Interface for module Dlx

This module implements Knuth's dancing links algorithm, also known as DLX (see *http://en.wikipedia.org/wiki/Dancing_Links*).

This is a backtracking algorithm to find all solutions of the exact cover problem, which is stated as follows: given a matrix of 0s and 1s, does it have a set of rows containing exactly one 1 in each column?

The first step is to build the matrix's columns, using the following abstract data type *column*. Each column is given a name, to be identified within solutions.

type *column*

val *create_column* : *string* → *column*

The next step is to fill the matrix, row by row. This is done using the following function *add_row* which inserts a new row in the matrix. A row is defined as the list of its columns containing a 1.

Note that row insertion is performed as a side effect on columns. A column is indeed a mutable data structure, and thus must not be shared across several exact cover problems.

val *add_row* : *column list* → *unit*

Finally, an exact cover problem is defined as the list of the matrix's columns. Note that this implementation deals with the generalized exact cover problem where the columns can be divided into primary and secondary columns. A solution must cover the primary columns exactly once and the secondary columns at most once. When creating a problem with *create_problem* one only gives the set of primary columns (and secondary columns are implicitely given when mentioned in *add_row*).

type *problem*

val *create_problem* : *column list* → *problem*

Finally, *find_all_solutions* p f finds out all solutions for problem p and applies f on each solution.

type *solution*

val *find_all_solutions* : *problem* → (*solution* → *unit*) → *unit*

The type of solution is abstract but a solution can be unpacked with *unpack_solution* as the list of rows, each row being the set of its columns containing a 1. A function *print_solution* is also provided, which is more efficient than calling *unpack_solution* and then printing the result. *print_solution* prints each row of the solution on a separate line, and each row as the names of its columns containing a 1.

val *unpack_solution* : *solution* → *string list list*

```
val print_solution : Format.formatter → solution → unit
```

The following function counts the number of solutions. It is slightly more efficient than using *find_all_solutions* and incrementing a reference for each solution.

```
val count_all_solutions : problem → int
val count_all_solutions_int64 : problem → int64
```

Module Dlx

See Knuth's Dancing Links paper for details on the following code.

```
open Format

type node = {
  mutable left : node;
  mutable right : node;
  mutable up : node;
  mutable down : node;
  mutable col : node;
  name : string; (* header only *)
  mutable size : int;
}

type column = node

type problem = node

let create_node name =
  let rec n =
    { left = n; right = n; up = n; down = n; col = n; name = name; size = 0; }
  in
  n

let create_column = create_node
```

make a left-right doubly linked list from a list of nodes

```
let make_dllist = function
  | [] → ()
  | first :: _ as l →
      let rec make = function
        | [] → assert false
        | [n] → first.left ← n; n.right ← first
        | n :: (m :: _ as l) → n.right ← m; m.left ← n; make l
      in
      make l
```

```
let create_problem cl =
  let h = create_node "root" in
  make_dllist (h :: cl);
  h

let add_row cl =
  let one c = (* inserts a new 1 node in column c and returns it *)
    c.size ← c.size + 1;
    let n = create_node "" in
    n.col ← c;
    n.down ← c.down;
    c.down ← n;
    n.up ← c;
    n.down.up ← n;
    n
  in
  let nodes = List.map one cl in
  make_dllist nodes

let iter_up c f =
  let rec iter r = if r ≢ c then begin f r; iter r.up end in iter c.up

let iter_down c f =
  let rec iter r = if r ≢ c then begin f r; iter r.down end in iter c.down

let fold_down c f =
  let rec fold r acc = if r ≢ c then fold r.down (f r acc) else acc in
  fold c.down

let iter_left c f =
  let rec iter r = if r ≢ c then begin f r; iter r.left end in iter c.left

let iter_right c f =
  let rec iter r = if r ≢ c then begin f r; iter r.right end in iter c.right

let cover_column c =
  assert (c.col ≡ c); (* it is a column *)
  c.right.left ← c.left;
  c.left.right ← c.right;
  iter_down c
    (fun i →
       iter_right i (fun j →
                        j.down.up ← j.up; j.up.down ← j.down;
                        j.col.size ← j.col.size − 1))
```

```
let uncover_column c =
  assert (c.col ≡ c); (* it is a column *)
  iter_up c
    (fun i →
       iter_left i (fun j →
                      j.down.up ← j; j.up.down ← j;
                      j.col.size ← j.col.size + 1));
  c.right.left ← c;
  c.left.right ← c

let choose_column h =
  let s = ref h.right.size in
  let c = ref h.right in
  iter_right h (fun j → if j.size < !s then begin s := j.size; c := j end);
  assert (!c ≢ h);
  !c
```

finds all solutions and applies f on each

```
let find_all_solutions h f =
  let rec search sol =
    if h.right ≡ h then
      f sol
    else begin
      let c = choose_column h in
      if c.down ≡ c then raise Exit; (* empty column = no solution *)
      cover_column c;
      iter_down c
        (fun r →
           iter_right r (fun j → cover_column j.col);
           begin try search (r :: sol) with Exit → () end;
           iter_left r (fun j → uncover_column j.col);
        );
      uncover_column c
    end
  in
  search []

type solution = node list

let unpack_solution s =
  let unpack_row r =
    let l = ref [r.col.name] in
```

```
    iter_left r (fun c → l := c.col.name :: !l);
    !l
  in
  List.map unpack_row s

let print_solution fmt =
  let print_col_name o = fprintf fmt "%s␣" o.col.name in
  let print_row r =
    print_col_name r; iter_right r print_col_name;
    printf "@."
  in
  List.iter print_row

let count_all_solutions h =
  let rec search count =
    if h.right ≡ h then
      count + 1
    else begin
      let c = choose_column h in
      if c.down ≡ c then
        count (* empty column = no solution *)
      else begin
        cover_column c;
        let count =
          fold_down c
            (fun r count →
              iter_right r (fun j → cover_column j.col);
              let count = search count in
              iter_left r (fun j → uncover_column j.col);
              count)
            count
        in
        uncover_column c;
        count
      end
    end
  in
  search 0

let count_all_solutions_int64 h =
  let rec search count =
    if h.right ≡ h then
```

```
      Int64.succ count
    else begin
      let c = choose_column h in
      if c.down ≡ c then
        count (* empty column = no solution *)
      else begin
        cover_column c;
        let count =
          fold_down c
            (fun r count →
              iter_right r (fun j → cover_column j.col);
              let count = search count in
              iter_left r (fun j → uncover_column j.col);
              count)
            count
        in
        uncover_column c;
        count
      end
    end
  in
  search 0L
```