编译原理实验三 中间代码生成

141270022 刘少聪 141270037 汪值

·实验三任务是在词法分析、语法分析和语义分析基础上将 C—源代码翻译成中间代码。
中间代码的输出形式为线性结构，使用虚拟机小程序来测试中间代码的运行结果。
选做要求：一维数组类型变量可以作为函数参数，可以出现高维数组变量。

线性中间代码的数据结构：
根据实验指导书上面的定义了两个结构体 struct Operand_t 和 struct InterCode_t

```
//单条中间代码的数据结构
typedef struct Operand_t* Operand;
typedef struct Operand_t {...
}Operand_t;

typedef struct InterCode_t* InterCode;
typedef struct InterCode_t {...
}InterCode_t;
```

使用动态数组方式来存放所有的中间代码的指针
InterCode *IRList 为动态数组

```
InterCode *IRList;
int IRlength;
int IRcapacity;
#define IRLIST_INIT_SIZE 10

void initIRList() {
    IRList = (InterCode*)malloc(IRLIST_INIT_SIZE * sizeof(InterCode));
    if (IRList == NULL) {
        printf("IRList Error!\n");
        return;
    }
    IRcapacity = IRLIST_INIT_SIZE;
    IRlength = 0;
}
```

动态数组数据长度 IRlength，容量 IRcapacity，每次扩容以倍增方式 realloc

根据实验指导书的“表 3-1 中间代码的形式及操作规范”，表中有 19 条中间代码形式，因此 InterCode 中定义了 19+1 个枚举变量，多的一个是便于生成调试信息。
三地址代码的形式有多种：一个操作数的 singleOP，两个操作数的有 doubleOP，三个操作数的 tripleOP，比较特殊的有 ifgotoOP 和 decOP
Operand 中区分了变量，地址，临时变量，常量，标号等。如下代码创建临时变量或标号：

```
    Operand widthOp = malloc(sizeof(Operand_t));
    memset(widthOp, 0, sizeof(Operand_t));
    widthOp->kind = CONSTANT_OP;
    int elem_width = getSize(typ1, 2);
    sprintf(widthOp->u.value, "%d", elem_width);
```

如下代码生成一条带有多个操作数的中间代码，并将中间代码插入到 IRList 数组中。

```
InterCode offsetIR = (InterCode)malloc(sizeof(InterCode_t));
memset(offsetIR, 0, sizeof(InterCode_t));
offsetIR->kind = STAR_IR;
offsetIR->u.tripleOP.result = offsetOp;
offsetIR->u.tripleOP.op2 = widthOp;
offsetIR->u.tripleOP.op1 = subscripOp;
insertCode(offsetIR);
```

翻译模式：
实验二中已经写好了每个产生式的规则定义，基本上为每个产生式定义了一个处理函数，基本表达式、语句、条件表达式的翻译模式基本上按照实验指导书上面的进行。
函数调用翻译模式
程序中函数调用的翻译模式没有按照实验指导书，首先在符号表中插入 write 和 read 函数。因为在实验二中的产生式中没有写 Args 的产生式规则， 而是用 while 循环解决的，因此没有写 tanslate_args 函数，而参数传递是栈式的逆序传递，因此循环难以解决，故使用了一个 Operand* argsList = (Operand*)malloc(sizeof(Operand) * 30)的数组来存放参数，数组大小为 30 个，故最多只能存放 30 个参数(bug)。

数组翻译模式
数组的翻译是使用递归模式，一维数组的翻译是先计算数组的下标，然后*4 得到偏移量，并且计算偏移量是否为 0，为 0 时可以简化代码，在生成中间代码时对数组名取地址 &，然后加上偏移量 offset，可以实现一维数组的翻译。
多维数组采用递归的模式，先识别最高维的部分，此时通过 int getSize(TypePtr typ, int t);函数来计算偏移量, getSize()函数中 t=0 时计算 DEC 数组的大小, 而 t=2 时计算偏移量 offset，此时只在最高位对数组名取地址 &，地位计算偏移量时不进行取地址操作。

优化部分：
Exp() 产生式函数调用中进行了部分优化若果 Exp -> INT 时，不会生成临时变量，而是直接生成一个 CONSTANT_OP 的 Operand 操作数。这样可以减少临时变量的生成和中间代码的条数。

开始在 optimize()函数中写了部分优化，如将 条件表达式的 true 或者 false 设置为 fall，此时对 relop 取反，将【IF t1 relop t2 GOTO LABEL label1 GOTO LABEL label2 LABEL label1】转换为【IF t1 ! relop t2 GOTO LABEL label2 LABEL label1】
这样可以删去 GOTO LABEL label2，此时中间代码减少了，但是运行时的条数增加了。所以取消了这个优化选项。

也进行了 GOTO label1 LABEL label1 这样的优化，但在运行过程中出现了标号冲突的 bug，由于试验时间的限制，故也没有继续进行更多的优化。

在测试过程中也发现程序也存在其他 bug, 如多个函数多次直接间接递归调用之后存在结果运行不正确的问题。