lab3 实验报告

PB20051150　陆子睦

实验要求

1. 阅读cminus-f 的语义规则
2. 阅读LightIR 核心类介绍
3. 阅读实验框架，理解如何使用框架以及注意事项
4. 修改 `src/cminusfc/cminusf_builder.cpp` 来实现 IR 自动产生的算法，使得它能正确编译任何合法的 cminus-f 程序。

实验难点

首先是对于实验框架的阅读，感觉需要不少精力才能看懂。有很多代码一开始不知道是干什么的，而且对于每一个变量的类型理解也不容易一下看懂。同时，由于代码中有很多继承之类的，所以有时候类之间的关系容易搞混，而且 auto 类型也给代码的阅读造成了一些困难。

其次是如何处理节点处理之间信息的传递，如何设计全局变量（由于一开始我并没有看到可以设全局变量，导致浪费了很多时间想如何用已经定义的变量来传）。

另外一个难点是如何使用 LightIR 的函数，由于对于那些函数并不是太熟，所以需要时常取查找需要的函数，导致整个编程过程比较缓慢（有时还由于函数用错了，所以出现了一些难调的bug）。

实验设计

全局变量

```
// store temperary values so that the result of the last node processed can be
seen in current node
Value *val_pre = nullptr;
// whether the scope has been entered or not
// is used when declaring a function and a scope is entered before the compound
statement
bool is_entered = false;
// whether the value returned is the address of variable or the value it stores
bool is_address = false;
```

`visit` 函数没有可传递的参数也没有返回值，所以节点之间信息传递主要靠全局变量。

首先显然很多 `visit` 函数都需要返回一个 `Value`，而函数又没有返回值，所以很自然想到需要这样一个全局变量，由于继承的关系，只需要把全局变量设为 `Value*` 类型就可以了。这样每一次只需要用全局变量把函数的返回传过来，然后存起来，就可以了。

然后我们会发现这样的在处理 `ASTVar` 节点的时候有两种情况，一种是变量在左值的位置，需要返回这个变量的地址，另一种是变量在右边的情况，需要返回它的值。而这个判断是在父节点确定的，所以需要定义一个全局变量 `bool is_address` 变量，从而标志是需要返回指针还是值。

第三个全局变量是在处理 `void CminusfBuilder::visit(ASTCompoundStmt &node)` 函数的时候添加的。一开始并不知道这个函数还缺失什么情况，后来经过分析发现，由于在函数定义的时候需要 `enter scope`，并插入形参，而这之后进入 `compound statement` 之后如何不加判断就会再次 `enter scope` 导致重复添加，所以需要一个全局变量来标记是否存在这种情况。所以添加了一个全局变量 `bool is_entered`.

类型转换

根据实验文档有以下五种类型转换。

- 赋值时
- 返回值类型和函数签名中的返回类型不一致时
- 函数调用时实参和函数签名中的形参类型不一致时
- 二元运算的两个参数类型不一致时
- 下标计算时

二元运算两个参数类型需要检测是否一致，如果不一致需要进行类型转换。由于判断较复杂，且比较重复，所以封装成一个函数，函数如下：

```
bool CminusfBuilder::convert(Value **l_ptr, Value **r_ptr) {
    auto &l_val = *l_ptr;
    auto &r_val = *r_ptr;
    bool is_integer;
    if (l_val->get_type() == r_val->get_type()) {
        is_integer = l_val->get_type()->is_integer_type();
    }
    else if (l_val->get_type() != r_val->get_type()) {
        is_integer = false;
        if (l_val->get_type()->is_integer_type()){
            l_val = builder->create_sitofp(l_val, FLOAT_T);
        }
        else{
            r_val = builder->create_sitofp(r_val, FLOAT_T);
        }
    }
    return is_integer;
}
```

其中返回值是标记最后运算类型是 `int` 还是 `float`，这样可以方便决定是需要使用哪个类型的指令。

其他几种情况只需要调用 `create_fptosi` 和 `create_sitofp` 函数按照情况转换一下就可以了。比如下标显然是 `int`，如果不是就转换一下。赋值需要和变量类型相同，函数调用需要和形参类型相同，返回值需要和返回值类型相同，判断比较简单，直接转换即可。

实验总结

本次实验中，我练习了阅读较大的代码框架的能力，进一步加深了对于访问者模式的理解。也增强了代码能力，学会了如何根据函数名和参数类型，调用封装好的函数。

同时，我更加清晰地了解了中间代码生成的过程，知道了如何基于已经生成的AST树来生成中间代码。

实验反馈 （可选 不计入评分）

本次实验有一定挑战性，代码量较大，不过实验收获也是很大的。