# 编译原理

# 实验三

## 1 程序功能

程序在实验一和二实现的词法分析、语法分析和语义分析的基础上，实现了对 C--源代码的中间代码生成。

本实验在给出的 7 个假设的基础上，还完成了要求 3.1 的内容，允许结构体类型的变量出现，并允许结构体类型的变量作为函数的参数。

为了方便后续实现对中间代码的优化，需要把生成的中间代码先保存到内存中，等全部翻译完，再做优化，最后再打印出来。于是，需要设计中间代码的表示形式，本实验采用了指导书上给出的线形数据结构，并使用双向链表来实现，数据结构如下：

```
typedef struct interCodes {
    pInterCode code;
    pInterCodes *prev, *next;
} InterCodes;
```

InterCode 中记录各条中间代码的类型与操作数信息，数据结构如下：

```
typedef struct interCode {
    enum {
        IR_LABEL,
        IR_FUNCTION,
        IR_ASSIGN,
        IR_ADD,
        IR_SUB,
        IR_MUL,
        IR_DIV,
        IR_GET_ADDR,
        IR_READ_ADDR,
        IR_WRITE_ADDR,
        IR_GOTO,
        IR_IF_GOTO,
        IR_RETURN,
        IR_DEC,
        IR_ARG,
        IR_CALL,
        IR_PARAM,
        IR_READ,
        IR_WRITE,
    } kind;

    union {
        struct {
            pOperand op;
        } oneOp;
        struct {
            pOperand right, left;
        } assign;
        struct {
            pOperand result, op1, op2;
        } binOp;
        struct {
            pOperand x, relop, y, z;
        } ifGoto;
        struct {
            pOperand op;
            int size;
        } dec;
    } u;
} InterCode;
```

Operand 参考了指导书上给出的示例，记录每一个操作数的类型与属性值（名或值），具体如下：

```
typedef struct operand {
    enum {
        OP_VARIABLE,
        OP_CONSTANT,
        OP_ADDRESS,
        OP_LABEL,
        OP_FUNCTION,
        OP_RELOP,
    } kind;

    union {
        int value;
        char* name;
    } u;
} Operand;
```

在进行中间代码生成时，需要借助实验二中得到的符号表，遍历实验一中构造出的语法分析树，每当遇到 ExtDefList 节点便意味着有可能出现变量、结构体或函数的定义，需要从该节点开始进行分析，对于符合要求的定义，要生成对应的中间代码并插入到双向链表中；对于不符合要求的定义（如高维数组变量、函数参数为数组等），需要进行错误的报告。在对 ExtDefList 节点进行中间代码生成时，会根据其子节点情况调用各种非终结符节点对应的中间代码生成流程，充分利用各个节点的综合属性与继承属性获取正在生成的中间代码所对应的类型和各个操作数的信息。这一过程的流程与实验二极为相似，此处不再详细叙述。

为了实现 3.1 的要求，需要考虑 VarDec 的子结点 ID 的类型为 STRUCTURE 的情况，对结构体变量定义的中间代码生成实现如下：

```
else if (type->kind == STRUCTURE) {
    // 3.1
    genInterCode(IR_DEC,
                 newOperand(OP_VARIABLE, newString(temp->field->name)),
                 getSize(type));
}
```

需要注意的是，在实现结构体类型的函数参数时，结构体参数应该传地址而非简单地传值。为了达到这一目的，首先需要修改实验二中的 FieldList 结构体，对一个变量是否为参数进行标记。修改结果如下：

```
typedef struct fieldList {
    char* name;
    pType type;
    boolean isArg;
    pFieldList tail;
} FieldList;
```

然后在处理函数的参数时，应该将结构体单独考虑，代码如下：

```
while (argTemp) {
    if (argTemp->op->kind == OP_VARIABLE) {
        pItem item = searchTableItem(table, argTemp->op->u.name);

        // 结构体作为参数需要传址
        if (item && item->field->type->kind == STRUCTURE) {
            pOperand varTemp = newTemp();
            genInterCode(IR_GET_ADDR, varTemp, argTemp->op);
            genInterCode(IR_ARG, varTemp);
        }

        // 一般参数直接传值
        else
            genInterCode(IR_ARG, argTemp->op);
    }

    argTemp = argTemp->next;
}
```

在进行赋值操作时，需要注意作为参数的结构体，已经传入了基地址，不可再次取址；而对于非参数的结构体，需要取基地址后计算每个域的真实地址，再赋值，如下：

```
pOperand target = newTemp();
if (item->field->isArg && item->field->type->kind == STRUCTURE)
    target = temp;
else
    genInterCode(IR_GET_ADDR, target, temp);
```

在对中间代码进行优化方面，实验代码并未采取太多措施。对于直接使用的变量名和立即数，不生成临时变量进行赋值，这样便简单地实现了一点优化，稍微减少了一些冗余代码。一个示例如下：

```
if (!strcmp(node->child->name, "ID")) {
    pItem temp = searchTableItem(table, node->child->val);
    pType type = temp->field->type;
    if (type->kind == BASIC) {
        if (place) {
            interCodeList->tempVarNum--;
            setOperand(place, OP_VARIABLE, (void*)newString(temp->field->name));
        }
    }
```

此处便是把原本的临时变量名改为了会直接使用到的变量名，而非进行无意义的赋值操作，减少了冗余代码量。

生成完中间代码，既可以通过阅读的方式检查结果的正确性，当然也可以借助指导书上给出的虚拟机小程序更方便、更快捷、更准确地测试结果。

## 2 程序运行

在源程序所在文件夹下进入终端，输入以下指令，即可得到编译结果 parser3:

```
zr@ubuntu:~/compilerlab/lab3$ gcc main.c syntax.tab.c semantic.c inter.c -lfl -o
 parser3
```

将测试用例也置于该文件夹下，测试时，在终端输入“./parser3 ”后加上输入文件名，再加上输出文件名即可得到结果。例如，若要对 test1.cmm 文件生成中间代码，并把结果写入 test1.ir，输入指令如下：

```
zr@ubuntu:~/compilerlab/lab3$ ./parser3 test1.cmm test1.ir
```