编译原理实验——中间代码生成器

缪源清 161220093 myqlily@126.com

一、实验环境

ubuntu16.04

二、实现功能

将语法树翻译成中间代码。可翻译结构体、数组。通过必做样例与选做样例的测试。

三、设计思路

实验二中用十字链表的方式实现了作用域的嵌套，在语义分析跳出当前作用域时会销毁此作用域的符号表。因此不便在语法分析结束后再遍历语法树生成中间代码，而是应该在语法分析的过程中生成中间代码。

实验三采用的是双向链表的数据结构。

代码:

```
typedef struct Operand_* Operand;
typedef struct InterCode_* InterCode;
typedef struct InterCodes_* InterCodes;
struct Operand_{
        enum { VARIABLE, INT_CONSTANT, FLOAT_CONSTANT, ADDRESS, TVARIABLE, LABEL, NAME }kind;
        union{
                int var_no;
                int temp_no;
                int lab_no;
                int int_value;
                float float_value;
                char* name;
        }u;
};
typedef struct Operand_ Operand_;

struct InterCode_{
        enum {ASSIGN, ADD, SUB, MUL,DIVI, LABEL_LINE, GOTOL, COND_GOTO, RETURNL, READL, CALLL, ARGSL, WRITEL, FUNCTIONL, PARAML, DEC}kind;
        union{
                struct {Operand right, left;}assign;
                struct {Operand result, op1, op2;}binop;
                struct {Operand t1,t2,lb;char* op;}condop;
                struct {Operand v;int len;}dec;
                Operand label;
        }u;
};

typedef struct InterCode_ InterCode_;
struct InterCodes_{
        InterCode code;
        InterCodes prev, next;
};
typedef struct InterCodes_ InterCodes_;
```

每条中间代码(InterCode_)都有 Operand 指针。设计中的一个原则是令中间代码指向的 Operand 各不相同(可通过复制实现)，这样在释放空间的时候不会发生错误。

在语义分析实验中设计的函数不能满足中间代码生成过程中参数传递的需求。为了尽量少地修改原有代码，采取了全局变量传参数的方法。

```
Operand place;
Operand label_true;
Operand label_false;
int irflag;
int called;
```

其中，变量 place 用于传递操作数；label_true, label_false 传递 label；irflag 和 called 用于控制代码的生成:

irflag=1 时，place 传递的是调用者分配为被调用者生成的操作数，irflag=0 时，传递的是被调用者返还给调用者的变量。举例来说，生成 id1 := #1 的中间代码时，id1 是由 EXP->ID 识别出的，返回的是操作数 id1，因此在上层调用时，应让 irflag 的值为 0；生成 id2 := id3 + id4 时，id3+id4 是由 EXP1->EXP2 PLUS EXP3 识别出来的，E 由上层为 EXP1 生成变量，传递给下层，用于存放 EXP2 PLUS EXP3 的结果。

在实验手册的基本表达式翻译模式表中，EXP1 RELOP EXP2 在 translate_Exp 和 translate_Cond 中都出现了。called 的设计就是为了区分当前的节点执行 translate_Exp 还是 translate_Cond。called=1 时，按后者模式执行；called=0 时，按前者模式执行。

在本实验的实现中，也对实验二的数据结构进行了修改。在符号表的符号项中增加了一个域（no），用来指示它在中间代码中的标号。

实验的代码基本参照基本表达式的翻译模式完成。下面说明对结构体和数组的翻译：

语义分析时，为判断结构体是否含某域而对结构体的域进行遍历。故可以在遍历的过程中对当前偏移进行记数，最终得到了该域的偏移量。

数组翻译稍微复杂一些。数组的生成式是 EXP1->EXP2 LB EXP3 RB，在语义分析中递归调用了对 EXP 的分析。递归的尽头是将第一个 EXP2 识别成 ID。 这时将 ID 和 EXP3 翻译出的变量*4 作为加类型的中间代码的操作数；新生成一个操作数，作为加操作的结果，生成代码。复制新生成的操作数，将复制出的操作数赋给 place 变量。这样就把 place 传递给了上一层的递归，在上一层生成新的中间代码。层层返回，翻译出了数组的地址。

数组和结构体都是以地址的形式作为实参的，但是它们在函数参数列表中的形式与它们相应的定义完全相同。语义分析时为了处理函数的参数不能与函数中定义的变量重名的情况，采用的方法是将函数列表的参数视为函数中定义的变量加入符号表中。但是在中间代码的生成过程中应该有所区分，为此采用的方法是将实参的”no”域的值设成负数。

四、对实验二的修正

语义分析过程中判断两个类型是否相同时有一个逻辑分支是:若它们都是数组，比较它们的基本类型、维数、每一维的大小。然而在 C 语言中并没有要求两个数组每一维大小相同的需求。故将对维数和每一维大小的比较判断删去。

五、实验的不足

没有对代码进行优化。

六、优点

由于中间代码生成过程中用了链表的数据结构并且有不少释放与分配的操作，因此需要注意内存的管理。优点在于考虑到了语义分析可能会出错。出错时，需要将已生成的中间代码的空间释放掉。因此程序具有更高的鲁棒性。

七、测试方法

在/Code 目录下 make

make t1:必做样例 1，生成的 ir 文件在/Test 目录下

make t2:必做样例 2，其余同上

make t3:必做样例 3，其余同上

make t4:必做样例 4，其余同上

将生成的 ir 文件导入虚拟机 qt 小程序中，执行即可。