lab3 实验报告

PB20000277 孙昊哲

实验要求

1. 完成`cminusf_builder.cpp`，使得可以将`cminus`源代码编译成`IR`指令
2. 熟练使用`GDB`等调试工具
3. 熟悉C++语言代码，较为熟练使用`G++`等开发工具

实验难点

1. 不熟悉访问者模式
2. 实验文档内容较少，需要自己探索的东西太多
3. 虽然提供了打印语法树的样例，但是能够提供的帮助非常有限
4. 使用`LightIR API`接口生成`IR`指令不够熟悉

实验设计

1. 本次实验加入了除了助教已经加入的三个全局变量`tmp_val`,`cur_fun`,`pre_enter_scope`外，我们额外添加了一个全局变量`var_l`,用来表明在`CminusfBuilder::visit(ASTVar &node)`函数中，应该求左值还是右值。也就是说对于左值和右值我们在这个函数中的处理逻辑是不同的
    - 对于左值来说，我们希望求得他的地址，也就是说我们要将一个数赋给这个地址，不需要`load`指令，仅需求得该变量的地址，并将这个地址存入`tmp_var`这个全局变量当中；
    - 而对于右值来说，我们就希望得到他的数值，也就是说我们要将这个`var`中的数值`load`进去`tmp_var`，方便后续做各种运算
2. 本次实验我们加入了一系列错误处理的机制
    - 在`CminusfBuilder::visit(ASTVar &node)`函数中，我们会检测当前引用的变量是否已经在当前作用域中声明，对于没有声明的变量，我们会有合适的报错机制来应对
    - 在`CminusfBuilder::visit(ASTVarDeclaration &node)`函数中，我们会检查是否已经在作用域中声明，若已经声明过，我们将提供相应的报错信息
    - 在`CminusfBuilder::visit(ASTVar &node)`中，我们会实现对于数组的上下标检测，要求数组的`num`大于等于零，若不在该范围内，我们将报错
3. 本次实验中我们针对`create_gep`函数提供了如下的设计
    - 因为我们惊喜地发现在`cminus`语法中，仅仅在函数中的形参需要使用`create_gep(ptr, {tmp_val})`这种调用方法，对于其他任何一种形式的数组参数都会使用`create_gep(ptr, {CONST_INT(0), tmp_val})`这种形式；
    - 造成上述地原因在于对于函数中的形参，我们使用下述方式声明

```
%3 = alloca i32*, align 8
store i32* %0, i32** %3, align 8
```

而对于其他地方声明的数组，我们采用类似(下面的声明方式是对于全局变量的，对于非全局变量只有声明地指令不同，而对于声明的类型是相同的)于如下的方式声明，

```
@x = common dso_local global [1 x i32] zeroinitializer, align 4
```

- 对于上述的情况，我们稍微扩展下了`Scope`对象，我们在存储变量的时候不仅存储变量`id`和变量`val`的键值对，我们还会存储该变量是否是函数中的形参，对于函数中的形参我们会有不同的`create_gep`格式

```
public:
    // push a name to scope
    // return true if successful
    // return false if this name already exits
    bool push(std::string name, Value *val, bool isFuncParam = false) {
        auto result = inner[inner.size() - 1].insert({name, val});
        isFParam[isFParam.size() - 1].insert({name, isFuncParam});
        return result.second;
    }

    bool is_func_param(std::string name) {
        for (auto s = isFParam.rbegin(); s != isFParam.rend(); s++) {
            auto iter = s->find(name);
            if (iter != s->end()) {
                return iter->second;
            }
        }

        return false;
    }

  private:
    std::vector<std::map<std::string, bool>> isFParam;
```

我们主要的任务是添加了`is_func_param(std::string name)`函数，根据变量的名称判断是否是函数的形参

4. 值得一提的是，对于`CminusfBuilder::visit(ASTVar &node)`中`val_l`的赋值一定需要小心，在使用过该变量后一定要立刻将这个变量置为`false`，不然在处理形如这样的表达式会遇到麻烦`x[n]`(这个bug调试了半天)
5. 需要注意的是`call`函数中需要注意到类型转换的问题，不然对于助教提供的前几个测试样例无法进行正常输出，这个bug又很隐秘，让人很难意识到是类型转换不正确的问题，觉得是自己浮点数运算的问题，浪费了很多时间在这上面
6. 本次实验中对于降低指令中的`IR`冗余没有过多的考虑，我们生成的`IR`冗余还是较多的，接下来我们可以考虑从如下的几点来考虑降低`IR`冗余：
    - 当前我们对于每一个错误处理都会生成一个新的`label`块

```
label:
  call void @neg_idx_except()
  ret i32 0
```

我们考虑将所有的报错都`br`到同一个`label`块，减少代码的冗余

- 对于一些编译器本身就能运算的东西，我们直接在编译器中将他算好，比如助教提供的test例子中经常会出现`output(24.68 / 2.)`这样的代码，我们考虑在编译器中就将他算好，在输出的`IR`直接输出`call void @output(float 12.34)`

7. 我们需要注意到每一个`BasicBlock`的结尾都应该以终止符结尾，所以我们需要给我们的每个`BB`块结束后检查是否有终止符结尾，对于没有终止符的我们应该为他们添加上适当的终止符
8. 并且我们需要注意类型转换，在下面五种情况我们需要使用类型转换：
   - 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

实验总结

1. 通过本次实验加深了对`IR`的理解，加深了对`Visitor Pattern`模式的了解
2. 通过本次实验熟练了`GDB`调试，并且对于`C++`以及面向对象的语言有了更深刻的体会
3. 同时因为该实验时间的紧迫性，我们仅仅初步生成了正确地代码，但我们并没有考虑编译器的鲁棒性，以及编译器对更多错误处理的反馈，并且我们生成的`IR`代码冗余度过高，我们希望能在今后对于这些不足进行一些针对性的优化

实验反馈 （可选 不计入评分）

无