实验三

171860572 侯策 171860572@smail.nju.edu.cn

171860588 史文泰 171860588@smail.nju.edu.cn

Operand思路

- `Operand`代表中间代码的一个操作数的类型，它们作为**语句四元式**的域填入。如下代码给出了各种`Operand`类型及其对应的使用到的**域类型**。

```
struct Operand_t{
    enum OperandKind kind;
    union{
        struct{
            int var_num;
            char* var_name;
        }variable;
//VARIABLE,ADDRESS,REFERENCE,STRUCTURE_ARRAY,STRUCTURE_STRUCTURE
        int num;          //TEMPORARY_VARIABLE,TEMPORARY_ADDRESS, LABEL_OPERAND
        int const_value; //CONSTANT
        char* name;       //FUNCTION_OPERAND
    }u;
    Operand next;
};
```

- **基础类型**及其对应的**域类型**：
  - `VARIABLE`：所有的变量（普通变量，数组变量，结构体变量），包含变量名及变量下标
  - `TEMPORARY_VARIABLE`：临时变量，包含临时变量下标
  - `CONSTANT`：常量，包含常量值
  - `FUNCTION_OPERAND`：函数名，包含函数名
  - `LABEL_OPERAND`：标号，包含标号下标
- 实际使用`Operand`完成操作数的保存和`InterCode`的生成时，设计了一些**特殊数据类型**用以实现不同的目标，如下：
  - **目标1**：希望能够通过类型名在中间代码打印时，知晓是否需要在其前面添加取地址(`&`)与解引用(`*`)。
    - `VARIABLE`类型对应中间代码中普通的`vi`（在程序中各种变量定义时会创建）
    - `REFERENCE`类型用于指示在生成中间代码时在其前面需要加上**&**的`VARIABLE`变体，表示该操作数需要取地址，用于后续操作，如访问结构体`x`时需通过`&x`获得`x`的首地址。
    - `ADDRESS`类型是需在其前面加上*****的`VARIABLE`变体，表示该操作数为地址类，访问时需要解引用。
    - `TEMPORARY_ADDRESS`类型是需在其前面加上*****的`TEMPORARY_VARIABLE`变体，表示该操作数为临时地址，如对于数组某个元素和结构体某个域的赋值，需要解引用。
  - **目标2**：一般来说，对于`X.Y`形式，处理之后的返回值为结构体某数据成员的索引地址，但当`Y`为结构体或数组类型时，需要在其生成的`Operand`节点中保存其`ID`，以用来在返回给上层后能通过`ID`在符号表中查询其具体类型，从而得到元素大小和域在结构体中的偏移量。

    - `STRUCT_ARRAY`类型，表示**结构体的数组域**，存入该数组的名称，用于后续使用数组特定元素时获得数组元素大小。
    - `STRUCTURE_STRUCTURE`类型，表示**结构体中的结构体或结构体数组类型**，存入该结构体的名称，用于后续使用结构体特定域时获得该域在结构体中的偏移量。

InterCode思路

- 采用**四元组**方式，利用双向链表组织所有的`InterCode`，对于每个`InterCode`类型，填充对应的`op1`，`op2`或`result`域。特殊处理如下：
    - 条件跳转语句由于条件不同无法统一在一个`InterCode`类型中，故细分为6个类型：`JE, JNE, JA, JAE, JB, JBE`，分别作为6种不同的跳转类型。
    - `x := *y`，`*x := y`，`x := &y`三个语法不作为独立的`InterCode`，而是将`*`和`&`作为一种特殊的`Operand`，在打印时加以区分。此时：
        - 对于`x := *y`，`*x := y`，`x := &y`，作为赋值语句构造。
        - 对于`IR`所支持的`x := *y + z`，`x := &y + *z`等，作为四则运算语句构造。
    - 对于`PARAM x`语句，无论`x`是普通变量还是结构体变量，均不打印`*`或`&`，但是若结构体变量`x`作为函数参数，则在后续使用时`x`作为结构体变量的指针，无需再使用`&x`获得地址。
    - 对于`READ x`语句，无论是否为地址类型，都不能解引用`*`，需新建`VARIABLE`类型的`Operand`插入`InterCode`结点。

数组赋值思路

- 数组赋值操作的特殊处理在`ASSIGN`语句中完成。在获得左操作数后判断是否为数组（包含**普通数组类型**`VARIABLE`**和结构体内部数组**`STRUCTURE_ARRAY`）。若是，则说明必然是数组赋值，右操作实也应为数组。获取右操作数后比较得出赋值个数，两者的数组元素所占大小，依次产生`ASSIGN`语句即可。需要注意的是：对于普通数组类型，需要取地址`&`得到其首地址，而对于结构体内部数组`STRUCTURE_ARRAY`类型，则获得的`Operand`返回值即为临时地址，代表着数组的首地址，无需取地址`&`。伪码如下：

```
leftOp = getLeftOperand
if(leftOp is ArrayType)                                 // 左操作数为数组
    rightOp = getRightOperand
    minSize = Min{leftOp.size, rightOp.size}            // 赋值元素个数
    length = leftOp.length                              // 元素长度
    if(leftOp/rightOp is Normal Array)
        address = new REFERENCE Operand                 // 需&获得首地址
    else
        address = new VARIABLE Operand                  // 无需&获得首地址
    for(i in range(size)){
        insert ASSIGN InterCode                         // 依次赋值
    }
else
    Normal ASSIGN                                       // 普通变量赋值
```

优化

- 在**中间代码生成过程中**对于**临时变量的生成**进行了一定的优化：

- 在处理`EXP`的分支中，参照实验手册中`translate_Exp`函数，为函数同时设置了`place`**参数和返回值**，参数`place`和函数返回值类型均为`Operand`类型。以此来实现初步的中间代码优化过程，比如：`x = y + z`，如果没有`place`参数，则生成类似`t1 = v1 + v2`，`v3 = t1`的两步代码，而如果使用`place`参数，将x作为参数传入`EXP->EXP PLUS EXP`的处理函数，则可以生成`v1 = v2 + v3`，仅需一句代码，如此，`place`参数的灵活使用，会令得到的中间代码中减少很多冗余的临时变量的传递。而当没有需要赋值的情形，利用返回值即可将得到的操作数节点返回给上层使用。使用`place`参数时，不需要使用返回值，使用返回值时不需要使用`place`参数，使得刚解析出的操作数节点既可在当前层参与生成中间代码，也可以返回给上层调遣，使用灵活。

- **常量的计算与折叠**：
  - **常量计算**，**常量折叠**，**过河拆桥**：遍历初步生成的中间代码，可以通过立即数之间的加减乘除计算出的临时变量计算出来，令其从加减乘除语句变为赋值语句。例如：`t1 = #3 * #11`转换为`t1 = #33`。计算出该句后，从该句向后遍历，将其中所有该临时变量均替换为立即数，用于后续继续常量计算。在上述例子中即后面出现的所有`t1`均替换为`#33`。由于后面所有对当前的临时变量的使用已经全部替换成了立即数，当前的这句临时变量赋值为立即数也就失去了意义（`t1 = #33`），将其在中间代码链表中删除，并从该处继续向后遍历，寻求下一个常量计算的机会。
  - 能够如此优化的原因：对于临时变量`ti`，仅当`ti`初始化时会被赋值，后续`ti`不会再变化，可以直接替换。相对应的，普通变量`vi`则无法直接替换，需要进行**常量传播**计算确定哪些为常量。
- `IF`**语句的优化**
  - 实际运行的结果显示如下的优化并不会显著改善运行代码条数，但仍不失为一种优化方式。

```
    IF x relop y GOTO Label1
    GOTO Label2
Label1:
        ……
Label2:
        ……
```

对于上述代码，可以通过对relop取反来删去一些GOTO语句和Label语句，如下：

```
    IF x ~relop y GOTO Label2
        ……
Label2：
        ……
```