南京大学本科生实验报告

课程名称： **编译原理**

学号 191220029 **姓名** 傅小龙

1.实验名称

中间代码生成

2.实验内容

本次实验中我实现了所有基础功能（中间代码生成）和所有的拓展功能（要求3.1，要求3.2）

2.1基础功能

在语法分析开始之前，需要将`(void) write(int)`和`(int) read()`函数添加到符号表中，这一操作实现在`/Code/semanteme.c (void) initTable()`中.

在完成语法分析，开始中间代码生成之前，为符号表中所有的变量（即函数、结构体定义除外的符号）生成变量类型的`Operand*`操作符，并统计它们的大小。故符号表表项`HashTable`的数据结构需要添加`Operand*`来存储对应符号的操作符。相关实现详见`/Code/IR.c (void) init_Var()`.

数组大小的计算采用递归方式实现，逐维度地统计数组大小，数组最外层的`Type.u.array.size`的值代表了整个数组的大小。结构体大小的计算则是计算所有成员变量大小后求和。相关实现详见`/Code/IR.c (void) init_Size()`.

本次实验中，中间代码采用线形结构，采用双向链表的方式实现。数据结构和实验手册（P86）中相似，这里不再赘述。

从语法树根节点开始，自顶向下遍历语法树中的每个结点并根据树结构生成相应的中间代码。为每个类型语法结点设计翻译函数。这些函数的实现思想和实验手册4.2.5~4.2.8给出的内容相同，不再具体介绍。内容详见`./Code/IR.h, ./Code/IR.c`

2.2 拓展功能

2.2.1 要求3.1：结构体变量及结构体变量的作参数

结构体变量：

`Exp -> Exp1 DOT ID`：当`Exp1 -> ID1`时，可以直接通过查找符号表获取`Exp1`结点所代表的结构体变量信息。若`Exp1`由其他产生式产生，那么可通过语法分析得到的`Exp1`结点的综合属性获取结构体变量信息。根据结构体定义时的成员顺序遍历结构体所有成员，直到找到`ID`对应的成员，在遍历途中对遇到的成员大小求和，即可得到表达式中成员变量相对结构体的偏移量`offset`. 接下来只需生成中间代码:`t1 = &structVar + #offset, place = *t1`. 当然，在实际运行中，需要根据`strcutVar`变量是普通变量还是地址（即指针）采用不同的取值方式求上述`t1`的值。如果是地址，那么不需要取址符`&`.

`VarDec -> ID`：给翻译`VarDec, DecList, Dec`结点的函数添加参数`bool structFlag`,当该参数值为`true`时表示该节点下有结构体变量的声明。对于结构体变量的声明，需要找到符号表中该结构体变量对应的操作符相匹配的变量`structVar`和大小`structSize`，并生成指令`DEC structVar #structSize`.

结构体变量作参数：

`VarDec -> ID`：结构体变量作为参数的处理和普通变量一致。给翻译`VarDec, ParamDec, VarList`结点的函数添加参数`bool funcFlag`，当该参数值为`true`时表示该节点下有函数参数的声明。在符号表中找到结构体变量的操作符变量，生成指令`PARAM structVar`.

2.2.2 要求3.2 一维数组参数及高维数组的变量

一维数组作为参数：

`VarDec -> VarDec LB INT RB`: 和普通变量作为参数时的处理方式一致：当`funcFlag`为`true`时，在符号表中找到该数组对应的操作符变量并添加`PRAM arrayVar`指令。

高维数组变量：

`VarDec -> VarDec1 LB INT RB`: 当`funcFlag`为`false`时，表示当前结点属于数组的声明部分。如果`VarDec1 -> VarDec2 LB INT RB`，那么该数组是高维数组，找到子树最下层的`ID`结点，即`VarDec -> VarDec1... -> VarDec2 ... ->...-> ID`，根据`ID`结点给出的数组名找到该数组的操作符变量和大小，生成`DEC arrayVar arraySize`指令。

`Exp -> Exp1 LB Exp RB`: 如果`Exp1 -> Exp2 LB Exp RB`，那么该表达式含有高维数组。偏移量计算从高维到低维逐层计算。由于此处的产生式是从上到下遍历的，即从低维到高维，为方便计算，这里用一个栈式结构`ArrayIndex`存储各维度的下标。然后根据符号表中数组各维度大小，依次对栈做`pop`操作计算各维度下标产生的偏移量。

2.3 机器无关优化

优化方面主要实现了简单版的常量折叠和传播赋值。

`Exp->INT|ID`:当任何非左值的`Exp`的子节点是`INT`或`ID`时，可以直接将`INT`对应的常数或`ID`对应的变量作为该`Exp`产生的操作符。

对于四则运算和取负操作(`Exp -> MINUS EXP`), 如果运算符两侧均是常数，那么对于运算结果也可以直接用一常数代替。这样可以省去运算指令。

以上两条简单的优化规则能够减少约30%的中间代码行数。

3. 编译本程序

在提交文件的根目录下，`/Code`文件夹内为实现实验手册中所有基础和选做要求的中间代码生成器。请使用这个文件夹下的`Makefile`编译生成可执行对象。

在`/Code`目录下的终端输入`make`即可生成可执行目标`parser`，用形如`./parser <test file> [dstPath]`的指令对`test file`生成中间代码. `dstPath`是中间生成的中间代码文件的路径，这一参数是可选的，缺省值为`./output/output.ir`. **注意：**如果生成文件的路径采用缺省值，请确保`paser`的同目录下文件夹`/output/`是存在的，否则运行`./parser`时将导致`Segmentation Fault`.

4.致谢

由于我的Ubuntu虚拟机所使用的软件源中没有远古的qt4, 因此无法运行最后一次修改时间为2014年的古董级.pyc文件。故中间代码的测试环节在陈诚（学号：*191220005*）同学的机器上完成。特在此感谢陈诚同学给我提供的帮助。