lab3 实验报告

学号 PB20000034
姓名 林宸昊

实验要求

得到一个初级cimusf编译器，以使用访问者模式实现IR的自动生成。

实验难点 以及 实验设计

1. 处理Num:

- Num中的值可以通过`node.i_val`或`node.f_val`取得，难点在于每一个节点的处理函数都不会引入其它节点作为参数，换言之，只能通过全局变量传递。因此，引入第一个全局变量：

```
Value *cur_num = nullptr;
```

然后注意利用CONST方法将`node.i_val`或`node.f_val`转化为Value*类型；

2. 处理$\text{var-declaration} \rightarrow \text{type-specifier}\ \mathbf{ID}\ \mathbf{;} \mid \text{type-specifier}\ \mathbf{ID}\ [\ \mathbf{INTEGER}\ ]\ \mathbf{;}$

- 首先为分数组与非数组两类，再将二者分为全局与非全局两类，最后区分为整型和浮点型两类。通过`scope.in_global()`判断是否全局，全局变量采用全局变量的创建方法（如实验2），注意初始化为0；

3. 处理$\text{fun-declaration} \rightarrow \text{type-specifier}\ \mathbf{ID}\ (\ \text{params}\ )\ \text{compound-stmt}$

- 事实上根据`ast.hpp`，`ASTFunDeclaration`已经包含了`ASTParam`中的全部信息并可以直接处理，因此可以省去后者；
- 首先按照类型（函数参数的类型可以有void）将参数类型存入数组并创建该函数，然后根据所给代码引入第二个全局变量：

```
Function *cur_fun = nullptr;
```

使得后续基本块的创建可以明确知道应该创建于哪个函数块中；

- 然后对于传入的每一个实际参数，按照已经得到的参数类型对应存入值；
- 最后为了防止函数的`compound_stmt`无法使函数正常结束，给予一个必要的返回值；

4. 处理$\begin{aligned}\text{selection-stmt} \rightarrow\ & \mathbf{if}\ (\ \text{expression}\ )\ \text{statement} \\ & \mid \mathbf{if}\ (\ \text{expression}\ )\ \text{statement}\ \mathbf{else}\ \text{statement}\end{aligned}$

- 如实验3，需要三个基本块来描述选择分支语句，然后通过创建条件跳转实现。但需要注意的是，不同于实验2，`cur_num`可能是浮点数，负数等，且类型一般为`INT32`而不是`INT1`，不可直接用于比较，故需要额外定义一个用于比较的变量，通过一个比较将`cur_num`的值强制转换为用于比较的1位0，1：

```
Value *cond_val;
if(cur_num->get_type()->is_integer_type())
    cond_val = builder->create_icmp_ne(cur_num, CONST_INT(0));
else
    cond_val = builder->create_fcmp_ne(cur_num, CONST_FP(0.));
```

- 如果不存在else语句，那么跳转的另一个分支就是end分支，否则跳转至else分支，再直接跳转至end分支。值得注意的是由于先前创建了else专用的基本块，在没有else分支时会在ll文件中产生一个空的基本块，即代码冗余。一种方法是不提前建立基本块，而是将创建放进判断中，一种是通过调用`BasicBlock.h`中声明的`erase_from_parent()`函数将该块从所属的函数模块中删去，以防止空基本块的产生；
- 需要注意的是，不可为创建的基本块命名。如果给定特定名字，那么所有以此法创建的基本块都将强制变为此名，导致跳转时无法选定跳转对象，若不命名则会自动编号防止此问题；
- 迭代分支与选择分支代大体相似，不多赘述；

5. 处理$\text{return-stmt} \rightarrow \textbf{return ;} \mid \textbf{return}\ \text{expression}\ \textbf{;}$

- 需要注意的是返回值类型的优先级要高于右侧表达式得到的值的优先级，因此需要作类型转换；

6. 处理
$\text{expression} \rightarrow \text{var} = \text{expression} \mid \text{simple-expression}\$以及\$\text{var} \rightarrow \textbf{ID} \mid \textbf{ID}\ [\ \text{expression}]$

- `var`的出现有两种情况——被赋值，或者作为一个`factor`提供它的值。而处理它的函数需要知道属于哪一种情况，以给`cur_num`赋以正确的值进行传递——如果是被赋值，那么直接传递`var`这个地址，如果使用它的值，那么传递从`var`中取出来的值，因此我们引入第三个全局变量：

```
bool assign_var = false;
```

用于告诉处理`var`的函数是否需要赋值；

- 利用`scope.find()`得到对应的地址，然后对不同情况进行分类。如果是`ID`：
    - 如果需要赋值，直接传给`cur_num`；
    - 如果不需要赋值，那么地址有指向int,float,pointer以及作为一个数组名四种类型，这些类型都可以通过：

      ```
      cur_p->get_type()->get_pointer_element_type()->is_xxx_type();
      ```

      得到。如果是前三种类型，那么可以直接用load方法，如果是后一种，那么可以直接使用gep方法；
- 如果是`ID[expression]`：
    - 同样需要判断下标是否为负，并强制转换为整型；
    - 如果是pointer类型，则先载入地址中存放的基地址，然后根据索引找到对应值：

      ```
      auto array_temp = builder->create_load(cur_p);
        array_p = builder->create_gep(array_temp, {idx});
      ```

    - 如果是int或float类型，则直接根据索引找到对应值：

      ```
      array_p = builder->create_gep(cur_p, {idx});
      ```

    - 如果是array类型，则采用第二种gep方法（如实验3）：

```
array_p = builder->create_gep(cur_p, {CONST_INT(0), idx});
```

    - 如果需要赋值，则直接传递给`cur_num`；
    - 如果不需要赋值，则直接从`array_p`中取值。
- 处理实际赋值节点时，将`assign_var`改为`true`，并注意右值类型优先级低于左值，需要强制转换；

7. 处理 $\text{simple-expression} \rightarrow \text{additive-expression relop additive-expression} \mid \text{additive-expression}$:

- 需要做一个较复杂的处理，将opcode左右的操作数的类型统一，并返回一个控制值告诉函数统一后的类型；
- 在得到比较值之后，需要注意将得到的`INT1`类型比较值转换为`INT32`类型以满足其他函数的需求；
- 除关系符外，运算符处理方法类似，不多赘述；

8. 处理 $\text{call} \rightarrow \textbf{ID}\ (\ \text{args})$:

- 利用cpp的强制类型转换从scope中得到需要调用的函数；
- 由于同样存在传入参数可能与声明参数类型不匹配的问题，需要将传入的参数作类型转换。对于二者的比较，可以利用`Type.h`中`FunctionType`的内部函数`param_begin()`获取参数列表的头指针，然后与传入的参数逐个比较，修改后再传入`agrs`，最后调用该函数。

实验总结

1. c++语言的运用更加熟练——如全局变量的使用以及各种函数的处理；能够从头文件中获取有用的信息辅助完成函数；
2. 能够较熟练的运用访问者模式实现对cimunsf的编译与IR生成，能够通过实际代码实现简单的语义规则；
3. 能够从头文件定义的多种函数中选择有用的尝试对冗余进行优化，但空间应该还很大。

实验反馈 （可选 不计入评分）