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

1. Intro. This program contains two implementations of the Koda–Ruskey algorithm for generating all ideals of a given forest poset [*Journal of Algorithms* **15** (1993), 324–340]. The common goal of both implementations is, in essence, to generate all binary strings $b_0 \ldots b_{n-1}$ in which certain bits are required to be less than or equal to specified bits that lie to their right. (For some values of j there is a value of $k > j$ such that we don't allow b_k to be 0 when $b_j = 1$.) Moreover, each binary string should differ from its predecessor in exactly one bit position; the algorithm therefore defines a generalized reflected Gray code.

The given forest is represented by n pairs of nested parentheses. For example, `()()()()()` represents five independent bits, while `((((()))))` represents five bits with $b_0 \le b_1 \le b_2 \le b_3 \le b_4$. A more interesting example, `((())(()()))`, represents six bits subject to the conditions $b_0 \le b_1$, $b_1 \le b_5$, $b_2 \le b_4$, $b_3 \le b_4$, $b_4 \le b_5$. Each pair of parentheses corresponds to a bit that must not exceed the bit of its enclosing pair, if any, and the pairs are ordered by the appearances of their right parentheses.

The first implementation uses n coroutines, which call each other in a hierarchical fashion. The second uses multilinked data structures in a loopless way, so that each generation step performs a bounded number of operations to obtain the next element. I couldn't resist writing this program, because both implementations turn out to be quite interesting and instructive.

Indeed, I think it's a worthwhile challenge for people who study the science of computer programming to verify that these two implementations both define the same sequence of bitstrings. Even more challenging would be to derive the second implementation "automatically" from the first.

```
#include <stdio.h>
  ⟨Type definitions 4⟩
  ⟨Global variables 3⟩
  int main(int argc, char *argv[])
  {
    register int j, k, l;
    ⟨Process the command line, parsing the given forest 2⟩;
    printf("Bitstrings␣generated␣from␣\"%s\":\n", argv[1]);
    ⟨Generate the strings with a coroutine implementation 9⟩;
    printf("\nTrying␣again,␣looplessly:\n");
    ⟨Generate the strings with a loopless implementation 15⟩;
    return 0;
  }
```

2. In this step we parse the forest into an array of “scopes”: *scope*[j] is the index of the smallest descendant of node j, including node j itself.

```
#define abort(m, i)
          { fprintf(stderr, m, argv[i]); return −1; }
#define stacksize 100    /* max levels in the forest */
#define forestsize 100    /* max nodes in the forest */

⟨ Process the command line, parsing the given forest 2 ⟩ ≡
  if (argc ≠ 2 ∨ argv[1][0] ≠ '(') abort("Usage:␣%s␣\"nestedparens\"\n", 0);
  for (j = k = l = 0; argv[1][k]; k++)
    if (argv[1][k] ≡ '(') {
      stack[l++] = j;
      if (l ≡ stacksize) abort("Stack␣overflow␣---␣\"%s\"␣is␣too␣deep␣for␣me!\n", 1);
    } else if (argv[1][k] ≡ ')') {
      if (−−l < 0) abort("Extra␣right␣parenthesis␣in␣\"%s\"!\n", 1);
      scope[j++] = stack[l];
      if (j ≡ forestsize) abort("Memory␣overflow␣---␣\"%s\"␣is␣too␣big!\n", 1);
    } else abort("The␣forest␣spec␣\"%s\"␣should␣contain␣only␣parentheses!\n", 1);
  if (l) abort("Missing␣right␣parenthesis␣in␣\"%s\"!\n", 1);
  nn = j;
```

This code is used in section 1.

3.
```
⟨ Global variables 3 ⟩ ≡
  int stack[stacksize];    /* nodes preceding each open leftparen, while parsing */
  int scope[forestsize];    /* table that exhibits each rightparen's influence */
  int nn;    /* the actual number of nodes in the forest */
```

See also sections 11 and 17.

This code is used in section 1.

4. The coroutine implementation. Our first implementation uses a system of n cooperating programs, each of which represents a node in the forest. For convenience we will call the associated record a "cnode." If p points to a cnode, $p\to child$ points to the cnode representing its rightmost child, and $p\to sib$ points to the cnode representing its nearest sibling on the left, in the given forest.

Each cnode corresponds to a coroutine whose job is to generate all the ideals of the subforest it represents. Whenever the coroutine is invoked, it either changes one of the bits in its scope and returns *true*, or it changes nothing and returns *false*. Initially all the bits are 0; when it first returns *false*, it will have generated all legitimate bit patterns, ending with some nonzero pattern. Subsequently it will generate the patterns again in reverse order, ending with all 0s, after which it will return *false* a second time. Invoking it again and again will repeat the same process, going forwards and backwards, ad infinitum.

Each coroutine has the same basic structure, which can be described as follows in an ad hoc extension of C language:

```
coroutine p()
  {
    while (1) {
      p->bit = 1; return true;
       while (p->child()) return true;
      return p->sib();
       while (p->child()) return true;
      p->bit = 0; return true;
      return p->sib();
    }
  }
```

If either $p\to child$ or $p\to sib$ is Λ, the corresponding coroutine $\Lambda(\,)$ is assumed to simply return *false*.

Suppose $p\to child(\,)$ first returns *false* after it has been called r times; thus $p\to child(\,)$ generates r different patterns, including the initial pattern of all 0s. Similarly, suppose that $p\to sib(\,)$ generates l different patterns before first returning *false*. Then the coroutine $p(\,)$ itself will generate $l(r+1)$ patterns in between the times when it returns *false*. The final bit pattern for p will be the final bit pattern for $p\to sib$, together with either $p\to bit = 1$ and the final bit pattern of $p\to child$ (if l is odd) or with $p\to bit = 0$ and all 0s in $p\to child$ (if l is even).

```
⟨ Type definitions 4 ⟩ ≡
  typedef enum {
    false, true
  } boolean;

  typedef struct cnode_struct {
    char bit;    /* either 0 or 1; always 1 when a child's bit is set */
    char state;    /* the current place in this cnode's coroutine */
    struct cnode_struct *child;    /* rightmost child in the given forest */
    struct cnode_struct *sib;    /* nearest left sibling in the given forest */
    struct cnode_struct *caller;    /* which coroutine invoked this one */
  } cnode;
```

See also section 14.

This code is used in section 1.

5. When coroutine p calls coroutine q, it sets $p\to state$ to an appropriate number and also sets $q\to caller = p$. Then control passes to q at the place determined by $q\to state$.

When coroutine q wants to return a boolean value, it sets *coresult* to this value; then it passes control to $p = q\to caller$ at the place determined by $p\to state$.

This program simulates coroutine linkage with a big switch statement. Actually the notion of "passing control" really means that we simply assign a value to the variable *cur_cnode*.

The value of $q\to caller$ for every cnode q is completely determined by the structure of the given forest, so we could set it once and for all during the initialization instead of setting it dynamically as done here. But what the heck.

```
#define cocall(q, s)
        { cur_cnode→state = s;
          if (q) q→caller = cur_cnode, cur_cnode = q;
          else coresult = false;
          goto cogo; }
#define bitchange(b, s)
        { cur_cnode→bit = b, coresult = true; coreturn(s); }
#define coreturn(s)
        { cur_cnode→state = s, cur_cnode = cur_cnode→caller;
          goto cogo; }
```

⟨ Repeatedly switch to the proper part of the current coroutine 5 ⟩ ≡

```
cogo: switch (cur_cnode→state) {
    ⟨ Cases for coroutine states 6 ⟩;
  default: abort("%s:␣Unknown␣state␣code␣(this␣can't␣happen)!\n", 0);
  }
```

This code is used in section 9.

6. In its initial state 0, a coroutine turns its bit on, returns *true*, and enters state 1.

⟨ Cases for coroutine states 6 ⟩ ≡

```
case 0: bitchange(1, 1);
```

See also sections 7, 8, and 12.

This code is used in section 5.

7. The purpose of state 1 is to run through all bit patterns of the current node's children, starting with all 0s and ending when they reach their final pattern. At that point we invoke the current node's nearest left sibling and enter state 3. An intermediate state 2 is defined for the purpose of examining the result after calling the child coroutine.

The purpose of state 3 is simply to return to whoever called us, passing along the information in *coresult*, which tells whether any of our left siblings has changed one of its bits. Then we will continue in state 4.

⟨ Cases for coroutine states 6 ⟩ +≡

```
case 1: cocall(cur_cnode→child, 2);
case 2: if (coresult) coreturn(1);
  cocall(cur_cnode→sib, 3);
case 3: coreturn(4);
```

8. State 4 is rather like state 1, except that the child coroutine is now running through its bit patterns in reverse order. Finally it reduces them all to 0s, and returns *false* the next time we attempt to invoke it. At that point we reset the current bit, return *true*, and enter state 6.

State 6 invokes the sibling coroutine, leading to state 7. And state 7 is like state 3, but it takes us back to state 0 instead of state 4.

⟨Cases for coroutine states 6⟩ +≡
case 4: *cocall*(*cur_cnode*→*child*, 5);
case 5: **if** (*coresult*) *coreturn*(4);
  bitchange(0, 6);
case 6: *cocall*(*cur_cnode*→*sib*, 7);
case 7: *coreturn*(0);

9. Hey, the implementation is done already, except that we have to get it started and write the code that controls it at the outermost level.

⟨Generate the strings with a coroutine implementation 9⟩ ≡
  {
    register cnode ∗*cur_cnode*;
    ⟨Initialize the cnode structure 10⟩;
    ⟨Repeatedly switch to the proper part of the current coroutine 5⟩;
  }

This code is used in section 1.

10. We allocate a special cnode to represent the external world outside of the given forest.

#**define** *root_cnode* *cnode_table*[*nn*].*child*

⟨Initialize the cnode structure 10⟩ ≡
  scope[*nn*] = 0;
  for ($k = 0$; $k \le nn$; k++)
    if (*scope*[k] < k) {
      cnode_table[k].*child* = *cnode_table* + $k - 1$;
      for ($j = k - 1$; *scope*[j] > *scope*[k]; j = *scope*[j] − 1) *cnode_table*[j].*sib* = *cnode_table* + *scope*[j] − 1;
    }
  cur_cnode = *cnode_table* + *nn*;
  goto *upward_step*;

This code is used in section 9.

11. ⟨Global variables 3⟩ +≡
  cnode *cnode_table*[*forestsize* + 1];   /∗ the cnodes ∗/
  boolean *coresult*;   /∗ value returned by a coroutine ∗/

12. States 8 and greater are reserved for the external (outermost) level, which simply invokes the coroutine for the entire forest and prints out the results, until the bit patterns have been generated in both the forward and reverse directions.

```
⟨ Cases for coroutine states 6 ⟩ +≡
case 8: if (coresult) {
  upward_step: ⟨ Print out all the current cnode bits 13 ⟩;
    cocall(root_cnode, 8);
  }
  printf("...␣and␣now␣we␣generate␣them␣in␣reverse:\n");
  goto downward_step;
case 9: if (coresult) {
  downward_step: ⟨ Print out all the current cnode bits 13 ⟩;
    cocall(root_cnode, 9);
  }
  break;
```

13. ⟨ Print out all the current cnode bits 13 ⟩ ≡

```
  for (k = 0; k < nn; k++) putchar('0' + cnode_table[k].bit);
  putchar('\n');
```

This code is used in section 12.

14. The loopless implementation. Our coroutine implementation solves the generation problem in a nice and natural fashion, but it can be inefficient if the given forest has numerous nodes of degree one. For example, a one-tree forest like `((...()...))` with n pairs of parentheses will need approximately $\binom{n}{2}$ coroutine invocations to generate $n+1$ bitstrings.

Our second implementation reduces the work in such cases to $O(n)$; in fact, it needs only a bounded number of operations to generate each bitstring after the first. It does, however, need a slightly more complex data structure with four link fields.

The basic idea is to work with a dynamically varying list of nodes called the current *fringe* of the forest. The fringe consists of all node whose bit is 1, together with their children. We maintain it as a doubly linked list, so that $p\to left$ and $p\to right$ are the neighbors of p on the left and right. A special node *head* is provided to make the list circular; thus $head\to right$ and $head\to left$ are the leftmost and rightmost fringe nodes.

A fringe node is said to be said to be either *active* or *passive*. Every node is active when it joins the fringe, but it becomes passive for at least a short time when its bit changes value; at such times the node is essentially shifting direction between going forward or backward, as in the coroutine implementation. (A passive node corresponds roughly to a coroutine that is asking its siblings to make the next move.) We save time jumping across such call-chains by using a special link field called the *focus*: If p is a passive fringe node whose righthand neighbor $p\to right$ is active, $p\to focus$ is the rightmost active node to the left of p in the fringe; otherwise $p\to focus = p$. (The special *head* node is always considered to be active, for purposes of this definition, but it is not strictly speaking a member of the fringe.)

The loopless implementation works with records called lnodes, just as the coroutine implementation worked with cnodes. Besides the dynamic *bit* and *left* and *right* and *focus* fields already mentioned, each lnode also has a static field called *lchild*, representing its leftmost child. (There is no need for an *rchild* field, since $p\to rchild = p - 1$ when $p\to lchild \neq \Lambda$.)

If p is not in the fringe, $p\to focus$ should equal p. Also, $p\to left$ and $p\to right$ are assumed to equal the nearest siblings of p to the left and right, respectively, if such siblings exist; otherwise $p\to left$ and/or $p\to right$ are undefined.

⟨ Type definitions 4 ⟩ +≡

```
typedef struct lnode_struct {
  char bit;   /* either 0 or 1; always 1 when a child's bit is set */
  struct lnode_struct *left, *right;   /* neighbors in the forest and/or fringe */
  struct lnode_struct *lchild;   /* leftmost child */
  struct lnode_struct *focus;   /* red-tape cutter for efficiency */
} lnode;
```

15. Here now is the basic outline of the loopless implementation:

⟨ Generate the strings with a loopless implementation 15 ⟩ ≡

```
{
  register lnode *p, *q, *r;
  ⟨ Initialize the lnode structure, putting all roots into the fringe 16 ⟩;
  while (1) {
    ⟨ Print out all the current lnode bits 22 ⟩;
    ⟨ Set p to the rightmost active node of the fringe, and activate everything to its right 18 ⟩;
    if (p ≠ head) {
      if (p→bit ≡ 0) {
        p→bit = 1;   /* moving forward */
        ⟨ Insert the children of p after p in the fringe 19 ⟩;
      } else {
        p→bit = 0;   /* moving backward */
        ⟨ Delete the children of p from the fringe 20 ⟩;
      }
    } else if (been_there_and_done_that) break;
    else {
      printf("...␣and␣now␣we␣generate␣them␣in␣reverse:\n");
      been_there_and_done_that = true; continue;
    }
    ⟨ Make node p passive 21 ⟩;
  }
}
```

This code is used in section 1.

16. Initialization of the lnodes is similar to initialization of the cnodes, but more links need to be set up.

```
#define head (lnode_table + nn)
```

⟨ Initialize the lnode structure, putting all roots into the fringe 16 ⟩ ≡

```
for (k = 0; k ≤ nn; k++) {
  lnode_table[k].focus = lnode_table + k;
  if (scope[k] < k) {
    for (j = k − 1; scope[j] > scope[k]; j = scope[j] − 1) {
      lnode_table[j].left = lnode_table + scope[j] − 1;
      lnode_table[scope[j] − 1].right = lnode_table + j;
    }
    lnode_table[k].lchild = lnode_table + j;
  }
}
head→left = head − 1, (head − 1)→right = head;
head→right = head→lchild, head→lchild→left = head;
```

This code is used in section 15.

17. ⟨ Global variables 3 ⟩ +≡

```
lnode lnode_table[forestsize + 1];   /* the lnodes */
boolean been_there_and_done_that;
```

18. ⟨ Set p to the rightmost active node of the fringe, and activate everything to its right 18 ⟩ ≡

```
q = head→left;
p = q→focus;
q→focus = q;
```

This code is used in section 15.

19. ⟨ Insert the children of p after p in the fringe 19 ⟩ ≡

```
if (p→lchild) {
  q = p→right;
  q→left = p − 1, (p − 1)→right = q;
  p→right = p→lchild, p→lchild→left = p;
}
```

This code is used in section 15.

20. ⟨ Delete the children of p from the fringe 20 ⟩ ≡

```
if (p→lchild) {
  q = (p − 1)→right;
  p→right = q, q→left = p;
}
```

This code is used in section 15.

21. At this point we know that p→*right* is active.

⟨ Make node p passive 21 ⟩ ≡

```
p→focus = p→left→focus;
p→left→focus = p→left;
```

This code is used in section 15.

22. ⟨ Print out all the current lnode bits 22 ⟩ ≡

```
for (k = 0; k < nn; k++) putchar('0' + lnode_table[k].bit);
putchar('\n');
```

This code is used in section 15.

23. I used the following code when debugging.

```
#define rel(f) (lnode_table[k].f ? lnode_table[k].f − lnode_table : −1)
```

⟨ Print out the whole lnode structure 23 ⟩ ≡

```
for (k = 0; k ≤ nn; k++) {
  printf("lnode␣%d:␣bit=%d,␣", k, lnode_table[k].bit);
  printf("focus=%d,␣left=%d,␣right=%d,␣lchild=%d\n", rel(focus), rel(left), rel(right), rel(lchild));
}
```

24. Index.

KODA-RUSKEY