编译原理Lab3实验报告

1. 完成情况：本次实验完成了所有选做要求，通过了OJ所有测试。
2. 编译方法：`make`生成可执行文件; `./parser <input>  out1.ir` 编译input测试用例，输出到`out1.ir`中。
3. 在实验2语义分析的基础上，我选择了新开一个文件`translator.c`，来实现IR的语法制导翻译，以取得更加良好的模块性。
4. 对IR的设计，我在`ir.h`中定义了如下结构体：

    1. ```
       struct Operand_{
           enum {VARIABLE, CONSTANT, TEMPORARY, ADDRESS, REFERENCE} kind;
           union {
               int value;
               Operand pt;
           } u;
       };

       struct InterCode_ {
           enum {ASSIGN, ADD, SUB, MUL, DIV,
                 ADDR, LOAD, STORE,
                 LABEL, FUNCT, JUMP, BRANCH,
                 RETURN, DEC, ARG, CALL, PARAM,
                 READ, WRITE} kind;
           union {
               struct {Operand right, left;} assign;
               struct {Operand result, op1, op2;} binop;
               struct {Operand result, op;} unop;
               int num;
               char *name;
               InterCode dest;
               Operand var;
               struct {Operand var, size;} dec;
               struct {InterCode dest; Operand op1, op2; enum {LSS, GRT, LEQ,
       GEQ, EQ, NEQ} relop;} branch;
               struct {Operand result; Function callee;} call;
           } u;
           InterCode prev, next;
       };

       struct Function_ {
           char *name;
           OpList params;
           InterCode entry;
           InterCode tail;
       };

       struct FunctionList_{
           Function func;
           FunctionList next;
       };
       ```
       ```

```
struct Module_ {
    FunctionList func_list;
    FunctionList func_tail;
};
```

2. 其中Module维护了一个Function的链表，Function则存有函数名，参数列表，第一条IR和最后一条IR（方便随时插入）
3. 对于Operand，我设计了五种类型：
    1. 三种基本类型：常量，变量，临时变量；对于这样的Operand, u.value中存放常量值/变量编号。
    2. 两种扩展类型：取地址，解引用，u.pt中存放指向的Operand
    3. 对一个基本类型，我实现了一个`makeAddress`函数来返回它的取地址类型；解引用同理。
4. 对于这些类型，都实现了他们的构造函数和打印函数（想念C++的构造函数）

5. 语法制导翻译部分，我选择了按照讲义上的翻译方法：模板进行实现，但是做了一些改动：翻译函数不返回IR，而是维护一个当前的Function，每次在Function->tail后插入一条指令。对于Exp的翻译，我选择让Exp返回Operand，这样就免去了传递place参数的麻烦。
6. 实现数组/结构体的Load/Store是相对比较困难的一部分。
    1. 对于结构体，需要获取偏离量。我修改了原先的语义分析，让符号表支持存储ident的偏移量，对于StructSpecifier，在得到它的类型后，更新structure中的所有变量的偏移量。
    2. 数组需要计算偏移量，因此我实现了一个计算类型大小的函数，只需要读取数组元素类型，并计算出常量，将其与index相乘就得到了offset。
    3. 无论是结构体还是数组都需要维护当前的类型，这部分只需要在读取到ID时查询符号表，并将其作为属性传回来即可。
    4. 对于数组的计算，我将Exp的计算分成了`translate_LExp`和`translate_Exp`：
        1. 我希望LExp能返回一个代表地址的Operand，而Exp能返回它的值。
        2. 以Exp => EXP1 ASSIGNOP EXP2举例，我希望这里的EXP1返回地址，而EXP2返回值。假设EXP1=a, EXP2=b, 那么LExp(a) = a, Exp(b) = t1, 并添加IR: `t1 = b`
        3. 在这种情况下，对Exp => EXP1 LB EXP2 RB 的计算：对EXP1采用LExp翻译，因为我希望得到这个数组的地址——对EXP2采用Exp翻译，因为EXP2是代表下标的右值。
    5. 选做部分允许传递数组/结构体参数，因此我遇到了不少问题：
        1. 对于参数列表来说，参数是RExp, 因此在目前的翻译模式下，传递数组参数，会多出不必要的对数组元素的取值。但是在传递一个参数前，是无法确定这是一个数组参数，还是一个值参数。我考虑了根据参数类型来判断，但这样很麻烦。因此我选择在`translate_Exp`中对当前翻译的类型进行判断，如果不是基本类型，就不进行取值。代码如下：
        2.
```
            case LB_NODE: {
                Operand base = translate_LExp(lhs_node);
                Operand addr = newTemp();
                Operand offset = newTemp();
                son = son->succ;
                Operand index = translate_Exp(son);
                left_type = left_type->u.array.elem;
                int size = get_type_size(left_type);
                Operand op = newConstant(size);
                InterCode ir1 = newBinaryIR(offset, index, op, MUL);
```

```
                InterCode ir2 = newBinaryIR(addr, base, offset,
ADD);
                if (left_type->kind != BASIC) {
                    insert_IR(func, ir1);
                    insert_IR(func, ir2);
                    return addr;
                }
                else {
                    Operand result = newTemp();
                    addr = makeReference(addr);
                    InterCode ir3 = newAssignIR(result, addr);
                    insert_IR(func, ir1);
                    insert_IR(func, ir2);
                    insert_IR(func, ir3);
                    return result;
                }
            }
            break;
            case DOT_NODE: {
                Operand base = translate_LExp(lhs_node);
                Operand addr = newTemp();
                son = son->succ;
                char *id = son->val.id;
                int offset = table_getoffset(id);
                Operand op = newConstant(offset);
                InterCode ir1 = newBinaryIR(addr, base, op, ADD);
                left_type = get_field(left_type, id);
                if (left_type->kind != BASIC) {
                    insert_IR(func, ir1);
                    return addr;
                }
                else {
                    Operand result = newTemp();
                    addr = makeReference(addr);
                    InterCode ir2 = newAssignIR(result, addr);
                    insert_IR(func, ir1);
                    insert_IR(func, ir2);
                    return result;
                }
            }
            break;
```

7. 除此之外的内容都比较基本，在建立完框架的基础上能简单地完成。
8. 总结：这次实验的过程中还是遇到了不小的挑战，对于IR的调试比较困难，发现了错误之后需要花不少时间去定位，找出错误。
9. 其他：虚拟机中的语法`DEC x [size]`返回的x竟然不是一个地址，但作为参数传递的结构体/数组是！我的感觉是这样从语法统一性上看不是很优美，也对实现造成了一些细节上的困难。