(See https://cs.stanford.edu/~knuth/programs.html for date.)

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 $B(z) = z \exp\left(B(z) - \frac{1}{2}B(z^2)\frac{1}{3}B(z^3) - \cdots\right)$. 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 $A(z) = z \exp\left(A(z) + \frac{1}{2}A(z^2) + \frac{1}{3}A(z^3) + \cdots\right)$; 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 $v(x_l) \geq v(x_{rl})$. 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

$$f(0) = 0$$
; $f(2^a + b) = 1 + f(a) + f(b)$, when $0 \le b < 2^a$.

Let L(1) = 1 and $L(n) = \lfloor \lg n \rfloor L(\lfloor \lg n \rfloor)$ 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 $f(n) \leq \lfloor cL(n)/2^{\lg *n} \rfloor - 2$ for all n > 0. It is quite easy to prove the weaker bound $f(n) \leq 4L(n) - 1$ 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 $2^{2^b} \uparrow 2^{2^a} = 2^{2^{a*b}}$, 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.

2 INTRODUCTION TCALC §1

Other problems concern time bounds for the algorithms below.

§2 TCALC INPUT CONVENTIONS

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
\langle \text{Global variables 2} \rangle \equiv
  char *helps[128];
                             /* strings describing each operator */
                              /* the user's input goes here */
  char buf[buf\_size];
  char *loc;
                     /* where we are looking in the buffer */
  char op;
                   /* the current operator */
                     /* the parameter to the current operator */
  int param;
See also sections 12, 24, 28, 45, 50, 53, 58, and 63.
This code is used in section 1.
3. \langle Prompt the user for a command and execute it _3\rangle \equiv
  {
      \langle \text{ Fill } buf \text{ with the user's next sequence of commands } 9 \rangle;
      \langle \text{ Clear the current stack 29} \rangle;
     while (1)
        \langle \text{ Set } op \text{ and } param \text{ for the next operator } 10 \rangle;
        \mathbf{switch} \ (\mathit{op}) \ \{
        case '\n': goto dump\_stack;
           (Cases for one-character operators 6)
        default: printf("Unknown operator '%c'!\n", op);
  dump\_stack: \langle Display and save all trees currently in the stack 30 <math>\rangle;
     \langle Check that the saved trees account for all the used nodes 61\rangle;
This code is used in section 1.
4. Here's an example of how each operator is introduced; we begin with the 'help' feature.
\langle \text{ Define the help strings 4} \rangle \equiv
  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.
     \langle Initialize the data structures 5\rangle \equiv
  \langle \text{ Define the help strings 4} \rangle;
See also section 46.
This code is used in section 1.
```

4 INPUT CONVENTIONS TCALC $\S6$

```
I must remember to say break at the end of the program for each case.
\langle \text{ Cases for one-character operators } 6 \rangle \equiv
case 'h': printf("The_following_operators_are_currently_implemented:\n");
  for (k = 0; k < 128; k \leftrightarrow)
     if (helps[k]) printf("_{\sqcup\sqcup}%c%s_{\sqcup}%s_{\sqcup},k,(*helps[k] \equiv '.' ? ":_{\sqcup\sqcup\sqcup}" : "<n>:"), <math>helps[k]+1);
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.
\langle \text{ Define the help strings 4} \rangle + \equiv
  helps['q'] = ".quit_{\square}the_{\square}program";
8. \langle \text{Cases for one-character operators } 6 \rangle + \equiv
case 'q': printf("Type, <return>, to, confirm, quitting:");
  if (qetchar() \equiv ' \ ) return 0;
  fgets(buf, buf\_size, stdin); /* flush the rest of that line */
  goto dump_stack;
9. \langle Fill buf with the user's next sequence of commands \rangle \equiv
  printf("?_{\sqcup}");
                     /* this is the prompt */
  if (fgets(buf, buf\_size, stdin) \equiv 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 */
\langle \text{ Set } op \text{ and } param \text{ for the next operator } 10 \rangle \equiv
  while (loc < \&buf[buf\_size] \land (*loc \equiv ' \cup ' \lor *loc < 0 \lor *loc \ge 128)) loc ++;
        /* bypass blanks and exotic characters */
  param = 0; /* assign the default value */
  if (loc \equiv \&buf[buf\_size]) op = '\n';
  else {
     op = *loc ++;
     if (op \neq '\n')
       while (loc < \&buf[buf\_size] \land (*loc \equiv `_{\sqcup}` \lor (*loc \leq `9` \land *loc \geq `0`))) {
          if (*loc \neq ' \cup ') {
             if (param \ge 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 TCALC DATA STRUCTURES 5

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

```
\langle \text{Type definitions } 11 \rangle \equiv
  typedef struct node_struct {
                  /* a value temporarily stored with this node */
                                       /* left and right subtree pointers */
     struct node_struct *l, *r;
  } node;
This code is used in section 1.
12. Storage allocation is dynamic. We will explicitly free nodes when they are no longer active.
\langle \text{Global variables } 2 \rangle + \equiv
                 /* this many nodes are active */
  int used;
  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 */
                      /* head of list of recycled nodes */
  node *avail;
                   /* the number of memory references to node pointers */
  int mems;
13. #define nodes_per_block 1000
\langle \text{ Basic subroutines } 13 \rangle \equiv
  node *qet_avail()
                         /* allocate a node */
     register node *p;
     if (avail) {
       p = avail;
       avail = p \neg r;
     else {
       if (cur\_node \equiv bad\_node) {
          cur\_node = (\mathbf{node} *) \ calloc(nodes\_per\_block, \mathbf{sizeof}(\mathbf{node}));
          if (\neg cur\_node) {
            printf("Omigosh, \_the\_memory\_is\_all\_gone! \n");
             exit(-1);
          bad\_node = cur\_node + nodes\_per\_block;
       p = cur\_node ++;
     p \rightarrow l = p \rightarrow r = \Lambda;
     mems ++;
     used ++;
     return p;
See also sections 14, 15, 16, 17, 18, 19, 55, 60, and 62.
This code is used in section 1.
```

6 DATA STRUCTURES TCALC §14

```
\langle \text{ Basic subroutines } 13 \rangle + \equiv
  void free\_node(p)
                               /* deallocate a node */
        \mathbf{node}\ *p;
     p \rightarrow r = avail;
      avail = p;
      used --:
      mems ++;
15. We often want to free all nodes of a tree that has served its purpose.
\langle \text{Basic subroutines } 13 \rangle + \equiv
   void recycle(p)
                             /* deallocate an entire tree */
        \mathbf{node} *p;
      if (\neg p) return;
      recycle(p \rightarrow l);
      recycle(p \rightarrow 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.;

```
\langle \text{Basic subroutines } 13 \rangle + \equiv
                        /* get the left subtree of a nonempty binary tree */
  \mathbf{node} * left(p)
        \mathbf{node}\ *p;
     mems ++;
     return p \rightarrow l;
                          /* get the right subtree of a nonempty binary tree */
  node *right(p)
       node *p;
     mems ++;
     return p \rightarrow 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.

```
\langle \text{Basic subroutines } 13 \rangle + \equiv
  \mathbf{node} * copy(p)
                       /* make a fresh copy of a binary tree */
       node *p;
    register node *q;
    if (\neg p) return \Lambda;
    q = get\_avail();
    change(\&q \rightarrow l, copy(left(p)));
    change(\&q \neg r, copy(right(p)));
    return q;
  }
18. Sometimes I want to copy tree behind the scenes; then I don't want to count mems.
\langle Basic subroutines 13 \rangle + \equiv
                           /* make a copy with no mem cost */
  node * cheap\_copy(p)
       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.
\langle \text{Basic subroutines } 13 \rangle + \equiv
                          /* determine whether p is less than, equal to, or greater than q *
  int compare(p,q)
       node *p, *q;
    register int k;
    if (\neg p) {
       if (\neg q) return 0; /* they were both empty */
       return -1;
                      /* only p was empty, so it's less */
    if (\neg q) return 1;
                          /* only q was empty, so p was greater */
    k = compare(left(p), left(q));
    if (k \neq 0) return k;
    return compare(right(p), right(q));
```

ξ20

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++'.

```
\langle \text{Subroutines } 20 \rangle \equiv
                      /* add one to tree p */
  void succ(p)
       node *p;
     \textbf{register node} *pr, *pl, *prr, *prl;
     pr = right(p);
     if (\neg pr) {
       pr = get\_avail();
        change(\&p \neg r, pr);
     else succ(pr);
     prr = right(pr);
     if (\neg 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) \equiv 0) {
                                          /* yes, we should */
          recycle(pr);
          change(\&p \neg r, \Lambda);
          if (pl) succ(pl);
          else change(\&p \rightarrow l, get\_avail());
  }
See also sections 21, 25, 26, 27, 44, 47, 54, and 57.
```

This code is used in section 1.

§21 TCALC ADDITION 9

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}}.
\langle \text{Subroutines } 20 \rangle + \equiv
                              /* compute the sum of two binary trees */
  node *sum(p,q)
        node *p, *q;
     register node *pl, *ql;
     register int s;
     easy = 1;
     if (\neg p) return q;
     if (\neg q) return p;
     pl = left(p);
     ql = left(q);
     s = compare(pl, ql);
     if (s \equiv 0) {
        \langle \operatorname{Add} right(p) \operatorname{to} right(q) \operatorname{and} \operatorname{append} \operatorname{this} \operatorname{to} \operatorname{succ}(pl) \ 23 \rangle
        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);
                                          /* does a carry need to be propagated? */
           s = compare(pl, ql);
           if (s \equiv 0) {
                                /* yup */
              change(\&p \neg r, right(q));
              recycle(ql); free\_node(q);
              if (pl) succ(pl);
              else change(\&p \rightarrow l, get\_avail());
              return p;
           else easy = 1; /* nope */
     no\_sweat: change(\&p \neg r, q);
        return p;
```

}

10 ADDITION TCALC §22

```
\langle \text{Swap } p \text{ and } q \text{ so that } p > q | 22 \rangle \equiv
22.
     pl = ql;
     ql = p;
     p = q;
     q = ql;
This code is used in section 21.
23. \langle \text{Add } right(p) \text{ to } right(q) \text{ and append this to } succ(pl) \ 23 \rangle \equiv
   recycle(ql);
  if (pl) succ(pl);
  else change(\&p \neg l, get\_avail());
  change(\&p \neg r, sum(right(p), right(q)));
  free\_node(q);
This code is used in section 21.
24. \langle Global variables 2\rangle + \equiv
                   /* communication parameter for the sum routine */
  int easy;
25. One nice spinoff of the addition routine is the following procedure for normalization:
\langle Subroutines 20\rangle + \equiv
                                 /* change p to normal form without changing the value */
  \mathbf{node} * normalize(p)
        node *p;
     register node *q, *qq;
     if (\neg p) return \Lambda;
     q = qq = left(p);
     q = normalize(q);
     if (q \neq qq) change (\&p \neg l, q);
     q = qq = right(p);
     q = normalize(q);
     change(\&p \rightarrow r, \Lambda);
     return sum(p,q);
```

§26 TCALC MULTIPLICATION 11

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 $2^a b = 2^{a+b_1} - \cdots + 2^{a+b_k}$.)

```
 \langle \text{Subroutines 20} \rangle +\equiv \\ \mathbf{node} *ez\_prod(p,q) \quad /* \text{ add } p \text{ to exponents of } q */\\ \mathbf{node} *p, *q; \\ \{ \\ \mathbf{register node} *qq, *qqr; \\ \mathbf{if} (\neg q) \{ \\ recycle(p); \\ \mathbf{return} \ \Lambda; \\ \} \\ \mathbf{for} (qq = q; \ qq; \ qq = qqr) \ \{ \\ qqr = right(qq); \\ \mathbf{if} (qqr) \ change(\&qq\neg l, sum(left(qq), copy(p))); \\ \mathbf{else} \ change(\&qq\neg l, sum(left(qq), p)); \\ \} \\ \mathbf{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.

```
 \langle \text{Subroutines 20} \rangle +\equiv \\ \mathbf{node} *prod(p,q) & /* \text{ form the product of } p \text{ and } q */\\ \mathbf{node} *p, *q; \\ \{ \\ \mathbf{register node} *pp, *ppr, *ss; \\ \mathbf{if} \ (\neg p \lor \neg q) \ \{ \\ recycle(p); \\ recycle(q); \\ \mathbf{return} \ \Lambda; \\ \} \\ \mathbf{for} \ (pp = p, ss = \Lambda; \ pp; \ pp = ppr) \ \{ \\ ppr = right(pp); \\ \mathbf{if} \ (ppr) \ ss = sum(ss, ez\_prod(left(pp), copy(q))); \\ \mathbf{else} \ ss = sum(ss, ez\_prod(left(pp), q)); \\ free\_node(pp); \\ \} \\ \mathbf{return} \ ss; \\ \}
```

12 STACK DISCIPLINE TCALC §28

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 */
\langle \text{Global variables } 2 \rangle + \equiv
  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 */
                      /* the number of items on the stack */
  int stack_ptr;
  int showing_mems;
                            /* should we tell the user how many mems were used? */
                          /* should we tell the user how big each tree is? */
  int showing_size;
  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 */
     \langle Clear the current stack 29\rangle \equiv
  stack_ptr = 0;
  mems = 0;
This code is used in section 3.
      The tree most recently in the stack is kept in saved[0].
#define operand(n) stack[stack\_ptr - (n)]
\langle \text{Display and save all trees currently in the stack 30} \rangle \equiv
  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 \equiv 0 \land k > 0) {
       recycle(saved[0]);
       saved[0] = copy(saved[k]);
     \langle \text{ Display tree } saved[k] | 59 \rangle;
  if (showing\_mems \land 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. \langle \text{ Define the help strings 4} \rangle + \equiv
  helps['S'] = ":show_tree_sizes,_if_<n>_is_nonzero";
  helps['T'] = ":show_computation_time_in_mems,_if_<n>_is_nonzero";
  helps['U'] = ":show unode usage, uif u < n > uis unonzero";
  helps['k'] = ":kill_{\square}% < n>_{\square} to_{\square} conserve_{\square} memory";
```

```
32.
      \langle Cases for one-character operators 6 \rangle + \equiv
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! \ ", param, param);
     recycle(saved[param]);
     saved[param] = \Lambda;
  break;
33. One way to put a new item on the stack is to copy an old item.
\langle \text{ Define the help strings 4} \rangle + \equiv
  helps[,\%] = ":recall_a_previously_computed_tree";
  helps['d'] = ":duplicate_{\sqcup}a_{\sqcup}tree_{\sqcup}that's_{\sqcup}already_{\sqcup}on_{\sqcup}the_{\sqcup}stack";
34. \langle Cases for one-character operators 6 \rangle + \equiv
case '%':
  if (param > save\_ptr) {
     printf("(\%\%d_is_unknown;_uI'm_using_u\%\%0_instead)\n", param);
  operand(0) = cheap\_copy(saved[param]);
inc\_stack:
  if (stack\_ptr < stack\_size) {
     stack_ptr++;
     break:
  }
  printf("Oops---the_lstack_loverflowed!\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;
\langle Cases for one-character operators _{6}\rangle +\equiv
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.
\langle \text{ Define the help strings 4} \rangle + \equiv
  helps['p'] = ".pop_{\sqcup}the_{\sqcup}top_{\sqcup}tree_{\sqcup}off_{\sqcup}the_{\sqcup}stack";
```

 $helps['x'] = ".exchange_{\sqcup}the_{\sqcup}top_{\sqcup}two_{\sqcup}trees";$

14 STACK DISCIPLINE TCALC §37

```
Of course I could generalize these commands so that param is relevant.
\langle Cases for one-character operators _{6}\rangle +\equiv
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;
38. Now we implement the arithmetic operators. Later we'll define an operator t such that 'tj' replaces
tree a by 2^a.
\langle \text{ Define the help strings 4} \rangle + \equiv
  helps['1'] = ".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.
\langle Cases for one-character operators _{6}\rangle +\equiv
case 'n': check\_stack(1);
                              /* normalization */
  operand(1) = normalize(operand(1));
  break:
40. And another, only slightly more tricky.
\langle \text{ Cases for one-character operators } 6 \rangle + \equiv
case 's': check\_stack(1);
                            /* the succ operation */
  if (operand(1)) succ(operand(1));
  else operand(1) = get\_avail();
  break;
```

§41 TCALC STACK DISCIPLINE 15

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

```
\langle \text{ Cases for one-character operators } 6 \rangle + \equiv
case '1': check_stack(1);
                                 /* the log operation */
  p = operand(1);
  if (\neg p) printf("(log_0\u00cuis_undefined;_\u1'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 (\neg p) printf("(rem_{\cup}0_{\cup}is_{\cup}undefined;_{\cup}I'm_{\cup}using_{\cup}0)\n");
     operand(1) = right(p);
    recycle(left(p));
    free\_node(p);
  break;
42. Binary operations are equally simple.
\langle Cases for one-character operators 6 \rangle + \equiv
case 'j': check\_stack(2);
                                 /* prepare for joining */
  stack\_ptr --;
  p = get\_avail();
  p \rightarrow l = operand(1);
  p \rightarrow 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;
                                 /* prepare for power-of-2 multiplication */
case 'm': check\_stack(2);
  stack_ptr --;
  operand(1) = ez\_prod(operand(1), operand(0));
  break;
```

16 STACK DISCIPLINE TCALC $\S43$

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.

```
 \begin{array}{lll} \langle \operatorname{Cases} \ \operatorname{for} \ \operatorname{one-character} \ \operatorname{operators} \ 6 \rangle + \equiv \\ \operatorname{case} \ ' \ ' : \ \operatorname{check\_stack}(2); & / * \ \operatorname{prepare} \ \operatorname{for} \ \operatorname{exponentiation} \ * / \\ stack\_ptr --; & p = \operatorname{operand}(1); & \\ \operatorname{if} \ (\neg p) \ \{ & & \\ \operatorname{if} \ (\operatorname{operand}(0)) \ \operatorname{recycle}(\operatorname{operand}(0)); & \\ \operatorname{else} \ p = \operatorname{get\_avail}(); & / * \ 0^0 = 1 \ * / \\ \} & \\ \operatorname{else} \ \operatorname{if} \ (\operatorname{right}(p)) \ \{ & & \\ \operatorname{printf}("Sorry, \sqcup I \sqcup \operatorname{don't} \sqcup \operatorname{do}_{\sqcup} a \operatorname{b}_{\sqcup} \operatorname{unless}_{\sqcup} a \sqcup \operatorname{power}_{\sqcup} \operatorname{of}_{\sqcup} 2! \operatorname{n"}); & \\ \operatorname{stack\_ptr} ++; & & \\ \operatorname{goto} \ \operatorname{dump\_stack}; & \\ \} & \\ \operatorname{else} \ \operatorname{change}(\&p \dashv l, \operatorname{prod}(\operatorname{left}(p), \operatorname{operand}(0))); & \\ \operatorname{goto} \ \operatorname{return\_p}; & \\ \end{array}
```

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 'tn', 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.

```
 \begin{array}{lll} \langle \operatorname{Subroutines} \ 20 \rangle \ + \equiv \\ & \mathbf{node} \ *normal\_tree(n) & /* \ \operatorname{generate} \ \operatorname{the} \ \operatorname{standard} \ \operatorname{tree} \ \operatorname{representation} \ \operatorname{of} \ n \ */ \\ & \mathbf{int} \ n; \\ \{ & \mathbf{register} \ \mathbf{int} \ k; \\ & \mathbf{register} \ \mathbf{node} \ *p; \\ & \mathbf{if} \ (\neg n) \ \mathbf{return} \ \Lambda; \\ & \mathbf{for} \ (k=0; \ (1 \ll k) \leq n; \ k++) \ ; & /* \ \operatorname{compute} \ k = 1 + \lfloor \lg n \rfloor \ */ \\ & p = get\_avail(); \ mems --; \\ & p \neg l = normal\_tree(k-1); \\ & p \neg r = normal\_tree(n-(1 \ll (k-1))); \\ & \mathbf{return} \ p; \\ \} \end{array}
```

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.

```
\langle Global variables 2\rangle +\equiv 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.

```
\langle Initialize the data structures 5\rangle + \equiv
  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));
  }
    The btree subroutine is called only when 0 \le m < cat[n].
\langle Subroutines 20\rangle + \equiv
  node *btree(n, m)
                            /* generate the mth binary tree that has n nodes */
       int n, m;
     register node *p;
     register int k;
     if (\neg n) return \Lambda;
     for (k = 0; cat[k] * cat[n - 1 - k] \le m; k++) m -= cat[k] * cat[n - 1 - k];
    p = get\_avail(); mems --;
    p \rightarrow l = btree(k, (\mathbf{int})(m/cat[n-1-k]));
    p \rightarrow r = btree(n-1-k, m \% cat[n-1-k]);
     return p;
  }
```

TCALC

18

```
48. \langle Define the help strings 4 \rangle + \equiv
   helps['t'] = ": the_{\sqcup} standard_{\sqcup} tree_{\sqcup} that_{\sqcup} represents_{\sqcup} < 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.

```
\langle Cases for one-character operators 6 \rangle + \equiv
case 't': operand(0) = normal\_tree(param);
  goto inc_stack;
case 'b':
  \textbf{for} \ (k=0; \ cat[k] \leq 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 $threshold \ge 19$ 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.

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 '0'.

```
\langle \text{Global variables } 2 \rangle + \equiv
                                         /* columns needed to display a subtree */
  int width[max\_tree] = \{1\};
  int height[max_tree];
                                  /* extra rows needed to display a subtree */
                                /* compressed numerical value, or large */
  int code [max_tree];
  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. \langle Subroutines 20 \rangle + \equiv
  void qet\_stats(p)
                             /* walk the tree and determine widths, lengths, etc. */
        node *p:
     register int j, jl, jr;
     if (\neg p) return;
     get\_stats(p \rightarrow l); get\_stats(p \rightarrow r);
                                            /* postorder traversal */
     jl = (p \rightarrow l ? p \rightarrow l \rightarrow val : 0);
     jr = (p \rightarrow r ? p \rightarrow r \rightarrow val : 0);
     p \rightarrow val = j = ++ count;
     if (count < max\_display\_size) \langle Compute stats for j from the stats of jl, jr 56\rangle;
```

55. We need a subroutine to compute the width of a decimal number. $\langle \text{ Basic subroutines } 13 \rangle + \equiv$ int dwidth(n)/* how many digits do we need to print n? */ int n; register int j, k; for $(j = 1, k = 10; n \ge k; j++, k *= 10); /* k = 10^{j} */$ return j; } **56.** Here we assume that the constant called large is 10000000000. We use the facts that threshold $\leq large$ and $2^{29} < large < 2^{30}$. Also the fact that large + large < maxint. #define lg_large 29 $/* |\log_2 large| */$ \langle Compute stats for j from the stats of jl, jr 56 $\rangle \equiv$ { $/* 2^{jl}$, or large */ register int til; $tjl = (code[jl] \le lg_large ? 1 \ll code[jl] : large);$ **if** $(tjl \leq threshold)$ { if $(code[jr] < tjl \wedge tjl + code[jr] \le threshold)$ { 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 $(p - r \equiv 0)$ lcode[j] = 0;

This code is used in section 54.

}

else lcode[j] = large, width[j] += 2;

if (height[jr] > height[j]) height[j] = height[jr];

height[j] = 1 + height[jl];

TCALC

22

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(' \cup '); \}
#define print\_digs(n)
           \{ align\_to(c); printf("%d", n); col += dwidth(n); \}
#define print\_char(n)
           { align_to(c); putchar(n); col ++; }
\langle \text{Subroutines } 20 \rangle + \equiv
  void print_rep(p, l, c)
                                  /* print the representation of p */
        \mathbf{node} *p;
                      /* the subtree in question */
        int l, c;
                        /* the starting line and column positions */
     register int j = (p ? p \rightarrow val : 0);
     if (code[j] < large) {
        if (l \equiv h) print_digs (code[j]);
     else if (lcode[j] \wedge lcode[j] < large) {
        if (l \equiv h) print_digs(lcode[j]);
     else {
        register int jl = (p \rightarrow l ? p \rightarrow l \rightarrow val : 0);
        if (l \equiv h) print\_char('2');
        if (l < h \land l + 1 + height[jl] \ge h) print_rep(p \rightarrow l, l + 1, c + 1);
     if (lcode[j]) {
                                         /* we know that p r \neq \Lambda */
        register jr = p \rightarrow r \rightarrow val;
        if (l + height[jr] \ge h) {
           c += width[j] - width[jr] - 1;
           if (l \equiv h) print_char('+');
           print\_rep(p \rightarrow r, l, c + 1);
58. \langle \text{Global variables } 2 \rangle + \equiv
               /* the row currently being printed */
  int h;
  int col:
                 /* the col currently being printed */
```

§59 TCALC

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

```
\langle \text{ Display tree } saved[k] \text{ 59} \rangle \equiv
  count = 0;
  p = saved[k];
  get\_stats(p);
  if (count \ge max\_display\_size) printf("%%%d=large", k);
  else
     for (h = (p ? height[count] : 0); h \ge 0; h--) {
       if (h \equiv 0) printf("%%%d=", k);
        col = (h \equiv 0 ? dwidth(k) + 2 : 0);
        print_rep(p, 0, dwidth(k) + 2);
        \quad \textbf{if} \ (h) \ \mathit{printf}("\n");
        else if (showing_size) {
          int c = dwidth(k) + 2 + width[count];
           align\_to(c);
        }
  if (showing\_size) printf("u(%dunodes)\n", count);
  else printf("\n");
This code is used in section 30.
```

24 DEBUGGING TCALC §60

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.

```
 \langle \text{ Basic subroutines } 13 \rangle + \equiv \\ \text{ int } eval(p) \qquad / * \text{ fills the } val \text{ fields of nodes } */\\ \text{ node } *p; \\ \{ \\ \text{ register int } lv, \ rv; \\ \text{ if } (\neg p) \text{ return } 0; \\ lv = eval(p \neg l); \\ rv = eval(p \neg r); \\ p \neg val = (lv \leq lg \neg large ? 1 \ll lv : large) + rv; \\ \text{ if } (p \neg val > large) \ p \neg val = large; \\ \text{ return } p \neg val; \\ \}
```

int time_stamp = large; /* unique number */

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) \equiv
  ++time\_stamp;
  count = 0;
  for (k = 0; k \leq save\_ptr; k++) stamp(saved[k]);
  if (count \neq used) printf("We_lost_ltrack_lof_l%d_nodes!\n", used - count);
This code is used in section 3.
62. \langle \text{Basic subroutines } 13 \rangle + \equiv
  void stamp(p)
                          /* stamp all nodes of p with time\_stamp, and count them */
        node *p;
     if (\neg p) return;
     stamp(p \rightarrow l);
     stamp(p \rightarrow r);
     if (p \rightarrow val \equiv time\_stamp) \ printf("***Node\_overlap!!\n");
     p \rightarrow val = time\_stamp;
     count ++;
63. \langle \text{Global variables } 2 \rangle + \equiv
```

 $\S64$ TCALC INDEX 25

64. Index.

mems: 12, 13, 14, 16, 18, 28, 29, 30, 44, 47. align_to: $\underline{57}$, $\underline{59}$. avail: 12, 13, 14. n: 44, 47, 55. bad_node : $\underline{12}$, $\underline{13}$. $no_sweat: \underline{21}.$ btree: $\underline{47}$, 49. **node**: 1, <u>11</u>, 12, 13, 14, 15, 16, 17, 18, 19, 20, 21, buf: $\underline{2}$, 8, 9, 10. 25, 26, 27, 28, 44, 47, 54, 57, 60, 62. $buf_size: 2, 8, 9, 10.$ $node_struct: \underline{11}.$ $c: \ \underline{57}, \ \underline{59}.$ $nodes_per_block$: 13. calloc: 13. $normal_tree$: 44, 49. $cat: \underline{45}, 46, 47, 49.$ $normalize: \underline{25}, 39.$ change: <u>16,</u> 17, 20, 21, 23, 25, 26, 43. old_mems: $\underline{28}$, 30. $cheap_copy\colon \ \underline{18},\ 34,\ 35.$ op: $\underline{2}$, 3, 10, 35. check_stack: <u>35,</u> 37, 39, 40, 41, 42, 43. operand: <u>30,</u> 34, 35, 37, 39, 40, 41, 42, 43, 49. code: 53, 56, 57. p: 1, 13, 14, 15, 16, 17, 18, 19, 20, 21, 25, 26, $col: 57, \underline{58}, 59.$ <u>27</u>, <u>44</u>, <u>47</u>, <u>54</u>, <u>57</u>, <u>60</u>, <u>62</u>. $compare\colon \ \underline{19},\ 20,\ 21,\ 49.$ param: 2, 10, 32, 34, 35, 37, 49, 52. copy: 17, 18, 26, 27, 30. $pl: \ \underline{20}, \ \underline{21}, \ 22, \ 23.$ count: $\underline{53}$, 54, 59, 61, 62. $pp: \underline{27}.$ cur_node : $\underline{12}$, $\underline{13}$. $ppr: \underline{27}.$ $dump_stack: \ \ \underline{3}, \ 8, \ 34, \ 35, \ 43.$ $pr: \underline{20}.$ dwidth: <u>55</u>, 56, 57, 59. $print_char$: 57. easy: 21, $\underline{24}$. $print_digs: \underline{57}.$ $print_rep: \underline{57}, \underline{59}.$ eval: $\underline{60}$. exit: 13.printf: 3, 6, 8, 9, 10, 13, 30, 32, 34, 35, 41, 43, $ez_prod: 26, 27, 42.$ 52, 57, 59, 61, 62. *fgets*: 8, 9. $prl: \underline{20}.$ free_node: <u>14</u>, 15, 21, 23, 27, 41. $prod: \ \underline{27}, \ 42, \ 43.$ get_avail: 13, 17, 20, 21, 23, 40, 42, 43, 44, 47. $prr: \underline{20}.$ get_state : 53. putchar: 57. get_stats : $\underline{54}$, $\underline{59}$. q: 16, 17, 18, 19, 21, 25, 26, 27.getchar: 8. $ql: \ \underline{21}, \ 22, \ 23.$ *h*: 58. qq: 25, 26.height: 53, 56, 57, 59.qqr: $\underline{26}$. helps: 2, 4, 6, 7, 31, 33, 36, 38, 48, 51.quot: $\underline{46}$. inc_stack : 34, 35, 49. $r: \underline{11}.$ $j: \ \underline{54}, \ \underline{55}, \ \underline{57}.$ recycle: 15, 20, 21, 23, 26, 27, 30, 32, 34, 37, 41, 43. $rem: \underline{46}.$ jl: 54, 56, 57. $jr: \underline{54}, 56, \underline{57}.$ $return_p: \underline{42}, \underline{43}.$ $right: \ \ \underline{16},\ 17,\ 19,\ 20,\ 21,\ 23,\ 25,\ 26,\ 27,\ 41,\ 43.$ k: 1, 19, 44, 47, 55. l: 11, 57. $rv: \underline{60}.$ $s: \underline{21}.$ larg: $\underline{10}$. large: <u>10, 52, 53, 56, 57, 60, 63.</u> $save_ptr$: 28, 30, 32, 34, 61. $save_size$: 28, 30. lcode: 53, 56, 57.*left*: <u>16,</u> 17, 19, 20, 21, 25, 26, 27, 41, 43. saved: 28, 30, 32, 34, 59, 61. $lg_large: \underline{56}, 60.$ showing_mems: $\underline{28}$, 30, 32. loc: $\underline{2}$, 9, 10. showing_size: $\underline{28}$, $\underline{32}$, $\underline{59}$. $showing_usage$: 28, 30, 32. lv: 60.m: 18, 47. $ss: \underline{27}.$ $main: \underline{1}.$ stack: 28, 30. $stack_ptr: 28, 29, 30, 34, 35, 37, 42, 43.$ $max_display_size$: $\underline{50}$, 52, 54, 59. $stack_size$: 28, 34. $max_tree: \underline{50}, 52, 53.$ maxint: 56.stamp: $61, \underline{62}$.

26 INDEX TCALC $\S 64$

stdin: 8, 9.

 $succ: \ \underline{20}, \ 21, \ 23, \ 40.$

sum: 21, 23, 24, 25, 26, 27, 42.

threshold: 44, <u>50</u>, 52, 56. $time_stamp$: 61, 62, <u>63</u>.

tjl: $\underline{56}$.

 $used: \ \underline{12}, \ 13, \ 14, \ 30, \ 61.$ $val: \ \underline{11}, \ 54, \ 57, \ 60, \ 62.$ $width: \ \underline{53}, \ 56, \ 57, \ 59.$

TCALC NAMES OF THE SECTIONS 27

```
\langle \text{Add } right(p) \text{ to } right(q) \text{ and append this to } succ(pl) 23 \rangle Used in section 21.
⟨ Basic subroutines 13, 14, 15, 16, 17, 18, 19, 55, 60, 62 ⟩ Used in section 1.
\langle \text{Cases for one-character operators 6, 8, 32, 34, 35, 37, 39, 40, 41, 42, 43, 49, 52} \rangle 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 V 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.
\langle \text{ Display tree } saved[k] \text{ 59} \rangle Used in section 30.
\langle \text{Fill } buf \text{ with the user's next sequence of commands } 9 \rangle Used in section 3.
\langle Global \ variables \ 2, 12, 24, 28, 45, 50, 53, 58, 63 \rangle Used in section 1.
(Initialize the data structures 5, 46) Used in section 1.
\langle Prompt the user for a command and execute it 3 \rangle Used in section 1.
\langle \text{ Set } op \text{ and } param \text{ for the next operator } 10 \rangle Used in section 3.
(Subroutines 20, 21, 25, 26, 27, 44, 47, 54, 57) Used in section 1.
\langle \text{Swap } p \text{ and } q \text{ so that } p > q \text{ 22} \rangle Used in section 21.
\langle Type definitions 11\rangle Used in section 1.
```

TCALC

	Section	Page
Introduction	1	. 1
Input conventions	2	3
Data structures	11	5
Simple tree operations	17	7
Addition	21	9
Multiplication	26	11
Stack discipline	28	12
Generating binary trees	44	17
Displaying the results	50	19
Debugging	6 0	24
Index	64	25