编译原理实验三报告

中间代码生成

1190200122 袁野

一、实现功能

程序实现了对 C—源代码进行中间代码生成的操作。

1、实验二基础上的改进

1.1 删除掉 lexical 中正则表达式 RELOP 的定义，并在构建语法树时为他们不同的词法节点的行号赋特定的值方便后边识别。

```
IGNORE [\r \t]
CHANGELINE [\n]
INT 0|[1-9][0-9]*
FLOAT {INT}\.[0-9]+
ID [_a-zA-Z][0-9_a-zA-Z]*
RELOP >|<|>=|<=|==|!=
TYPE int|float
DIGIT [0-9]
```

```
">" {
    yylval = create_Node_1("RELOP\0");
    yylval -> linenumber = 3;
    return RELOP;
}

"<" {
    yylval = create_Node_1("RELOP\0");
    yylval -> linenumber = 2;
    return RELOP;
}

">=" {
    yylval = create_Node_1("RELOP\0");
    yylval -> linenumber = 5;
    return RELOP;
}

"<=" {
    yylval = create_Node_1("RELOP\0");
    yylval -> linenumber = 4;
    return RELOP;
}

"==" {
    yylval = create_Node_1("RELOP\0");
    return RELOP;
}
```

1.2 在 semantic.c 中，添加 temp_num 变量用来为每个非函数的变量名赋序号。

```
void insert(FieldList f,int line,int is_def, int is_from_fun) {
    unsigned int val = hash_pjw(f->name);
    if (table[val] == NULL) {
        table[val] = (TABLE)malloc(sizeof(struct TABLE_));
        table[val] -> field = f;
        table[val] -> is_def_struct = is_def;
        table[val] -> next = NULL;
        table[val] -> linenumber=line;
        table[val] -> is_from_func = is_from_fun;
        if (f -> type -> kind != FUNCT) {
            table[val] -> variable=temp_num;
            temp_num++;
        }
    } else {
        TABLE new_table = (TABLE) malloc(sizeof(struct TABLE_));
        new_table -> field = f;
        new_table -> linenumber = line;
        new_table -> is_def_struct = is_def;
        new_table -> next = NULL;
        new_table -> is_from_func = is_from_fun;
        if (f -> type -> kind != FUNCT) {
            new_table -> variable = temp_num;
            temp_num++;
        }
        TABLE rem = table[val];
        while(rem -> next != NULL) rem = rem -> next;
        rem -> next = new_table;
    }
}
```

2、Operand 及中间代码语句：

```
struct Operand_{
    enum{FROM_ARG,VARIABLE,TEMP,CONSTANT,ADDRESS,WADDRESS,FUNCTION,LABEL,RELOP}kind;
    int u_int;
    char* u_char;
    Type type;
};

struct InterCode_{
    enum{ILABEL,IFUNCTION,ASSIGN,ADD,SUB,MUL,DIV,ADDRASS1,ADDRASS2,ADDRASS3,
        GOTO,IF,RETURN,DEC,ARG,CALL,PARAM,READ,WRITE}kind;
    union{
        //LABEL,FUNCTION,GOTO,RETURN,ARG
        //PARAM,READ,WRITE
        struct{Operand op;}ulabel;
        //ASSIGN,CALL
        //ADDRASS1,ADDRASS2,ADDRASS3
        struct{Operand op1,op2;}uassign;
        //ADD,SUB,MUL,DIV
        struct{Operand result,op1,op2;}ubinop;
        //IF
        struct{Operand x,relop,y,z;}uif;
        //DEC
        struct{Operand op;int size;}udec;
    }u;
    InterCode before;
    InterCode next;
};
```

3、函数实现

对于不同的语法单元以及产生式，都有单独的翻译函数进行中间代码生成。

3.1 FunDec -> ID LP VarList RP| ID LP RP

在哈希表中寻找之前记录过的该函数，并挨个访问其参数，如果是基本变量就记录为 VARIABLE 类型，否则记录为 ADDRESS，并对他们逐个生成 PARAM 语句。

3.2 Stmt ->IF LP Exp RP Stmt | IF LP Exp RP Stmt ELSE Stmt

如果没有 ELSE,那么分别翻译 Exp 和 Stmt 部分，并产生对应的 Label，即可。如果有 ELSE 部分，则再添加一个 Label，控制不满足条件的话跳向这个 Label 即可。

3.3 Stmt -> WHILE LP Exp RP Stmt

和 IF 类似，对 Exp 和 Stmt 部分进行翻译，并分别设置好自己所对应的 Label，产生语句控制满足条件跳向的 Label 和不满足时跳向得到 Label。

3.4 Exp -> Exp DOT ID

首先翻译 Exp 取出结构体变量的首地址，然后通过其 temp1 在表中的对应 type 类型以及 ID 中的域名信息，生成计算出该域名在结构体中的偏移量的代码。记录 place->type=ID.type，通过在符号表中查找 ID 取出。

3.5 Exp -> Exp LB Exp RB

首先翻译第一个 Exp，生成取出该数组对应的首地址以及单个位置的大小的代码，然后翻译第二个 Exp，生成计算出偏移量，相结合计算出地址的代码即可

3.6 Exp -> ID LP Args RP

如果当前 ID 时 write 函数，直接对其进行翻译，并生成将临时变量输出的语句即可。

否则生成类型为 CALL 的语句，用来调用该函数。

3.7 Exp -> ID LP RP

如果是 read 函数，则生成临时变量 t 并生成 read(t)。

否则生成类型为 CALL 的语句，用来调用该函数。

3.8 translate_Args: Args -> Exp COMMA Args| Exp

正向翻译 Exp 时，利用双向链表记录中间代码。

所有 Exp 语句翻译完成，从链表尾部向前将中间代码加入总的 Intercode 中。

3.9 数组与结构体定义

VarDec -> ID | VarDec1 LB INT RB (VarDec1->ID)

StructSpecifier -> STRUCT Tag (Tag -> ID)

在哈希表中找到 ID 对应的类型信息，并计算其大小/单个位置大小等信息，生成对应语句即可。

二、编译过程

通过借助网上的参考资料，构建了一个 makefile 文件来编译这个程序，命令如下：

make 编译该项目并生成 parser 可执行文件

make test 测试实验指导书中的前 3 个样例

make clean 删除所有编译过程中产生的新文件