report

选做3.2

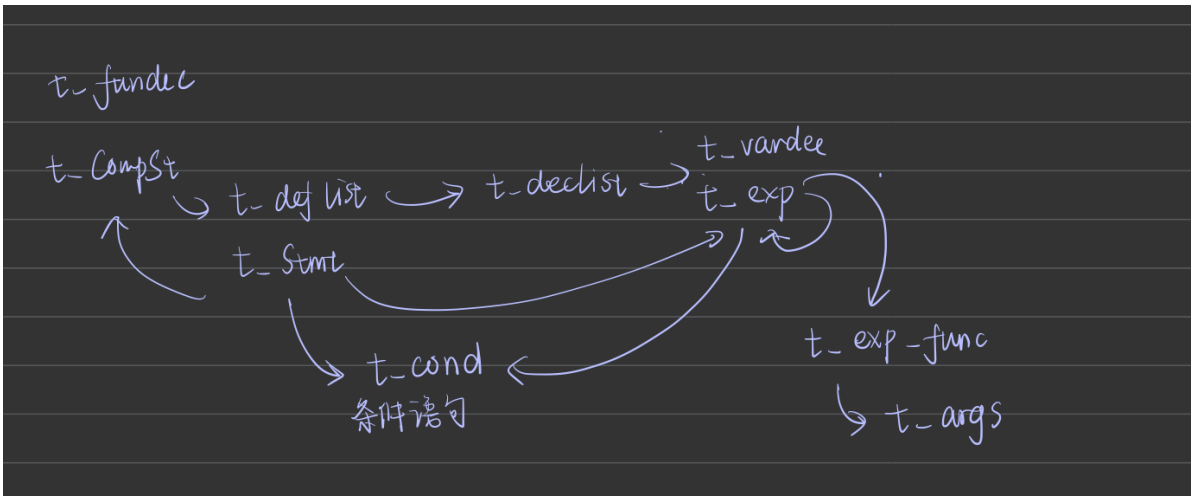
- 一维数组类型的变量可以作为函数参数（但函数不会返回一维数组类型的值）。
- 可以出现高维数组类型的变量（但高维数组类型的变量不会作为函数的参数或返回类值）。

我的理解： 需要处理的是数组单个元素的读写和数组整体的读写。参数传入本质是传地址。

结构设计

为了让代码保持便于维护的模块化结构，另外编写了模块在中间代码生成阶段重新遍历语法树。

与语法检查阶段不同，此次遍历不包括每一个节点 只在必要节点。经过观察，在只完成选做3.2的情况下，代码主要由函数构成（包括main函数） 因此入口模块是 FunDec 和 CompSt。中间节点的处理也基于语法正确的情况下做了情况的简化。



中间代码生成的设计

由于是程序是单线程线性的，因此采用的方法是向文件中按调用顺序写入中间代码。其中place变量既可能向被调用函数传递信息，也可能帮助调用函数接收信息，用途灵活。

place设计为结构体Reg以兼容argList是连续参数列表的情况，结构体的信息包括分配的变量名称，分配的空间大小（为数组准备信息），内部保存的是否是地址（为函数调用参数是数组准备）。同时在实验二的数据结构基础上给符号表表项信息添加了Reg属性，方便全局查找。

Exp的处理

包括普通运算和函数调用两种情况。Exp中设计大量关于place的操作，其中包括给空place填写信息，向被调用者传递变量信息，。如作为被操作数，需要向上层传递变量信息；而作为左侧操作数，接受调用者处理得到的结果。

数组的处理

修改了实验二中对数组的处理，把数组长度直接计算好，在生成DEC指令时直接输出。这一步也便于后续处理高维数组。

读写数组时，首先判断当前的变量var对应的是否是数组地址信息，如果不是需要加取地址符号；在传入参数时，Reg中的isAddr会被设置为真，则在函数体内不需要再加取地址符号。

计算offset

对于访问数组某一元素，由于计算公式所需要的信息包括每一维度的信息，而这些信息只有在访问到ID时才能获取，因此设计了一个递归调用的函数calOffset来完成取值的计算和中间代码生成。

```
//Exp是传入的数组表达式，offset是计算出的偏移量的中间代码形式，type是从高维向低维数组元素的类型信息，为了化简，可以计算出的常数部分保存在constant中
Reg calOffset(Node* Exp, char* offset, Type* type,int* constant)
{
    Reg id = NULL;
    if(Exp可以直接解析出ID)
    {
        Node* idNode = Exp->child;
        item = findSymbolItem(idNode.valName);//查询数组名
        *type = item->type;//记录下type信息
        id = item->reg;//取出数组对应的中间代码变量信息
    }
    else//不可以 依旧是数组的形式 但是已经解析出一层
    {
        id = calOffset(Exp->child,offset,type,constant);
        *type = (*type)->info.arrayInfo.elemType;//接下来计算需要的是该层数组元素的大小
    }
    if(Exp可以被解析成数组的形式而非只有ID)
    {
        解析偏移量信息idx;
        int size = 4;//假设最小单位是基本类型
        if((*type)->kind==ARRAY)
            size = (*type)->info.arrayInfo.size;
```

```
        //得到本层数组单元的大小
        if(idx是ID)
        {
            idIdx = findSymbolItem(idx);
            fprintf(code,"t1 := idIdx * #%d",size);
            if(offset不为空)
                fprintf(code,"t1 := t1 + %s",offset);
            fprintf(offset,"+ t1",);
        }
        else if(idx是int)
        {
            *constant = *constant + idx.val*size;
        }
    }
}
```

有了计算偏移量的函数，处理数组读写的任务变得更加清晰简单：

```
Reg arr = calOffset(Exp,offset,&type,constant);
Reg tmp = newReg(newTempName());
if(arr->isAddr)
    生成中间代码时不需要加取地址符号
else
    生成中间代码时需要加取地址符号
if(*constant!=0)
    fprintf(code,"tmp := tmp + %d",constant);
if(arrRW)
    fprintf(code,"*tmp := place");
else
    fprintf(code,"place := *tmp");
```

arrRW是用于标识数组读写状态的全局变量，由于单线程线性处理的性质，一次只会有一个数组被处理，多数情况下都是读（如作为操作数参与运算，数组元素作为参数传递），目前只有在数组在赋值符号左侧以及定义的时候会出现写操作。

数组赋值

根据实验二的说法，只要数组元素类型相同即可，不需要size相同，赋值按照填满为止或者源数组数据全部填入为止。高维数组可以认为展开后是一维数组。因此仿照循环，用中间代码硬编码了一段循环填入中间代码作为数组元素赋值操作。