编译原理第三次实验报告

1.实验目标

在词法分析，语法分析和语义分析的基础上将 C--程序翻译成中间代码。中间代码在内部的表示形式为三地址代码。

2.实验数据结构

Operand

```
truct Operand_ {
        enum {VARIBLE,VARIBLE_ADDRESS, CONSTANT,ADDRESS,TEMP_VARIBLE,LA
BEL,REF,DEREF,FUNC/*,....*/} kind;
        union {
                int var_no; //变量在符号表中的序号
                int value; //立即数的值
                int temp_var_no;//临时变量的序号
                int label_no; //Labe 的序号
                        char funcname[32]; //函数名
                Operand op;                 //引用和解引用对应的操作数
        };
};
```

注:1.VARIBLE_ADDRESS 是针对一维数组作为函数参数情况进行处理的，此时传入的 v1 是一个地址，与数组作为局部变量处理时 v 为数值相区别。

2.ADDRESS 与 TEMP*VARIBLE 共同使用 temp*var_no 来命名。

InterCode

因实验报告空间有限，故数据结构就不列出了。与教程上的中心思路是相似的

InterCodes

```
struct InterCodes_ {
        InterCode code;
        InterCodes prev;
        InterCodes next;
};
```

采用双向链表结构来存储线性 IR,以实现的代价换取极大的灵活性。

3.实验模块

我组实验思路为单独在一个文件实现生成中间代码功能

主要函数

主要函数如下所示

```
void Free_intercodes(InterCodes head);
void myfree2(); //释放malloc 申请的空间
void Init_InterCode(); //初始化双向链表
bool Insert_InterCode(InterCode i);//将某个节点插入链表
void Print_InterCode(char *filename);//打印双向链表中所有节点
void Print_Operand(Operand p, FILE *P);//打印某个操作数
//bool Delete_InterCode(InterCode i);
Type Translate_Exp(struct node *ast,Operand o,Type t); //翻译Exp 节点
void Translate_Logical(struct node *ast,Operand o,Type t);//翻译Exp 中逻辑运算
//以下均为语法树节点过度或者实验已有模板
void Translate_Cond(struct node *ast,Operand l1,Operand l2,Type t);
void Translate_Cond_Other(struct node *ast,Operand l1,Operand l2,Type t);
void Translate_Program(struct node *ast);
void Translate_ExtDefList(struct node *ast);
void Translate_ExtDef(struct node *ast);
Type Translate_Specifier(struct node *ast);
void Translate_DefList(struct node *ast);
void Translate_Def(struct node *ast);
void Translate_DecList(struct node *ast,Type t);
void Translate_Dec(struct node *ast,Type t);
Operand Translate_VarDec(struct node *ast,Type t);//翻译局部变量
Operand Translate_VarDec2(struct node *ast,Type t);//翻译形参
void Translate_Stmt(struct node *ast);
void Translate_CompSt(struct node *ast);
void Translate_StmtList(struct node *ast);
void Insert_Args(Operand arg_list[],Operand t); //将实参转换为Operand 形式并插入实参列表
//void Init_Args();
void Translate_Args(struct node *ast,Operand arg_list[]); //函数参数的翻译模式
void Translate_ParamDec(struct node *ast);
void Translate_FunDec(struct node *ast);//函数定义的翻译模式
```

所有中间代码节点都是生成即插入。

选做思路

我组选做的是局部变量支持高维数组，函数参数支持一维数组部分。

高维数组(局部变量)

在完成高维数组之前，我们必须要获取数组每一维的 type,所以首先修改 translate_exp 函数的返回类型为 type

当翻译 Exp->ID 时，在符号表中查找 ID,并返回符号的 type

当翻译 Exp*1-> Exp*2 LP Exp_3 RP 时,

调用 translate*Exp(Exp*2)获得当前维度的 type 以及当前的地址

调用 translate*Exp(Exp*3)获得操作数数值

place :=place(translate*Exp(Exp*2)) + place(translate*Exp(Exp*3)) *(Exp2 返回的 type 所对应的地址增量 * 4)

最后根据类型判断进行解引用

程序返回 tanslate*Exp(Exp*2)->array.elem

函数形参与实参部分

遇到 arg_list 本来准备偷个懒，设置成全局变量，随用随取，用完就清空，方便简洁。开始简单的测试用例都是可以过的，直到遇到函数 a 的实参里面调用了函数 b 的情况，此时两个函数都执行了一次对 arg_list 的清空，导致外函数之前的实参丢失。尝试仍在全局变量基础上通过调整清空指令的执行逻辑来解决，宣告失败，于是还是老老实实把 arg_list 设为局部变量并作为实参一步步传下去，遂解决。

4.优化

对于 Exp->ID 以及 Exp -> INT 来说 place := ...这一步完全是多余的，只要传出一个对应 ID/INT 的操作数就行了

对于 Stmt -> Exp SEMI 来说，除非是 exp->exp assignop exp 或者 read write 函数调用或者函数参数存在数组，其他情况下生成的中间代码都可以被优化。