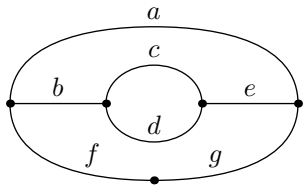
1. Introduction. This program generates all spanning trees of a given series-parallel graph, changing only one edge at a time, using an interesting algorithm.

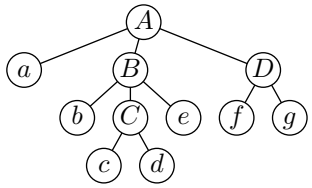
The given graph is specified using a simple right-Polish syntax

$$G \to \texttt{-} \mid G\,G\,\texttt{s} \mid G\,G\,\texttt{p}$$

so that, for example, the specifications `----ps-sp--sp` and `----p-ss--spp` both denote the graph



which can also be represented as a tree:



Branch nodes of the tree are either S nodes or P nodes, alternating from level to level.

As we do the computation, we count the total number of spanning trees that were generated and the total number of memory references that were needed.

```
#define o mems++
#define oo mems += 2
#define ooo mems += 3
#define oooo mems += 4
#define call oo /* let's say that a subroutine call costs two mems */
#define verbose (argc > 2) /* show the edges of each spanning tree */
#define extraverbose (argc > 3) /* show inner workings of the program */
#include <stdio.h>
  ⟨ Type definitions 6 ⟩
  ⟨ Global variables 3 ⟩
  unsigned int trees, mems;
  ⟨ Subroutines 9 ⟩
  main(int argc, char *argv[])
  {
    register int j, k;
    if (argc ≡ 1) {
      fprintf(stderr, "Usage: %s SPformula [[gory] details]\n", argv[0]); exit(0);
    }
    ⟨ Parse the formula argv[1] and set up the tree structure 2 ⟩;
    ⟨ Prepare the first spanning tree 14 ⟩;
    printf("(%u mems to get started)\n", mems); mems = 0;
    ⟨ Do the algorithm 29 ⟩;
    printf("Altogether %u spanning trees, %u additional mems.\n", trees, mems);
  }
```

2. Parsing and preparation. We begin by converting the Polish notation into a binary tree.

In the following code, we have scanned j binary operators and there are k items on the stack.

```
#define abort(mess)
        { fprintf(stderr, "Parsing␣error:␣%.*s|%s,␣%s!\n", p - argv[1], argv[1], p, mess); exit(-1); }
⟨ Parse the formula argv[1] and set up the tree structure 2 ⟩ ≡
  {
    register char *p = argv[1];
    for (j = k = 0; *p; p++)
      if (*p ≡ '-') ⟨ Create a new leaf 4 ⟩
      else if (*p ≡ 's' ∨ *p ≡ 'p') ⟨ Create a new branch 5 ⟩
      else abort("bad␣symbol");
    if (k ≠ 1) abort("disconnected␣graph");
    ⟨ Create the main tree 8 ⟩;
  }
```

This code is used in section 1.

3. `#define maxn 1000` `/* the maximum number of leaves; not checked */`

```
⟨ Global variables 3 ⟩ ≡
  int stack[maxn];      /* stack for parsing */
  int llink[maxn], rlink[maxn];      /* binary subtrees */
```

See also section 7.

This code is used in section 1.

4. Mems are not counted in this phase of the operation, because the program is essentially assumed to begin with the graph represented as a tree.

```
⟨ Create a new leaf 4 ⟩ ≡
  stack[k++] = 0;
```

This code is used in section 2.

5.

```
⟨ Create a new branch 5 ⟩ ≡
  {
    if (k < 2) abort("missing␣operand");
    rlink[++j] = stack[--k];
    llink[j] = stack[k - 1];
    stack[k - 1] = (*p ≡ 's' ? #100 : 0) + j;
  }
```

This code is used in section 2.

6. Now we convert the binary tree to the desired working tree, whose branch nodes appear in preorder.

```
⟨ Type definitions 6 ⟩ ≡
  typedef struct node_struct {
    int typ;      /* 1 for series nodes, otherwise 0 */
    struct node_struct *lchild;      /* leftmost child; Λ for a leaf */
    struct node_struct *rchild;      /* rightmost child; Λ for a leaf */
    struct node_struct *rsib;      /* right sibling; wraps around cyclically */
    ⟨ Additional fields of a node 13 ⟩
  } node;
```

This code is used in section 1.

7. The first half of *nodelist* contains up to *maxn* leaves; the other half contains up to *maxn* branches.

```
⟨ Global variables 3 ⟩ +≡
  node nodelist[maxn + maxn];      /* nodes of the tree */
  node *curleaf;      /* the leftmost not-yet-allocated leaf node */
  node *curnode;      /* the rightmost allocated branch node */
  node *root, *topnode;      /* root of the tree and its parent */
```

8. A recursive subroutine called *build* will govern the construction process.

```
#define isleaf(p) ((p) < nodelist + maxn)
⟨ Create the main tree 8 ⟩ ≡
  curleaf = nodelist;
  topnode = curnode = nodelist + maxn;
  curnode→typ = 2;      /* special typ code for the outer level */
  root = build(stack[0], curnode);
  root→rsib = root;      /* unnecessary but tidy */
```

This code is used in section 2.

9. When we *build* a leaf node, we simply allocate it. When we *build* a branch node, we link its children together via their sibling links.

Only one complication arises: We must prevent serial nodes from having serial children and parallel nodes from having parallel children. In such cases the child's family is merged with that of the parent, and the child goes away.

```
⟨ Subroutines 9 ⟩ ≡
  node *build(int stackitem, node *par)
  {
    register node *p, *l, *r, *lc, *rc;
    register int t, j;
    if (stackitem ≡ 0) return curleaf++;
    t = stackitem ≫ 8, j = stackitem & #ff;      /* type and location of a binary op */
    if (t ≠ par→typ) p = ++curnode, p→typ = t;
    else p = par;
    l = build(llink[j], p), lc = l→lchild, rc = l→rchild, r = build(rlink[j], p);
    if (l ≡ p) ⟨ Incorporate left child into node p 11 ⟩
    else if (r ≡ p) ⟨ Incorporate right child into node p 10 ⟩
    else p→lchild = l, p→rchild = r, l→rsib = r, r→rsib = l;
    return p;
  }
```

See also sections 15, 16, 17, 18, and 19.

This code is used in section 1.

10.

```
⟨ Incorporate right child into node p 10 ⟩ ≡
  r = p→lchild, p→lchild = l, l→rsib = r, p→rchild→rsib = l;
```

This code is used in section 9.

11.

```
⟨ Incorporate left child into node p 11 ⟩ ≡
  if (r ≡ p) ⟨ Incorporate both children into node p 12 ⟩
  else p→rchild = r, rc→rsib = r, r→rsib = lc;
```

This code is used in section 9.

12. ⟨ Incorporate both children into node p 12 ⟩ ≡

```
rc→rsib = p→lchild, p→lchild = lc, p→rchild→rsib = lc;
```

This code is used in section 11.

13. OK, the tree has been set up; our next goal is to decorate it. First let's take a closer look at the problem we're trying to solve.

Each node of the tree corresponds to a series-parallel graph between two vertices u and v, in a straightforward way: A leaf is a single edge $u \mathrel{—} v$. A nonleaf node p corresponds to a "superedge" formed from the edges or superedges $u_1 \mathrel{—} v_1, \ldots, u_k \mathrel{—} v_k$ of its $k \geq 2$ children. If p is a series node, its children are joined so that $v_j = u_{j+1}$ for $1 \leq j < k$; if p is a parallel node, its children are joined together so that $u_1 = \cdots = u_k$ and $v_1 = \cdots = v_k$. In both cases p is then considered to be a superedge between u_1 and v_k.

Let us say that a *near-spanning tree* of a series-parallel graph between u and v is a spanning forest that has exactly two components, where u and v lie in different components.

If p is a series superedge, its spanning trees are spanning trees of all its children; its near-spanning trees are obtained by designating some child, then constructing a near-spanning tree for that child and a spanning tree for each of the other children.

If p is a parallel superedge, the roles are reversed: Its near-spanning trees are near-spanning trees of all its children; its spanning trees are obtained by designating some child, then constructing a spanning tree for that child and a near-spanning tree for each of the other children.

We shall assign a Boolean value $p\to val$ to each leaf node p, specifying whether the corresponding edge is present or absent in the current spanning tree being considered. The $p\to val$ field of a branch node, similarly, will specify whether the corresponding superedge currently has a spanning tree or a near-spanning tree.

In the following algorithm every branch node p has a designated child, $p\to des$, with the property that $p\to val = p\to des\to val$.

Only certain combinations of values are legal; the legal ones, according to the discussion above, are characterized by two rules:

All non-designated children of a series node have value 1;

All non-designated children of a parallel node have value 0.

In other words, if q is the parent of node p,

$$p\to val = \begin{cases} q\to val, & \text{if } p = q\to des; \\ q\to typ, & \text{if } p \neq q\to des. \end{cases}$$

For any choice of the designated children, we obtain a unique spanning tree or near-spanning tree for node p by setting $p\to val$ to 1 or 0, respectively, and using this equation to propagate values down to the leaves.

Thus we can generate all the spanning trees of the graph (namely the spanning trees corresponding to the *root* node) by setting $root\to val = 1$ and considering all possible settings of designated children $p\to des$ throughout the tree.

However, many settings of the $p\to des$ pointers will produce the same result: The value of $p\to des$ is irrelevant for serial nodes of value 1 and for parallel nodes of value 0. We will return to this problem later; meanwhile let's put the necessary information into our data structure.

⟨ Additional fields of a **node** 13 ⟩ ≡

```
int val;      /* 0 = off, open, near-spanning; 1 = on, closed, spanning */
struct node_struct *des;      /* the designated child */
```

See also sections 22 and 25.

This code is used in section 6.

14. To start things off, we might as well designate each node's leftmost child.

Mems are computed under the assumption that a node's *typ* and *val* can be fetched and stored in a single operation.

```
⟨ Prepare the first spanning tree 14 ⟩ ≡
  o, topnode→typ = 1;
  call, init_tree(root, topnode);
  trees = 1;
  if (verbose) ⟨ Print the first tree 20 ⟩;
```

This code is used in section 1.

15. A few amendments to the data structure will be desirable later, but we're ready now to write most of the tree-initializing routine.

```
⟨ Subroutines 9 ⟩ +≡
  void init_tree(node *p, node *par)      /* par is the parent of p */
  {
    register node *q;
    ooo, p→val = (par→des ≡ p ? par→val : par→typ);
    if (isleaf(p)) ⟨ Further initialization of a leaf node 26 ⟩
    else {
      oo, p→des = p→lchild;
      for (q = p→lchild; ; q = q→rsib) {
        call, init_tree(q, p);
        if (o, q→rsib ≡ p→lchild) break;
      }
      ⟨ Further initialization of a branch node 27 ⟩;
    }
  }
```

16. Diagnostic routines. Several simple subroutines are used to print all or part of our data structure, as aids to debugging and/or when the user wants to examine all the spanning trees.

We name the leaves `a`, `b`, `c`, etc., and the branches `A`, `B`, `C`, etc., as in the example at the beginning of this program.

When I'm debugging this program I plan to save keystrokes and mental energy by typing, say, *xx*(`'A'`) when I want a pointer to node `A`.

```
#define leafname(p) ('a' + ((p) - nodelist))
#define branchname(p) ('A' + ((p) - root))
#define nodename(p) (isleaf(p) ? leafname(p) : branchname(p))
⟨ Subroutines 9 ⟩ +≡
  node *xx(char c)
  {
    if (c ≥ 'a') return nodelist + (c - 'a');
    return nodelist + maxn + (c - '@');
  }
```

17. ⟨ Subroutines 9 ⟩ +≡

```
  void printleaf(node *p)
  {
    printf("%c:%c␣rsib=%c\n", leafname(p), p→val + '0', nodename(p→rsib));
  }
  void printbranch(node *p)
  {
    printf("%c:%c␣rsib=%c␣lchild=%c␣des=%c␣rchild=%c", branchname(p), p→val + '0',
        nodename(p→rsib), nodename(p→lchild), nodename(p→des), nodename(p→rchild));
    ⟨ Print additional fields of a branch node 28 ⟩;
    printf("\n");
  }
  void printnode(node *p)
  {
    if (isleaf(p)) printleaf(p);
    else printbranch(p);
  }
```

18. ⟨ Subroutines 9 ⟩ +≡

```
  void printtree(node *p, int indent)
  {
    register node *q;
    register int k;
    for (k = 0; k < indent; k++) printf("␣");
    printnode(p);
    if (¬isleaf(p))
      for (q = p→lchild; ; q = q→rsib) {
        printtree(q, indent + 1);
        if (q→rsib ≡ p→lchild) break;
      }
  }
```

19. ⟨ Subroutines 9 ⟩ +≡

```
void printedges(node *p)      /* print the leaves whose value is 1 */
{
  register node *q;
  if (isleaf(p)) {
    if (p→val) printf("%c", leafname(p));
  } else for (q = p→lchild; ; q = q→rsib) {
      printedges(q);
      if (q→rsib ≡ p→lchild) break;
    }
}
```

20. ⟨ Print the first tree 20 ⟩ ≡

```
{
  if (extraverbose) printtree(root, 0);
  printf("The␣first␣spanning␣tree␣is␣");
  printedges(root);
  printf(".\n");
}
```

This code is used in section 14.

21. Overview of the algorithm. A branch node p will be called *easy* if $p\neg val = p\neg typ$. In such cases the designated child $p\neg des$ has no effect on the spanning tree or near-spanning tree, because all children have the same value.

Let's say for convenience that the *configs* of p are its spanning trees if $p\neg val = 1$, its near-spanning trees if $p\neg val = 0$. Our problem is to generate all configs of the root.

If p is easy, its configs are the Cartesian product of the configs of its children. But if p is uneasy, its configs are the union of such Cartesian products, taken over all possible choices of $p\neg des$.

Easy nodes are relatively rare: At most one child of an uneasy node (namely the designated child) can be easy, and all children of easy nodes are uneasy unless they are leaves.

```
#define easy(p) o,p→typ ≡ p→val
```

22. Cartesian products of configurations are easily generated in Gray-code order, using essentially a mixed-radix Gray code for n-tuples. (See Section 7.2.1.1 of *The Art of Computer Programming*.) In this program I'm using a "modular" code instead of a "reflected" one, because the modular code requires only *rsib* links to cycle through the possible choices of $p\neg des$.

Let's include a new field $p\neg leaf$ in each node, pointing to the leaf that lies at the end of the path from p to its designated descendants $p\neg des$, $p\neg des\neg des$, etc. All the *val* fields on this path are the same as $p\neg val$.

When $p\neg des$ is changed from one child to another, say from q to r, only two edge values of the overall spanning tree are affected. Namely, we have $q\neg typ \neq p\neg typ$ and $r\neg typ = p\neg typ$, so $q\neg leaf\neg val$ becomes $r\neg typ$ and $r\neg leaf\neg val$ becomes $q\neg typ$. Therefore such a change is pleasantly "Gray."

```
⟨ Additional fields of a node 13 ⟩ +≡
  struct node_struct *leaf;      /* the end of the designated path */
  struct node_struct *parent;    /* parent of this node */
```

23. These considerations lead us to the following algorithm to generate all spanning trees: Begin with all uneasy branch nodes active. Then repeatedly

1) Select the rightmost active node, p, in preorder.
2) Change $p\neg des$ to $p\neg des\neg rsib$, update all values of the tree, and visit the new spanning tree.
3) Activate all uneasy nodes to the right of p.
4) If $p\neg des$ has run through all children of p since p last became active, make node p passive.

A field $p\neg done$ is introduced in order to implement step (4): Node p becomes passive when $p\neg des = p\neg done$, and at such a time we reset $p\neg done$ to the previous value of $p\neg des$.

Actually $p\neg done$ is initially equal to $p\neg rchild$, and the *rchild* pointers are not needed by the main algorithm. So we can equate $p\neg done$ with $p\neg rchild$.

```
#define done rchild      /* the new meaning of the rchild field */
```

24. For example, let's apply the algorithm to the series-parallel graph illustrated in the introduction. Since A is a parallel node and since each leftmost child is initially designated, *init_tree* sets $A{\rightarrow}val = 1$, $B{\rightarrow}val = 0$, $C{\rightarrow}val = 1$, $D{\rightarrow}val = 0$, and the first spanning tree consists of edges $aceg$. All four branch nodes are initially uneasy. (That's just a coincidence, not a general rule.)

The current state of the algorithm can be indicated by writing each designated child as a subscript, by enclosing easy nodes in parentheses, and by placing a hat over passive nodes. With these conventions, the algorithm proceeds as follows:

branch node states				spanning tree
A_a	B_b	C_c	D_f	$aceg$
A_a	B_b	C_c	$\widehat{D}_g$	$acef$
A_a	B_b	$\widehat{C}_d$	D_g	$adef$
A_a	B_b	$\widehat{C}_d$	$\widehat{D}_f$	$adeg$
A_a	B_C	(C_d)	D_f	$abeg$
A_a	B_C	(C_d)	$\widehat{D}_g$	$abef$
A_a	$\widehat{B}_e$	C_d	D_g	$abdf$
A_a	$\widehat{B}_e$	C_d	$\widehat{D}_f$	$abdg$
A_a	$\widehat{B}_e$	$\widehat{C}_c$	D_f	$abcg$
A_a	$\widehat{B}_e$	$\widehat{C}_c$	$\widehat{D}_g$	$abcf$
A_B	(B_e)	C_c	D_g	$bcef$
A_B	(B_e)	C_c	$\widehat{D}_f$	$bceg$
A_B	(B_e)	$\widehat{C}_d$	D_f	$bdeg$
A_B	(B_e)	$\widehat{C}_d$	$\widehat{D}_g$	$bdef$
$\widehat{A}_D$	B_e	C_d	(D_g)	$bdfg$
$\widehat{A}_D$	B_e	$\widehat{C}_c$	(D_g)	$bcfg$
$\widehat{A}_D$	B_b	C_c	(D_g)	$cefg$
$\widehat{A}_D$	B_b	$\widehat{C}_d$	(D_g)	$defg$
$\widehat{A}_D$	$\widehat{B}_C$	(C_d)	(D_g)	$befg$

Thus, we first change $D{\rightarrow}des$ from f to g and passivate D; then we change $C{\rightarrow}des$ from c to d and passivate C. After four steps we change $B{\rightarrow}des$ from b to C, making C easy; and so on.

25. So-called "focus pointers" can be used to implement steps (1) and (3) very efficiently, as discussed in Algorithm 7.2.1.1L. We set $p{\rightarrow}focus = p$ except when p is an uneasy node such that the nearest uneasy node to its right is active. We also imagine that an artificial uneasy active node appears to the right of *curnode*, which is the rightmost branch node of the entire tree in preorder. Then the simple operations

$$p = r{\rightarrow}focus,\quad r{\rightarrow}focus = r$$

implement (1) and (3), when r is the rightmost uneasy node—in spite of the fact that step (2) changes some nodes from easy to uneasy and vice versa(!).

Furthermore, we can passivate node p in step (4) by the simple operations

$$p{\rightarrow}focus = l{\rightarrow}focus,\quad l{\rightarrow}focus = l$$

when l is the rightmost uneasy node to the left of p. We imagine that *topnode*, which lies to the left of everything in preorder, is always uneasy and active; therefore l always exists. Step (1) stops if $p = topnode$, since we have then generated all the spanning trees.

⟨ Additional fields of a **node** 13 ⟩ +≡

```
struct node_struct *focus;      /* the magical Gray-oriented focus pointer */
```

26. We can easily incorporate the new fields into our initialization routine. It will turn out that the algorithm doesn't really have to look at *leaf* or *parent* pointers, so no mems are charged for the cost of computing them.

⟨ Further initialization of a leaf node 26 ⟩ ≡
```
p→leaf = p, p→parent = par;
```
This code is used in section 15.

27. ⟨ Further initialization of a branch node 27 ⟩ ≡
```
p→leaf = p→des→leaf, p→parent = par;
o, p→focus = p;
```
This code is used in section 15.

28. ⟨ Print additional fields of a branch node 28 ⟩ ≡
```
printf("␣leaf=%c", leafname(p→leaf));
if (p→focus ≠ p) printf("␣focus=%c", branchname(p→focus));
```
This code is used in section 17.

29. Doing it. Let's go ahead now and implement the algorithm just sketched.

```
⟨ Do the algorithm 29 ⟩ ≡
  topnode→focus = topnode;
  while (1) {
    register node *p, *q, *l, *r;
    for (r = curnode; easy(r); r--) ;      /* find the rightmost uneasy node */
    oo, p = r→focus, r→focus = r;      /* steps (1) and (3) */
    if (p ≡ topnode) break;
    ⟨ Change p→des and visit a new spanning tree 31 ⟩;
    if (o, p→des ≡ p→done) ⟨ Passivate p 30 ⟩;
  }
```

This code is used in section 1.

30. All uneasy nodes to the right of p are now active, and l is the former $p\rightarrow des$.

```
⟨ Passivate p 30 ⟩ ≡
  {
    o, p→done = l;
    for (l = p − 1; easy(l); l--) ;      /* find the first uneasy node to the left */
    ooo, p→focus = l→focus, l→focus = l;
  }
```

This code is used in section 29.

31. If the user has asked for *verbose* output, we print only the edge that has entered the spanning tree and the edge that has left.

```
⟨ Change p→des and visit a new spanning tree 31 ⟩ ≡
  oo, l = p→des, r = l→rsib;
  o, k = p→val;      /* k = l→val ≠ r→val */
  for (q = l; ; o, q = q→des) {
    o, q→val = k ⊕ 1;
    if (isleaf(q)) break;
  }
  if (verbose) printf("␣%c%c", k ? '-' : '+', leafname(q));
  for (q = r; ; o, q = q→des) {
    o, q→val = k;
    if (isleaf(q)) break;
  }
  if (verbose) printf("%c%c\n", k ? '+' : '-', leafname(q));
  o, p→des = r, trees++;      /* "visiting" */
  for (q = p; q→des ≡ r; r = q, q = q→parent) q→leaf = r→leaf;
      /* that loop was optional, so it costs no mems */
  if (extraverbose) {
    printedges(root);
    printf(";␣now␣%c->leaf=%c\n", branchname(r), leafname(r→leaf));
  }
```

This code is used in section 29.

32. A loopless version. The algorithm implemented here contains four loops. Two of them skip over easy nodes when finding r and l in the list of branches; two of them go down from branches to leaves when changing the *val* fields.

The amortized cost of those loops is constant per new spanning tree. But it can be instructive to search for an algorithm that is entirely loopless, in the sense that the number of operations per new tree is bounded (once the algorithm has initialized itself in linear time).

Loopless algorithms tend to run slower than their loopy counterparts, especially in cases like the present where the additional overhead needed to avoid looping appears to be substantial. So the search for a loopless implementation is strictly academic. Yet it still was fascinating enough to keep me working on it for three days during my recent vacation.

I believe I see how to do it. But I don't have time to carry through the details, so I've decided just to sketch them here. Maybe somebody else will be inspired to work them out and to compare the loopless mem-counts with those of the present implementation.

The first two loops can be avoided by changing the tree dynamically, so that the designated child is always the leftmost. In such cases it's easy to see that no two easy nodes can be consecutive in preorder. My planned implementation swaps the rightmost child into the leftmost position when $p\rightarrow des$ is supposed to change. This swapping causes two adjacent substrings of the preordered node list to change places. The node list should be doubly linked; to do the swap, we need a new field $p\rightarrow scope$ that points to the rightmost branch that is descended from p in the current list.

The other two loops can be avoided if we update the *val* fields lazily, starting at the bottom. But then the pointer $p\rightarrow leaf$ becomes crucial, not optional, because the leaf nodes are encountered first, and because we need to know both $p\rightarrow leaf$ and $p\rightarrow rchild\rightarrow leaf$ when reporting the edges that enter and leave the spanning tree.

Of course the introduction of two required fields $p\rightarrow scope$ and $p\rightarrow leaf$ means that we must maintain them, and that seems to require additional loops that were not needed in the present implementation. Fortunately we don't have to update them instantly; they only have to be valid when p is the critical node in step (2) of the algorithm.

My solution is to introduce two additional fields for "registration." Consider a sequence of nodes p_1, p_2, $\ldots$, p_k where p_{j+1} is the rightmost child of p_j for $1 \le j < k$, and where p_1 and p_k are active but the others are either easy or passive. The easy ones among p_2, $\ldots$, p_{k-1} are not consecutive; the uneasy ones most recently went passive in order, from left to right. When $p = p_k$ is the critical node, we're going to rearrange the tree below p; and if p is then going to become passive, we have reached our last chance to update the scope link of p_1.

Node p can find p_1 using focus pointers, because p_1 is the rightmost active node to its left. But we need to verify that there really is a path from p_1 to p_k as described, because we mustn't screw up the *scope* links of random nodes. Let p be the critical node, and let q be the first active node to its left. Go up one or two levels from p via *parent* pointers until reaching an uneasy node, say u; but stop if this upward motion is not from the rightmost branch-child to a parent. Otherwise, if $q = u$, great; we update $q\rightarrow scope$ and we're done. Or if $q = u\rightarrow registry$, where *registry* is a new field to be discussed further, again we update $q\rightarrow scope$. Otherwise we conclude that q is not the top of the food chain to p.

When the critical node p becomes passive, after a case where $q\rightarrow scope$ has been updated, we set $p\rightarrow registry = q$, and $u\rightarrow registry = \Lambda$ in the case that $q = u\rightarrow registry$. This handshaking passes the required information down the tree, and doesn't leave spurious non-null *registry* values that could lead to false diagnoses.

A similar method works to maintain the *leaf* pointers, which are similar but based on leftmost instead of rightmost children. Instead of $p\rightarrow registry$, I should have spoken of $p\rightarrow scope\_registry$ and $p\rightarrow leaf\_registry$.

(Whew.)

33. Index.

SPSPAN