DS6 (2 heures)

Lisez tout le texte avant de commencer. La plus grande importance sera attachée à la clarté, à la précision et à la concision de la rédaction. Toute réponse non justifiée ne sera pas prise en compte. Si vous repérez ce qui vous semble être une erreur d'énoncé, signalez la sur votre copie et poursuivez votre composition en expliquant les raisons de vos éventuelles initiatives. L'usage de tout dispositif électronique est interdit.

Exercice 1

Question 1. φ et ψ désignent deux formules logiques.
□ **1.1.** En déduction naturelle, prouver le séquent : $\vdash (\neg\varphi \vee \psi) \to (\varphi \to \psi)$.
□ **1.2.** Justifier que le séquent $\vdash (\varphi \to \psi) \to (\neg\varphi \vee \psi)$ est sémantiquement vrai. Sans construire d'arbre de preuve, en déduire qu'il prouvable.

Question 2. φ et ψ désignent deux prédicats.
□ **2.1.** En déduction naturelle, prouver le séquent : $\vdash (\forall x\ \varphi \vee \forall x\ \psi) \to \forall x\ (\varphi \vee \psi)$.
□ **2.2.** Qu'en est-il du séquent : $\vdash \forall x\ (\varphi \vee \psi) \to (\forall x\ \varphi \vee \forall x\ \psi)$?

Exercice 2

Le *hanjie* est un jeu de réflexion qui consiste à retrouver une image par le noircissement de certaines cases d'une grille rectangulaire, sur la base d'indications laissées sur les côtés de la grille. Pour chaque rangée, qu'elle soit horizontale ou verticale, le joueur dispose d'une suite d'entiers non nuls t_1, t_2, t_3, etc. qui indiquent que la rangée contient une série de t_1 cases noires consécutives, suivie plus loin d'une série de t_2 cases noires consécutives, et ainsi de suite. Un nombre quelconque de cases blanches peut se trouver en tête ou en queue de rangée ; au moins une case blanche sépare deux séries de cases noires.

On présente ci-dessous un exemple résolu à la main. Une grille vide, de taille 5×7, est définie (figure 1). Tant que la couleur d'une case est inconnue, elle est marquée par un symboles `#`. Les rangées sont numérotées à partir de 0, de gauche à droite pour les colonnes, du haut vers le bas pour les lignes.
La colonne 0 et la colonne 5 sont de longueur 5. L'indication est $[5]$ dans les deux cas. Elles doivent donc chacune contenir 5 blocs noirs consécutifs. On les noircit donc en totalité (figure 2). Sur la ligne 0, il n'y a qu'une seule manière de placer un bloc de 4 cases noires puis 2 cases noires. De même, sur la ligne 2, il n'y a qu'une seule manière de positionner le bloc de longueur 2 : il doit se trouver au milieu entre les deux blocs déjà isolés. Sur la ligne 3, on a déjà placé deux cases noires : les autres sont donc toutes blanches. Enfin, sur la ligne 4, il n'y a plus qu'une seule manière de placer un bloc de 6 cases noires (figure 3). En reprenant les indications des colonnes, dans les colonnes 1, 2 et 4, la dernière case inconnue est blanche. Dans la colonne 3 et 6 en revanche, on doit noircir la dernière case inconnue. On obtient ainsi une unique solution du hanjie (figure 4).

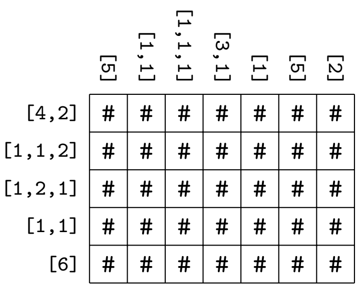


Figure 1

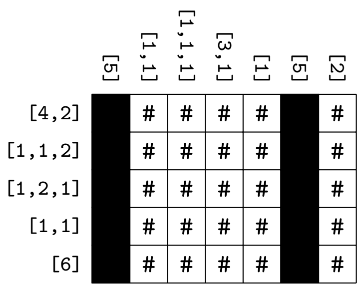


Figure 2

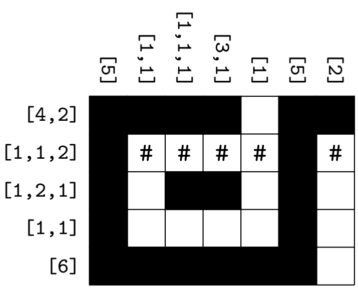


Figure 3

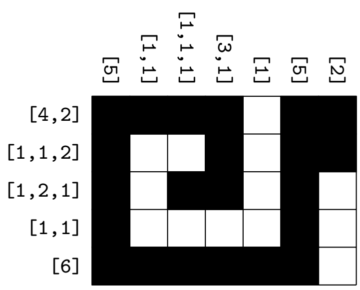
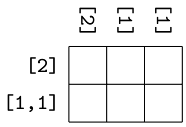
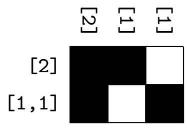


Figure 4

Pour déterminer la couleur de certaines cases, on peut écrire des formules de logique propositionnelle. On se limite ici au hanjie H défini par la grille de gauche ci-dessous. Une solution est donnée par la grille du milieu.





x_0	x_1	x_2
x_3	x_4	x_5

Question 1. Par un raisonnement en langue française, établir l'unicité de la solution du hanjie H.

Question 2. On introduit six variables booléennes, nommées $x_0, x_1, \dots, x_5$ et correspondant aux cases de la figure de droite ci-dessus. On associe la valeur de vérité V (vrai) à la couleur noire et F (faux) à la couleur blanche.
□ **2.1.** Soit L_0 le prédicat : *l'indication de la première ligne du hanjie H est satisfaite.* Dresser la table de vérité du prédicat L_0 portant sur les variables x_0, x_1, x_2. En déduire une formule de logique φ sous forme normale conjonctive qui décrit le prédicat L_0.
□ **2.2.** Soit C_1 le prédicat : *l'indication de la colonne du milieu du hanjie H est satisfaite.* Dresser la table de vérité du prédicat C_1 portant sur les variables x_1, x_4. En déduire une formule de logique ψ sous forme normale conjonctive qui décrit le prédicat C_1.

Question 3. Les règles d'inférence de la déduction naturelle sont rappelées dans l'annexe.
□ **3.1.** Construire un arbre de preuve qui démontre le séquent $\varphi \vdash x_1$.
□ **3.2.** Construire de même un arbre de preuve qui démontre le séquent $\psi, x_1 \vdash \neg x_4$.
□ **3.3.** On note ψ' la formule logique obtenue à partir de ψ en remplaçant la variable x_1 par x_2 et la variable x_4 par x_5. Démontrer qu'il n'existe pas d'arbre de preuve qui démontre la formule $\varphi \wedge \psi' \vdash \neg x_2$.

Exercice 3

L'annexe fournit les fonctions utiles pour manipuler les *threads* et les *mutex* en OCaml.

Question 1. On donne le programme suivant. Expliquer le rôle des deux fonctions `calc` et `main`. Qu'affiche le programme après son exécution ?

```
let calc (arr, idx_min, idx_max, retval) =
  let s = ref 0 in
  for i = idx_min to idx_max-1 do s := !s + arr.(i) done;
  retval := !s
;;

let main () =
  let arr = [|3;1;4;1;5;9;2;6|] in
  let n = Array.length arr in
  let s = ref 0 in
  calc (arr, 0, n, s);
  Printf.printf "s = %d\n" !s
;;

let () = main ();;
```

Question 2. On souhaite calculer la somme des éléments d'un tableau d'entiers de taille n en utilisant k fils d'exécution, avec $k \in [\![1, n]\!]$.
□ **2.1.** Justifier la nécessité de recourir à un *mutex* pour calculer cette somme.
□ **2.2.** Le tableau est découpé en k portions délimitées par les indices $(d_0, d_1, \dots, d_k)$ tels que la i-ième portion est comprise entre les indices d_i inclus et d_{i+1} exclu. Que valent d_0 et d_k ?
□ **2.3.** On souhaite que les portions soient de tailles décroissantes :

$$\forall i \in [\![0, k-2]\!] \quad d_{i+2} - d_{i+1} \leqslant d_{i+1} - d_i$$

et de tailles qui diffèrent d'au plus 1 : $d_1 - d_0 \leqslant d_{k+1} - d_k + 1$.
▷ **2.3.1.** Quelles sont les valeurs des indices si $n = 15$ et $k = 4$?
▷ **2.3.2.** Écrire une fontion `idx : int -> int -> int -> int` qui prend en arguments les entiers n, k, i et qui renvoie l'indice d_i.

Question 3. Proposer deux nouvelles fonctions `calc_thrd` et `main` de signatures :

```
calc_thrd : int array * int * int * int ref * Mutex.t -> unit
main : int -> unit
```

qui permettent le calcul de la somme à l'aide de fils d'exécution et d'un *mutex*.

Exercice 4

Deux fils d'exécution (*threads*), désignés par T_0 et T_1, souhaitent accéder à une ressource partagée. Cette dernière comporte une *section critique*. L'algorithme de Peterson permet l'accès à la section critique en satisfaisant les quatre contraintes suivantes.

- *Exclusion mutuelle* – L'algorithme garantit l'exclusivité de l'accès à la section critique à un seul *thread*.
- *Absence de famine* – Lorsqu'un *thread* souhaite accéder à la section critique, il finit toujours par y parvenir.
- *Absence d'interblocage* – Si les deux threads souhaitent accéder à la section critique, l'un d'eux y parvient toujours.
- *Attente bornée* – Si un thread attend pour enter en section critique, le nombre de fois où l'autre thread y entre avant lui est borné.

Pour mettre en œuvre l'algorithe de Peterson, on définit :

- une variable *turn* qui peut prendre soit la valeur 0 associée au thread T_0, soit la valeur 1 associée au thread T_1 ; initialement, *turn* peut indifférement recevoir 0 ou 1 ;
- un tableau de deux booléens *flag[2]* initialisé à *false*.

flag[i] indique si T_i peut entrer en section critique et la valeur de *turn* indique quel *thread* y entre. Le pseudo-code suivant propose une mise en œuvre de l'algorithme de Peterson pour chacun des deux *threads*.

Algorithme 1 : Thread T_0

```
tant que (true) faire
    // section non critique
    flag[0] ← true
    turn ← 1
    tant que (flag[1]) && (turn = 1) faire
        rien
    // section critique
    flag[0] ← false
    // section non critique
```

Algorithme 2 : Thread T_1

```
tant que (true) faire
    // section non critique
    flag[1] ← true
    turn ← 0
    tant que (flag[0]) && (turn = 0) faire
        rien
    // section critique
    flag[1] ← false
    // section non critique
```

On suppose que les deux *threads* souhaitent accéder à la ressource partagée et, notamment, entrer dans la section critique.

Question 1. Montrer que l'algorithme satisfait la contrainte d'*exclusion mutuelle*.

Question 2. Montrer qu'il satisfait la contrainte d'*absence de famine*.

Question 3. Montrer qu'il satisfait la contrainte d'*absence d'interblocage*.

Question 4. Montrer qu'il satisfait la contrainte d'*attente bornée*.

Annexes

Règles de la déduction naturelle

Règles de →

$$\frac{\Gamma, \varphi_1 \vdash \varphi_2}{\Gamma \vdash \varphi_1 \to \varphi_2} \to_i \qquad \frac{\Gamma \vdash \varphi_1 \to \varphi_2 \qquad \Gamma \vdash \varphi_1}{\Gamma \vdash \varphi_2} \to_e$$

Règles de ∧

$$\frac{\Gamma \vdash \varphi_1 \qquad \Gamma \vdash \varphi_2}{\Gamma \vdash \varphi_1 \wedge \varphi_2} \wedge_i \qquad \frac{\Gamma \vdash \varphi_1 \wedge \varphi_2}{\Gamma \vdash \varphi_1} \wedge_e \qquad \frac{\Gamma \vdash \varphi_1 \wedge \varphi_2}{\Gamma \vdash \varphi_2} \wedge_e$$

Règles de ∨

$$\frac{\Gamma \vdash \varphi_1}{\Gamma \vdash \varphi_1 \vee \varphi_2} \vee_i \qquad \frac{\Gamma \vdash \varphi_2}{\Gamma \vdash \varphi_1 \vee \varphi_2} \vee_i \qquad \frac{\Gamma \vdash \varphi_1 \vee \varphi_2 \qquad \Gamma, \varphi_1 \vdash \psi \qquad \Gamma, \varphi_2 \vdash \psi}{\Gamma \vdash \psi} \vee_e$$

Règles de ¬

$$\frac{\Gamma, \varphi \vdash \bot}{\Gamma \vdash \neg\varphi} \neg_i \qquad \frac{\Gamma \vdash \neg\varphi \qquad \Gamma \vdash \varphi}{\Gamma \vdash \bot} \neg_e$$

Raisonnement par l'absurde

$$\frac{\Gamma, \neg\varphi \vdash \bot}{\Gamma \vdash \varphi} \textit{raa}$$

Tiers exclu

$$\frac{}{\Gamma \vdash \varphi \vee \neg\varphi} \textit{te}$$

Règle de ⊥

$$\frac{\Gamma \vdash \bot}{\Gamma \vdash \varphi} \bot_e$$

Règle de ⊤

$$\frac{}{\Gamma \vdash \top} \top_i$$

Règles de ∀

$$\frac{\Gamma \vdash \varphi \qquad x \text{ n'apparaît pas libre dans } \Gamma}{\Gamma \vdash \forall x\, \varphi} \forall_i \qquad \frac{\Gamma \vdash \forall x\, \varphi}{\Gamma \vdash \varphi^{\{x \leftarrow t\}}} \forall_e$$

Règles de ∃

$$\frac{\Gamma \vdash \varphi^{\{x \leftarrow t\}}}{\Gamma \vdash \exists x\, \varphi} \exists_i \qquad \frac{\Gamma \vdash \exists x\, \varphi \qquad \Gamma, \varphi \vdash \psi \qquad x \text{ n'apparaît libre ni dans } \Gamma \text{ ni dans } \psi}{\Gamma \vdash \psi} \exists_e$$

Threads et *mutex* OCaml

Fonction	Description
`Thread.create`	`('a -> 'b) -> 'a -> Thread.t` `Thread.create f v` crée un nouveau *thread* pour exécuter l'appel `f v`. La fonction s'exécute en même temps que le *thread* appelant. La valeur retour de `f` n'est pas utilisée.
`Thread.join`	`Thread.t -> unit` `Thread.join t` suspend l'exécution du *thread* appelant jusqu'à ce que `t` ait terminé son exécution.
`Mutex.create`	`unit -> Mutex.t` `Mutex.create ()` crée un nouveau *mutex*.
`Mutex.lock`	`Mutex.t -> unit` Un appel `Mutex.lock m` verrouille le *mutex* `m`. Un seul *thread* à la fois peut verrouiller ce *mutex*. Si un *thread* `t2` essaie de verrouiller `m` alors qu'il l'est déjà par `t1`, le *thread* `t2` est mis en attente jusqu'à ce que `t1` déverrouille `m`.
`Mutex.unlock`	`Mutex.t -> unit` Un appel `Mutex.unlock m` déverrouille le *mutex* `m`. Tous les *threads* suspendus parce qu'ils ont essayé de verrouiller `m` sont réveillés (pour tenter à nouveau de verrouiller `m`).