Module Hashtbl

1. On définit un type de liste de couples d'entiers, pour économiser de l'espace par rapport aux listes prédéfinies.

```
type bucketlist =
    Empty
  | Cons of int × int × bucketlist

let size = 1466333

let table = Array.create size Empty
```

2. On hache de manière efficace avec la somme des deux entiers, en tenant compte du fait que les deux bits de poids faible du premier valent 0.

```
let hash (a1, a2) = (a1 lsr 2) + a2
```

3. Le reste est classique...

```
let add ((a, b) as key) =
  let i = (hash key) mod size in
  table.(i) ← Cons(a, b, table.(i))

let mem ((a, b) as key) =
  let rec mem_in_bucket = function
  | Empty →
      false
  | Cons(c, d, rest) →
      (a ≡ c ∧ b ≡ d) ∨ mem_in_bucket rest
  in
  mem_in_bucket table.((hash key) mod size)
```

Module Coups

4. Une position est représentée par un entier (a, b), dont les bits codent la présence ou non d'une bille. Le codage est le suivant :

	0	1	2	3	4	5	6
0	a_0	a_1		$\ldots$			a_6
1	a_7	a_8		$\ldots$			a_{13}
2	a_{14}	a_{15}		$\ldots$			a_{20}
3	a_{21}	a_{22}		$\ldots$			a_{27}
4	b_0	b_1		$\ldots$			b_6
5	b_7	b_8		$\ldots$			b_{13}
6	a_{14}	b_{15}		$\ldots$			b_{20}

```
type position = int × int
```

5. On hash les positions de manière efficace, avec le module spécialisé précédemment défini.

```
module HashPos = Hashtbl
```

6. Pour simplifier la manipulation de ces bits par la suite, on crée une matrice $case(i, j)$ qui retourne le couple d'entier représentant exactement la case (i, j), pour $i, j \in \{0, \ldots, 6\}$.

```
let case =
  let m = Array.make_matrix 7 7 (0,0) in
  for i = 0 to 6 do
    for j = 0 to 6 do
      m.(i).(j) ←
        if j < 4 then (1 lsl (i + 7 × j), 0) else (0, 1 lsl (i + 7 × (j − 4)))
    done
  done;
  m
```

7. Les positions de départ et d'arrivée sont définies facilement.

```
let arrivee = case.(3).(3)
```

```
let depart = (1110111111111100111000011100₂, 001110000111001111111₂)
```

8. Affichage d'une position.

```
let affiche (a, b) =
  for j = 0 to 6 do
    for i = 0 to 6 do
      let (a', b') = case.(i).(j) in
      if (a land a' ≡ a') ∧ (b land b' ≡ b') then
        print_string "X␣" else print_string ".␣"
    done;
    print_newline()
  done
```

9. Test d'arrêt sur une position : il ne reste qu'une bille. (Utilisé pour rechercher la première solution plutôt que la ¡¡ vraie ¿¿ solution du solitaire.) Attention : *un_bit* est correct parce que l'on sait que $n > 0$.

```
let rec un_bit n = n ≡ 1 ∨ (n land 1 ≡ 0 ∧ un_bit (n lsr 1))
let test_arret (a, b) = ((un_bit a) ∧ b ≡ 0) ∨ ((un_bit b) ∧ a ≡ 0)
```

10. Un coup est représenté par deux ¡¡ positions ¿¿, la première donnant les deux cases devant contenir une bille, la deuxième la case devant être vide.

```
type coup = position × position
```

11. Dès lors, tester qu'un coup est possible dans une position donnée revient à tester les présence des deux billes et de la case vide par simples opérations de filtrage sur les bits.

```
let est_possible (p1, p2) ((b1, b2), (v1, v2)) =
    (p1 land b1 ≡ b1)
  ∧ (p2 land b2 ≡ b2)
  ∧ ((lnot p1) land v1 ≡ v1)
  ∧ ((lnot p2) land v2 ≡ v2)
```

12. Jouer un coup revient à appliquer le négatif de ces deux filtres ($\bullet \bullet \circ \rightarrow \circ \circ \bullet$).

```
let joue (p1, p2) ((b1, b2), (v1, v2)) =
  ((p1 lxor b1) lor v1, (p2 lxor b2) lor v2)
```

13. La liste de tous les coups.

```
let tous_les_coups =
  let l = ref [] in
  (* ajoute un coup et son inverse *)
  let ajoute (i1, j1) (i2, j2) (i3, j3) =
    let (a1, b1) as c1 = case.(i1).(j1) in
    let (a2, b2) = case.(i2).(j2) in
    let (a3, b3) as c3 = case.(i3).(j3) in
    l := ((a1 lor a2, b1 lor b2), c3) ::
          ((a2 lor a3, b2 lor b3), c1) :: !l
  in
  for j = 0 to 6 do
    ajoute (2, j) (3, j) (4, j);
    ajoute (j, 2) (j, 3) (j, 4)
  done;
  for i = 0 to 4 do
    ajoute (2, i) (2, i + 1) (2, i + 2);
    ajoute (3, i) (3, i + 1) (3, i + 2);
    ajoute (4, i) (4, i + 1) (4, i + 2);
    ajoute (i, 2) (i + 1, 2) (i + 2, 2);
    ajoute (i, 3) (i + 1, 3) (i + 2, 3);
    ajoute (i, 4) (i + 1, 4) (i + 2, 4)
  done;
  !l
```

14. Affichage d'une solution.

```
let affiche_coup ((b1, b2), cv) =
  let iv = ref 0 and jv = ref 0 in
  for i = 0 to 6 do
    for j = 0 to 6 do
      if case.(i).(j) = cv then begin iv := i; jv := j end
    done
  done;
  let dir = ref "" in
  for i = 0 to 6 do
    for j = 0 to 6 do
      let (c1, c2) = case.(i).(j) in
      if (b1 land c1 ≠ 0) ∨ (b2 land c2 ≠ 0) then
        if i < !iv then dir := "gauche"
        else if i > !iv then dir := "droite"
        else if j > !jv then dir := "bas"
```

```
      else dir := "haut"
    done
  done;
  Printf.printf "(%d,␣%d)␣%s\n" !iv !jv !dir

let affiche_solution l =
  List.iter affiche_coup (List.rev l)
```

Module Solitaire

15. L'algorithme suivant est un backtracking classique, où l'on teste si une position a déjà été rencontrée.

```
exception Echec

let rec backtrack sol p =
  if p = arrivee then
    sol
  else begin
    HashPos.add p;
    let rec essaye = function
      | [] → raise Echec
      | c :: r →
          if est_possible p c then
            let p′ = joue p c in
            if ¬ (HashPos.mem p′) then
              try backtrack (c :: sol) p′ with Echec → essaye r
            else
              essaye r
          else
            essaye r
    in
    essaye tous_les_coups
  end
```

16. Le programme principal appelle *backtrack* sur la position de départ, et affiche la solution.

```
let _ =
  let s = backtrack [] depart in
  affiche_solution s
```

Index