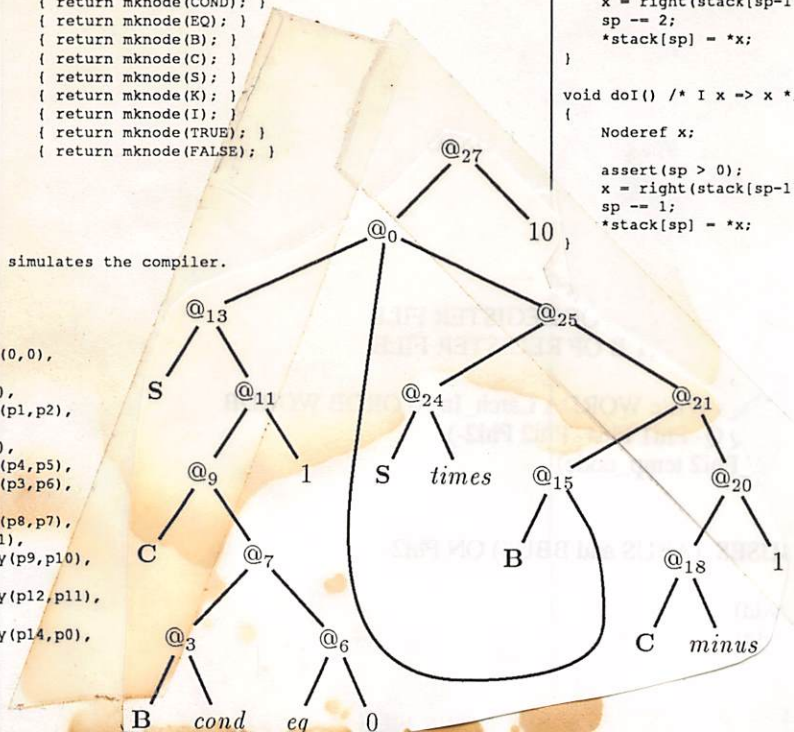
```
#include <stdio.h>
#include <malloc.h>
#include <assert.h>

    /*
     *
     * Graphreduction                5/5/90 --kjepo
     *
     * [Turner '79, A new implementation technique for Applicative Languages]
     *
     */

typedef enum {
    APPLY, NUM, B, C, S, K, I, Y, PLUS, MINUS, TIMES, COND, EQ, TRUE, FALSE
} Nodetype;

typedef struct Node {
    Nodetype kind;
    union {
        struct {                /* kind=APPLY */
            struct Node *left;
            struct Node *right;
        } apply;
        int val;                /* kind=NUM */
    } u_node;
} *Noderef;

#define kind(p)         ((p)->kind)
#define left(p)         ((p)->u_node.apply.left)
#define right(p)        ((p)->u_node.apply.right)
#define num(p)          ((p)->u_node.val)

Noderef stack[100];
int sp;

void reduce(Noderef graph, int stack_bot);

/*------------------------------------------------------------------*/

Noderef mknode(Nodetype tag)
{
    Noderef p = (Noderef) malloc(sizeof(struct Node));
    assert(p);
    kind(p) = tag;
    return p;
}

Noderef mkapply(Noderef l, Noderef r)
{
    Noderef p = mknode(APPLY);
    left(p) = l;
    right(p) = r;
    return p;
}

Noderef mknum(int i)
{
    Noderef p = mknode(NUM);
    num(p) = i;
    return p;
}

Noderef mkPLUS()        { return mknode(PLUS); }
Noderef mkMINUS()       { return mknode(MINUS); }
Noderef mkTIMES()       { return mknode(TIMES); }
Noderef mkCOND()        { return mknode(COND); }
Noderef mkEQ()          { return mknode(EQ); }
Noderef mkB()           { return mknode(B); }
Noderef mkC()           { return mknode(C); }
Noderef mkS()           { return mknode(S); }
Noderef mkK()           { return mknode(K); }
Noderef mkI()           { return mknode(I); }
Noderef mkTRUE()        { return mknode(TRUE); }
Noderef mkFALSE()       { return mknode(FALSE); }

Noderef init()
{
    /*
     *
     * This function simulates the compiler.
     *
     */

    Noderef
        p0 = mkapply(0,0),
        p1 = mkB(),
        p2 = mkCOND(),
        p3 = mkapply(p1,p2),
        p4 = mkEQ(),
        p5 = mknum(0),
        p6 = mkapply(p4,p5),
        p7 = mkapply(p3,p6),
        p8 = mkC(),
        p9 = mkapply(p8,p7),
        p10 = mknum(1),
        p11 = mkapply(p9,p10),
        p12 = mkS(),
        p13 = mkapply(p12,p11),
        p14 = mkB(),
        p15 = mkapply(p14,p0),
        p16 = mkC(),
        p17 = mkMINUS(),
        p18 = mkapply(p16,p17),
        p19 = mknum(1),
        p20 = mkapply(p18,p19),
        p21 = mkapply(p15,p20),
        p22 = mkS(),
        p23 = mkTIMES(),
        p24 = mkapply(p22,p23),
        p25 = mkapply(p24,p21),
        p26 = mknum(10),
        p27 = mkapply(p0,p26);

    left(p0) = p13;
    right(p0) = p25;

    return p27;
}

void doB() /* B f g x => f (g x) */
{
    Noderef f, g, x;

    assert(sp > 2);
    f = right(stack[sp-1]);
    g = right(stack[sp-2]);
    x = right(stack[sp-3]);
    sp -= 3;
    /* kind(stack[sp]) = APPLY; */
    left(stack[sp]) = f;
    right(stack[sp]) = mkapply(g, x);
}

void doC() /* C f g x => f x g */
{
    Noderef f, g, x;

    assert(sp > 2);
    f = right(stack[sp-1]);
    g = right(stack[sp-2]);
    x = right(stack[sp-3]);
    sp -= 3;
    /* kind(stack[sp]) = APPLY; */
    left(stack[sp]) = mkapply(f, x);
    right(stack[sp]) = g;
}

void doS() /* S x y z => x z (y z) */
{
    Noderef x, y, z;

    assert(sp > 2);
    x = right(stack[sp-1]);
    y = right(stack[sp-2]);
    z = right(stack[sp-3]);
    sp -= 3;
    /* kind(stack[sp]) = APPLY; */
    left(stack[sp]) = mkapply(x, z);
    right(stack[sp]) = mkapply(y, z);
}

void doK() /* K x y => x */
{
    Noderef x;

    assert(sp > 1);
    x = right(stack[sp-1]);
    sp -= 2;
    *stack[sp] = *x;
}

void doI() /* I x => x */
{
    Noderef x;

    assert(sp > 0);
    x = right(stack[sp-1]);
    sp -= 1;
    *stack[sp] = *x;
}
```




```
void doY() /* Y h = h(Y h) = h(h(h(....))) */
{
    Noderef h;

    assert(sp > 0);
    h = right(stack[sp-1]);
    sp --1 ;
    /* kind(stack[sp]) = APPLY; */
    left(stack[sp]) = h;
    right(stack[sp]) = stack[sp]; /* tie the knot */
}

void doPLUS() /* PLUS x y => x+y */
{
    Noderef x, y;
    int xval, yval;

    assert(sp > 1);
    x = right(stack[sp-1]);
    y = right(stack[sp-2]);
    if (kind(x) != NUM) {
```

```
        reduce(x, sp);          /* recursively evaluate x */
        x = stack[sp];
    }
    xval = num(x);
    if (kind(y) != NUM) {
        reduce(y, sp);
        y = stack[sp];
    }
    yval = num(y);
    sp -= 2;
    kind(stack[sp]) = NUM;
    num(stack[sp]) = xval + yval;
}

void doMINUS() /* MINUS x y => x-y */
{
    Noderef x, y;
    int xval, yval;

    assert(sp > 1);
    x = right(stack[sp-1]);
    y = right(stack[sp-2]);
    if (kind(x) != NUM) {
        reduce(x, sp);
        x = stack[sp];
    }
    xval = num(x);
    if (kind(y) != NUM) {
        reduce(y, sp);
        y = stack[sp];
    }
    yval = num(y);
    sp -= 2;
    kind(stack[sp]) = NUM;
    num(stack[sp]) = xval - yval;
}

void doTIMES() /* TIMES x y => x*y */
{
    Noderef x, y;
    int xval, yval;

    assert(sp > 1);
    x = right(stack[sp-1]);
    y = right(stack[sp-2]);
    if (kind(x) != NUM) {
        reduce(x, sp);
        x = stack[sp];
    }
    xval = num(x);
    if (kind(y) != NUM) {
        reduce(y, sp);
        y = stack[sp];
    }
    yval = num(y);
    sp -= 2;
    kind(stack[sp]) = NUM;
    num(stack[sp]) = xval * yval;
}

void doCOND() /* COND TRUE x y => x, COND FALSE x y => y */
{
    Noderef pred, tnod, fnod;

    assert(sp > 2);
    pred = right(stack[sp-1]);
    tnod = right(stack[sp-2]);
    fnod = right(stack[sp-3]);
    if (kind(pred) != TRUE && kind(pred) != FALSE) {
        reduce(pred, sp);
        pred = stack[sp];
    }
    sp -= 3;
    switch (kind(pred)) {
      case TRUE:
        *stack[sp] = *tnod;
        break;
      case FALSE:
        *stack[sp] = *fnod;
        break;
      default:
        fprintf(stderr, "predicate wasn't boolean.\n");
        abort();
    }
}

void doEQ() /* EQ x y => TRUE if x=y, /* EQ x y => FALSE otherwise */
{
    Noderef x, y;
    int xval, yval;

    assert(sp > 2);
    x = right(stack[sp-1]);
    y = right(stack[sp-2]);
    if (kind(x) != NUM) {
        reduce(x, sp);
        x = stack[sp];
    }
    xval = num(x);
    if (kind(y) != NUM) {
        reduce(y, sp);
        y = stack[sp];
    }
    yval = num(y);
    sp -= 2;
    if (xval == yval)
        kind(stack[sp]) = TRUE;
    else
        kind(stack[sp]) = FALSE;
}

void push(Noderef n)
{
    assert(sp < sizeof(stack));
    stack[++sp] = n;
}

void reduction()
{
    while (kind(stack[sp]) == APPLY)
        push(left(stack[sp]));

    switch (kind(stack[sp])) {
      case B:
        doB();
        break;
      case C:
        doC();
        break;
      case S:
        doS();
        break;
      case K:
        doK();
        break;
      case I:
        doI();
        break;
      case Y:
        doY();
        break;
      case PLUS:
        doPLUS();
        break;
      case MINUS:
        doMINUS();
        break;
      case TIMES:
        doTIMES();
        break;
      case COND:
        doCOND();
        break;
      case EQ:
        doEQ();
        break;
      case NUM:
        fprintf(stderr, "number applied to something.\n");
        abort();
      case TRUE:
      case FALSE:
        fprintf(stderr, "boolean applied to something.\n");
        abort();
      default:
        fprintf(stderr, "tag field corrupt in node.\n");
        abort();
    }
}

void reduce(Noderef graph, int stack_bot)
{
    int save_sp = sp;

    sp = stack_bot;
    stack[stack_bot] = graph;
    while (kind(stack[stack_bot]) == APPLY)
        reduction();
    sp = save_sp;
}

void main()
{
    Noderef graph;

    graph = init();
    reduce(graph, 0);
    switch (kind(graph)) {
      case NUM:
        printf("%d\n", num(graph));
        break;
      case TRUE:
        printf("true\n");
        break;
      case FALSE:
        printf("false\n");
        break;
      default:
        fprintf(stderr, "result can not be printed.\n");
        abort();
    }
}
```