1. Introduction. I'm writing this program to experiment with recursive algorithms on trees that I think are educational and fun. It's an interactive system that reads online commands in a primitive language and displays the results.

The algorithms are based on a recursive way to represent nonnegative integers by binary trees: The empty tree is $r(0)$, the representation of zero. And the tree that represents the number $2^a + b$, where $0 \le b < 2^a$, is the binary tree whose left and right subtrees are $r(a)$ and $r(b)$.

Stating this another way, let $\star$ be the binary operation on nonnegative integers defined by the rule $a \star b = 2^a + b$. This operator is not associative, so we need to insert parentheses to indicate the meaning; however, right associativity is implied whenever there's any doubt, so that $a \star b \star c$ means $a \star (b \star c)$. Notice that $a \star b \star c = 2^a + a^b + c$, so the partial commutative law $a \star b \star c = b \star a \star c$ is valid. We can use $\star$ to assign a number $v(T)$ to each binary tree T, by saying that $v(\Lambda) = 0$ and $v(T) = v(T_l) \star v(T_r)$ when T is nonempty.

A binary tree is a *normal form* if it is the representation of some integer as described above. It isn't hard to prove that this condition holds if and only if each node x that has a right child x_r satisfies the condition $v(xl) > v(x_{rl})$.

The main algorithms in this program compute the sum and product of binary trees, in the sense that $v(T + T') = v(T) + v(T')$ and $v(T \times T') = v(T)v(T')$. If the tree operands are normal, the results are too. Otherwise the sum and product operations are a bit peculiar, but they are still well defined (they don't blow up), and they might even turn out to define interesting groupoids.

Lots of interesting research problems arise immediately in this study, and I haven't time to answer them now. So I'll just state a few of the more obvious ones. For example: How many n-node binary trees are in normal form? Call this number b_{n+1}. It can be shown that the generating function $B(z)$ is defined by the formula $B(z) = z \exp\bigl(B(z) - \frac{1}{2}B(z^2) \frac{1}{3}B(z^3) - \cdots\bigr)$. I'm virtually certain that a little analysis will establish the formula $b_n \sim c\alpha^n / n^{3/2}$, where $c \approx 0.36$ and $\alpha \approx 2.52$, using methods that Pólya applied to the similar equation $A(z) = z \exp\bigl(A(z) + \frac{1}{2}A(z^2) + \frac{1}{3}A(z^3) + \cdots\bigr)$; see *Fundamental Algorithms*, exercise 2.3.4.4–4. The latter equation, incidentally, enumerates binary trees that are in normal form under the weaker condition $v(x_l) \ge v(x_{rl})$. The operator corresponding to this weaker condition is $a \star b = \omega^a + b$; it gives all the "small" *ordinal* numbers. The free groupoid on one letter satisfying the axiom $a \star b \star c = b \star a \star c$ is isomorphic to the binary trees that have this weaker normal form.

Another tantalizing problem: Estimate the size of the binary tree that represents n, when n is large. This is the solution to the recurrence

$$f(0) = 0\,; \qquad f(2^a + b) = 1 + f(a) + f(b)\,, \quad \text{when } 0 \le b < 2^a.$$

Let $L(1) = 1$ and $L(n) = \lfloor \lg n \rfloor L\bigl(\lfloor \lg n \rfloor\bigr)$ for $n > 1$. (Thus, $L(n) = \lg n (\lg \lg n)(\lg \lg \lg n) \ldots$, rounding each factor down to an integer and continuing until that integer reaches 1). Then it can be shown that $f(n) = \lfloor cL(n)/2^{\lg^* n} \rfloor - 2$ when n has the special form $2 \uparrow\uparrow m - 1$ (namely a stack of 2s minus one): 1, 3, 7, 65535, $2^{65536} - 1$, etc. Here $\lg^* 1 = 0$ and $\lg^* n = 1 + \lg^* \lfloor \lg n \rfloor$ when $n > 1$. I conjecture that $f(n) \le \lfloor cL(n)/2^{\lg^* n} \rfloor - 2$ for all $n > 0$. It is quite easy to prove the weaker bound $f(n) \le 4L(n) - 1$ by induction.

How many binary trees give the value n? If this number is c_n, the generating function $C(z)$ satisfies

$$C(z) = \frac{1}{1 - c_0 z - c_1 z^2 - c_2 z^4 - c_3 z^8 - \cdots}\,,$$

so we find that $C(z) = 1 + z + 2z^2 + 3z^3 + 7z^4 + 12z^5 + 23z^6 + 41z^7 + 81z^8 + 149z^9 + 282z^{10} + \cdots$; what is the asymptotic growth?

How many distinct values can you get by inserting parentheses into the expression $2 \uparrow 2 \uparrow \cdots \uparrow 2$, when there are n 2s? Since $2^{2^b} \uparrow 2^{2^a} = 2^{2^{a \star b}}$, this is the same as the number of distinct values you can get by inserting parentheses into the expression $0 \star 0 \star \cdots \star 0$ when there are n 0s. So it's the number of distinct values obtainable from $n-1$-node binary trees. This sequence begins $1, 1, 1, 2, 4, 8, 17, 36, 78, 171, 379$, according to Guy and Selfridge [*AMM* **80** (1973), 868–876], but its general characteristics are unknown.

Other problems concern time bounds for the algorithms below.

```
#include <stdio.h>
  ⟨ Type definitions 11 ⟩
  ⟨ Global variables 2 ⟩
  ⟨ Basic subroutines 13 ⟩
  ⟨ Subroutines 20 ⟩
  main( )
  {
    register int k;
    register node *p;
    ⟨ Initialize the data structures 5 ⟩;
    while (1) ⟨ Prompt the user for a command and execute it 3 ⟩
  }
```

2. Input conventions. The user types short commands in a simple postfix language. For example, '`f2 f3 + s`' means, "fetch a copy of tree"2 (the output of a previous step), also fetch tree 3, add them, and compute the successor." This tree will be displayed, and it might be fetched in later commands.

Spaces in the input are ignored. Numbers are treated as decimal parameters for the operator that they follow; the default is zero if no explicit parameter is given. Each operator is a single character with mnemonic significance.

The "user manual" is distributed throughout this program, since I keep adding features as I go. The operator '`h`' gives online help—a brief summary of all operators.

```
#define buf_size 200
⟨ Global variables 2 ⟩ ≡
  char *helps[128];      /* strings describing each operator */
  char buf[buf_size];    /* the user's input goes here */
  char *loc;     /* where we are looking in the buffer */
  char op;       /* the current operator */
  int param;     /* the parameter to the current operator */
```

See also sections 12, 24, 28, 45, 50, 53, 58, and 63.

This code is used in section 1.

3. ⟨ Prompt the user for a command and execute it 3 ⟩ ≡

```
{
  ⟨ Fill buf with the user's next sequence of commands 9 ⟩;
  ⟨ Clear the current stack 29 ⟩;
  while (1) {
    ⟨ Set op and param for the next operator 10 ⟩;
    switch (op) {
    case '\n': goto dump_stack;
      ⟨ Cases for one-character operators 6 ⟩
    default: printf("Unknown operator '%c'!\n", op);
    }
  }
dump_stack: ⟨ Display and save all trees currently in the stack 30 ⟩;
  ⟨ Check that the saved trees account for all the used nodes 61 ⟩;
}
```

This code is used in section 1.

4. Here's an example of how each operator is introduced; we begin with the 'help' feature.

```
⟨ Define the help strings 4 ⟩ ≡
  helps['h'] = ".helpful summary of all known operators";
    /* '.' means that this operator ignores its parameter */
```

See also sections 7, 31, 33, 36, 38, 48, and 51.

This code is used in section 5.

5. ⟨ Initialize the data structures 5 ⟩ ≡

```
⟨ Define the help strings 4 ⟩;
```

See also section 46.

This code is used in section 1.

6. I must remember to say **break** at the end of the program for each case.

```
⟨ Cases for one-character operators 6 ⟩ ≡
case 'h': printf("The␣following␣operators␣are␣currently␣implemented:\n");
  for (k = 0; k < 128; k++)
    if (helps[k]) printf("␣␣%c%s␣%s\n", k, (*helps[k] ≡ '.' ? ":␣␣␣" : "<n>:"), helps[k] + 1);
  break;
```

See also sections 8, 32, 34, 35, 37, 39, 40, 41, 42, 43, 49, and 52.

This code is used in section 3.

7. Here's another easy case: The normal way to stop, instead of resorting to control-C, is to give the quit command.

```
⟨ Define the help strings 4 ⟩ +≡
  helps['q'] = ".quit␣the␣program";
```

8.

```
⟨ Cases for one-character operators 6 ⟩ +≡
case 'q': printf("Type␣<return>␣to␣confirm␣quitting:");
  if (getchar( ) ≡ '\n') return 0;
  fgets(buf, buf_size, stdin);      /* flush the rest of that line */
  goto dump_stack;
```

9.

```
⟨ Fill buf with the user's next sequence of commands 9 ⟩ ≡
  printf("?␣");      /* this is the prompt */
  if (fgets(buf, buf_size, stdin) ≡ 0) return 0;      /* we quit at end of file */
  loc = buf;      /* get ready to scan the buffer */
```

This code is used in section 3.

10. The scanning routine is intentionally simple.

```
#define large 1000000000      /* parameter numbers aren't allowed to get this big */
#define larg 100000000      /* large/10 */
⟨ Set op and param for the next operator 10 ⟩ ≡
  while (loc < &buf[buf_size] ∧ (*loc ≡ '␣' ∨ *loc < 0 ∨ *loc ≥ 128)) loc++;
      /* bypass blanks and exotic characters */
  param = 0;      /* assign the default value */
  if (loc ≡ &buf[buf_size]) op = '\n';
  else {
    op = *loc++;
    if (op ≠ '\n')
      while (loc < &buf[buf_size] ∧ (*loc ≡ '␣' ∨ (*loc ≤ '9' ∧ *loc ≥ '0'))) {
        if (*loc ≠ '␣') {
          if (param ≥ larg) printf("(I'm␣reducing␣your␣large␣parameter␣mod␣%d)\n", large);
          param = ((param % larg) * 10) + *loc - '0';
        }
        loc++;
      }
  }
```

This code is used in section 3.

11. Data structures. I'm representing trees in the obvious way: Each node consists of two pointers and one integer field for multipurpose use.

```
⟨ Type definitions 11 ⟩ ≡
  typedef struct node_struct {
    int val;      /* a value temporarily stored with this node */
    struct node_struct *l, *r;      /* left and right subtree pointers */
  } node;
```

This code is used in section 1.

12. Storage allocation is dynamic. We will explicitly free nodes when they are no longer active.

```
⟨ Global variables 2 ⟩ +≡
  int used;      /* this many nodes are active */
  node *cur_node;      /* the next node to be allocated when we run out */
  node *bad_node;      /* if cur_node equals bad_node, we need another block */
  node *avail;      /* head of list of recycled nodes */
  int mems;      /* the number of memory references to node pointers */
```

13. `#define nodes_per_block 1000`

```
⟨ Basic subroutines 13 ⟩ ≡
  node *get_avail( )      /* allocate a node */
  {
    register node *p;
    if (avail) {
      p = avail;
      avail = p→r;
    }
    else {
      if (cur_node ≡ bad_node) {
        cur_node = (node *) calloc(nodes_per_block, sizeof(node));
        if (¬cur_node) {
          printf("Omigosh, the memory is all gone!\n");
          exit(−1);
        }
        bad_node = cur_node + nodes_per_block;
      }
      p = cur_node++;
    }
    p→l = p→r = Λ;
    mems++;
    used++;
    return p;
  }
```

See also sections 14, 15, 16, 17, 18, 19, 55, 60, and 62.

This code is used in section 1.

14. ⟨ Basic subroutines 13 ⟩ +≡

```
void free_node(p)      /* deallocate a node */
    node *p;
{
  p→r = avail;
  avail = p;
  used−−;
  mems++;
}
```

15. We often want to free all nodes of a tree that has served its purpose.

⟨ Basic subroutines 13 ⟩ +≡

```
void recycle(p)      /* deallocate an entire tree */
    node *p;
{
  if (¬p) return;
  recycle(p→l);
  recycle(p→r);
  free_node(p);
}
```

16. The algorithms for arithmetic are careful (I hope) to access node pointers only via the subroutines *left*, *right*, and *change*. This makes the program longer and slightly less readable, but it also ensures that *mems* will be properly counted.

In my first draft of this code I implemented a reference counter scheme, but I soon found that explicit deallocation was much better.;

⟨ Basic subroutines 13 ⟩ +≡

```
node *left(p)      /* get the left subtree of a nonempty binary tree */
    node *p;
{
  mems++;
  return p→l;
}
node *right(p)      /* get the right subtree of a nonempty binary tree */
    node *p;
{
  mems++;
  return p→r;
}
void change(p, q)      /* change pointer field p to q */
    node **p;
    node *q;
{
  *p = q;
  mems++;
}
```

17. Simple tree operations. The multiplication routine needs to make several copies of subtrees, and the copying algorithm is one of the simplest we will need. So let's start with it. We can do the harder stuff once we get into the groove.

```
⟨ Basic subroutines 13 ⟩ +≡
  node *copy(p)      /* make a fresh copy of a binary tree */
      node *p;
  {
    register node *q;
    if (¬p) return Λ;
    q = get_avail( );
    change(&q→l, copy(left(p)));
    change(&q→r, copy(right(p)));
    return q;
  }
```

18. Sometimes I want to copy tree behind the scenes; then I don't want to count mems.

```
⟨ Basic subroutines 13 ⟩ +≡
  node *cheap_copy(p)      /* make a copy with no mem cost */
      node *p;
  {
    register node *q;
    register int m = mems;
    q = copy(p);
    mems = m;
    return q;
  }
```

19. Another easy case, frequently needed for arithmetic, is the lexicographic comparison of binary trees. The *compare* subroutine returns -1, 0, or $+1$ according as $p < q$, $p = q$, or $p > q$.

```
⟨ Basic subroutines 13 ⟩ +≡
  int compare(p, q)      /* determine whether p is less than, equal to, or greater than q */
      node *p, *q;
  {
    register int k;
    if (¬p) {
      if (¬q) return 0;      /* they were both empty */
      return −1;      /* only p was empty, so it's less */
    }
    if (¬q) return 1;      /* only q was empty, so p was greater */
    k = compare(left(p), left(q));
    if (k ≠ 0) return k;
    return compare(right(p), right(q));
  }
```

20. Having laid the groundwork, we come now to the first interesting case: The *succ* function adds one to the value represented by a binary tree.

This function would have been more efficient if we had represented numbers from right to left instead of from left to right. (A dual representation considers $2^a + b$ where b is a multiple of 2^{a+1} rather than a number less than 2^a. The same number of nodes appears in both representations; the difference is sort of a reflection of the tree about a diagonal line, with a few additional alterations.) But the dual representation makes the comparison operation slower, and comparisons is more important than successions.

The *succ* routine is given a pointer to a nonempty binary tree. It changes that tree T so that the new tree T' has $v(T') = v(T) + 1$; furthermore, T' has the same root node as T. Thus this operation is sort of like 'T++'.

```
⟨ Subroutines 20 ⟩ ≡
  void succ(p)      /* add one to tree p */
      node *p;
  {
    register node *pr, *pl, *prr, *prl;
    pr = right(p);
    if (¬pr) {
      pr = get_avail( );
      change(&p→r, pr);
    }
    else succ(pr);
    prr = right(pr);
    if (¬prr) {      /* the successor of pr was a power of two; should we propagate a carry? */
      pl = left(p);
      prl = left(pr);
      if (compare(pl, prl) ≡ 0) {      /* yes, we should */
        recycle(pr);
        change(&p→r, Λ);
        if (pl) succ(pl);
        else change(&p→l, get_avail( ));
      }
    }
  }
```

See also sections 21, 25, 26, 27, 44, 47, 54, and 57.

This code is used in section 1.

21. Addition. Arithmetic is now within our grasp. Again we need to think a bit about how to do it from left to right (i.e., from most significant bit to least significant). Here is a way that keeps the number of subtree comparisons to essentially the same amount as would be needed if we went from right to left.

Trees p and q are destroyed by the action of the *sum* procedure, which returns a tree that represents $v(T)+v(T')$. It also sets the global variable *easy* nonzero if no carry propagated from the most significant bits.

Incidentally, this addition operation is not associative on abnormal trees. The associative law fails, for example, on the first case I tried when I got the program working:

$$b_{123}+(b_{456}+b_{789})=2^{1+1+4}+2^{1+1}+4+2+2^{1+2^{1+1}}+1\,;$$
$$(b_{123}+b_{456})+b_{789}=2^{1+1+4}+2^{1+1}+4+1+2+2^{1+2^{1+1}}\,.$$

```
⟨ Subroutines 20 ⟩ +≡
  node *sum(p, q)      /* compute the sum of two binary trees */
      node *p, *q;
  {
    register node *pl, *ql;
    register int s;
    easy = 1;
    if (¬p) return q;
    if (¬q) return p;
    pl = left(p);
    ql = left(q);
    s = compare(pl, ql);
    if (s ≡ 0) {
      ⟨ Add right(p) to right(q) and append this to succ(pl) 23 ⟩
      easy = 0; return p;
    }
    else {
      if (s < 0) ⟨ Swap p and q so that p > q 22 ⟩;
      q = sum(right(p), q);
      if (easy) goto no_sweat;
      else {
        ql = left(q);
        s = compare(pl, ql);      /* does a carry need to be propagated? */
        if (s ≡ 0) {      /* yup */
          change(&p→r, right(q));
          recycle(ql); free_node(q);
          if (pl) succ(pl);
          else change(&p→l, get_avail( ));
          return p;
        }
        else easy = 1;      /* nope */
      }
    no_sweat: change(&p→r, q);
      return p;
    }
  }
```

22. ⟨ Swap p and q so that $p > q$ 22 ⟩ ≡

```
{
  pl = ql;
  ql = p;
  p = q;
  q = ql;
}
```

This code is used in section 21.

23. ⟨ Add *right*(*p*) to *right*(*q*) and append this to *succ*(*pl*) 23 ⟩ ≡

```
recycle(ql);
if (pl) succ(pl);
else change(&p→l, get_avail( ));
change(&p→r, sum(right(p), right(q)));
free_node(q);
```

This code is used in section 21.

24. ⟨ Global variables 2 ⟩ +≡

```
int easy;    /* communication parameter for the sum routine */
```

25. One nice spinoff of the addition routine is the following procedure for normalization:

⟨ Subroutines 20 ⟩ +≡

```
node *normalize(p)    /* change p to normal form without changing the value */
    node *p;
{
  register node *q, *qq;
  if (¬p) return Λ;
  q = qq = left(p);
  q = normalize(q);
  if (q ≠ qq) change(&p→l, q);
  q = qq = right(p);
  q = normalize(q);
  change(&p→r, Λ);
  return sum(p, q);
}
```

26. Multiplication. A moment's thought reveals the somewhat surprising fact that it's easier to multiply by 2^a than by a. (Because if $b = 2^{b_1} + \cdots + 2^{b_k}$, we have $2^a b = 2^{a+b_1} - \cdots + 2^{a+b_k}$.)

```
⟨ Subroutines 20 ⟩ +≡
  node *ez_prod(p, q)      /* add p to exponents of q */
      node *p, *q;
  {
    register node *qq, *qqr;
    if (¬q) {
      recycle(p);
      return Λ;
    }
    for (qq = q; qq; qq = qqr) {
      qqr = right(qq);
      if (qqr) change(&qq→l, sum(left(qq), copy(p)));
      else change(&qq→l, sum(left(qq), p));
    }
    return q;
  }
```

27. Full multiplication is, of course, a sum of such partial multiplications. I am not implementing it in the cleverest way, since I compute the final sum as $(\cdots((2^{a_1}b + 2^{a_2}b) + 2^{a_3}b) + \cdots) + 2^{a_k}b$, thereby passing k times over many of the nodes. It's obvious how to reduce this to $\log k$ times per node, but is there a better way? I leave that as an open problem for now.

A bit of experimentation shows that the product of abnormal trees might not even be commutative, much less associative.

```
⟨ Subroutines 20 ⟩ +≡
  node *prod(p, q)      /* form the product of p and q */
      node *p, *q;
  {
    register node *pp, *ppr, *ss;
    if (¬p ∨ ¬q) {
      recycle(p);
      recycle(q);
      return Λ;
    }
    for (pp = p, ss = Λ; pp; pp = ppr) {
      ppr = right(pp);
      if (ppr) ss = sum(ss, ez_prod(left(pp), copy(q)));
      else ss = sum(ss, ez_prod(left(pp), q));
      free_node(pp);
    }
    return ss;
  }
```

28. Stack discipline. Some commands put trees on the stack; others operate on those trees. Everything left on the stack at the end of a command line is displayed, and assigned an identification number for later use.

```
#define stack_size 20 /* this many trees can be on the stack at once */
#define save_size 1000 /* this many trees can be recalled */
⟨ Global variables 2 ⟩ +≡
  node *saved[save_size]; /* trees that the user might recall */
  int save_ptr; /* the number of saved trees */
  node *stack[stack_size + 1]; /* there's one extra slot for breathing space */
  int stack_ptr; /* the number of items on the stack */
  int showing_mems; /* should we tell the user how many mems were used? */
  int showing_size; /* should we tell the user how big each tree is? */
  int showing_usage; /* should we tell the user how many nodes are active? */
  int old_mems; /* holding place for mems until we're ready to report it */
```

29. ⟨ Clear the current stack 29 ⟩ ≡

```
stack_ptr = 0;
mems = 0;
```

This code is used in section 3.

30. The tree most recently in the stack is kept in *saved*[0].

```
#define operand(n) stack[stack_ptr - (n)]
⟨ Display and save all trees currently in the stack 30 ⟩ ≡
  old_mems = mems;
  while (stack_ptr) {
    stack_ptr--;
    if (++save_ptr < save_size) k = save_ptr;
    else {
      k = 0;
      recycle(saved[0]);
      save_ptr = save_size - 1;
    }
    saved[k] = operand(0);
    if (stack_ptr ≡ 0 ∧ k > 0) {
      recycle(saved[0]);
      saved[0] = copy(saved[k]);
    }
    ⟨ Display tree saved[k] 59 ⟩;
  }
  if (showing_mems ∧ old_mems) printf("Operations cost %d mems\n", old_mems);
  if (showing_usage) printf("(%d nodes are now in use)\n", used);
```

This code is used in section 3.

31. ⟨ Define the help strings 4 ⟩ +≡

```
helps['S'] = ":show tree sizes, if <n> is nonzero";
helps['T'] = ":show computation time in mems, if <n> is nonzero";
helps['U'] = ":show node usage, if <n> is nonzero";
helps['k'] = ":kill %<n> to conserve memory";
```

32. ⟨ Cases for one-character operators 6 ⟩ +≡

```
case 'S': showing_size = param; break;
case 'T': showing_mems = param; break;
case 'U': showing_usage = param; break;
case 'k':
  if (param > save_ptr)
    printf("You can't do k%d, because %%%d doesn't exist!\n", param, param);
  else {
    recycle(saved[param]);
    saved[param] = Λ;
  }
  break;
```

33. One way to put a new item on the stack is to copy an old item.

⟨ Define the help strings 4 ⟩ +≡

```
  helps['%'] = ":recall a previously computed tree";
  helps['d'] = ":duplicate a tree that's already on the stack";
```

34. ⟨ Cases for one-character operators 6 ⟩ +≡

```
case '%':
  if (param > save_ptr) {
    printf("(%%%d is unknown; I'm using %%0 instead)\n", param);
    param = 0;
  }
  operand(0) = cheap_copy(saved[param]);
inc_stack:
  if (stack_ptr < stack_size) {
    stack_ptr++;
    break;
  }
  printf("Oops---the stack overflowed!\n");
  recycle(operand(0));
  goto dump_stack;
```

35. The command '`d`' duplicates the top tree on the stack. Similarly, '`d3`' duplicates the item three down from the top.

```
#define check_stack(k)
        if (stack_ptr < k) {
          printf("Not enough items on the stack for operator %c!\n", op);
          goto dump_stack;
        }
```

⟨ Cases for one-character operators 6 ⟩ +≡

```
case 'd': check_stack(param + 1);
  operand(0) = cheap_copy(operand(param + 1));
  goto inc_stack;
```

36. Here are two trivial operations that seem pointless, because I haven't allowed the user to define macros. But in fact, users do have macros, because they can run TCALC from an emacs shell.

⟨ Define the help strings 4 ⟩ +≡

```
  helps['p'] = ".pop the top tree off the stack";
  helps['x'] = ".exchange the top two trees";
```

37. Of course I could generalize these commands so that *param* is relevant.

```
⟨ Cases for one-character operators 6 ⟩ +≡
case 'p': check_stack(1);
  stack_ptr−−;
  recycle(operand(0));
  break;
case 'x': check_stack(2);
  p = operand(2); operand(2) = operand(1); operand(1) = p;
  break;
```

38. Now we implement the arithmetic operators. Later we'll define an operator `t` such that '`tj`' replaces tree a by 2^a.

```
⟨ Define the help strings 4 ⟩ +≡
  helps['l'] = ".replace␣tree␣by␣its␣log␣(the␣left␣subtree)";
  helps['r'] = ".replace␣tree␣by␣its␣remainder␣(the␣right␣subtree)";
  helps['s'] = ".replace␣tree␣by␣its␣successor";
  helps['n'] = ".normalize␣a␣tree";
  helps['+'] = ".replace␣a,b␣by␣a+b";
  helps['*'] = ".replace␣a,b␣by␣ab";
  helps['^'] = ".replace␣a,b␣by␣a^b,␣assuming␣that␣a␣is␣a␣power␣of␣2";
  helps['j'] = ".replace␣a,b␣by␣2^a+b";     /* j is for "join" */
  helps['m'] = ".replace␣a,b␣by␣2^a␣b";
```

39. Here's a typical unary operator.

```
⟨ Cases for one-character operators 6 ⟩ +≡
case 'n': check_stack(1);      /* normalization */
  operand(1) = normalize(operand(1));
  break;
```

40. And another, only slightly more tricky.

```
⟨ Cases for one-character operators 6 ⟩ +≡
case 's': check_stack(1);      /* the succ operation */
  if (operand(1)) succ(operand(1));
  else operand(1) = get_avail( );
  break;
```

41. The `l` and `r` operators are charged as many mems as it takes to recycle the discarded nodes of the tree.

```
⟨ Cases for one-character operators 6 ⟩ +≡
case 'l': check_stack(1);  /* the log operation */
  p = operand(1);
  if (¬p) printf("(log␣0␣is␣undefined;␣I'm␣using␣0)\n");
  else {
    operand(1) = left(p);
    recycle(right(p));
    free_node(p);
  }
  break;
case 'r': check_stack(1);  /* the rem operation */
  p = operand(1);
  if (¬p) printf("(rem␣0␣is␣undefined;␣I'm␣using␣0)\n");
  else {
    operand(1) = right(p);
    recycle(left(p));
    free_node(p);
  }
  break;
```

42. Binary operations are equally simple.

```
⟨ Cases for one-character operators 6 ⟩ +≡
case 'j': check_stack(2);  /* prepare for joining */
  stack_ptr−−;
  p = get_avail( );
  p→l = operand(1);
  p→r = operand(0);
return_p: operand(1) = p;
  break;
case '+': check_stack(2);  /* prepare for addition */
  stack_ptr−−;
  operand(1) = sum(operand(1), operand(0));
  break;
case '*': check_stack(2);  /* prepare for multiplication */
  stack_ptr−−;
  operand(1) = prod(operand(1), operand(0));
  break;
case 'm': check_stack(2);  /* prepare for power-of-2 multiplication */
  stack_ptr−−;
  operand(1) = ez_prod(operand(1), operand(0));
  break;
```

43. Here's the only one that's not quite trivial. Strictly speaking, I should disallow 0^x; but the implementation is so easy, I went ahead and did it.

```
⟨ Cases for one-character operators 6 ⟩ +≡
case '^': check_stack(2);      /* prepare for exponentiation */
  stack_ptr--;
  p = operand(1);
  if (¬p) {
    if (operand(0)) recycle(operand(0));
    else p = get_avail( );      /* 0^0 = 1 */
  }
  else if (right(p)) {
    printf("Sorry, I don't do a^b unless a is a power of 2!\n");
    stack_ptr++;
    goto dump_stack;
  }
  else change(&p→l, prod(left(p), operand(0)));
  goto return_p;
```

44. Generating binary trees. But how do the trees get built in the first place? One useful way to get a fairly big tree is to ask for '`t`n', the tree that canonically represents n. Then we can get bigger by multiplication and exponentiation, etc.

If this program is working properly, and if n does not exceed the *threshold* for compression to be described below, the binary tree created here will be displayed simply as the integer n.

```
⟨ Subroutines 20 ⟩ +≡
  node *normal_tree(n)      /* generate the standard tree representation of n */
      int n;
  {
    register int k;
    register node *p;
    if (¬n) return Λ;
    for (k = 0; (1 ≪ k) ≤ n; k++) ;      /* compute k = 1 + ⌊lg n⌋ */
    p = get_avail( ); mems −−;
    p→l = normal_tree(k − 1);
    p→r = normal_tree(n − (1 ≪ (k − 1)));
    return p;
  }
```

45. There's also a convenient way to build random binary trees, so that we can experiment with abnormal structures.

For these, it's handy to have a table of the Catalan numbers, which enumerate the binary trees that have n nodes.

```
⟨ Global variables 2 ⟩ +≡
  int cat[20];      /* the first twenty Catalan numbers; cat[19] = 1767263190 */
```

46. We have to be careful when evaluating $cat[n] = (4n-2)cat[n-1]/(n+1)$, because the intermediate result might overflow even though the answer is a single-precision integer.

```
⟨ Initialize the data structures 5 ⟩ +≡
  cat[0] = 1;
  for (k = 1; k < 20; k++) {
    register int quot = cat[k − 1]/(k + 1), rem = cat[k − 1] % (k + 1);
    cat[k] = (4 * k − 2) * quot + (int)(((4 * k − 2) * rem)/(k + 1));
  }
```

47. The *btree* subroutine is called only when $0 \le m < cat[n]$.

```
⟨ Subroutines 20 ⟩ +≡
  node *btree(n, m)      /* generate the mth binary tree that has n nodes */
      int n, m;
  {
    register node *p;
    register int k;
    if (¬n) return Λ;
    for (k = 0; cat[k] * cat[n − 1 − k] ≤ m; k++) m −= cat[k] * cat[n − 1 − k];
    p = get_avail( ); mems −−;
    p→l = btree(k, (int)(m/cat[n − 1 − k]));
    p→r = btree(n − 1 − k, m % cat[n − 1 − k]);
    return p;
  }
```

48. ⟨ Define the help strings 4 ⟩ +≡

```
helps['t'] = ":the␣standard␣tree␣that␣represents␣<n>";
helps['b'] = ":the␣binary␣tree␣of␣rank␣<n>␣in␣lexicographic␣order";
```

49. Lexicographic order of binary trees is taken to mean that we order them first by number of nodes, then recursively by the order of the *compare* function.

⟨ Cases for one-character operators 6 ⟩ +≡

```
case 't': operand(0) = normal_tree(param);
  goto inc_stack;
case 'b':
  for (k = 0; cat[k] ≤ param; k++) param −= cat[k];
  operand(0) = btree(k, param);
  goto inc_stack;
```

50. Displaying the results. And finally, the grand climax—the most interesting algorithm in this whole program.

A special form of display is appropriate for the binary trees we're considering. If a tree is in normal form, we can describe it by simply stating its value. However, a small binary tree can have a super-astronomical value; there is in fact a tree with six nodes whose numerical value involves more decimal digits than there are molecules in the universe! So we use power-of-two notation whenever the value of a subtree exceeds a given *threshold*.

If *threshold* = 0, for example, the printed representation of 19, a tree of seven nodes, takes five lines:

```
0
2
2     0
2     2   0
2    +2  +2
```

(There's one '2' for each node, and one '+' for each node with a nonnull right subtree.) But if *threshold* = 1, the displayed output will be

```
1
2
2    1
2   +2 +1
```

And with *threshold* = 2 it becomes simpler yet:

```
2
2
2  +2+1
```

With *threshold* = 3 the '2+1' becomes '3', and with *threshold* ≥ 19 the whole tree is displayed simply as '19'.

If a binary tree is not in normal form, its normal-form subtrees are displayed as usual but its abnormal subtrees are displayed as if the threshold were exceeded. For example, a two-node tree that has no left subtree will be displayed as '1+1' for all values of *threshold* > 0. This convention ensures that the tree structure is uniquely characterized by the display.

Some binary trees are so huge, we don't want to see them displayed in full. The user can suppress detailed output of any tree with *max_display_size* or more nodes. The value of *max_display_size* must exceed the default value of 1000.

```
#define max_tree 1000    /* we don't display trees having this many nodes */
⟨ Global variables 2 ⟩ +≡
  int threshold;    /* trees are compressed if their value is at most this */
  int max_display_size = max_tree;    /* trees are shown if their size is less than this */
```

51. ⟨ Define the help strings 4 ⟩ +≡

```
  helps['M'] = ".use␣maximum␣possible␣compression␣threshold␣for␣tree␣display";
  helps['N'] = ":compress␣tree␣displays␣only␣for␣t0..t<n>";
  helps['O'] = ":omit␣display␣of␣trees␣having␣<n>␣or␣more␣nodes";
```

52. ⟨ Cases for one-character operators 6 ⟩ +≡

```
case 'M': param = large − 1;
case 'N': threshold = param;
  break;
case 'O':
  if (param > max_tree) {
    printf("(I've␣changed␣O%d␣to␣the␣maximum␣permitted␣value,␣O%d)\n", param, max_tree);
    param = max_tree;
  }
  max_display_size = param;
  break;
```

53. The idea we'll use to display a tree is to tackle the job in two phases. First, we compute statistics about the tree nodes, so that the root of the tree "knows" about its subtrees. Then we recursively print each line of the display.

The statistics-gathering phase is handled by a routine called *get_state*. It first stamps each node with a serial number j, which turns out to be the index of that node in postorder. Then it computes several important facts about that node's subtree: $width[j]$, the number of columns needed to display this subtree; $height[j]$, the number of rows needed to display this subtree, not counting the base row; $code[j]$, the numerical value of this subtree; and $lcode[j]$, which is zero if no `+` sign will be printed for this subtree, otherwise it's the code for the part that precedes the `+`. An abnormal subtree is always considered *large*.

Initialization constants here apply to the empty binary tree, whose width is 1 because it's always displayed as '`O`'.

⟨ Global variables 2 ⟩ +≡

```
int width[max_tree] = {1};      /* columns needed to display a subtree */
int height[max_tree];           /* extra rows needed to display a subtree */
int code[max_tree];             /* compressed numerical value, or large */
int lcode[max_tree];            /* extra info when this subtree needs a + sign */
int count;      /* this will be set to the number of nodes in the tree */
```

54. ⟨ Subroutines 20 ⟩ +≡

```
void get_stats(p)       /* walk the tree and determine widths, lengths, etc. */
    node *p;
{
  register int j, jl, jr;
  if (¬p) return;
  get_stats(p→l); get_stats(p→r);       /* postorder traversal */
  jl = (p→l ? p→l→val : 0);
  jr = (p→r ? p→r→val : 0);
  p→val = j = ++count;
  if (count < max_display_size) ⟨ Compute stats for j from the stats of jl, jr 56 ⟩;
}
```

55. We need a subroutine to compute the width of a decimal number.

```
⟨ Basic subroutines 13 ⟩ +≡
  int dwidth(n)      /* how many digits do we need to print n? */
      int n;
  {
    register int j, k;
    for (j = 1, k = 10; n ≥ k; j++, k *= 10) ;      /* k = 10^j */
    return j;
  }
```

56. Here we assume that the constant called *large* is 1000000000. We use the facts that $threshold \le large$ and $2^{29} < large < 2^{30}$. Also the fact that $large + large < maxint$.

```
#define lg_large 29      /* ⌊log2 large⌋ */
⟨ Compute stats for j from the stats of jl, jr 56 ⟩ ≡
  {
    register int tjl;      /* 2^jl, or large */
    tjl = (code[jl] ≤ lg_large ? 1 ≪ code[jl] : large);
    if (tjl ≤ threshold) {
      if (code[jr] < tjl ∧ tjl + code[jr] ≤ threshold) {
        code[j] = tjl + code[jr];
        lcode[j] = 0;
        width[j] = dwidth(code[j]);
        height[j] = 0;
      }
      else {
        code[j] = large;
        lcode[j] = tjl;
        width[j] = dwidth(tjl) + width[jr] + 1;
        height[j] = height[jr];
      }
    }
    else {
      code[j] = large;
      width[j] = width[jl] + width[jr];
      if (p-r ≡ 0) lcode[j] = 0;
      else lcode[j] = large, width[j] += 2;
      height[j] = 1 + height[jl];
      if (height[jr] > height[j]) height[j] = height[jr];
    }
  }
```

This code is used in section 54.

57. The second phase is governed by another recursive procedure. This one, called *print_rep*, has three parameters representing a subtree to be displayed and its starting line and column numbers. Lines are numbered 0 and up from bottom to top.

Global variable h contains the line actually being printed. If $l \neq h$, we keep track of our position but don't emit any characters. The subroutine is called only when $l \le h \le l + height[j]$, where j is the postorder index of the subtree being printed.

Another global variable, *col*, represents the number of columns output so far on line h.

```
#define align_to(c)
        while (col < c) { col++; putchar('␣'); }
#define print_digs(n)
        { align_to(c); printf("%d", n); col += dwidth(n); }
#define print_char(n)
        { align_to(c); putchar(n); col++; }
⟨ Subroutines 20 ⟩ +≡
  void print_rep(p, l, c)      /* print the representation of p */
      node *p;      /* the subtree in question */
      int l, c;     /* the starting line and column positions */
  {
    register int j = (p ? p->val : 0);
    if (code[j] < large) {
      if (l ≡ h) print_digs(code[j]);
    }
    else if (lcode[j] ∧ lcode[j] < large) {
      if (l ≡ h) print_digs(lcode[j]);
    }
    else {
      register int jl = (p->l ? p->l->val : 0);
      if (l ≡ h) print_char('2');
      if (l < h ∧ l + 1 + height[jl] ≥ h) print_rep(p->l, l + 1, c + 1);
    }
    if (lcode[j]) {
      register jr = p->r->val;      /* we know that p->r ≠ Λ */
      if (l + height[jr] ≥ h) {
        c += width[j] − width[jr] − 1;
        if (l ≡ h) print_char('+');
        print_rep(p->r, l, c + 1);
      }
    }
  }
```

58.
```
⟨ Global variables 2 ⟩ +≡
  int h;      /* the row currently being printed */
  int col;    /* the col currently being printed */
```

59. OK, we've built the necessary recursive mechanisms; now we just have to supply the driver program.

```
⟨ Display tree saved[k] 59 ⟩ ≡
  count = 0;
  p = saved[k];
  get_stats(p);
  if (count ≥ max_display_size) printf("%%%d=large", k);
  else
    for (h = (p ? height[count] : 0); h ≥ 0; h−−) {
      if (h ≡ 0) printf("%%%d=", k);
      col = (h ≡ 0 ? dwidth(k) + 2 : 0);
      print_rep(p, 0, dwidth(k) + 2);
      if (h) printf("\n");
      else if (showing_size) {
        int c = dwidth(k) + 2 + width[count];
        align_to(c);
      }
    }
  if (showing_size) printf("␣(%d␣nodes)\n", count);
  else printf("\n");
```

This code is used in section 30.

60. Debugging. Finally, here are some quick-and-dirty routines that might be useful while I'm debugging.

The *eval* routine, which is invoked only by the debugger, computes $x_l \star x_r$ at every node of a possibly abnormal tree, and leaves these values in the *val* fields. It also returns the value of the whole tree.

```
⟨ Basic subroutines 13 ⟩ +≡
  int eval(p)      /* fills the val fields of nodes */
      node *p;
  {
    register int lv, rv;
    if (¬p) return 0;
    lv = eval(p→l);
    rv = eval(p→r);
    p→val = (lv ≤ lg_large ? 1 ≪ lv : large) + rv;
    if (p→val > large) p→val = large;
    return p→val;
  }
```

61. The next routine is used to check that I've recycled all the nodes. I could take it out, now that the program appears to work; but what the heck, this isn't a production program.

```
⟨ Check that the saved trees account for all the used nodes 61 ⟩ ≡
  ++time_stamp;
  count = 0;
  for (k = 0; k ≤ save_ptr; k++) stamp(saved[k]);
  if (count ≠ used) printf("We␣lost␣track␣of␣%d␣nodes!\n", used − count);
```

This code is used in section 3.

62.

```
⟨ Basic subroutines 13 ⟩ +≡
  void stamp(p)      /* stamp all nodes of p with time_stamp, and count them */
      node *p;
  {
    if (¬p) return;
    stamp(p→l);
    stamp(p→r);
    if (p→val ≡ time_stamp) printf("***Node␣overlap!!\n");
    p→val = time_stamp;
    count++;
  }
```

63.

```
⟨ Global variables 2 ⟩ +≡
  int time_stamp = large;      /* unique number */
```

64. Index.

⟨ Add *right*(*p*) to *right*(*q*) and append this to *succ*(*pl*) 23 ⟩ Used in section 21.
⟨ Basic subroutines 13, 14, 15, 16, 17, 18, 19, 55, 60, 62 ⟩ Used in section 1.
⟨ Cases for one-character operators 6, 8, 32, 34, 35, 37, 39, 40, 41, 42, 43, 49, 52 ⟩ Used in section 3.
⟨ Check that the saved trees account for all the *used* nodes 61 ⟩ Used in section 3.
⟨ Clear the current stack 29 ⟩ Used in section 3.
⟨ Compute stats for *j* from the stats of *jl*, *jr* 56 ⟩ Used in section 54.
⟨ Define the help strings 4, 7, 31, 33, 36, 38, 48, 51 ⟩ Used in section 5.
⟨ Display and save all trees currently in the stack 30 ⟩ Used in section 3.
⟨ Display tree *saved*[*k*] 59 ⟩ Used in section 30.
⟨ Fill *buf* with the user's next sequence of commands 9 ⟩ Used in section 3.
⟨ Global variables 2, 12, 24, 28, 45, 50, 53, 58, 63 ⟩ Used in section 1.
⟨ Initialize the data structures 5, 46 ⟩ Used in section 1.
⟨ Prompt the user for a command and execute it 3 ⟩ Used in section 1.
⟨ Set *op* and *param* for the next operator 10 ⟩ Used in section 3.
⟨ Subroutines 20, 21, 25, 26, 27, 44, 47, 54, 57 ⟩ Used in section 1.
⟨ Swap *p* and *q* so that $p > q$ 22 ⟩ Used in section 21.
⟨ Type definitions 11 ⟩ Used in section 1.

TCALC