Exercice 1

1.

(a) $u_0 = 4$ et $u_1 = 7$. On a $u_{n+2} = u_{n+1} + 2u_n$, $\forall n \in \mathcal{N}$.

(b) On montre cela par récurrence sur n.

- initialisation : Les cas $n = 0$ et $n = 1$ terminent puisqu'ils s'agit des cas terminaux de cette fonction récursive.
- hérédité : Soit $n \geqslant 2$. On suppose que `u k` termine pour tout $k < n$.
  L'appel `u n` calcul alors `u (n-1)` et `u (n-2)` qui terminent avant de renvoyer la quantité $u_{n-1} + 2u_{n-2}$ d'où le résultat.

(c) Notons A_n le nombre d'opérations arithmétiques nécessaires au calcul de `u n`. On a alors la relation suivante :

$$A_0 = 0, \quad A_1 = 0, \quad A_{n+2} = A_{n+1} + A_n + 2, \quad \forall n \in \mathbb{N}$$

On pose $B_n = A_n + 2$ et on se rend compte que $B_{n+2} = B_{n+1} + B_n$ On peut donc trouver une formule pour B_n grâce au cours de mathématiques sur les suites récurrentes linéaires d'ordre 2. On en déduit alors A_n.

(d) On en déduite une complexité exponentielle.

2.

(a)

```
let rec u_term a b n =
   match n with
      | 0 -> a
      | _ -> u_term b (b+2*a) (n-1)
;;
```

(b)

```
let u2 n =
   u_term 4 7 n
;;
```

(c) Notons encore une fois A_n, le nombre d'opérations arithmétiques nécessaires au calcul de `u2 n`. On réalise cette fois que l'on a la relation $A_{n+1} = A_n + 2$ avec $A_0 = 0$. On en déduit $A_n = 2 * n$.

(d) On a donc une complexité linéaire $\mathcal{O}(n)$.

3.

```
let u3 n =
   let terme = ref 4 in
   let terme_suiv = ref 7 in
   let temp = ref 0 in
   for  i = 0 to n-1 do
      temp := !terme_suiv ;
      terme_suiv := !terme_suiv + !term ;
      terme := !temp
   done;
   !terme
;;
```

Exercice 2

1.

```
let indice_min_tableau tab =
   let n = Array.length tab in
   if n=0 then
      failwith "tableau vide"
   else
      let indice = ref 0 in
      for  i = 1 to n-1 do
         if tab.(i) < tab.(!indice) then
            indice := i
      done;
      !indice
;;
```

On montre la correction en démontrant l'invariant de boucle suivant :

Inv_i : "Au début du $i^{\text{ième}}$ tour de boucle for, la variable `indice` contient l'indice du premier des plus petits éléments du tableau entre les indice 0 et i".

- initialisation : Le cas $i = 1$ correspond à une sous partie du tableau constituée d'un seul élément.
- hérédité : Soit $i \geqslant 1$. On suppose l'invariant au rang i. Alors un tour de boucle for compare l'élément en position `indice` et l'élément en position `i` et change `indice` si et seulement si on a détecté un élément strictement plus petit. D'où le résultat.

2.

```
let rec max_liste liste =
   match liste with
      | [] -> failwith "liste vide"
      | [a] -> a
      | a::tl ->
         let temp = max_liste tl in
         if temp < a then
            a
         else
            temp
;;
```

On montre la correction par récurrence sur n la taille de la liste d'entrée.

- initialisation : Le cas $n = 1$ correspond à un cas terminal ou l'on renvoie l'unique élément de la liste.
- hérédité : Soit $n \geqslant 1$. On suppose le résultat au rang 1. L'appel `max_liste liste` sur une liste de taille $n + 1$ va faire un appel récursif qui, par hypothèse de récurrence, va enregistrer le maximum de la queue de `liste` dans une variable `temp` et en comparant cette dernière à la tête de la liste `liste` va renvoyer le bon résultat.

Exercice 3

1.

```
let rec insertion x l =
   match l with
      | [] -> [x]
      | a::tl when a<x -> a::(insertion x tl)
      | _ -> x::l
;;
```

2.

```
let rec tri_insertion l =
   match l with
     | [] -> []
     | a::tl -> insertion a (tri_insertion
      tl)
;;
```

3.
- meilleur cas : La liste est déjà triée. Chaque appel de la fonction `insertion` ne fera alors qu'une seule comparaison et la complexité sera alors linéaire en la taille de la liste.
- pire cas : Si la liste est triée dans l'ordre décroissant alors chaque appel de la fonction `insertion` devra réinsérer l'élément considéré à la fin de la liste de sortie en construction ce qui mènera à une complexité quadratique.

Exercice 4

1.
   (a) On montre cela par induction :
   - cas de base : Le résultat est évidemment vrai pour le mot vide et les éléments de $\Sigma\backslash\{[,]\}$ puisqu'il n'y a pas de parenthèses dedans.
   - cas d'induction : Soient $u, v \in E$, on suppose le résultat acquis sur u et v. Alors $p([u]v) = 1 + p(u) - 1 + p(v) = 0$. D'où le résultat.

   (b) On montre également cela par induction :
   - cas de base : Le résultat est évidemment vrai pour le mot vide et les éléments de $\Sigma \setminus \{[,]\}$.
   - cas d'induction : Soient $u, v \in E$, on suppose le résultat acquis sur u et v. Alors un préfixe de $[u]v$ est constitué soit de $[$ avec un préfixe de u soit de $[u]$ avec un préfixe de v. Dans tous les cas le résultat est immédiat.
2. On peut montrer cela par récurrence sur la taille n du mot u :
   - initialisation : Les cas $n = 0$ et $n = 1$ sont évident puisqu'il s'agit soit du mot vide (pour $n = 0$) soit, par propriété (a) d'un élément de $\Sigma \setminus \{[,]\}$.
   - hérédité : Soit $n \geqslant 2$. On suppose le résultat acquis pour tout mot u de taille $k < n$. Soit u un mot de taille n. On considère alors le plus petit préfixe t de u tel que $p(t) = 0$. Ce dernier existe en particulier parce que $p(u) = 0$ par propriété (a). On peut alors décomposer $u = tv$. De plus t doit finir par un parenthèse fermante, sinon on aurait un préfixe strict de t qui serait soit de poids nul soit qui contredirait la propriété (b). Ainsi on peut écrire $t = [s]$ et $u = [s]v$. Or, le respect de u des propriétés (a) et (b) force le respect de ces mêmes propriétés par s et v. D'où le résultat.
3. Il suffit alors de définir un compteur, initialisé à 0. On parcours le mot de gauche à droite en faisant $+1$ si l'on tombe sur une parenthèse ouvrante et -1 si l'on tombe sur une fermante. Si à aucun moment le compteur n'est devenu strictement négatif et qu'à la fin il vaut 0 on a affaire à un mot bien parenthésé.

Exercice 5

1. $F \equiv (A_1 \wedge B_1 \wedge C_1) \vee (\neg A_1 \wedge B_1 \wedge C_1) \vee (\neg A_1 \wedge \neg B_1 \wedge C_1) \vee (\neg A_1 \wedge \neg B_1 \wedge \neg C_1)$
2. $A_1 \equiv D \vee G$, $B_1 \equiv \neg D$ et $C_1 \equiv P \wedge G$.
3.
$$\begin{aligned} F \equiv &((D \vee G) \wedge \neg D \wedge P \wedge G) \vee (\neg D \wedge \neg G \wedge \neg D \wedge P \wedge G) \vee (\neg D \wedge \neg G \wedge D \wedge P \wedge G) \vee (\neg D \wedge \neg G \wedge D \wedge \neg(P \wedge G)) \\ \equiv &((D \vee G) \wedge \neg D \wedge P \wedge G) \\ \equiv &\neg D \wedge P \wedge G \end{aligned}$$

   Ainsi, un kjalt n'a pas de dard mais possède des griffes et des pinces.
4. $F \equiv (A_2 \wedge B_2 \wedge A_3) \vee (\neg A_2 \wedge \neg B_2 \wedge \neg A_3)$
5. $A_2 \equiv M \wedge \neg J$, $B_2 \equiv \neg V$ et $A_3 \equiv J \Rightarrow V$.

6.

M	V	J	A_2	B_2	A_3	F
0	0	0	0	1	1	0
0	0	1	0	1	0	0
0	1	0	0	0	1	0
0	1	1	0	0	1	0
1	0	0	1	1	1	1
1	0	1	0	1	0	0
1	1	0	1	0	1	0
1	1	1	0	0	1	0

On en déduit qu'un lyop ne peut qu'être mauve.