《编译原理》Lab3实验报告

姓名：李诗宇

学号：3210100999

实验目的

利用`lab1`,`lab2`建立的语法树和符号表，将语法树AST转为中间代码。

实验亮点

1.基于AST遍历时的Dump方法和动态重建符号表

由于我在lab2中, 符号表建立与AST的生成是同步在`yyparse()`函数内进行的，而在lab3中，我的实现思路是在`yyparse()`结束之后，对其建立的AST调用`Dump()`method, 遍历整个AST，在符号表的帮助下生成`IR`，写入`output_file`:

```
std::unique_ptr<BaseAST> ast;
auto ret = yyparse(ast);//语法与语义分析，将生成的AST的根节点存入ast变量中

ast->Print();   //打印ast, 供debug
ast->Dump();    //生成IR
```

而此时，若仍然在`yyparse()`函数中建立符号表，到执行`ast->Dump();`时，符号表已经失效，因此我们需要在执行`Dump()`遍历AST时重新动态建立符号表。

为此，我在AST相应节点中的`Dump()`method中加入了重建符号表的动作，以`FunDefAST`节点为例:

```
class FuncDefAST : public BaseAST { //FuncDef ::= FuncType IDENT "(" [FuncFParams] ")" Block;
    public:
    std::string fun_type;
    std::string ident;
    std::unique_ptr<BaseAST> params;
    std::unique_ptr<BaseAST> block;

    void Dump() const override {
        string code;
        meta m;
        std::vector<Param> fun;
        if(pt1!=NULL){
                for (const auto& param : pt1->funparams) {  //建立符号表中函数的参数列表信息
                    Param elm = Param(param.is_lval, param.ident, param.value, param.bracketlist);
                    fun.push_back(elm);
                }
            }
        if(fun_type == "INT"){
            m = init_meta(FUNCTION_INT, {}, fun, 0);    //建立符号表中函数的信息
        }
        else m = init_meta(FUNCTION_VOID, {}, fun, 0);
        insert(ident, m, table_dump);   //建立符号表中函数的信息

        beginscope(table_dump);
        if(pt1!=NULL){
                for (const auto& param : pt1->funparams) {
                    vector<int> var = param.bracketlist;
                    meta n = init_meta(VARIABLE, var, {}, 0);
                    insert(param.ident, n, table_dump);     //将函数的参数列表作为局部变量加入符号表
                }
```

```
            }
        fprintf(yyout, "FUNCTION %s:\n", ident.c_str());    //打印IR函数名
        code += params->translate_exp("PARAM");          //调用params的成员函数打印IR下的参数列表
        fprintf(yyout, "%s", code.c_str());
        block->Dump();                                  //打印函数体
        endscope(table_dump);                           //结束作用域
    }
};
```

2.数组变量的存取操作

数组变量相对普通变量的难点在于它的地址偏移量计算。

当数组变量出现在Exp或赋值Stmt中时，我们需要：

1.查找符号表，找到该数组变量在IR表示绑定的临时变量`t*`,该临时变量中存储了数组变量的首地址

2.根据**符号表中数组定义的维数信息**与**访问该变量的index**，计算地址偏移量

3.首地址+偏移量，进行相应存取操作

实现如下：

```
m = lookup(ident, table_dump);  //查找符号表
for(int i = explist.size(); i < m->var.size(); i++){
    d *= m->var[i];             //m->var中存储数组定义的维数及size
}
p1 = generateTemp();
p2 = generateTemp();
p3 = generateTemp();
p4 = generateTemp();
p5 = generateTemp();
p6 = generateTemp();
p7 = generateTemp();
code += p3 + " = #" + to_string(d) +"\n";   //p3 = d
code += p2 + " = #0\n"; //p2 = offset
for(int i = explist.size() - 1; i >= 0; i--){
    ExpAST * pt = dynamic_cast<ExpAST *>(explist[i]);
    code += pt->translate_exp(p1);              //计算"[]"内的表达式，将结果存入临时变量p1
    code += p6 + " = " + p2 + "\n";
    code += p7 + " = " + p3 + " * " + p1 + "\n";
    code += p2 + " = " + p6 + " + " + p7 + "\n";
    // offset += d * bracketlist[i];            //计算地址偏移量（用IR语句实现）
    code += p4 + " = #" + to_string(m->var[i]) + "\n";
    code += p5 + " = " + p4 + " * " + p3 + "\n";
    code += p3 + " = " + p5 + "\n";
    // d *= m->var[i];
}
t1 = generateTemp();
//code += t1 + " = #" + to_string(offset) + "\n";
code += t1 + " = " + p2 + " + t" + to_string(m->id) + "\n"; //首地址+偏移量
code += place + " = *" + t1 + "\n";                     //进行存取操作
```

3.数组列表初始化

通过对文法规则的分析，依次处理初始化列表内的元素, 元素的形式无非就两种可能: 整数, 或者另一个初始化列表，因此设计如下的树状结构**(left child right sibling)**来存储初始化列表

```
class InitValListAST : public BaseAST {
    public:
    InitValListAST* next;  // 指向下一个节点，值或子列表
    bool isValue;            // 是否为值，0代表子列表，1代表值
    union {          // 值节点的值
            int value;
            InitValListAST* child; // 子列表节点
    } data;
    BaseAST * expr;
};
```

然后我采用北京大学编译原理实验指导中的建议，先按照规则对初始化列表补零，将多维数组打平到一维数组(`int * val`)，然后按照该一维数组对数组进行赋值：

```
int zero_fill(InitValListAST * root, int size, int &index, int * val, int * array_size, int length){
      InitValListAST* head = root;
      InitValListAST* current = root->data.child;
      while(current){
            if(current->isValue){   //遇到整数，直接一次填值
                  val[index++] = current->data.value;
            }
            else {  //遇到子列表节点
                  int t, mod;
                  for(t = 0, mod = 1; t < length; t++){ //检查当前对齐到了哪一个边界，
                        mod *= array_size[t];
                        if(t == length - 1) break;
                        else if(index % mod != 0) break;
                  }
                  int new_index = index + mod / array_size[t];  //然后将当前初始化列表视作这个边界所对应的最长维度的数
                  zero_fill(current, new_index, index, val, array_size, t-1);//组的初始化列表，并递归处理
                  index = new_index;
            }
            current = current->next;
      }
      printf("index : %d, size : %d\n" ,index, size);
      if(index > size) return -1;//数组越界
      else return 0;
}
```

其他

其余IR生成的**表达式代码生成**、**语句代码生成**、**条件判断语句代码生成**基本全部按照ZJU编译原理实验指导实现，故不加赘述。

本实验的实现还参考了**北京大学**与**南京大学**的编译原理指导。

编译

在提交的zip文件中包含了`Makefile`文件，只需进入Makefile所在的目录执行：

```
make
```

`src`文件夹中的源代码即可完成编译，生成执行文件`compiler`，即可进行测试。