编译原理实验三

实验目标

在前两次实验检查后，将没有词法、语法和语义错误源代码翻译为中间代码

实验步骤

- 对语法树进行遍历
- 在遍历的过程中对不同的语法单元进行翻译
- 将翻译的中间代码进行组合拼接
- 输出最终的中间代码

遍历语法树

这部分代码框架和语义分析中的类似，使用深度优先搜索进行遍历

翻译语法单元

首先实现储存中间代码的数据结构，这里使用双向链表，并且实现了表示中间代码的结构体。

```
struct Label {
    enum LabelKind kind;
    union { char *name; int value; };
};

struct LabelList_ {
    Label label;
    LabelList next;
};

struct Code {
    enum CodeKind kind;
    union {
        struct { Label x, y; } assign;
        struct { enum DeOp op; Label x, y; } dereference;
        struct { Label x, y; } reference;
        struct { enum BinOp op; Label x, y, z; } binop;
        struct { enum LabelOp op; Label x; } labelop;
        struct { Label x, f; } call;
        struct { int relop; Label x, y, z; } condjmp;
        struct { Label x; int size; } dec;
    };
};

struct InterCode_ {
    Code code;
    InterCode prev, next;
```

```
};
```

翻译语法单元

```
static InterCode Cond(Node node, Label label_true, Label label_false) {
    Node first = getChild(node, 0);
    Node second = getChild(node, 1);
    if (second != NULL && strcmp(second->nodeName, "AND") == 0) {
        Label l1 = new_label();
        InterCode code1 = Cond(getChild(node, 0), l1, label_false);
        InterCode code2 = new_labelop(LABELOP_DEFLABEL, l1);
        InterCode code3 = Cond(getChild(node, 2), label_true, label_false);
        contact(&code2, &code3);
        contact(&code1, &code2);
        return code1;
    } else if (second != NULL && strcmp(second->nodeName, "OR") == 0) {
        Label l1 = new_label();
        InterCode code1 = Cond(getChild(node, 0), label_true, l1);
        InterCode code2 = new_labelop(LABELOP_DEFLABEL, l1);
        InterCode code3 = Cond(getChild(node, 2), label_true, label_false);
        contact(&code2, &code3);
        contact(&code1, &code2);
        return code1;
    } else if (second != NULL && strcmp(second->nodeName, "RELOP") == 0) {
        Label t1 = new_temp();
        Label t2 = new_temp();
        InterCode code1 = Exp(getChild(node, 0), t1, RIGHT);
        InterCode code2 = Exp(getChild(node, 2), t2, RIGHT);
        InterCode code3 = new_condjmp(second->valInt, t1, t2, label_true);
        InterCode code4 = new_labelop(LABELOP_GOTO, label_false);
        contact(&code3, &code4);
        contact(&code2, &code3);
        contact(&code1, &code2);
        return code1;
    } else if (strcmp(first->nodeName, "NOT") == 0) {
        return Cond(getChild(node, 1), label_false, label_true);
    } else {
        Label t1 = new_temp();
        InterCode code1 = Exp(node, t1, RIGHT);
        InterCode code2 = new_condjmp(RELOP_NEQ, t1, (Label){ LABEL_CONSTANT,
.value = 0 }, label_true);
        InterCode code3 = new_labelop(LABELOP_GOTO, label_false);
        contact(&code2, &code3);
        contact(&code1, &code2);
        return code1;
    }
}
```

特别的，对于数组和结构体，使用符号表进行记录，便于查询数组大小和结构体内部结构。

为了处理结构体作为函数参数，设置了两种不同的参数类型：变量和地址。

结构体传参传递地址，在函数中通过地址和偏移量计算结构体成员的地址，通过指针解引用操作获取成员变量的值。

在翻译表达式时，针对左值和右值表达式需要不同的翻译操作。左值表达式因为可能被赋值，所以需要返回地址，在需要取值时再解引用。而右值表达式可以直接返回变量的值，省略取地址操作。

遇到逻辑表达式，通过特殊处理实现短路求值，可以加快计算。

总结

本次实验需要考虑的细节很多，为了增强代码的可读性和模块化，我将翻译部分实现成了新的模块。通过定义一些常用的函数减少了重复代码，降低了出错率。

作为复杂的部分是实现结构体，需要分析当前应该生成的是地址还是变量，对于嵌套的域需要递归的计算偏移量。

为了实现简单，此次翻译出来的中间代码包含了很多冗余操作，有很大的优化空间。比如一些中间变量的赋值操作可以省略，条件跳转语句和label可以重新编排，计算的中间变量可以复用减少重复计算等。但为了优先保证正确性，本次实验并没有进行优化，可以在将来的实验中改进。