exp;

    node->exp->accept(*this);
    auto right = newExp->unEx(&temp_map)->exp;

    newNode = new tree::Move(left, right);
}
```

5. 条件语句（if）

条件语句通过 `tree::Cjump` 和标签语句实现。

- 首先翻译条件表达式，生成一个 `tree::Cjump`。
- 然后翻译 `if` 和 `else` 分支，分别跳转到对应的标签。

```
void ASTToTreeVisitor::visit(fdmj::If* node) {
    node->exp->accept(*this);
    auto exp_cx = newExp->unCx(&temp_map);

    auto L_true = temp_map.newlabel();
    auto L_false = temp_map.newlabel();
    auto L_end = temp_map.newlabel();

    exp_cx->true_list->patch(L_true);
    exp_cx->false_list->patch(L_false);

    vector<tree::Stm*>* sl = new vector<tree::Stm*>();
    sl->push_back(exp_cx->stm);
    sl->push_back(new tree::LabelStm(L_true));

    node->stm1->accept(*this);
    sl->push_back(static_cast<tree::Stm*>(newNode));
    sl->push_back(new tree::Jump(L_end));

    sl->push_back(new tree::LabelStm(L_false));
    if (node->stm2) {
        node->stm2->accept(*this);
        sl->push_back(static_cast<tree::Stm*>(newNode));
    }

    sl->push_back(new tree::LabelStm(L_end));
    newNode = new tree::Seq(sl);
}
```

6. 循环语句（while）

循环语句通过循环标签和条件跳转实现。

- 首先生成循环的入口标签。
- 然后翻译条件表达式，生成一个 `tree::Cjump`。
- 最后翻译循环体，并跳转回入口标签。

```
void ASTToTreeVisitor::visit(fdmj::While* node) {
    auto L_while = temp_map.newlabel();
    auto L_true = temp_map.newlabel();
    auto L_end = temp_map.newlabel();

    node->exp->accept(*this);
    auto exp_cx = newExp->unCx(&temp_map);

    exp_cx->true_list->patch(L_true);
    exp_cx->false_list->patch(L_end);

    vector<tree::Stm*>* sl = new vector<tree::Stm*>();
    sl->push_back(new tree::LabelStm(L_while));
    sl->push_back(exp_cx->stm);
    sl->push_back(new tree::LabelStm(L_true));

    node->stm->accept(*this);
    sl->push_back(static_cast<tree::Stm*>(newNode));
    sl->push_back(new tree::Jump(L_while));
    sl->push_back(new tree::LabelStm(L_end));

    newNode = new tree::Seq(sl);
}
```

7. 数组初始化 (VarDecl->ARRAY)

数组初始化通过 `tree::ExtCall` 调用 `malloc` 函数分配内存，并设置数组的大小和初始值。

- 首先计算数组的大小（如果维数为空, 则需要考虑初始化长度）。
- 调用 `malloc` 分配内存，并将数组的大小存储在数组的第一个位置。
- 如果数组有初始值列表，则依次将初始值存储到数组中。

8. 数组存取 (ArrayExp)

数组存取通过计算数组的偏移量并访问对应的内存地址实现。

- 首先检查数组下标是否越界（`index >= size`），如果越界则调用 `exit(-1)` 终止程序。
- 计算数组元素的地址: `*(arr + (index + 1) * int_length)`。
- 返回对应的内存值。

9. 数组长度 (Length)

获取数组长度通过访问数组的第一个位置实现。

- 数组的长度存储在数组的第一个位置（`arr[0]`）。
- 直接返回该位置的值。

Q3.1: 你是如何重命名method的?

采用了 **类名+方法名** 对方法进行重命名
例如，类 `A` 中的方法 `foo` 会被重命名为 `A^foo`

具体实现如下:

- 在 `MethodDecl` 节点的翻译中，使用 `class_name + "^" + method_name` 作为方法的唯一标识符。
- 在类方法调用时，通过类名和方法名查找方法的偏移量，并生成对应的调用代码。

Q3.2: main method和class method的参数列表有何不同（hint:this）

`main` 方法和类方法的参数列表主要区别在于类方法会隐式包含一个 `this` 指针，而 `main` 方法没有。

Q3.3: 你是如何处理class method中的this的?

1. 在进入类方法时，生成一个 `this_temp` 指针的临时变量:

```
this_temp = new tree::TempExp(tree::Type::PTR, temp_map.newtemp());
```

2. 在访问类成员变量或方法时，通过 `this_temp` 指针计算偏移量，生成对应的访存代码:

```
auto var_mem = new tree::Mem(var_type, new tree::Binop(tree::Type::PTR, "+", this_temp, new
tree::Const(offset)));
```

Q3.4: 你是如何记录不同class的变量和方法的（hint:Unified Object Record）

使用了一个全局统一的对象记录来存储类的变量和方法信息

- 在生成类表时，计算每个成员变量和方法的偏移量，并存储在 `Class_table` 中:
- 在访问类成员变量或方法时，通过类表查找偏移量，并生成对应的内存访问代码。

Q3.5: 你是如何处理多态的?

在处理多态时，需要根据实际调用的对象类型来确定方法的真实类名

- 首先获取当前对象的类名 `class_name` 和方法名 `method_name`。
- 然后通过 `name_maps` 不断向上查找父类，
  直到找到第一个与当前类的返回形参结点地址不同的类 (即实现方法不同)
- 最后一个相同的类就是方法的真实类名 `method_real_class`。

```
// 找到第一个return_formal不同的
auto cur_class_name = class_name;
auto par_class_name = name_maps->get_parent(class_name);
while (par_class_name != "") {
    auto cur_f_return = name_maps->get_method_return_formal(cur_class_name, method_name);
    auto par_f_return = name_maps->get_method_return_formal(par_class_name, method_name);
    if (par_f_return != cur_f_return)
        break;

    method_real_class = par_class_name;
    cur_class_name = par_class_name;
    par_class_name = name_maps->get_parent(cur_class_name);
}
```

Q3.6: 你是如何翻译有关class的操作的?

1. 类的初始化

在类的初始化过程中，我们为类的每个实例分配内存，并初始化其成员变量和方法表:

- 为类实例分配内存:

```
auto class_malloc = new tree::ExtCall(tree::Type::PTR, "malloc", class_malloc_args);
```

- 初始化成员变量(带初始化):

```
auto var_mem = new tree::Mem(tree::Type::PTR, new tree::Binop(tree::Type::PTR, "+",
class_temp, new tree::Const(offset)));
newNodes.push_back(new tree::Move(var_mem, array_init));
```

- 初始化方法表:

```
auto method_mem = new tree::Mem(tree::Type::PTR, new tree::Binop(tree::Type::PTR, "+",
class_temp, new tree::Const(offset)));
auto method_nameExp = new Name(temp_map.newstringlabel(method_real_class + "^" +
method_name));
newNodes.push_back(new tree::Move(method_mem, method_nameExp));
```

2. 访问类变量

通过 `this` 指针和变量的偏移量计算变量的地址，并生成访存代码:

```
auto var_mem = new tree::Mem(var_type, new tree::Binop(tree::Type::PTR, "+", this_temp, new
tree::Const(offset)));
```

3. 访问类方法

通过 `this` 指针和方法的偏移量查找方法的入口地址，并生成调用代码:

```
auto method_mem = new tree::Mem(tree::Type::PTR, new tree::Binop(tree::Type::PTR, "+", objExp,
new tree::Const(offset)));
auto method_call = new tree::Call(return_type, method_name, method_mem, args);
```

Git Graph