编译原理实验三

2021110892 赵宇川

一、程序实现了哪些功能

实验三是在词法分析、语法分析和语义分析的基础上，将 C--代码翻译为中间代码。依据实验的假设，和中间代码形式的要求，完成了基本的前两个测试用例的翻译。没有完成对结构体的翻译和对高维数组的翻译。

二、如何编译和运行源代码

在源代码根目录下（有 Makefile 文件的目录），运行以下指令进行编译：

```
make build
make rerun
```

之后可以对用例进行测试，在这个实验中由于只实现了前两个，所以可以执行：

```
make my_test num=1
make my_test num=2
```

之后，在同级的 out 目录下会生成 out1.ir 和 out2.ir 两个文件，这是中间代码的文件。

三、如何实现这些功能

Operand 和 interCode 的结构体定义参考了讲义中给出的建议，不再细说，然后采用了链表式 IR：

```
// ir 代码定义为

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

typedef struct _interCodeList {
    pInterCodes head;
    pInterCodes cur;
    int tempVarNum;          // 临时变量，t0-t.. t[0..tempVarNum]
    int labelNum;            // 标签数量 同临时变量命名相同形式
} InterCodeList;

```

总的来说，在语义分析结束之后，我们会得到程序的符号表，若确定语义分析没有错误，那么我们再沿着语法树从根节点开始先序遍历，同样依据语法规则对不同的语句进行判断，再执行不同的动作。

另外对中间代码的生成没有做太多优化，主要对直接使用的符号和立即数进行了一步去掉一步创建新临时变量的优化。
具体来说，就是在分配一个临时变量时，若我们发现是为一个立即数分配一个 t_num 的临时变量，之后 t_num 又被赋值给某一个变量，那么这就多了一步不必要的赋值。所以在这个情况下，我们去除这个临时变量，直接进行赋值。
具体代码如下：

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

这里的 Operand 是操作数，当 kind 是 OP_CONSTANT，即立即数的时候，我们将原先分配的 u.value（即 t_num）改写为这个立即数的值（即 val）。

```
void setOperand(pOperand p, int kind, void* val) {
    assert(p != NULL);
    assert(kind >= 0 && kind < 6);
    p->kind = kind;
    switch (kind) {
        case OP_CONSTANT:
            p->u.value = (int)val;
            break;
        case OP_VARIABLE:
        case OP_ADDRESS:
        case OP_LABEL:
        case OP_FUNCTION:
        case OP_RELOP:
            if (p->u.name) free(p->u.name);
            p->u.name = (char*)val;
            break;
    }
}
```

其余方面，有关结构体和高维数组并为实现，所以示例 3、4 的 out 文件是不会有结果的。