实验三　中间代码生成

131220128 杨帆

一、实验环境

操作系统：　GNU Linux Release: Ubuntu 12.04
软件版本：　GNU Flex version 2.5.35；　GNU Bison version 2.5

二、实验内容

本次实验任务是在词法分析、语法分析和语义分析程序的基础上，将 C--源代 码翻译为中间代码。我们要求将中间代码生成为线性结构，可以使用我们提供的虚拟机小程序来测试中间代码的运行结果。

三、实验过程

整体思路：

在实验二完成了语义分析后，我们将程序中的变量、函数、参数等信息记录到对应的符号表中。本次实验我们需要再次遍历生成的语法树，并将其翻译为中间代码，需要查找符号表中的信息。

参照了书上的翻译方案和数据结构，我们在深度遍历语法树的过程中，生成的中间代码返回给上一层，并通过连接中间代码的函数 codeBind()进行代码连接。

```
// binding of intercodes
InterCodes codeBind(InterCodes code1, InterCodes code2) {
    if (code1 == NULL) return code2;
    if (code2 == NULL) return code1;
    InterCodes p = code1;
    while (p->next != NULL) {
        //printf("%d\n", p->code->kind);
        p = p->next;
    }
    //printf("!!!\n");
    p->next = code2;
    code2->prev = p;
    return code1;
}
```

最终返回给遍历根节点的函数 program()时便是完整的中间代码。

完成的功能点：

必做内容：1）~11）
选做内容：3.1 结构体变量的实现

程序结构与编译过程：

在实验二的基础上新加入intercode.c　intercode.h 两个文件
指令：bison –d syntax.y　　flex lexical.l
gcc main.c semantic.c intercode.c syntax.tab.c –lfl –ly –o parser
或实验中使用的大班提供的makefile文件，直接执行make语句即可。

重要数据结构的表示和主要内容的实现方式：

```
struct Operand_ {
    OType kind;
    union{
        int var_no;
        int tmp_no;
        int lab_no;
        int addr_no;
        int value;
        char *fname;              // function name
        char *pname;              // param name
    } u;
};
```

```
struct InterCode_ {
    IType kind;
    union{
        struct { Operand right, left; } assign;
        struct { Operand result, op1, op2; } binop;
        struct { Operand op; } uniop;
        struct { RType rtype; Operand label, relop1, relop2; } cond;
    } u;
};

struct InterCodes_ {
    InterCode code;
    struct InterCodes_ *prev, *next;
};
```

参照实验讲义，以上分别定义了操作数的数据结构、单条中间代码的数据结构，并用双向链表的连接方式将中间代码进行连接。

当遇到一个新的变量时，通过在 semantic.c 中添加的 lookupstable()和 getVar 函数查找普通变量、数组变量、结构体变量的信息。例如：若是调用函数，调用 new_func()函数定义这个操作数，操作数类型为 FUNCTION:

```
Operand new_func(char *fname) {
    Operand opf = (Operand)malloc(sizeof(struct Operand_));
    opf->kind = FUNCTION;
    opf->u.fname = fname;
    return opf;
}
```

对于变量和临时变量的空间申请采用类似的操作，变量的序号在 SymbolMsg 中新加入一个属性 var_no，新定义一个变量便赋其值为+1，临时变量 t 的序号由一个全局变量 tmp_no 来给定，每出现一个临时变量，便执行 tmp_no++。

在访问到语法树底层时（即要生成中间代码时），例如 IF (a>b) GOTO label1 ELSE GOTO label2， if …goto…属于条件语句， a>b 属于双操作数语句，所以我分别编写了对应的生成中间代码的函数：

```
InterCodes Init_codes(InterCodes a);
InterCodes gen_assign_ir(IType itype, Operand l, Operand r);
InterCodes gen_uniop_ir(IType itype, Operand p);
InterCodes gen_binop_ir(IType itype, Operand result, Operand op1, Operand op2);
InterCodes gen_cond_ir(RType rtype, Operand label, Operand op1, Operand op2);
```

调用 gen_cond_ir()处理条件语句，调用 gen_binop()处理双操作数的语句，最后在调用它们的 stmt()过程中将代码拼接起来。生成完整的 stmt 中间代码后，返回给调用 stmt()的上一层，以此类推。

在处理一维数组和结构体时，需要确定访问对应变量的地址，通过 cal_offset()函数计算偏移量，因为只实现了一维数组，故数组的偏移量只需要第一维的 size*4 即可（因为只有 int 类型变量），结构体偏移量按照讲义上的算法进行计算。

然而如何确定访问的变量是左值还是右值，何时需要打印&、*，何时不需要。我在处理 Exp 节点时，传入一个 left 参数，后将 left 参数传入对数组和结构处理的过程中。若 left = 1 则说明传入的节点为左值，最终需要访问其内存位置并更改，即“*x = a”；若 left = 0，则说明传入参数为右值，最后生成“t = *x, v = t”的形式，x 为计算出的地址。处理函数定义处的参数时，在符号表中加入属性 isParam 判断变量是否是函数的参数，若在参数定义函数中使用该变量，则变量前不需要加“&”。

```
int cal_offset(struct SymbolMsg *s) {
    //printf("in call offset\n");
    if (s->isArray == 0 && s->symType == INT || s->symType == FLOAT) {
        return 4;
    }
    else if (s->symType == STRUCT) {
        //printf("in calloff struct\n");
        int time = 0, off = 0;
        struct FieldMsg *flist = s->fieldList;
        while (flist != NULL) {
            time = time + 1;
            struct SymbolMsg *sis = getVar(flist->name);
            off = off + cal_offset(sis);
            flist = flist->nextField;
        }
        //printf("time: %d\n", time);
        return off;
    }
    else if (s->isArray == 1) {
        int off = 4;
        struct ArrMsg *aMsg = s->arr;
        while (aMsg != NULL) {
            off = off * aMsg->size;
            aMsg = aMsg->nextarrType;
        }
        return off;
    }
```

四、未能实现的部分以及程序的一些缺陷:

1. 未能实现对多维数组的中间代码翻译。实验二中数组实现的结构很差，多维数组也存在 bug，导致本次实验对数组的判断、信息读取造成了困难。
2. 未能实现结构体数组的中间代码翻译。
3. 本次实验代码结构相对于实验二可读性增强，但仍有很多冗余的地方，和实验二函数接口实现得不好有很大关系。

五、实验体会

在实验二的基础上，本次实验的框架的实现可以说并不困难，而且也不像上次实验中有大量的继承属性、综合属性需要传递。但是在生成中间代码的实现方案和实现数组和结构体的翻译是难点。本次实验让我了解了中间代码生成过程的具体过程，也加强了我对指针、链表的实现和处理。本次实验并没有测试很多测试样例，存在着一些尚未发现的 bug，我会在接下来的实验中努力进行完善。实验耗时整整一周，痛并快乐着 QAQ。