HW4-实验报告

代码实现

[详见仓库][xht03/FudanCompilerH-25: Lab repo of FDU Compiler 2025 (H).](https://github.com/xht03/FudanCompilerH-25)

```
(base) keats@OMEN-Yanxu:~/FudanCompilerH2025/HW4$ git log --graph
* commit 11b70ef85a419cd60531947017198b0850fce509 (HEAD -> master)
| Author: xht03 <1620318777@qq.com>
| Date:   Thu Apr 10 23:27:51 2025 +0800
|
|     seemingly completed without bug.
|
* commit 561c881021fd1284f05f00df4b76503ac5a6f386
| Author: xht03 <1620318777@qq.com>
| Date:   Thu Apr 10 23:11:00 2025 +0800
|
|     If not support nested If.
|
* commit 03a7b6b27a01f21463909db5bd98e87ac1a467ab
| Author: xht03 <1620318777@qq.com>
| Date:   Thu Apr 10 23:03:01 2025 +0800
|
|     segmentation fault.
|
* commit e63b144b9e250312b92c21fdd6cae3a8ba46ba93
| Author: xht03 <1620318777@qq.com>
| Date:   Thu Apr 10 21:39:07 2025 +0800
|
|     prepare to accomplish Tr_exp.
|
* commit cd43f461cbbb2a4554f6380c64e95c9b93b9e0ee
| Author: xht03 <1620318777@qq.com>
| Date:   Thu Apr 10 20:20:57 2025 +0800
|
|     simplify generate_class_table and so on, to debug HW4
|
* commit 652bb712d13ab4136735530ef290c54c5b613b70
| Author: xht03 <1620318777@qq.com>
| Date:   Thu Apr 10 20:13:06 2025 +0800
|
|     finish generate_class_table and so on.
|
* commit b3d6cfe3c0be7dce47cd305f148e88302d09543e
| Author: xht03 <1620318777@qq.com>
| Date:   Thu Apr 10 19:25:07 2025 +0800
|
|     finish visitor.
|
* commit 283fb615db6e4ac61f4a409b1298433b735cfbed (origin/main)
| Author: xht03 <1620318777@qq.com>
| Date:   Thu Apr 10 18:27:37 2025 +0800
|
|     almost finish writing.
|
```

思考题

Q1.1: `treep.hh` 中有许多 tiger irp 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
- **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。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）。
- `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。
- `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。

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。

```
    // 算数运算
    vector<string> algo_op = { "+", "-", "*", "/" };
    if (find(algo_op.begin(), algo_op.end(), op) != algo_op.end()) {
        auto left = left_exp->unEx(temp_map)->exp;
        auto right = right_exp->unEx(temp_map)->exp;
        visit_exp_result = new Tr_ex(new tree::Binop(result_type, op, left,
right));
        return;
    }
```

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。

```
    // 比较运算
    vector<string> logic_op = { "==", "!=", "<", ">", "<=", ">=" };
    if (find(logic_op.begin(), logic_op.end(), op) != logic_op.end()) {
        auto left = left_exp->unEx(temp_map)->exp;
        auto right = right_exp->unEx(temp_map)->exp;

        // 构造CJump
        Label* t = temp_map->newlabel();
        Label* f = temp_map->newlabel();
        auto cjump = new tree::Cjump(op, left, right, t, f);

        // 添加修补列表
        auto true_list = new Patch_list();
        auto false_list = new Patch_list();
        true_list->add_patch(t);
        false_list->add_patch(f);
        visit_exp_result = new Tr_cx(true_list, false_list, cjump);
```

```
        return;
    }
```

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。

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) {
    // 首先访问左侧表达式
    node->left->accept(*this);
    tree::Exp* dst = visit_exp_result->unEx(temp_map)->exp;

    // 访问右侧表达式
    node->exp->accept(*this);
    tree::Exp* src = visit_exp_result->unEx(temp_map)->exp;

    visit_tree_result = new tree::Move(dst, src);
}
```

5. **条件语句**

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。

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) {
    // 条件表达式
    node->exp->accept(*this);
    Tr_cx* cond_exp = visit_exp_result->unCx(temp_map);

    auto L1 = cond_exp->true_list;
    auto L2 = cond_exp->false_list;

    auto L_while = temp_map->newlabel();
    auto L_true = temp_map->newlabel();
    auto L_end = temp_map->newlabel();

    L1->patch(L_true);
    L2->patch(L_end);

    vector<tree::Stm*>* stm_list = new vector<tree::Stm*>();
```

```
    stm_list->push_back(new tree::LabelStm(L_while));
    stm_list->push_back(cond_exp->stm);

    stm_list->push_back(new tree::LabelStm(L_true));
    if (node->stm) {
        current_loop_start_label = L_while;
        current_loop_end_label = L_end;
        node->stm->accept(*this);
        auto stm = static_cast<tree::Stm*>(visit_tree_result);
        stm_list->push_back(stm);
    }
    stm_list->push_back(new tree::Jump(L_while));
    stm_list->push_back(new tree::LabelStm(L_end));

    visit_tree_result = new tree::Seq(stm_list);
}
```

Q3: 你是如何重命名`method`的? `main method`和`class method`的参数列表有何不同（hint: `this`）？你是如何处理`class method`中的`this`的？你是如何记录不同`class`的变量和方法的（hint: `Unified Object Record`）？你是如何处理多态的？你是如何翻译有关`class`的操作（初始化、访问类变量、访问类方法……）的？请用中文语言描述（不要包含大段代码），并在实验报告中包含。

1. **方法重命名**

在`ASTToTreeVisitor::visit(fdmj::MethodDecl* node)`方法中，通过组合类名和方法名创建唯一标识符：

```
string method_name = current_class_name + "^" + node->id->id;
```

使用`^`符号连接类名与方法名，确保即使在不同类中存在同名方法也能正确区分。例如，A类的foo方法会被命名为"A^foo"。

2. **main方法与类方法的参数列表区别**

- **main方法**：不包含this指针，只有普通参数（在示例代码中参数列表为空）
- **类方法**：始终将this指针作为第一个参数，然后才是方法声明的其他参数：

```
// 第一个参数是 this 指针
tree::Temp* this_temp = temp_map->newtemp();
args->push_back(this_temp);
```

3. **this指针的处理**

对于类方法，this指针被作为特殊的局部变量处理：

- 创建一个临时变量表示this指针
- 将其作为方法的第一个参数
- 在方法变量表中注册this：

```
method_var_table->var_temp_map->insert({"this", this_temp});
method_var_table->var_type_map->insert({"this", tree::Type::PTR});
```

- 在方法内部可以通过查找变量表访问this引用

4. **统一对象记录模式**

使用`Class_table`类实现统一对象记录模式（Unified Object Record）：

- 所有类共享同一个类表布局
- 使用`var_pos_map`记录变量在对象内存中的偏移量
- 使用`method_pos_map`记录方法在虚函数表中的索引位置
- 这种设计使得所有对象的内存布局一致，可以通过相同的偏移量机制访问字段和方法

5. **类相关操作的翻译**

- **类初始化**：分配内存并设置虚表指针，可能调用构造函数
- **访问类变量**：
  - 获取变量偏移量：`offset = class_table->get_var_pos(var_name)`
  - 计算变量地址：`对象指针 + 偏移量`
  - 通过内存访问操作获取/设置值
- **调用类方法**：
  - 找到对象虚表
  - 通过方法偏移量确定方法入口地址
  - 将对象指针作为第一个参数传入
  - 调用方法