**1. Intro.** This program implements the Koda-Ruskey algorithm for generating all ideals of a given forest poset [*Journal of Algorithms* **15** (1993), 324–340].

It was inspired by two implementations of this algorithm by D. E. Knuth (available at `http://www-cs-staff.Stanford.EDU/~knuth/programs.html#Gray`). This implementation demonstrates the adequacy of a functional programming language for this particular algorithm. Indeed, it has a very nice and concise implementation using higher-order functions. (The algorithm itself, given in section 8, is only 9 lines long).

**2.** Trees and forests. In a given forest, the $n$ nodes are numbered from 0 to $n-1$ according to a postfix traversal.

```
type tree = Node of int × forest
and forest = tree list
```

**3.** From/to nested parentheses. There is a one-to-one mapping between forests and nested parentheses, where each pair of parentheses is the child of the immediatly enclosing pair of parentheses.

```
let from_paren s =
  let rec parse stack i = parser
    | [⟨ '(' ; st ⟩] →
        (* we push a new forest on top of stack *)
        parse ([] :: stack) i st
    | [⟨ ')' ; st ⟩] →
        (* we make a node with the forest f1 on top of stack *)
        (match stack with
           | f1 :: f2 :: s →
               let t = Node (i, List.rev f1) in
               parse ((t :: f2) :: s) (i + 1) st
           | _ → failwith "unbalanced␣parenthesis")
    | [⟨ ⟩] →
        (* end of input: there must be exactly one forest on top of stack *)
        (match stack with
           | [f] → List.rev f
           | _ → failwith "unclosed␣parenthesis")
  in
  parse [[]] 0 (Stream.of_string s)

open Printf
```

```
let rec print_tree (Node (_, f)) =
   "(" ^ print_forest f ^ ")"
and print_forest = function
   | [] → ""
   | t :: f → print_tree t ^ print_forest f
```

**4.** Nice rendering with dot. We append all the ideals in a single `dot` file `"tree.dot"`. Bits are given as an array *bits* of zeros and ones, and nodes are displayed in white or grey accordingly. (Each ideal goes on a separate page; thus stepping from page to page gives a nice animation.)

```
let cout = open_out "tree.dot"

let to_dot f bits =
  fprintf cout "digraph g {\n";
  let rec print_tree (Node (i, f)) =
    let color = if bits.(i) ≡ 0 then "" else "[style=filled]" in
    fprintf cout "b%d %s;\n" i color;
    List.iter (fun (Node (j, _)) → fprintf cout "b%d -> b%d;\n" i j) f;
    print_forest f
  and print_forest f =
    List.iter print_tree f
  in
  print_forest f;
  fprintf cout "}\n"
```

**5.** Size.

```
let rec tree_size (Node (_, f)) = succ (forest_size f)

and forest_size f = List.fold_left (fun s t → s + tree_size t) 0 f
```

**6.** Command line options.

```
let display = ref true
let dot = ref false
```

**7. Purely functional implementation.**

```
type state = int array × string

type computation = state → state

let create f = (Array.create (forest_size f) 0, "")

let update i v (t, d) = let t' = Array.copy t in t'.(i) ← v; (t', d)

let get i (t, _) = t.(i)

let msg m ((t, d) as s) = if !display then (t, d ^ m ^ "\n") else s

let print (t, d) =
  (t, (Array.fold_left (fun d b → d ^ string_of_int b) d t) ^ "\n")
```

```
let (++) f g s = g (f s)

let ideals0 f =
  let rec loop_f k f s = match f with
    | [] → k s
    | t :: f → loop_t (loop_f k f) t s
  and loop_t k (Node (i, f)) s =
    if get i s ≡ 0 then
      (k ++ update i 1 ++ loop_f k f) s
    else
      (loop_f k f ++ update i 0 ++ k) s
  in
  let s = create f in
  let s =
    msg (sprintf "Bitstrings␣generated␣from␣\"%s\":" (print_forest f)) s in
  let s = loop_f print f s in
  let s = msg "...␣and␣now␣we␣generate␣them␣in␣reverse:" s in
  let s = loop_f print f s in
  if !display then print_string (snd s)
```

**8. Koda-Ruskey algorithm.** The function *ideals* generates all ideals of a given forest, starting from $00\cdots0$ (all zeros). Then it generates the patterns again, in reverse order.

The code works as follows: *loop_f* $k$ $f$ generates all the ideals of a given forest $f$ and *loop_t* $k$ $t$ all the ideals of a given tree $t$. In both case, the function $k$ is to be called in between the computation steps. Both functions *loop_f* and *loop_t* can generate the patterns in both ways, depending on the state of the bits when called. The main call is on *loop_f*, with an "empty" function $k$.

If for instance *loop_f* is called on a forest $[t_1; t_2]$, it will call itself recursively on $[t_2]$ with a continuation $k$ which calls *loop_t* on $t_1$. Therefore, all patterns will be generated for $t_1$ each time a progress is made in the generation of patterns for $t_2$.

*loop_t* tests the bit of the node to determine in which way patterns have to be generated. If zero, we know that all bits below are zeros; thus we call $k$, we set the bit to one and we call *loop_f* on the children with the same continuation $k$. If one, we know that the patterns have to be generated in reverse order; thus we do the converse: we call *loop_f* on the children, resulting with all children being set to zero, we set the node to zero and we call $k$.

```
let ideals f =
  let n = forest_size f in
  let bits = Array.create n 0 in
  let dump () = for i = 0 to n - 1 do printf "%d" bits.(i) done; printf "\n" in
  let rec loop_f k = function
    | [] → k ()
    | t :: f → loop_t (fun () → loop_f k f) t
  and loop_t k (Node (i, f)) =
    if bits.(i) ≡ 0 then begin
      k (); bits.(i) ← 1; loop_f k f
    end else begin
      loop_f k f; bits.(i) ← 0; k ()
    end
  in
  let k0 () = if !display then dump (); if !dot then to_dot f bits in
  if !display then
    printf "Bitstrings␣generated␣from␣\"%s\":\n" (print_forest f);
  loop_f k0 f;
  if !display then printf "...␣and␣now␣we␣generate␣them␣in␣reverse:\n";
  loop_f k0 f
```

**9. A better implementation.** The above code is nice but is actually building the same closures many times. Indeed, we build the same closure each time we traverse the same tree or the same forest. The following code factorizes this closure construction.

Functions *loop_f* and *loop_t* now return a function, of type *unit*→ *unit*, which generates the patterns. The main call consists in calling *loop_f* to get a function *loop* and then to call this function (first to get the patterns and then a second time to get the patterns in reverse order).

Note how the recursive calls to *loop_f* in *loop_t* have been factorized, out of the closure fun () → .... Therefore, the partial evaluation (*loop_t k t*) in *loop_f* is really performing computations.

This code is roughly 33% faster than the previous one.

```
let ideals2 f =
  let n = forest_size f in
  let bits = Array.create n 0 in
  let dump () = for i = 0 to n − 1 do printf "%d" bits.(i) done; printf "\n" in
  let rec loop_f k = function
    | [] → k
    | t :: f → loop_t (loop_f k f) t
  and loop_t k (Node (i, f)) =
    let lf = loop_f k f in
    fun () →
      if bits.(i) ≡ 0 then begin
        k (); bits.(i) ← 1; lf ()
      end else begin
        lf (); bits.(i) ← 0; k ()
      end
  in
  let k0 () = if !display then dump (); if !dot then to_dot f bits in
  let loop = loop_f k0 f in
  if !display then
    printf "Bitstrings␣generated␣from␣\"%s\":\n" (print_forest f);
  loop ();
  if !display then printf "...␣and␣now␣we␣generate␣them␣in␣reverse:\n";
  loop ()
```

**10. A defunctionalized implementation.** The above code is nice but makes many calls to unknown functions, which incurs a high speed penalty on modern processors. Instead, we represent closures using an explicit datatype.

Defunctionalization is originally due to Reynolds. It has been recently studied again by Danvy and employed by Cejtin, Jagannathan and Weeks in the MLton compiler.

This approach seems roughly 25% faster than the previous one, when compiled with `ocamlopt`.

```
type continuation =
  | Continue of int × continuation × continuation (* values for i, k and k′ *)
  | Init

let ideals3 f =
  let n = forest_size f in
  let bits = Array.create n 0 in
  let dump () = for i = 0 to n − 1 do printf "%d" bits.(i) done; printf "\n" in
  let rec loop_f k = function
    | [] → k
    | t :: f → loop_t (loop_f k f) t
  and loop_t k (Node (i, f)) =
    Continue (i, k, loop_f k f)
  in
  let rec exec = function
    | Continue (i, k, k′) →
        if bits.(i) ≡ 0 then begin
          exec k; bits.(i) ← 1; exec k′
        end else begin
          exec k′; bits.(i) ← 0; exec k
        end
    | Init →
        if !display then dump (); if !dot then to_dot f bits
  in
  let loop = loop_f Init f in
  if !display then
    printf "Bitstrings␣generated␣from␣\"%s\":\n" (print_forest f);
  exec loop;
  if !display then printf "...␣and␣now␣we␣generate␣them␣in␣reverse:\n";
  exec loop
```

**11. An "even more defunctionalized" implementation.** It is easy to see that the execution of *loop_f* above does nothing but create exactly one *Continue* node per node in the forest. So, the work performed during this phase essentially amounts to associating two nodes, namely the nodes associated with the continuations $k$ and $k'$, to every node. Then, why not represent this association using two integer arrays? This is what we do below. The arrays *ak* and $ak'$ associate a continuation to every node. The special value $-1$ is used to represent the initial continuation.

This approach seems roughly 3% faster than the previous one, when compiled with `ocamlopt -unsafe`.

```
let ideals4 f =
  let n = forest_size f in
  let bits = Array.create n 0 in
  let ak = Array.create n (−1)
  and ak′ = Array.create n (−1) in
  let dump () = for i = 0 to n − 1 do printf "%d" bits.(i) done; printf "\n" in
  let rec loop_f k = function
    | [] → k
    | t :: f → loop_t (loop_f k f) t
  and loop_t k (Node (i, f)) =
    ak.(i) ← k;
    ak′.(i) ← loop_f k f;
    i
  in
  let rec exec = function
    | (−1) →
        if !display then dump (); if !dot then to_dot f bits
    | i →
        if bits.(i) ≡ 0 then begin
          exec ak.(i); bits.(i) ← 1; exec ak′.(i)
        end else begin
          exec ak′.(i); bits.(i) ← 0; exec ak.(i)
        end
  in
  let loop = loop_f (−1) f in
  if !display then
    printf "Bitstrings␣generated␣from␣\"%s\":\n" (print_forest f);
  exec loop;
  if !display then printf "...␣and␣now␣we␣generate␣them␣in␣reverse:\n";
  exec loop
```

**12. Count.** The following functions *count_f* and *count_t* compute the number of patterns, respectively for forests and trees. We use 64-bits integers to avoid overflows.

```
let rec count_f f =
  List.fold_left (fun n t → Int64.mul n (count_t t)) Int64.one f
and count_t (Node (_,f)) =
  Int64.succ (count_f f)
```

**13. Main program.** The forest is given on the command line. Options may also be passed to first print the number of patterns (`-count`), to select a `dot` rendering (`-dot`), to suppress the printings on standard output (`-silent`) and/or to select the desired implementation (`-algo1` and `-algo2` respectively).

```
let forest = ref []
let algo0 = ref false
let algo1 = ref false
let algo2 = ref false
let algo3 = ref false
let algo4 = ref false
let count = ref false

let _ =
  Arg.parse
    [ "-dot", Arg.Set dot, "display␣patterns␣using␣dot";
      "-silent", Arg.Clear display, "do␣not␣display␣patterns␣on␣stdout";
      "-count", Arg.Set count, "print␣number␣of␣patterns";
      "-algo0", Arg.Set algo0, "use␣algo␣0";
      "-algo1", Arg.Set algo1, "use␣algo␣1";
      "-algo2", Arg.Set algo2, "use␣algo␣2";
      "-algo3", Arg.Set algo3, "use␣algo␣3";
      "-algo4", Arg.Set algo4, "use␣algo␣4" ]
    (fun s → forest := from_paren s)
    "usage:␣koda-ruskey␣[options]␣\"nested␣parentheses\""

let _ =
  if !count then begin
    printf "nb␣patterns␣=␣%s\n" (Int64.format "%d" (count_f !forest));
    flush stdout
  end;
  if !algo0 then ideals0 !forest;
  if !algo1 then ideals !forest;
  if !algo2 then ideals2 !forest;
  if !algo3 then ideals3 !forest;
```

```
if !algo4 then ideals4 !forest;
flush stdout;
if !dot then begin
  if ¬ !algo1 ∧ ¬ !algo2 ∧ ¬ !algo3 ∧ ¬ !algo4 then
    to_dot !forest (Array.create (forest_size !forest) 0);
  close_out cout;
  ignore (Sys.command "dot␣-Tps␣-o␣tree.ps␣tree.dot");
  ignore (Sys.command "gv␣-media␣automatic␣tree.ps")
end
```