7 NP 問題

最も有名な NP 問題の SAT のアルゴリズムを見る．(NP ながら) 効率の良いアルゴリズムがあるので，実際に問題を SAT に還元してから計算するのがよかったりする．

```
type cnf = (string * bool) list list                          (* 論理積標準形 *)

module Val = Map.Make(struct type t = string let compare = compare end)
module Val :
  sig
    type key = string
    type +'a t                                        (* 均衡木構造に基いた写像の型 *)
    val empty : 'a t                                                (* 空の写像 *)
    val add : key -> 'a -> 'a t -> 'a t                            (* 関係の追加 *)
    val find : key -> 'a t -> 'a                                        (* 探索 *)
    val mem : key -> 'a t -> bool
    val fold : (key -> 'a -> 'b -> 'b) -> 'a t -> 'b -> 'b
    ...
  end

let positives v =                          (* true になっている要素のリストを作る *)
  List.rev (Val.fold (fun x b l -> if b then x :: l else l) v [])
val positives : bool Val.t -> Val.key list

exception Found of (bool Val.t)                    (* 割り当てを例外として返す *)

let find_opt x v =                              (* find の結果を option として返す *)
  try Some (Val.find x v) with Not_found -> None
val find_opt : Val.key -> 'a Val.t -> 'a option

let rec fast_solve v (problem : cnf) =
  match problem with
    [] -> raise (Found v)                                    (* 割り当てを返す *)
  | conj :: rem ->
      let rec solve_one v conj =
        match conj with
          [] -> ()
        | (x,b) :: conj' ->
            match find_opt x v with
              Some b' ->
                if b = b' then fast_solve v rem          (* 既に満されている *)
                else solve_one v conj'             (* この割当てでリテラルが偽 *)
            | None ->
                fast_solve (Val.add x b v) rem;  (* 真にして問題の残りを解く *)
                solve_one (Val.add x (not b) v) conj'    (* だめなら次を試す *)
      in
      solve_one v conj
val fast_solve : bool Val.t -> cnf -> unit

let fast_sat problem =
  try fast_solve Val.empty problem; None        (* 例外でなければ，答えがない *)
```

```
  with Found v -> Some (positives v)      (* 例外としてもらった答えを変換する *)
val fast_sat : cnf -> Val.key list option
```

他の問題を SAT に還元するとき，変数を生成する必要がある．

```
let rec gen m n f =                       (* m から n の整数に対して f を呼ぶ *)
  if m <= n then f m :: gen (m+1) n f else []
val gen : int -> int -> (int -> 'a) -> 'a list

let color_choice v n =                    (* 各頂点の可能な色を論理和として表現 *)
  gen 1 n (fun i -> v ^ string_of_int i, true)
val color_choice : string -> int -> (string * bool) list
```

練習問題 7.1 *1.* グラフを入力に取り，n 彩色問題を SAT として表現するプログラムを作れ．上の color_choice を使うといい．

2. 上記のアルゴリズムは割り当てを作りながらチェックをする．NP の定義のとおり，チェックのみを行う関数を定義せよ．