lab3 实验报告

本实验已完成所有的必做和选做内容

一、编译与测试命令

编译可以使用：

> make parser（或直接make）

通过makefile提供的方式可以对本实验项目进行编译，生成程序名为parser，此外我们写了一个简单的脚本用于对`../Test/`中的所有单元测试进行一键测试，使用

> ./test.sh

二、实现过程

1.总体设计

Lab3的任务是将C--源代码翻译为线性结构（三地址代码）形式的中间代码，由于之前的实验实现中已经处理保存了语法树的信息，因此Lab3的实现方式主要就是再次遍历之前语法制导翻译得到的语法树，并且在发现特定结构时进行翻译，将生成中间代码保存在内存中，待全部翻译完毕后打印到指定文件中。

1.1 模块化与模块之间的复用

实现Lab3的具体代码主要在`intercode.h`和`intercode.c`中，这样新建文件编写而非在Lab2实现的`semantic.c`中继续添加函数是遵循了模块化的思想，在系统主函数`main.c`调用semantic模块进行语义分析确认没有语义错误后，会调用`PrintInterCode(char *filename)`函数进入中间代码翻译模块执行。
但是，模块化编写也会导致冗余的问题，由于两个模块中都会使用到分析并记录元素类型信息的代码，为了避免重复的代码实现，同时提高编译器的效率，我们在`intercode.h`中引用了`semantic.h`并修改了`semantic.c`中的部分代码，使得在在翻译中间代码时可以获取语义检测模块记录在`FieldList`和`Type`这两个结构体中的类型信息。

1.2 保存中间代码的数据结构：结构体链表

为了实现所有翻译完成后再全部打印的，需要将所有中间代码保存在内存中，而中间代码的逻辑结构是多条单行代码的线性组合，因此我们使用链表记录中间代码，由于下文提到的顺序性实现，单向链表就已经够了。每条中间代码语句由于是三地址结构，受到三地址代码的四元式启发，我们所采用的结构体用枚举形式记录op的信息，用三个元素的结构体数组分别记录三个操作数的信息，而在每个操作符中只需要记录其类型，并附加一个值记录其名称/数值信息即可，大体如下：

```
typedef struct Operand_* Operand;
struct Operand_ {
    enum { VARIABLE_OP,TMPVAR_OP,CONSTANT_OP,GETADDR_OP,...} kind;
    union {
        char* name;
        int value;
    } u;
};
```

```
struct InterCode_{
    enum { LABEL_IR,FUNC_IR,ASSIGN_IR,ADD_IR,SUB_IR,... } kind;
    Operand op[3];//操作数（指针）
};
```

其中在操作数类型上，首先区分了变量和临时变量，这是由于观察到，由于实验假设中所有变量不重名，所以C--源代码的变量名可以直接在中间代码中使用，不用额外维护符号表，但这样也会存在一个问题：源代码中的变量可能会和形如`t1`的中间代码临时变量重名，为了避免这种情况，使用了一个trick：每次在打印操作数发现其是以`t`开头的变量类型，就多打印一个t，这样就相当于为源代码的变量改名，避免了重名，且不用维护符号表。此外，将取地址和解引用这两种特殊的操作作为操作数的类型，是方便某个需要取地址或解引用的变量（主要是数组和结构体）的多次使用中都能正确取地址或解引用，而不必维护符号表、增加额外的中间代码类型去表示这一信息。

1.3 翻译和打印

代码的翻译主要由一系列`void Translate_X(Tnode *s)`函数构成，在设计函数时，考虑过每个函数返回记录中间代码信息的结构体指针，但是仔细思考后发现在遍历语法树时，完全可以做到通过合适的执行顺序使得代码完全按照正确顺序生成，因此只要在每次需要生成一行中间代码时，使用函数`void NewInterCode(int codekind,Operand op0,Operand op1,Operand op2)`传入一条中间代码所需要的类型和三个操作数，一步到位生成一条中间代码并插入链表末尾即可，这样顺序插入避免了写代码时繁琐的指针操作和运行时在语法树上递归向下时多次操作链表的开销。　　由于上文介绍的设计，打印中间代码的过程就非常简单了，只需要从头遍历存放中间代码的链表，每次拿出一条中间代码，使用switch语句根据类型的不同打印保留字符和三个操作数即可。打印操作数的函数`void PrintOperand(Operand op)`也是同理，根据操作数不同类型打印即可，其中对用户变量和临时变量的区分上文已介绍。

2.翻译中的细节处理

2.1 数组结构体的翻译

结构体和数组的翻译有以下细节需要处理：

- 1.记录类型信息得到地址位置：由于一个结构体或者数组变量名会被用来访问其内部的某个数据，所以需要记录类型信息，记录方式和代码实现其实已经在实验2中写过了，但由于作用域的影响在语义检查时每次退出一个作用域会删除记录的类型信息，所以本次实验只需要再次调用之前写好的函数去在一个哈希表中记录变量名和其对应的类型信息即可，由于没有全局变量使用，只需要在每个函数中对其参数列表VarList和函数体CompSt分别调用一次Lab2中对应函数即可，并且由于变量均不重名，这里也不需要考虑作用域问题，在退出函数时不需要删去类型信息。类型信息会在下一步翻译地址时用到。
- 2.结构体或数组可能作为函数的参数或者函数内部定义的局部变量，这一点需要在代码中进行区别，因为如果是函数内的定义后续使用需要取地址，而如果是参数，那么该结构体或数组名已经是其地址了。这里采用的区分方式是直接在记录类型信息时，对于参数列表VarList和函数体CompSt分别求类型信息，并打上标签作为区分，使得后文调用时明确这一操作数是否需要取地址。
- 3.结构体属性的翻译：对于形如`Exp DOT ID`的表达式，就是结构体类型的基本形式，由于表达式的性质，Exp迭代一定可以得到这个结构体变量的名字（Exp如果有其他运算的情况暂时不考虑），用这个名字查表得到其类型，然后的工作就很简单了，只要dfs查询这个类型表中的属性是否有匹配上属性名的，在dfs过程中加上每个属性的大小，遍历走过的大小就是这个属性的偏移量。另外需要注意的是由于属性名可能是数组等非基本类型的值，如果这个属性不是基本类型那么传出的place变量应该保持地址不变，不需要解引用，否则属性应该是一个INT值，最终结果要解引用。

- 4.数组类型的翻译：对于任意维数组，`Exp1 [Exp2]`的形式表明，第一个Exp是其开头地址，第二个Exp是表示偏移格数的一个数值，可以递归得到这两个Exp存放到中间变量，就可以得到存放位置为`Exp1 + Exp2*size`，其中size是单位偏移大小，但问题在于，由于不知道目前处于高维数组的哪个维度，就不知道偏移格数的大小，例如对于定义数组`int a[9][8][7]`在翻译`a[3][4][5]`递归到最后的`a[3]`时，size应该为8*7，但如何得到这个值呢？关键在于维数，只要得到了目前处于的维数，就可以通过之前记录的信息得到偏移量，而当前维数可以通过递归语法分析树上的`Exp1`看几层能达到`ID`就是维数，得到存放位置赋予place变量后将place这一操作数的类型改为解引用就能使得place成功得到数组的内容。

2.2 Read 和 Write函数处理

这两个特殊的函数调用需要在在Lab2的`semantic.c`中实现的符号表中，预先添加read和write这两个预定义的函数，否则将会导致语义检查出现函数未定义的问题，这里采用的实现方式较为简单，直接在记录信息的哈希表初始化时插入两个手写定义的函数即可。

2.3 中间代码翻译中的优化

由于需要考虑性能，中间代码的条数应该越少越好，因此需要加速优化，优化的重点是中间变量，因为中间变量的性质导致很多时候会出现冗杂的赋值或没有必要的运算，加速思路主要是：

- 1.在解析表达式中，对于一个Exp如果直接为变量名或常数（通过查看语法树上节点的孩子得到这一信息）时，可以跳过设置形如`t1 = Exp`这样的中间变量过渡来加速。
- 2.在算数运算语句中，如果两边都是立即数类型可以直接计算，如果有`Exp + 0`，`Exp * 1`这种语句也可以直接省略。