Exercice 1

1. On montre ce résultat par induction :
   - c'est vrai pour les trois cas de bases qui sont soit $\varnothing$, soit ε, soit $a \in \Sigma$;
   - l'induction se fait naturellement en remarquant que :
     * $e|\varnothing \simeq \varnothing|e \simeq e$;
     * $e\varnothing \simeq \varnothing e \simeq \varnothing$;
     * $\varnothing^* \simeq \varepsilon$.
2. On obtient :

```
let rec normaliser = function
    | Concat (e, f) -> let en = normaliser e and fn = normaliser f in
                       if en = Vide || fn = Vide then Vide
                       else Concat (en, fn)
    | Union (e, f)  -> let en = normaliser e and fn = normaliser f in
                       if en = Vide then fn
                       else if fn = Vide then en
                       else Union (en, fn)
    | Etoile e      -> let en = normaliser e in
                       if en = Vide then Epsilon
                       else Etoile en
    | e                                  -> e;;
```

   Notons qu'on a traité les trois cas de base en dernier car on ne modifie pas l'expression.
3. Il suffit de tester si l'expression normalisée est vide ou non.

```
let est_vide e = normaliser e = Vide;;
```

Exercice 2

1. Un simple parcours d'arbre. On peut ici regrouper des cas.

```
let rec nombre_lettres = function
    | Vide | Epsilon             -> 0
    | Lettre _                   -> 1
    | Union(e, f) | Concat(e, f) -> nombre_lettres e + nombre_lettres f
    | Etoile e                   -> nombre_lettres e;;
```

2. On commence par créer un tableau de caractères et une référence qui permettra de faire l'association des numéros avec une lettre. Pour chaque lettre rencontrée, on modifie le tableau et on incrémente la référence, avant de renvoyer l'expression modifiée.

```
let lineariser e =
    let k = nombre_lettres e in
    let t = Array.make k 'a' in
    let i = ref 0 in
    let rec aux_lin = function
        | Vide          -> Vide
        | Epsilon       -> Epsilon
        | Lettre a      -> t.(!i) <- a; incr i; Lettre (!i - 1)
        | Union (e, f)  -> Union (aux_lin e, aux_lin f)
        | Concat (e, f) -> Concat (aux_lin e, aux_lin f)
        | Etoile e      -> Etoile (aux_lin e) in
    aux_lin e, t;;
```

3. On a :

   - $V(\varnothing) = \emptyset$;
   - $V(\varepsilon) = \{\varepsilon\}$;
   - $\forall a \in \Sigma$, $V(a) = \emptyset$;
   - $V(e|e') = V(e) \cup V(e')$;
   - $V(ee') = V(e) \cap V(e')$;
   - $V(e^\star) = \{\varepsilon\}$.

   - $P(\varnothing) = \emptyset$;
   - $P(\varepsilon) = \emptyset$;
   - $\forall a \in \Sigma$, $P(a) = \{a\}$;
   - $P(e|e') = P(e) \cup P(e')$;
   - $P(ee') = P(e) \cup V(e)P(e')$;
   - $P(e^\star) = P(e)$.

   - $S(\varnothing) = \emptyset$;
   - $S(\varepsilon) = \emptyset$;
   - $\forall a \in \Sigma$, $S(a) = a$;
   - $S(e|e') = S(e) \cup S(e')$;
   - $S(ee') = S(e') \cup V(e')S(e)$;
   - $S(e^\star) = S(e)$.

   - $F(\varnothing) = \emptyset$;
   - $F(\varepsilon) = \emptyset$;
   - $\forall a \in \Sigma$, $F(a) = \emptyset$;
   - $F(e|e') = F(e) \cup F(e')$;
   - $F(ee') = F(e) \cup F(e') \cup S(e)P(e')$;
   - $F(e^\star) = F(e) \cup S(e)P(e)$.

4. On utilise les formules précédentes.

```
let rec mot_vide = function
    | Vide | Lettre _    -> false
    | Epsilon | Etoile _ -> true
    | Union (e, f)       -> mot_vide e || mot_vide f
    | Concat (e, f)      -> mot_vide e && mot_vide f;;
```

5. Pour l'union, on se contente de faire des ou logiques.

```
let union t1 t2 =
    let n = Array.length t1 in
    assert (n = Array.length t2);
    let t = Array.make n false in
    for i = 0 to n - 1 do
        t.(i) <- t1.(i) || t2.(i)
    done;
    t;;
```

6. On utilise les formules précédentes.

```
let rec prefixes n = function
    | Vide | Epsilon -> Array.make n false
    | Lettre i       -> let t = Array.make n false in
                        t.(i) <- true; t
    | Union(e, f)    -> union (prefixes n e) (prefixes n f)
    | Concat(e, f)   -> let t1 = prefixes n e in
                        if mot_vide e then union t1 (prefixes n f)
                                      else t1
    | Etoile e       -> prefixes n e;;
```

   De même pour les suffixes.

```
let rec suffixes n = function
    | Vide | Epsilon -> Array.make n false
    | Lettre i       -> let t = Array.make n false in
                        t.(i) <- true; t
    | Union(e, f)    -> union (suffixes n e) (suffixes n f)
    | Concat(e, f)   -> let t2 = suffixes n f in
                        if mot_vide f then union (suffixes n e) t2
                                      else t2
    | Etoile e       -> suffixes n e;;
```

7. On commence par écrire une fonction d'union de matrice pour alléger le code :

```
let union_matrice m1 m2 =
    let n = Array.length m1 in
    let m = Array.make_matrix n n false in
    for i = 0 to n - 1 do
        m.(i) <- union m1.(i) m2.(i)
    done;
    m;;
```

Dès lors, on peut faire le calcul inductif (les points délicats sont la concaténation et l'étoile).

```
let rec facteurs n = function
    | Vide | Epsilon | Lettre _ -> Array.make_matrix n n false
    | Union (e, f)              -> union_matrice (facteurs n e) (facteurs n f)
    | Concat (e, f)             ->
            let m = union_matrice (facteurs n e) (facteurs n f) in
            let suff = suffixes n e and pref = prefixes n f in
            for i = 0 to n - 1 do
                for j = 0 to n - 1 do
                    m.(i).(j) <- m.(i).(j) || (suff.(i) && pref.(j))
                done
            done;
            m
    | Etoile e                  ->
            let m = facteurs n e in
            let suff = suffixes n e and pref = prefixes n e in
            for i = 0 to n - 1 do
                for j = 0 to n - 1 do
                    m.(i).(j) <- m.(i).(j) || (suff.(i) && pref.(j))
                done
            done;
            m;;
```

8. On applique l'algorithme de Berry-Sethi :
   - on linéarise ;
   - on calcule V, P, S et F pour l'expression obtenue ;
   - on crée l'automate standard local associé à l'expression linéarisée, en mettant directement les bonnes étiquettes de transitions.

   Il faut ici faire attention : l'état initial rajouté doit être numéroté 0, ce qui décale tous les autres états.

```
let glushkov e =
    let elin, t = linearisation e in
    let n = Array.length t in
    let ve = mot_vide elin and
        pe = prefixes n elin and
        se = suffixes n elin and
        fe = facteurs n elin in
    let finaux = Array.make (n + 1) false and
        delta = Array.make (n + 1) [] in
    finaux.(0) <- ve;
    let transition q a q' = delta.(q) <- (a, q') :: delta.(q) in
    for q = 1 to n do
        if pe.(q - 1) then transition 0 t.(q - 1) q;
        finaux.(q) <- se.(q - 1);
        for q' = 1 to n do
            if fe.(q - 1).(q' - 1) then transition q t.(q' - 1) q'
        done
    done;
    {finaux = finaux; delta = delta};;
```

Exercice 3

1. Il s'agit de parcourir toutes les transitions depuis un des états de t. La fonction `traiter` permet de modifier le tableau de retour si on trouve une transition étiquetée par la bonne lettre.

```
let delta_ens aut x a =
    let n = Array.length x in
    let x' = Array.make n false in
    for q = 0 to n - 1 do
        let traiter (b, q') = x'.(q') <- x'.(q') || a = b in
        if x.(q) then List.iter traiter aut.delta.(q)
    done;
    x';;
```

2. On applique la fonction précédente pour chaque lettre du mot.

```
let delta_etoile aut x u =
    let x' = ref x in
    for i = 0 to String.length u - 1 do
        x' := delta_ens aut !x' u.[i]
    done;
    !x';;
```

3. Il s'agit ici de calculer $\Delta^*(\{0\}, u)$ et de vérifier si cet ensemble contient un état final (ce que fait la boucle `while`).

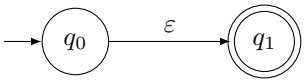
```
let reconnu aut u =
    let n = Array.length aut.finaux in
    let x = Array.make n false in
    x.(0) <- true;
    let x' = delta_etoile aut x u in
    let i = ref 0 in
    while !i < n && (not x'.(!i) || not aut.finaux.(!i)) do
        incr i
    done;
    !i < n;;
```

4.

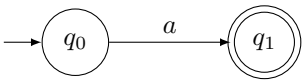
Exercice 4

1. On propose :
   - pour $\emptyset$:

   - pour ε :



   - pour $a \in \Sigma$:



2. – $L(A) \cup L(B)$: on crée une copie de A et de B et on rajoute un état initial q_0 et un état final q_f. On rajoute des ε-transition depuis q_0 vers les états initiaux de A et B et des ε-transitions des états finaux de A et B vers q_f.
   – $L(A)L(B)$: on crée une copie de A et de B et on rajoute une ε-transition entre l'état final de A et l'état initial de B. On considère que l'état initial de l'automate est celui de A et l'état final est celui de B.
   – $L(A)^*$: on crée une copie de A et on rajoute un état initial q_0 et un état final q_f. On rajoute des ε-transition depuis q_0 vers l'état initial de A et vers q_f, et des ε-transitions de l'état final de A vers son état initial et q_f.
3. On applique la construction proposée ci-dessus :

```
let rec thompson = function
    | Vide
       -> let initial = {lettre = None; sortie1 = None; sortie2 = None}
          and final = {lettre = None; sortie1 = None; sortie2 = None} in
          {initial; final}
    | Epsilon
       -> let initial = {lettre = None; sortie1 = None; sortie2 = None}
          and final = {lettre = None; sortie1 = None; sortie2 = None} in
          initial.sortie1 <- Some final;
          {initial; final}
    | Lettre a
       -> let initial = {lettre = Some a; sortie1 = None; sortie2 = None}
          and final = {lettre = None; sortie1 = None; sortie2 = None} in
          initial.sortie1 <- Some final;
          {initial; final}
    | Concat(e1, e2)
       -> let t1 = thompson e1 and t2 = thompson e2 in
          t1.final.sortie1 <- Some t2.initial;
          {initial = t1.initial; final = t2.final}
    | Union(e1, e2)
       -> let t1 = thompson e1 and t2 = thompson e2 in
          let initial = {lettre = None;
                         sortie1 = Some t1.initial;
                         sortie2 = Some t2.initial}
          and final = {lettre = None; sortie1 = None; sortie2 = None} in
          t1.final.sortie1 <- Some final;
          t2.final.sortie1 <- Some final;
          {initial; final}
    | Etoile e
       -> let th = thompson e in
          let final = {lettre = None; sortie1 = None; sortie2 = None} in
          let initial = {lettre = None;
                           sortie1 = Some th.initial;
                           sortie2 = Some final} in
          th.final.sortie1 <- Some th.initial;
          th.final.sortie2 <- Some final;
          {initial; final};;
```

4. Il suffit de tester les deux transitions sortantes pour chaque état.

```
let reconnu_thompson th u =
    let n = String.length u in
    let rec accepte q i = match q.lettre, q.sortie1, q.sortie2 with
        | None, None, None       -> q == th.final && i = n
        | Some a, Some p, None   -> i < n && u.[i] = a && accepte p (i + 1)
        | None, Some p, None     -> accepte p i
        | None, Some p1, Some p2 -> accepte p1 i || accepte p2 i
        | _ -> failwith "L'automate n'est pas normalisé" in
    accepte th.initial 0;;
```

5. Le problème est l'existence d'un cycle étiqueté par ε : le backtracking tournera en boucle sur les mêmes transitions.
6. On remarque que le problème se produit lorsqu'on calcule l'étoile d'une expression régulière dont l'interprétation contient le mot vide, car il existe alors un calcul de l'état initial à l'état final de l'automate de Thompson étiqueté par ε. On remarque que toute expression régulière e est

équivalente à une expression de la forme :

- e si $V(e) = \emptyset$;
- $e'|\varepsilon$ avec $\mathcal{L}(e') = \mathcal{L}(e) \setminus \{\varepsilon\}$ sinon.

On remarque notamment que si e et f sont des expressions régulières dont l'interprétation ne contient pas ε, alors :

- $e(f|\varepsilon) \simeq ef|e$, $(e|\varepsilon)f \simeq ef|f$, $(e|\varepsilon)(f|\varepsilon) \simeq (ef|e|f)|\varepsilon$;
- $e|(f|\varepsilon) \simeq (e|\varepsilon)|f \simeq (e|\varepsilon)|(f|\varepsilon) \simeq (e|f)|\varepsilon$;
- $(e|\varepsilon)^* \simeq ee^*|\varepsilon$.

On peut appliquer ce processus récursivement pour obtenir une expression régulière dont on est sûr qu'elle ne contient pas de cycle étiqueté par ε.

```
let rec reecriture = function
    | Vide          -> Vide
    | Lettre a      -> Lettre a
    | Epsilon       -> Union (Vide, Epsilon)
    | Concat (e, f) -> begin match reecriture e, reecriture f with
        | Union (e', Epsilon), Union (f', Epsilon) ->
              Union (Union (Union (Concat (e', f'), e'), f'), Epsilon)
        | Union (e', Epsilon), f'                  ->
              Union (Concat (e', f'), f')
        | e', Union (f', Epsilon)                  ->
              Union (Concat (e', f'), e')
        | e', f'                                   ->
              Concat (e', f') end
    | Union (e, f)  -> begin match reecriture e, reecriture f with
        | Union (e', Epsilon), Union (f', Epsilon)
        | Union (e', Epsilon), f'
        | e', Union (f', Epsilon)                  ->
              Union (Union (e', f'), Epsilon)
        | e', f'                                   ->
              Union (e', f') end
    | Etoile e      -> match reecriture e with
        | Union (e', Epsilon) | e' ->
              Union (Concat (e', Etoile e'), Epsilon);;
```

Dès lors, on peut créer l'automate de Thompson par :

```
let thompson_bis e = thompson (reecriture e);;
```