

# Examen surprise

## Correction

### Exercice 1

**Exercice 1.** Qu'est-ce qu'un algorithme ?

Un algorithme est une suite d'instructions *qui doivent permettre d'obtenir un résultat précis*.

Un algorithme peut être écrit dans un pseudo-langage naturel (c'est-à-dire en français) ou bien dans un langage de programmation tel que Java, OCaml, Python, C#, Haskell ou Pascal... Généralement, on commence par décrire l'algorithme en langage naturel ou/et mathématique avant de l'écrire dans un langage plus précis. Le langage plus précis que vous avez employé jusqu'à maintenant, inspiré de Pascal, se définit de la manière suivante :

- tout algorithme commence par un bloc optionnel de création de variables, suivi par **début**, le corps de l'algorithme, suivi de **fin** – on pourra négliger d'écrire **début** et **fin** si le corps de l'algorithme ne fait qu'une seule ligne et s'il n'y a pas de création de variable
- un bloc de création de variables commence par le mot **variables** et continue par une liste de variables, sous la forme *nom de la variable : type de la variables*
- pour le moment, les types de variables que vous avez vus sont entier, booléen ou tableau de  $t$ , où  $t$  est à son tour un type de variable – vous pouvez aussi employer des sous-ensembles, tels que entier compris entre  $-7$  et  $+7$
- le corps d'un algorithme est une suite d'instructions
- pour le moment, les instructions que vous avez vues sont
  - l'affectation du résultat de l'évaluation du résultat d'une expression  $e$  à une variable  $v$ , notée  $v \leftarrow e$ . Cette instruction n'a de sens que si la variable  $v$  a été définie et doit être comprise comme « évaluer l'expression  $e$ . Appelons  $r$  le résultat de cette évaluation. Pour la suite du programme, ou jusqu'à une nouvelle affectation à la variable  $v$ , ou jusqu'à ce que  $v$  ne soit plus définie, à chaque fois que l'on utilise le nom  $v$ , c'est pour faire référence au résultat  $r$ . »
  - une instruction de contrôle de flot  
si  $e$  alors  
 $a$   
  
éventuellement suivie de **fin si**, pour des questions de lisibilité, et où  $e$  est une expression dont le résultat est un booléen et  $a$  est un algorithme (n'oubliez pas que  $a$  peut à son tour contenir des créations de variables et *doit* contenir **début** et **fin**). Cette instruction doit être comprise comme « si  $e$  s'évalue en la valeur "vrai", alors exécuter  $a$ , sinon sauter  $a$  et passer à la suite. »
  - une instruction de contrôle de flot  
si  $e$  alors  
 $a$   
sinon  
 $b$

éventuellement suivie de fin si, pour des questions de lisibilité. Ici aussi,  $e$  est une expression dont le résultat est un booléen,  $a$  est un algorithme,  $b$  est un algorithme. Cette instruction doit être comprise comme « si  $e$  s'évalue en la valeur "vrai", alors exécuter  $a$ , sinon, exécuter  $b$ , puis, dans les deux cas, passer à la suite. »

- une boucle

```
tant que  $e$  faire
   $a$ 
```

éventuellement suivie de fin tant que, pour des questions de lisibilité. Ici aussi,  $e$  est une expression dont le résultat est un booléen et  $a$  est un algorithme. Cette instruction doit être comprise comme « si  $e$  s'évalue en la valeur "vrai", alors exécuter  $a$ , puis vérifier si  $e$  s'évalue toujours en la valeur "vrai", si oui exécuter de nouveau  $a$ , etc... et continuer tant que l'expression  $e$  s'évalue en la valeur "vrai". Une fois que  $e$  ne s'évalue plus en "vrai", passer à la suite. »

- la production d'un résultat par

```
afficher  $e$ 
```

Cette instruction doit être comprise comme « évaluer  $e$ , afficher le résultat et passer à la suite ». Notez que, à ce stade, il est généralement une mauvaise idée d'afficher plusieurs résultats supposés définitifs pour un même algorithme, surtout lorsque ces résultats sont contradictoires.

- pour le moment, les expressions que vous avez vues sont les expressions arithmétiques habituelles (addition, comparaison, etc.), les conjonctions d'expressions ( $e$  ET  $e'$ , aussi noté  $e \wedge e'$ ), les disjonctions d'expressions ( $e$  OU  $e'$ , aussi noté  $e \vee e'$ ) et les appels de fonctions ( $\sin(e)$  ...).

Vous verrez d'autres types d'instructions et d'expressions avec le temps.

**Important** Ce langage précis n'est pas le seul langage précis possible. C'est entre autres pour cela qu'il est nécessaire de commencer par préciser en français intelligible la liste des opérations que l'on veut entreprendre.

## Exercice 2

**Exercice 2.** (compléter l'énoncé)

### En français

Deux reines  $A$  et  $B$  peuvent se prendre mutuellement si elles sont sur la même ligne ( $x_A = x_B$ ), sur la même colonne ( $y_A = y_B$ ), sur la même diagonale Nord-Ouest/Sud-Est ( $x_A - x_B = y_A - y_B$ ) ou la même diagonale Nord-Est/Sud-Ouest ( $x_A + x_B = y_A + y_B$ ). On va donc vérifier si au moins une de ces quatre conditions est vérifiée.

### En pseudo-langage

```
début
```

```
  si  $x_A = x_B \vee y_A = y_B \vee x_A - x_B = y_A - y_B \vee x_A + x_B = y_A + y_B$  alors
```

```
    afficher "vrai"
```

```
  sinon
```

```
    afficher "faux"
```

```
  fin si
```

```
fin
```

ou, plus simplement,

```
afficher ( $x_A = x_B \vee y_A = y_B \vee x_A - x_B = y_A - y_B \vee x_A + x_B = y_A + y_B$ )
```

Vous noterez que l'expression  $x_A = x_B \vee y_A = y_B \vee x_A - x_B = y_A - y_B \vee x_A + x_B = y_A + y_B$  est une expression booléenne – qui s'évaluera donc en « vrai » ou « faux ». Il suffit donc d'afficher le résultat de cette évaluation.

### En Java

En Java, on pourrait écrire

```
{
  boolean result;
  if (xA == xB || yA == yB || xA-xB == yA-yB || xA+xB == yA+yB)
    result = true;
  else
    result = false;
  return result;
}
```

En Java, les accolades ont à peu de choses près le rôle de début et fin. L'opération ou s'écrit `||` et on emploie l'instruction spéciale `return` pour renvoyer un résultat.

On pourra aussi écrire, plus simplement,

```
return (xA == xB || yA == yB || xA-xB == yA-yB || xA+xB == yA+yB);
```

Si l'on souhaite réutiliser cet algorithme pour la suite, sous le nom `prise`, on écrira

```
boolean prise(xA : int, yA : int, xB : int, yB : int)
{
  return (xA == xB || yA == yB || xA-xB == yA-yB || xA+xB == yA+yB);
}
```

### En OCaml

En OCaml, on aurait par exemple l'algorithme suivant :

```
if xa = xb or ya = yb or xa-xb = ya-yb or xa+xb = ya+yb then
  true
else
  false
```

En OCaml, l'opération « ou » s'écrit `or` et il n'y a pas besoin de préciser que l'on renvoie un résultat.

On pourra aussi écrire, plus simplement

```
xa = xb or ya = yb or xa-xb = ya-yb or xa+xb = ya+yb
```

Si l'on souhaite réutiliser cet algorithme pour la suite, sous le nom `prise`, on écrira

```
let prise xa ya xb yb =
  xa = xb or ya=yb or xa-xb = ya-yb or xa+xb = ya+yb
```

## Exercice 3

**Exercice 3.** Soient  $n$  reines de coordonnées  $(x_1, y_1), (x_2, y_2) \dots (x_n, y_n)$  sur un échiquier de longueur *longueur* et de largeur *largeur*. Considérons une reine supplémentaire de coordonnées  $(x_A, y_A)$ . Écrire un algorithme dont le rôle est d'afficher « vrai » si la reine  $A$  est en sécurité par rapport aux  $n$  reines, « faux » dans le cas contraire.

## En français

Nous allons vérifier successivement pour chacune des  $n$  reines que la reine n'est pas en prise avec la nouvelle reine A.

## En pseudo-langage précis

En pseudo-langage précis, nous sommes obligés de choisir un ordre parmi les reines. On commencera à la reine 1, puis la reine 2... jusqu'à la reine  $n$ . Pour ce faire, nous emploierons une boucle. Cela donne :

```
variables
  index : entier entre 1 et n
  danger : booléen
début
  index ← 1
  danger ← faux
  tant que index ≤ n et danger = faux faire
  début
    si prise( $x_{\text{index}}, y_{\text{index}}, x_A, y_A$ ) alors
      danger ← vrai
    fin si
    index ← index+1
  fin
  afficher ¬danger
fin
```

Rappelons que le symbole  $\neg$  désigne la négation logique. C'est-à-dire que  $\neg$ vrai vaut faux et  $\neg$ faux vaut vrai.

## En Java

En Java, on aura

```
{
  int index = 1;
  bool danger = false;
  while(index <= n && danger == false)
  {
    if prise(x[index-1], y[index-1], xA, yA)
      danger = true;
    index = index+1;
  }
  return !danger;
}
```

En Java, nous avons employé un tableau  $x[]$  et un tableau  $y[]$  pour représenter les coordonnées des  $n$  reines. À cause de bizarreries de Java, nous devons utiliser  $\text{index}-1$  pour consulter les coordonnées de la reine numéro  $\text{index}$ , puisque Java considère que le premier élément est toujours numéroté 0. La négation est représentée par le point d'exclamation.

De manière plus concise, on pourra aussi écrire

```
{
  bool danger = false;
  for(int index = 1; index <=n && danger == false; ++index)
  {
    danger = prise(x[index-1], y[index-1], xA, yA);
  }
}
```

```

    return !danger
}

```

La notation `for` est juste une autre manière plus concise mais plus compliquée de noter tant `que`, prévue pour être employée dans des circonstances où il est nécessaire de compter – ici entre 1 et  $n$ . La notation `++index` est équivalente à `index = index+1`.

De manière encore plus concise, on pourra tirer parti du fait que l’instruction spéciale `return` achève immédiatement l’exécution de la routine en renvoyant un résultat.

```

{
  for(int index = 0; index < n; ++index)
  {
    if(prise(x[index], y[index], xA, yA)
       return false;
  }
  return true;
}

```

Si l’on veut réutiliser cet algorithme pour la suite, sous le nom `surete`, on écrira

```

bool surete(int xA, int yA, int x[], int y[])
{
  for(int index = 0; index < n; ++index)
  {
    if(prise(x[index], y[index], xA, yA)
       return false;
  }
  return true;
}

```

Pour des raisons techniques que nous ne détaillerons pas, il ne sera malheureusement pas possible dans la suite de réutiliser la version Java de cet algorithme.

## OCaml

En OCaml, on demandera directement de vérifier que, pour toute reine choisie dans la liste `reines`, la reine en question n’est pas en prise avec  $A$ .

```

for_all (fun (x_index, y_index) -> not (prise x_index y_index x_a y_a)) reines

```

Le terme `fun` désigne une fonction mathématique – nous avons ici défini une fonction mathématique qui, si on lui donne les coordonnées d’une reine, vérifie si cette reine est en prise avec  $a$  et renvoie `false` le cas échéant, `true` dans le cas contraire. La négation se dénote `not`. Enfin, à l’aide de `for_all` `… reines`, nous demandons à OCaml d’appliquer cette vérification à toute la liste des reines.

Si l’on veut réutiliser cet algorithme pour la suite, sous le nom `surete`, on écrira

```

let surete reines =
  for_all (fun (x_index, y_index) -> not (prise x_index y_index x_a y_a)) reines

```

## Exercice 4

**Exercice 4.** Considérons toujours  $n$  reines de coordonnées  $(x_1, y_1), (x_2, y_2) \dots (x_n, y_n)$  sur un échiquier de longueur *longueur* et de largeur *largeur*. Écrire maintenant un algorithme pour déterminer si chacune des  $n$  reines est en sécurité par rapport aux autres.

En français

L'idée générale est de prendre indépendamment chaque reine de la liste et d'employer l'algorithme de l'exercice précédent pour comparer cette reine avec toutes les autres. Nous pouvons cependant accélérer (et simplifier) largement l'algorithme en tirant parti du fait que si une reine  $A$  ne peut pas prendre une reine  $B$ , alors la reine  $B$  ne peut pas non plus prendre la reine  $A$ . Il suffit donc, par exemple, pour chaque reine de la liste de vérifier si elle ne peut prendre aucune des reines *suyvantes*.

### Pseudo-langage

En pseudo-langage précis, de nouveau, nous sommes obligés de choisir un ordre parmi les reines. Nous allons donc choisir comme reine en cours successivement la reine 1, la reine 2, ... la reine  $n - 1$  et à chaque fois, nous allons comparer avec les reines suivantes, dans l'ordre croissant, jusqu'à la reine  $n$ .

```
variables
  reine_en_cours : entier entre 1 et  $n - 1$ 
  danger          : booléen
début
  reine_en_cours  $\leftarrow 1$ 
  danger           $\leftarrow$  faux
  tant que reine_en_cours  $< n$  et danger = faux faire
  variables
    reine_suyvante : entier entre reine_en_cours et  $n$ 
  début
    reine_suyvante  $\leftarrow$  reine_en_cours+1
    tant que reine_suyvante  $\leq n$  et danger = faux faire
    début
      si prise( $x_{reinesuyvante}, y_{reinesuyvante}, x_{reineencours}, y_{reineencours}$ ) alors
      début
        danger  $\leftarrow$  vrai
      fin
      reine_suyvante  $\leftarrow$  reine_suyvante+1
    fin
    reine_en_cours  $\leftarrow$  reine_en_cours+1
  fin
  afficher  $\neg$ danger
fin
```

Notez que nous avons déclaré la variable `reine_suyvante` à l'intérieur de la boucle. Cela n'était pas obligatoire mais cela évite des erreurs.

### En Java

En Java, cela se traduit directement par

```
{
  int reineEnCours = 1;
  boolean danger   = false;
  while(reineEnCours < n && danger == false)
  {
    int reineSuyvante = reineEnCours+1;
    while(reineSuyvante <= n)
    {
      if(prise(x[reineSuyvante-1],y[reineSuyvante-1],
              x[reineEnCours-1],y[reineEnCours-1]))
      {
        danger = true;
      }
      reineSuyvante = reineSuyvante+1;
    }
  }
}
```

```

    }
    reineEnCours = reineEnCours+1
  }
  return !danger;
}

```

ou, de manière plus concise,

```

{
  boolean danger = false;
  for(int reineEnCours = 1; reineEnCours < n; ++reineEnCours)
  {
    for(int reineSuivante = reineEnCours+1; reineSuivante <= n; ++reineSuivante)
      danger = prise(
        x[reineSuivante-1],y[reineSuivante-1],
        x[reineEnCours-1],y[reineEnCours-1]
      );
  }
  return !danger;
}

```

ou, de manière encore plus concise,

```

{
  for(int reineEnCours = 0; reineEnCours < n-1; ++reineEnCours)
  {
    for(int reineSuivante = reineEnCours+1; reineSuivante < n; ++reineSuivante)
      if(prise(x[reineSuivante],y[reineSuivante],x[reineEnCours],y[reineEnCours]))
        return false;
  }
  return true;
}

```

## En Ocaml

En Ocaml, on aura plutôt le réflexe de se ramener à chaque fois à un cas précédent : si la liste des reines contient au moins une reine `reine_en_cours` et probablement d'autres reines `reines_suivantes`, on commence par vérifier comme dans l'exercice précédent qu'aucune reine de `reines_suivantes` ne peut prendre `reine_en_cours`. Si c'est bien le cas, on peut oublier `reine_en_cours` et ne se concentrer que sur les `reines_suivantes`. Enfin, si nous avons vérifié toutes les reines, c'est-à-dire que la liste des reines est vide, ou `[]`, nous avons vérifié toutes les reines avec succès et nous pouvons répondre « oui » :

```

let rec teste_reines reines = match reines with
| reine_en_cours::reines_suivantes ->
  (surete reine_en_cours reines_suivantes)
  & (teste_reines reines_suivantes)
| [] -> true

```

notez que ce programme porte, par construction, le nom `teste_reines`.

Le terme `let` signifie « soit », comme en mathématiques. Le terme `rec` désigne une définition récursive, c'est-à-dire une définition qui se ramène à chaque fois au cas précédent, jusqu'à obtenir un cas simple. Les termes `match ... with` signifient qu'on va regarder les différents cas possibles. Chaque cas est introduit par le caractère `|`.

On pourrait aussi écrire

```

let rec pour_tous test liste = match liste with

```

```

| en_cours::suivants -> (test en_cours suivants) & (pour_tous test suivants)
| []                 -> true
in
pour_tous surete reines

```

dans cette version, nous commençons par expliquer à OCaml comment appliquer un test `test` à une reine `en_cours` et à la liste des autres reines `suivants` – cette technique fonctionne en fait pour n’importe quelle liste, pas juste pour des reines. Enfin, nous demandons à OCaml d’appliquer le test de l’exercice 3 à notre liste de reines.

## Exercice 5

**Exercice 5.** Considérons toujours  $n$  reines de coordonnées  $(x_1, y_1), (x_2, y_2) \dots (x_n, y_n)$  sur un échiquier de longueur *longueur* et de largeur *largeur*. Écrire maintenant un algorithme pour trouver des coordonnées sûres pour une reine  $n + 1$ .

### En français

Pour chercher un emplacement sûr pour une nouvelle reine, il suffit d’essayer toutes les cases possibles jusqu’à en trouver une qui ne soit prise par personne. On pourra par exemple réutiliser le résultat de l’exercice 3 pour vérifier qu’un emplacement est sûr.

### En pseudo-langage

Pour essayer tous les emplacements, on commence par  $(1, 1)$ , puis  $(1, 2)$ , puis  $(1, 3)$  puis ... jusqu’à  $(1, \text{hauteur})$ , puis  $(2, 1)$ ,  $(2, 2)$ ... jusqu’à  $(2, \text{hauteur})$  et ainsi de suite jusqu’à  $(\text{largeur}, \text{hauteur})$ . C’est-à-dire que nous employons de nouveau une boucle (pour tous les  $x$  possibles), contenant une boucle (pour tous les  $y$  possibles), contenant encore une boucle (pour chaque reine déjà placée).

On aura quelque chose comme

```

variables
  x_en_cours : entier entre 1 et largeur
  y_en_cours : entier entre 1 et hauteur
  danger     : booléen
  trouvé     : booléen
début
  x_en_cours ← 1
  trouvé     ← faux
  tant que x_en_cours ≤ largeur et trouvé = faux faire
  début
    y_en_cours ← 1
    tant que y_en_cours ≤ hauteur et trouvé = faux faire
    variables
      reine_en_cours : entier entre 1 et n
    début
      reine_en_cours ← 1
      danger         ← faux
      tant que reine_en_cours ≤ n et danger = faux faire
      début
        si prise( $x_{\text{reineencours}}$ ,  $y_{\text{reineencours}}$ , x_en_cours, y_en_cours) alors
          danger ← vrai
        reine_en_cours ← reine_en_cours+1
      fin
    si danger = vrai faire
      trouvé ← vrai
  fin
fin

```

```

fin
si trouvé = vrai faire
    afficher x_en_cours, y_en_cours
sinon
    afficher ‘‘il n’y a nulle part où placer une nouvelle reine’’
fin

```

## Java

En Java, nous pouvons traduire cela à peu près directement.

```

{
    int xEnCours;
    int yEnCours;
    boolean danger = false;
    boolean trouve = false;
    for(xEnCours = 1; xEnCours <= largeur && trouve == false; ++xEnCours)
    {
        for(yEnCours = 1; yEnCours <= hauteur && trouve == false; ++yEnCours)
        {
            danger = false;
            for(int reineEnCours = 1; reineEnCours <= n && danger == false; ++reineEnCours)
            {
                danger = prise(x[reineEnCours-1],y[reineEnCours-1],xEnCours,yEnCours);
            }
            if(danger==false)
                trouve = true;
        }
    }
    if(trouve == true)
        return new Position(xEnCours,yEnCours);
    else
        return null;
}

```

notez que ceci utilise une construction `Position`, à définir séparément. La définition de `Position` n’est pas fournie car elle nécessite une cinquantaine de lignes supplémentaires – il s’agit de lignes simples mais qui elles-mêmes nécessitent l’explication de nombreuses autres constructions.

L’algorithme répond en renvoyant soit une nouvelle `Position` avec les coordonnées qui nous intéressent, soit la valeur spéciale `null`, qui signifie « aucune valeur ». Notez que s’il y avait plus de deux possibilités (soit une position, soit pas de solution), il serait nécessaire d’employer des techniques plus complexes.

De manière plus concise, nous pouvons réécrire

```

{
    boolean danger = false;
    for(int xEnCours = 1; xEnCours <= largeur; ++xEnCours)
    {
        for(int yEnCours = 1; yEnCours <= hauteur; ++yEnCours)
        {
            danger = false;
            for(int reineEnCours = 1; reineEnCours <= n && !danger; ++reineEnCours)
            {
                danger = prise(x[reineEnCours-1],y[reineEnCours-1],xEnCours,yEnCours);
            }
            if(!danger)

```

```

        return new Position(xEnCours, yEnCours);
    }
}
return null;
}

```

### En OCaml

En OCaml, la méthode la plus simple est à peu près la même.

```

type resultat = Position of (int * int) | Pas_de_solution

exception Fini of resultat

for x_en_cours = 1 to largeur do
  for y_en_cours = 1 to largeur do
    if surete (x_en_cours, y_en_cours) reines
    then
      raise (Fini (Position (x_en_cours, y_en_cours)))
    else
      ()
  done
done;
raise (Fini Pas_de_solution)

```

Nous commençons par définir un « cas exceptionnel », qui va nous servir à répondre soit que nous avons trouvé une solution avec des coordonnées données, soit que nous n’avons pas trouvé de solution – c’est la barre verticale | qui représente le « soit ». Cette définition en deux lignes, qui pourrait être simplement étendue pour rendre compte de plus de deux possibilités correspond grossièrement à la cinquantaine de lignes nécessaires pour définir `Position` en Java.

## Exercice 6

**Exercice 6.** Dans l’algorithme précédent, combien de fois  $\text{prise}(x_A, y_A, x_B, y_B)$  est-elle évaluée ?

Dans le pire des cas, `prises` est évaluée  $\text{largeur} \times \text{hauteur} \times n$  fois.

## Exercice 7

**Exercice 7.** Discuter comment résoudre complètement le problème des  $n$  reines.

### En français

La méthode la plus simple pour résoudre le problème des  $n$  reines est de commencer par poser une reine dans la case arbitraire puis de se ramener au problème des  $n - 1$  reines sur l’échiquier où toutes les cases prises par la première reine sont interdites. Si jamais on atteint une situation où on est arrivé à poser toutes les reines, le problème est résolu. Si jamais on se retrouve dans une situation où il reste des reines à poser mais nulle part où les poser, il n’est pas certain que le problème n’ait pas de solution, puisqu’on peut parfois trouver une solution en essayant une autre manière de poser les reines précédentes. Par conséquent, si la reine numéro  $i$  n’est pasposable, il faut revenir à la reine numéro  $i - 1$  et essayer de lui trouver une autre position, avant de recommencer à chercher une position pour la reine  $i$ . Si aucune des positions possibles de  $i - 1$  ne permet de placer  $i$ , il est nécessaire de revenir à la reine  $i - 2$ , avant de réessayer toutes les positions possibles de  $i - 1$ , etc. Si on a essayé toutes les manières possibles de poser toutes les reines et qu’aucune n’est valide, le problème n’a pas de solution.

Ce processus de retour sur ses pas s'appelle le « backtracking », nous en reparlerons. Au total, cet algorithme nécessite jusqu'à largeur  $\times$  hauteur  $\times n!$  essais, c'est-à-dire plus de 250000 essais pour 8 reines sur un échiquier  $8 \times 8$ .

## En Java

En Java, on peut écrire quelque chose comme

```
void reines(Position[] deja_places, int reste_a_placer)
{
    if(reste_a_placer==0)
        throw new Resultat(deja_places);
    else
    {
        for(int x = 0; x<this.largeur; ++x)
        {
            for(int y = 0; y<this.hauteur;++y)
            {
                if (this.securite(x, y, deja_places, reste_a_placer))
                {
                    deja_places[reste_a_placer-1] = new Position(x, y);
                    this.reines(deja_places, reste_a_placer-1);
                }
            }
        }
    }
}

boolean securite(int x, int y, Position[] autres, int reste_a_placer)
{
    for(int i = reste_a_placer; i < autres.length ++i)
    {
        Position reine_en_cours = autres[i];
        if(this.prise(x,y,reine_en_cours.x,reine_en_cours.y))
            return false;
    }
    return true;
}
```

Nous en avons profité pour donner des noms aux sous-programmes `prise`, `securite` ou `reines`. De nouveau, nous avons sauté la définition de `Resultat` et de `Position`.

## En Ocaml

En OCaml, on peut écrire

```
type resultat_final = Trouve of (int * int) list | Pas_de_solution

exception Fini of resultat_final

let rec place_reine reines_places reines_a_placer =
    if reines_a_placer = 0 then
        raise (Fini (Trouve reines_places))
    else
        for x = 1 to largeur do
            for y = 1 to hauteur do
```

```

    if surete x y reines_placees then
      place_reine ((x,y)::reines_placees) (reines_a_placer-1)
    else
      ()
    done
  done
in
  place_reine [] n; raise (Fini Pas_de_solution)

```

Pour information, confronté à un échiquier  $8 \times 8$  et à 8 reines, la version OCaml de ce programme trouve en quelques secondes le résultat suivant :

