lab3 实验报告

PB20111694 毛陈诚

实验要求

1. 阅读cminus-f 的语义规则成为语言律师，我们将按照语义实现程度进行评分
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 来实现自动 IR 产生的算法，使得它能正确编译任何合法的 cminus-f 程序

实验理解

1. 对访问者模式的理解
   利用accept()函数实现对抽象语法树的深度优先遍历，每次调用accept()函数意味着沿树自顶向下对子节点进行访问。访问者模式利用函数重载，面对不同类型输入节点时，可以调用对应的执行函数。
   譬如在Program访问函数中，由`program -> declaration-list`，我们调用accept()函数对其子节点`declaration-list`进行遍历

```
void CminusfBuilder::visit(ASTProgram &node) {
    // program -> declaration-list 深度优先遍历
    for(auto n : node.declarations){ // n 获取 declaration-list
        n->accept(*this);  // visit all declaration-list
    }
}
```

2. 对实现代码的理解
   本实验是对实验2手动完成的编译器的自动化实现。
3. 对类型以及类型转换,类型判断的理解与实现(特别是数组与指针)
   如下代码，在`main`函数中数组`a[10]`的`a`在`scope`里存储的是指向一段10个元素数组的指针即`*[int,10]`,而在`f`函数中传参`c[]`中，存储的是int的二级指针,即`int **`,所以这两种情形，使得在处理`main`函数中的`a[3]`与`f`函数中的`c[3]`是不同的。此外在`main`函数调用`f`函数也要注意传递`a`的地址而不是`a[10]`这个数组。(但貌似在`ASTFunDeclaration`判断`params`的时候用的区分变量是`isarray`,容易造成混淆了)

```
void f(int c[]) {
    output(c[3]);
    g(c);
    return;
}
void main(void) {
    int a[10];
    a[3] = 1024;
    f(a);
    return;
}
```

具体实现在`ASTVar`中可见

- 首先要对数组与指针进行分开判断

```
auto ispointer = var->get_type()->get_pointer_element_type()->get_pointer_element_type();
auto is_array =  var->get_type()->get_pointer_element_type()->get_array_element_type();
```

- 对数组与指针不同操作(当`node.expresion != NULL`)如果是数组要直接调用`gep`返回数组第n个元素的指针`int/float *p`,如果直接调用`load`返回的就是数组了;如果是指针先调用`load`得到一级指针,在用`gep`进行基址偏移

```
if(is_array){
    tmp_val = builder->create_gep(var, {CONST_INT(0),tmp_val});//返回数组第n个元素指针
}
else if(ispointer){
    auto array_load = builder->create_load(var);//**a->*a
    tmp_val = builder->create_gep(array_load, {tmp_val});
```

- 当`node.expresion == NULL`

```
if(is_array){
    tmp_val = builder->create_gep(var, {CONST_INT(0), CONST_INT(0)});
else if(ispointer){
    tmp_val = builder->create_load(var);//返回指针/数（上一级）
}
```

函数内部对不同变量类型的分类，由于变量有浮点型，整型，布尔型，指针等类型，不同类型要分类讨论执行不同操作，十分麻烦。此外还有强制类型转换等问题。

实验设计

请写明为了顺利完成本次实验，加入了哪些亮点设计，并对这些设计进行解释。
可能的阐述方向有:

1. 如何设计全局变量
   前两个存储访问中产生的临时变量，后两个判断变量与函数的作用域，最后一个用来标定函数。

```
Value *tmp_val = nullptr;
Value *tmp_val2 = nullptr;
bool need_exit_scope =false;
bool pre_enter_scope =false;// function that is being built
Function *cur_fun = nullptr;
```

2. 遇到的难点以及解决方案
   - 第一点指针与数组判断在`实验难点`里已经阐述
   - 第二点还是函数`ASTVar`,对于不同函数要求TOKEN的不同值，比如对于赋值语句，左值要求返回的是存放数指针，右值要求返回的是数。于是设计了两个临时变量`tmp_val`,`tmp_val2`分别返回地址与数。

```
else if(is_int || is_float ){
tmp_val = builder->create_load(var);//返回
tmp_val2 = var;
}
```

在`ASTAssignExpression`中进行判断如果不是数则返回地址

```
Value *addr;
node.var->accept(*this);
addr = tmp_val;
if(addr->get_type() == INT32_T ||addr->get_type() == FLOAT_T )
addr = tmp_val2;
```

3. 块最后如果没有终结语句，需要有跳转语句来跳转到其他块，这个很容易漏掉。于是添加了判断语句,来增加跳转语句

```
if (builder->get_insert_block()->get_terminator() == nullptr)
    builder->create_br(next);//块最后一定要有终结语句！！！
    builder->set_insert_point(iffalse);
    node.else_statement->accept(*this);
if (builder->get_insert_block()->get_terminator() == nullptr)
    builder->create_br(next);
    builder->set_insert_point(next);
```

4. 强制类型转换问题

   - 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

     举一例说明因为在各个函数里大同小异，要注意的是不要漏掉`INT1_T`的情况

```
if(return_type == INT32_T){
 if(tmp_val->get_type() == FLOAT_T){
     tmp_val = builder->create_fptosi(tmp_val,INT32_T);
    }
    if(tmp_val->get_type() == INT1_T){
        tmp_val = builder->create_zext(tmp_val,INT32_T);
    }
    else if(return_type == FLOAT_T){;
        if(tmp_val->get_type() == INT32_T){
           tmp_val = builder->create_sitofp(tmp_val,FLOAT_T);
    }
    if(tmp_val->get_type() == INT1_T){
         tmp_val = builder->create_zext(tmp_val,INT32_T);
         tmp_val = builder->create_sitofp(tmp_val,FLOAT_T);
    }
```

实验总结

实验收获很大

- 了解很多`c++`的扩展知识，包括继承重载容器等的应用
- 深入理解了中间代码的生成过程。从用把代码解析为TOKEN，然后生成语法树，再生成抽象语法树，最后在将抽象语法树转化为中间代码
- 深刻感受到编译器的威力与无私奉献，自己生成的中间代码与clang生成的相比存在很多的冗余代码，值得进一步提升
  （~~但实验耗时约40h。。。所以收获时间比很小~~