1 Une preuve de terminaison

1. (a) La réflexivité ne pose guère de problème. Pour l'antisymétrie comme pour la transitivité, il s'agit d'uue fascinante étude de cas... laissée au lecteur consciencieux.

   (b) Pour $N \in \mathbb{N}$, on ontrer la proposition $\mathcal{P}(N)$: "Il n'existe pas de suite infinie strictement décroissante pour $\leq_L$ de premier terme (N, p)".
   - $\mathcal{P}(0)$ ne pose guère de problème : une suite strictement décroissante issue de $(0, p)$ possède au plus p termes.
   - Fixons $N \in \mathbb{N}^*$, et supposons $\mathcal{P}(k)$ vérifiée pour tout $k \in [\![0, N-1]\!]$. Supposons par l'absude qu'il existe une suite infinie $((\alpha_n, \beta_n))_{n\in\mathbb{N}}$ strictement décroissante avec $\alpha_0 = N$. La suite d'entiers $(\alpha_n)_{n\in\mathbb{N}}$ ne peut être constante (sans quoi on a une suite $(\beta_n)_{n\in\mathbb{N}}$ d'entiers positifs strictement décroissante, ce qui n'est guère raisonnable). Il existe donc un rang n_0 tel que $\alpha_{n_0} < N$. Maizalors, la suite $((\alpha_n, \beta_n))_{n\geq n_0}$ est infinie strictement décroisante issue de $(\alpha_{n_0}, *)$, contredisant $\mathcal{P}(\alpha_{n_0})$, et fournissant une absurdité.
     Ceci prouve $\mathcal{P}(N)$.
   - Le principe de récurrence permet de conclure.

   (c) Si $\alpha \geq 1$ et $N \geq 1$, alors on peut trouver une suite strictement décroissante issue de (α, β) et de longueur $\geq N$:

   $$(\alpha_\beta) >_L (0, N+1) > (0, N) >_L \cdots >_L (0, 1) >_L (0, 0).$$

   Ceci répond à la question.

2. (a) Si le calcul `foo n` termine, le résultat retourné sera le nombre d'appels récursifs qui auront eu lieu pendant le calcul (en ne comptant pas le premier appel "extérieur". La preuve est immédiate par récurrence (forte) sur n.

   (b) Si $n \leq 1$, le calcul termine directement (et retourne 0). Sinon, supposons que $n > 1$, et que le calcul `foo n` ne termine pas. Il y a alors une infinité d'appels récursifs emboîtés. Notons n_k la valeur sur laquelle est exécutée le k-ième appel. Si n_k est pair, alors $n_{k+1} = \dfrac{n_k}{2}$, donc on supprime un zéro à droite dans la représentation binaire. Si n_k est impair, on a $n_{k+1} = n_k$. Dans ce cas, si on regarde la représentation binaire de n_k (pour $n_k \neq 2^r - 1$), on a : $n_k = \overline{1w01^s}$ et $n_{k+1} = \overline{1w10^s}$, donc la distance entre le premier et le dernier "un" a descendu strictement. Si on note α_k la distance entre le premier et le dernier "un" de la représentation binaire de n_k et β_k le nombre de "zéros" (2-valuation de n_k), on a alors $(\alpha_{k+1}, \beta_{k+1}) <_L (\alpha_k, \beta_k)$. On obtient alors une suite infinie strictement décroissante pour $<_L$: absurde.

   (c) A chaque appel récursif, on supprime un bit (0), ou on décale le 1 le plus à droite d'un certain nombre de crans, sans toucher à la longueur de la représentation binaire (sauf si $n = 2^k - 1$). Ainsi, si on part de l'entier $n = \overline{1w}$, avec $|w| = k-1$, alors il y aura $(k-1)+1 = k$ opérations $n \leftarrow \dfrac{n}{2}$ (le "+1" vient du fait qu'à une étape (et une seule) le nombre de bits augmente de 1), et "un certain nombre" d'opérations $n \leftarrow n+1$ (fonction des regroupements initiaux des "un" dans la représentation binaire de n). Puisque dans ces opérations, la distance entre le premier et le dernier 1 est diminuée strictement, et qu'elle vaut au plus $k-1$ au départ, il y aura au plus $k-1$ telles opérations. On obtient donc au plus $k + (k-1) = 2k-1$ appels récursifs avant d'atteindre $n = 1$.
   Maintenant, $64.1024 < 91171 < 128.1024$, donc $91171 = \overline{1w}$ avec $|w| = 16$ (donc $k = 17$), et les entiers ≤ 91171 ont une représentation $\overline{1w}$ avec $|w| \leq 16$, donc le nombre d'appels récursifs sera $\leq 2.17 - 1 = 33$. Si $n = 64.1024 + 1 = 2^{16} + 1$, il y aura effectivement 33 appels récursifs, qui est donc le maximum recherché. L'expérience confirme !

   ```
   #let m=ref 0 in for i=1 to 91171 do m:=max !m (foo i) done; !m;;
   - : int = 33
   #foo 65537;;
   - : int = 33
   ```

   (d) 33. Haha !

2 Calcul du langage d'un automate

2.1 Généralités

1. On réalise seulement la simplification `Plus(Vide,e)<-e` (et son symétrique) :

```
let additionner =function
  | (Vide,e)->e | (e,Vide)->e | (e1,e2)->Plus(e1,e2);;
(*additionner : 'a expr_rat * 'a expr_rat -> 'a expr_rat = <fun>*)
```

2. Pour la concaténation, on regarde si l'un des deux termes vaut `Vide` ou bien `Eps`. Pour étoiler, on regarde si l'expression est `Vide, Eps` ou encore de la forme `Etoile(e)` :

```
let concatener =function
  | (Vide,_)->Vide | (_,Vide)->Vide
  | (Eps,e)->e | (e,Eps)->e
  | (e1,e2)->Concat(e1,e2);;
(*concatener : 'a expr_rat * 'a expr_rat -> 'a expr_rat = <fun>*)
let etoiler=function
  | Vide -> Eps | Eps -> Eps | Etoile(e)->Etoile(e) | e->Etoile(e);;
(*etoiler : 'a expr_rat -> 'a expr_rat = <fun>*)
```

3. Je n'en demandais pas autant que ce qui suit :

```
let rec ecrire_exp_rat_general action=function
  | Vide -> print_string "Vide"
  | Eps -> print_string "Eps"
  | Lettre(a) -> action a
  | Etoile(Lettre l)-> action l;print_string "*"
  | Etoile(e)->print_string "(";
      ecrire_exp_rat_general action e; print_string ")*"
  | Concat(Plus(e1,e2),Plus(e3,e4))->print_string "(";
      ecrire_exp_rat_general action (Plus(e1,e2));print_string ")(";
      ecrire_exp_rat_general action (Plus(e3,e4));print_string ")"
  | Concat(Plus(e1,e2),e)->print_string "(";
      ecrire_exp_rat_general action (Plus(e1,e2));print_string ")";
      ecrire_exp_rat_general action e;
  | Concat(e,Plus(e3,e4))->
      ecrire_exp_rat_general action e;print_string "(";
      ecrire_exp_rat_general action (Plus(e3,e4));print_string ")"
  | Concat(e1,e2)->ecrire_exp_rat_general action e1;
      ecrire_exp_rat_general action e2
  | Plus(e1,e2)->ecrire_exp_rat_general action e1;
      print_string "+";ecrire_exp_rat_general action e2;;
ecrire_exp_rat_general : ('a -> unit) -> 'a expr_rat -> unit = <fun>
```

2.2 Méthode d'Arden

1. Déjà, $L = K^*M$ est bien solution, puisque $K(K^*M)+M = (K.K^*+\varepsilon)M = K^*M$. Supposons donc maintenant que $L = KL + M$. Déjà, $M \subset KL + M = L$, donc $M \subset L$, maizalors $KL \subset KL \subset L$, puis par récurrence imédiate, tous les K^nM sont inclus dans L, donc $K^*M \subset L$. Pour l'autre inclusion, on montre par récurrence avec prédécesseurs : "si w est de longueur n et est dans L, alors il est dans K^*M" : cette proposition est bien vérifiée pour $n = 0$ (si $\varepsilon \in L = KL + M$, alors il est forcément dans M puisque les mots de KL ont une longueur ≥ 1, donc $\varepsilon \in K^*M$). Si la proposition est vérifiée jusqu'au rang n et $w \in L$ est de longueur $n+1$, alors il est soit dans M, et c'est gagné, soit dans KL, et alors il se décompose $w = kl$ avec $k \in K$ et $l \in L$. Comme $k \neq \varepsilon$, l est de longueur $\leq n$, or il est dans L, donc dans K^*M, donc $w \in K^+M$, et c'est gagné.
2. (a) $\mathcal{L}(\mathcal{A})$ est clairement la réunion des L_{i_0}, pour i_0 décrivant I (états initiaux).

(b) Déjà, $\varepsilon \in L_i$ si et seulement si $i \in F$. Ensuite, si $w = \alpha w_1 \in L_i$, alors il existe un chemin indexé par αw_1 allant de i à un état final. Si on note j le premier état visité après la lecture de α, alors w étiquette un chemin de j vers un final, donc $w_1 \in L_j$, puis $w \in \alpha L_j$. Réciproquement, si $w = \alpha w_1$, avec $(i, \alpha, j) \in \Delta$ et $w_1 \in L_j$, alors $w \in L_i$. Ceci montre la proposition réclamée.

3. $L_1 = 0L_2$, $L_2 = 0L_2 + 1L_3$ et $L_3 = \varepsilon + 0L_1$, donc $L_2 = 0L_2 + 10L_1 + 1$ (la première ardenisation pour L_3 est inutile !), donc $L_2 = 0^*(10L_1 + 1)$, puis $L_1 = 00^*10L_1 + 00^*1$, et enfin

$$\mathcal{L}(\mathcal{A}) = L_1 = (00^*10)^*00^*1$$

(inutile de faire la dernière phase du calcul pour trouver L_2 et L_3 !).

4. (a)
```
Pour l allant de n à 1
      (* Ardeniser sur la ligne l*)
  M.(l)<-(K.(l).(l))*.(M.(l));
  Pour i=1 allant de 1 à l-1  : K.(l).(i)<-(K.(l).(l))*.K.(l).(i)
  K.(l).(l)<-Vide; (* pas indispensable, mais bon... *)
      (* reporter dans les equations precedentes *)
  Pour ligne allant de 1 à l-1
    Pour i allant de 1 à l-1 : K.(ligne).(i)<-K.(ligne).(i)+K.(ligne).(l).K.(l).(i)
    M.(ligne)<-M.(ligne)+(K.(ligne).(l)).(M.(l))
    K.(ligne).(l)<-Vide
```

(b) Au départ, les $K_{i,i}$ ne contiennent que des lettres (éventuellement aucune). Ensuite, évolutions ne feront jamais aparaître ε : la seule fois où une étoile apparaît, il s'agit de $K^*_{l,l}K_{l,i}$, et grâce à $K_{l,i}$, le mot vide ne sera pas présent ans la concaténation. Ceci justifie les Ardenisations.

(c) Il y a plus difficile...

```
Pour l allant de 2 à n
  Pour i allant de 1 à l-1
     M.(l)<-M.(l)+(K.(l).(i)).M.(i)
```

5. (a) On initialise à `Eps` tous les L_i pour tous les états finaux i, et on laisse les autres à `Vide`. C'est donc `a.finaux` qui sera donné à manger à `initialise_M`.

```
let rec initialise_M m=function
  | []->();
  | f::reste->m.(f)<-Eps; initialise_M m reste;;
(*initialise_M : 'a expr_rat vect -> int list -> unit = <fun>*)
```

(b) Pour chaque transition (i, α, j), on place la lettre α dans $K_{i,j}$. Il suffit donc de traiter les transitions les unes après les autres (c'est `a.transitions` qui sera donné en paramètre :

```
let rec initialise_K k=function
  | []->();
  | (d,l,f)::reste->
      k.(d).(f)<-additionner(Lettre l,k.(d).(f));
      initialise_K k reste;;
(*initialise_K : 'a expr_rat vect vect -> (int * 'a * int) list -> unit = <fun>*)
```

(c) On additionne les L_i, avec i décrivant les états initiaux. Ces derniers seront donc transmis, en plus du tableau `M`, qui contient les expressions rationnelles des L_i.

```
let rec reconstituer_L m=function
  | [] -> Vide
  | [f]-> m.(f)
  | f::reste->additionner(m.(f),reconstituer_L m reste);;
(*reconstituer_L : 'a expr_rat vect -> int list -> 'a expr_rat = <fun>*)
```

(d) Ben, on recolle les morceaux !

```
let arden a=
  let n=a.Q in
  let K=make_matrix (n+1) (n+1) Vide
  and M=make_vect (n+1) Vide
```

```
  in
    begin
      initialise_M M a.finaux;
      initialise_K K a.transitions;
      for l=n downto 1
      do
for i=1 to l-1
do  K.(l).(i)<-concatener(etoiler(K.(l).(l)),K.(l).(i)) done;
M.(l)<-concatener(etoiler(K.(l).(l)),M.(l));
K.(l).(l)<-Vide; (* pas indispensable, mais bon... *)
  (* reportons dans les equations precedentes *)
for ligne=1 to l-1
do
  for i=1 to l-1
  do
    K.(ligne).(i)<-additionner(
      K.(ligne).(i),
      concatener(K.(ligne).(l),K.(l).(i)))
  done;
  M.(ligne)<-additionner(M.(ligne),concatener(K.(ligne).(l),M.(l)));
  K.(ligne).(l)<-Vide
done;
      done;
      for l=2 to n
      do
for i=1 to l-1
do M.(l)<-additionner(M.(l),concatener(K.(l).(i),M.(i))) done
      done;
      reconstituer_L M a.initiaux
    end;;
(*arden : 'a automate -> 'a expr_rat = <fun>*)
```

Test :

```
ecrire_exp_rat_general print_int (arden auto1);;
(*0(00*10)*00*1- : unit = ()*)
```

2.3 Une autre méthode

1. $L_{i,j,0}$ est l'ensemble des mots permettant de passer de i à j SANS étape intermédiaire, donc en exécutant uue seule transition. Il sera donc constitué des lettres α telles $(i, \alpha, j) \in \Delta$.
2. Soit $k \in [\![0, n-1]\!]$. Pour aller de i à j via des états $\leq k+1$, on peut faire le chemin via seulement des états $\leq k$, ou bien aller une première fois à l'état $k+1$ (donc avec des intermédiaires $\leq k$), faire un certain nombre (éventuellement nul) de boucles allant de $k+1$ à $k+1$ via des états $\leq k$, puis aller de $k+1$ à j via des états $\leq k$. Ceci prouve :

$$\forall i, j \in [\![1, n]\!],\ \forall k \in [\![0, n-1]\!], \qquad L_{i,j,k+1} = L_{i,j,k} + L_{i,k+1,k}(L_{k+1,k+1,k})^* L_{k+1,j,k}.$$

   Pour le dessin, la lassitude ma gagne...
3. On regarde toutes les transitions...

```
let rec initialise k0=function
  | []-> ()
  | (i,a,j)::reste->
      k0.(i).(j)<-additionner(k0.(i).(j),Lettre(a));
      initialise k0 reste;;
(*initialise : 'a expr_rat vect vect -> (int * 'a * int) list -> unit = <fun>*)
```

4. On fait l'addition de tous les $L_{i,j,n}$, pour i décrivant I et j décrivant F :

```
let rec reconstituer_langage k initio fino=
  let rec un_initial i=
    (function
       | [] -> Vide
       | f::reste ->
   additionner(k.(i).(f),un_initial i reste))
  in match initio with
    | [] -> Vide
    | i::reste->additionner
(un_initial i fino,reconstituer_langage k reste fino);;
(*reconstituer_langage :
'a expr_rat vect vect -> int list -> int list -> 'a expr_rat = <fun>*)
```

5. Sans génie excessif, vu ce qui précède :

```
let pas_arden a=
  let n=a.Q in
  let K0=ref (make_matrix (n+1) (n+1) Vide)
  and K1=make_matrix (n+1) (n+1) Vide
  in
    begin
      initialise !K0 a.transitions;
      for k=0 to n-1
      do
for i=1 to n do for j=1 to n do
  K1.(i).(j)<-
  additionner(!K0.(i).(j),
      concatener(!K0.(i).(k+1),
 concatener(etoiler(!K0.(k+1).(k+1)),
    !K0.(k+1).(j))))
done done;
K0:=K1
      done;
    end;
    reconstituer_langage !K0 a.initiaux a.finaux;;
(*pas_arden : 'a automate -> 'a expr_rat = <fun>*)
```

2.4 Calcul des coûts

1. On peut donner une définition inductive de la taille d'une expression rationelle, qui va correspondre (à un facteur près) à la taille pour stocker en machine : $t(\varepsilon) = t(\emptyset) = t(\alpha) = 1$, $t(e^*) = 1 + t(e)$ et $t(e_1 + e_2) = t(e_1.e_2) = t(e_1) + t(e_2) + 1$.
2. Pour la deuxième méthode, il y a $n = |Q|$ étapes. A chacune de ces étapes, on calcule n^2 expressions. Ces expressions rationnelles ont une taille $O(4^k)$ à la k-ième étape. On récupère donc à la fin une expression rationnelle de taille $O(4^n)$, pour un temps de calcul de l'ordre de $n^2 \sum_{k=1}^{n} 4^k = O(n^2 4^n)$.
3. (a) Clairement, au début, $t(K_{i,j}) \leq 2\,|A|$ et $t(M_i) = 1$.
   (b) Hum... au moment de rédiger le corrigé, il apparait clairement que c'est abominable ! On doit pourvoir tout majorer très brutalement par des exponentielles... et encore, même ça, c'est sordide ! Attendons de voir les copies pour voir si vous avez dit des choses intéressantes ![1]
4. Et donc :
   (a) toujours rédiger le corrigé avant le DS...
   (b) c'était sacrément long, et il reste l'autre problème ! !

[1] Oui, bon, OK, je ne suis pas fier de moi...