Info A XULCR – 2019

Autour des sous-mots et des sur-mots

Olivier Brunet, Éric Detrez, Émeric Tourniaire

April 22, 2019

Question 1

a. Procédons par récurrence sur la taille des mots. Étant donné un plongement $p : [m] \to [n]$ de ua dans $u'a'$, deux cas sont possibles :

- si $p(m-1) = n-1$, alors nécessairement
$$a = (ua)[m-1] = (u'a')[n-1] = a'$$
et la restriction $p|_{[m-1]}$ est strictement croissante de $[m-1]$ dans $[n-1]$, i.e. c'est un plongement de u dans u', d'où $u \preccurlyeq u'$;
- sinon, $p\big([m]\big) \subseteq [n-1]$ et donc $ua \preccurlyeq u'$.

Réciproquement, il est clair que, de chaque cas de la disjonction de droite, on peut déduire que $ua \preccurlyeq u'a'$.

b. De la question précédente, on déduit le programme suivant, sachant que l'équivalence que l'on vient de montrer suppose que les deux mots comportent au moins une lettre. Pour raison d'efficacité, la disjonction précédente peut être transformée en deux cas *disjoints* :
$$(a \neq a' \text{ et } ua \preccurlyeq u') \text{ ou } (a = a' \text{ et } u \preccurlyeq u')$$
puisque si $ua \preccurlyeq u'$, alors nécessairement $u \preccurlyeq u'$.

```
let teste_sous_mot v u =
  let rec aux i j =
    if i = 0 then true
    else if j = 0 then false
    else if v.[i - 1] = u.[j - 1] then aux (j - 1) (i - 1)
    else aux j (i - 1)
  in aux (String.length v) (String.length u)
;;
```

Concernant la complexité, chaque appel récursif à `aux` se fait en temps constant, et puisque `i` décroit de 1 à chaque fois, le nombre d'appels est majoré par $|u|$. La complexité de la fonction est en $O\big(|u|\big)$.

1 Compter et construire

Question 2

a. On a bien $\binom{\texttt{abab}}{\texttt{ab}} = 3$, puisque c'est le nombre de façon de sélectionner dans `abab` un `a` puis un `b` :

$$\underline{\texttt{ab}}\texttt{ab} \qquad \underline{\texttt{a}}\texttt{ba}\underline{\texttt{b}} \qquad \texttt{ab}\underline{\texttt{ab}}$$

b. La donnée d'un plongement de a^m dans a^n revient à sélectionner m positions dans l'ensemble $[n]$ à n éléments. Il y a $\binom{n}{m}$ façons de faire, d'où :

$$\binom{a^n}{a^m} = \binom{n}{m}.$$

c. C'est une conséquence directe de la question **1.a.**, sachant que les ensembles de plongements de va dans u d'une part, et de v dans u d'autre part sont disjoints (du fait du cardinal de l'ensemble de départ).

Question 3

a. La terminaison de `nb_plongements` est assurée par le fait que les arguments `i` et `j` de la fonction auxilliaire `aux` sont des entiers positifs, et que les appels récursifs à cette même fonction font toujours décroître strictement `j`.

b. La correction de `nb_plongements` découle de la question **2.c.** et, plus généralement, de la question **1.a.** auquel on a ajouté les cas de base :

$$\binom{u}{\varepsilon} = 1 \qquad v \neq \varepsilon \implies \binom{\varepsilon}{v} = 0$$
$$\binom{ua}{va} = \binom{u}{va} + \binom{u}{v} \qquad a \neq a' \implies \binom{ua'}{va} = \binom{u}{va}$$

Ces différentes équations correspondent aux quatre branches de la fonction.

Question 4

a. Nous notons ici $T(i, j)$ à la place de $T(v, u)$ pour $i = |v|$ et $j = |u|$.

On a, dans le pire des cas (si $i \geq 1$, $j \geq 1$ et $u[i-1] = v[i-1]$)

$$T(i, j) \leq 1 + T(i, j-1) + T(i-1, j-1)$$

où la constante C tient compte des différents tests et opérations arithmétiques. De plus, ayant $T(i, 0) = 1$ (on effectue au plus deux tests donc, sans tenir compte de la récupération des longueurs des arguments, on a $A = 2$), il est clair par récurrence que :

$$\forall\, i, j \in \mathbf{N},\ T(i, j) \leq 2 \times 2^j - 1$$

puisque $T(i,0) \leq 1 = 2 \times 2^0 - 1$ et, pour l'hérédité :

$$\begin{aligned} T(i,j+1) &\leq 1 + T(i,j) + T(i-1,j) \\ &\leq 1 + 2 \times \left(2 \times 2^j - 1\right) = 2 \times 2^{j+1} - 1 \end{aligned}$$

En particulier, on a,

$$\forall\, i,j \in \mathbf{N},\, T(i,j) \leq 2 \times 2^j$$

soit, en termes de mots et en posant $C_1 = 2$,

$$\forall\, v,u \in A^\star,\, T(v,u) \leq 2^{|u|} \times C_1$$

b. Il n'est pas possible de majorer $T(v,u)$ par une fonction polynômiale de $\binom{u}{v}$ (et même par n'importe quelle fonction de $\binom{u}{v}$) puisque $T(v,u)$ peut prendre des valeurs arbitrairement grandes tout en ayant $\binom{u}{v} = 0$. On peut considérer par exemple des mots de la forme

$$u = \mathtt{a}^n \qquad v = \mathtt{ba}^m$$

puisque le mot u ne contient pas la lettre `b`, mais l'algorithme, en lisant les lettres de droite à gauche, doit au moins parcourir le mot u en entier.

c. En écrivant

$$\binom{u}{v} = 1 + 1 + 1 + \ \cdots\ + 1 + 1,$$

on remarque qu'il y a $\binom{u}{v}$ fois la constante 1, et $\binom{u}{v} - 1$ fois le signe $+$. Or, chacun de ces symboles correspond à un appel à la fonction `aux` (avec les cas `i = 0` pour 1, et `v.[i - 1] = u.[j - 1]` pour $+$). Cela implique que

$$T(v,u) \geq 2\binom{u}{v} - 1.$$

Question 5 On peut accélérer la fonction en sauvegardant les valeurs obtenus lors des appels récursifs dans une matrice d'entiers. Cela donne le programme suivant :

```
let nb_plongements_rapide (v : string) (u : string) =
  let len_u = String.length u
  and len_v = String.length v in
  let m = Array.make_matrix
        (len_v + 1) (len_u + 1) 0 in
  (* Si v = epsilon, (v parmi u) = 1 *)
  for j = 0 to len_u do
    m.(0).(j) <- 1
  done ;
  for j = 1 to len_u do
```

```
    for i = 1 to len_v do
      if v.[i - 1] = u.[j - 1]
      then
        m.(i).(j) <- m.(i).(j - 1) + m.(i - 1).(j - 1)
      else
        m.(i).(j) <- m.(i).(j - 1)
    done
  done ;
  m.(len_v).(len_u)
;;
```

La complexité est alors clairement en $O\big(|u| \times |v|\big)$ en temps comme en espace. En particulier, on a bien un temps polynomial en $|u| + |v|$.

Question 6

a. Remarquons tout d'abord que $\downarrow wav \subseteq (\downarrow wav) \cdot a \subseteq \downarrow wava$, ce qui assure l'inclusion

$$\downarrow wav \cup \big(\downarrow wav \setminus \downarrow w\big) \cdot a \subseteq \downarrow wava$$

Réciproquement, soit $u \in \downarrow wava \setminus \downarrow wav$. Il existe donc un plongement de u dans $wava$ mais il n'y a pas de plongement de u dans wav. Cela implique tout d'abord que la dernière lettre de u est a (car sinon, un plongement existe dans wav, toujours d'après la question **1.a.**). En particulier, en écrivant $u = u'a$, il y a un plongement de u' dans wav. Mais il n'y a pas de plongement de u dans wa (on en déduirait un plongement de u dans wav) et donc il n'y a pas de plongement de u' dans w. Ainsi, on a :

$$u \in (\downarrow wav \setminus \downarrow w) \cdot a.$$

On a donc montré que :

$$\downarrow wava = \downarrow wav \cup \big(\downarrow wav \setminus \downarrow w\big) \cdot a$$

b. Si $\downarrow wav \cap \big(\downarrow wav \setminus \downarrow w\big) \cdot a \neq \emptyset$, un mot u dans l'intersection se termine par un a (à cause du « $\cdot\, a$ »). Comme $u \in \downarrow wav$ et, en notant $u = u'a$, $u' \notin \downarrow w$, on a $u \notin \downarrow wa$. Cela entraîne que dans le plongement de u dans wav, le a final de u est nécessairement dans v, et donc que v contient la lettre a.

Réciproquement, si v contient la lettre a, alors on a $waa \in \downarrow wav$ ainsi que $wa \in \downarrow wa \setminus \downarrow w$, d'où

$$waa \in \downarrow wav \cap \big(\downarrow wav \setminus \downarrow w\big) \cdot a$$

En particulier, l'intersection est non vide.

En conclusion, l'union est disjointe si et seulement si le mot v contient la lettre a.

Question 7

a. Si la dernière lettre a d'un mot u apparaît au moins deux fois, on peut le mettre sous la forme $u = wava$ avec v qui ne contient pas la lettre a. Dans ce cas, d'après la question précédente, on a

$$\mathrm{Card}(\downarrow wava) = \mathrm{Card}(\downarrow wav) + \mathrm{Card}\big((\downarrow wav \setminus \downarrow w) \cdot a\big)$$

Mais, puisque $\downarrow w \subseteq \downarrow wav$, on a :

$$\mathrm{Card}\big((\downarrow wav \setminus \downarrow w) \cdot a\big) = \mathrm{Card}\big(\downarrow wav \setminus \downarrow w\big) = \mathrm{Card}(\downarrow wav) - \mathrm{Card}(\downarrow w)$$

Ainsi,

$$\mathrm{Card}(\downarrow wava) = 2\mathrm{Card}(\downarrow wav) - \mathrm{Card}(\downarrow w)$$

Par contre, si a apparaît une seule fois dans u (en dernière position, donc), en écrivant $u = wa$, il est clair que l'on a l'union disjointe

$$\downarrow u = \downarrow w \cup (\downarrow w) \cdot a.$$

On a alors

$$\mathrm{Card}(\downarrow u) = 2\mathrm{Card}(\downarrow w)$$

b. En suivant l'idée de programmation dynamique, on peut calculer efficacement le nombre de sous-mots en utilisant un tableau pour stocker le nombre de sous-mots des différents préfixes de `u` :

```
let nb_sousmots (u : string) =
  let m = Array.make (String.length u + 1) 0 in
  m.(0) <- 1 ;
  for i = 1 to String.length u do
    let a = u.[i - 1]
    and j = ref (i - 2) in
    while !j >= 0 && u.[!j] != a do
      decr j
    done ;
    if !j = -1 then
      (* a apparaît une unique fois *)
      m.(i) <- 2 * m.(i - 1)
    else
      m.(i) <- 2 * m.(i - 1) - m.(!j)
  done ;
  m.(String.length u)
;;
```

La fonction obtenue a une complexité temporelle en $O\big(|u|^2\big)$ de fait des deux boucles imbriquées, et une complexité spatiale en $O\big(|u|\big)$.

Remarque : à l'aide d'un dictionnaire qui indiquerait, pour chaque lettre, sa dernière occurence sous réserve de définition, la fonction précédente pourrait avoir une complexité en $O\big(|u|\big)$.

Question 8

a. Si $\text{pcsmc}(ua, vb) = wa$ alors vb est un sous-mot de w (puisque $b \neq a$) et comme ua est un sous-mot de wa, alors nécessairement u est un sous-mot de w. Mais si $w \neq \text{pcsmc}(u, vb)$, alors $\text{pcsmc}(u, vb)$ est strictement plus petit que w pour l'ordre lexicographique, et il en est de même pour $\text{pcsmc}(u, vb)\cdot a$ par rapport à wa. On a donc nécessairement $w = \text{pcsmc}(u, vb)$.

b. Clairement, on a $\text{pcsmc}(ua, va) = \text{pcsmc}(u, v)\cdot a$ alors que, dans le cas où les deux mots se terminent par des lettres différentes, on a soit $\text{pcsmc}(ua, vb) = \text{pcsmc}(u, vb)\cdot a$ ou bien $\text{pcsmc}(ua, vb) = \text{pcsmc}(ua, v)\cdot b$ (d'après la question précédente, sachant que la dernière lettre de $\text{pcsmc}(ua, vb)$ est nécessairement a ou b pour ne pas avoir de lettre inutile). Pour déterminer quelle possibilité suivre, il suffit de comparer les deux chaînes, et d'en prendre le plus petit selon la relation d'ordre indiquée : taille puis ordre lexicographique.

c. On en déduit la fonction suivante, toujours en utilisant les idées de programmation dynamique :

```
let pcsmc (u : string) (v : string) =
  let m = Array.make_matrix (String.length u + 1)
                            (String.length v + 1) "" in
  for i = 1 to String.length u do
    m.(i).(0) <- String.sub u 0 i
  done ;
  for j = 1 to String.length v do
    m.(0).(j) <- String.sub v 0 j
  done ;
  for i = 1 to String.length u do
    for j = 1 to String.length v do
      if u.[i - 1] = v.[j - 1] then
        m.(i).(j) <- m.(i - 1).(j - 1) ^ (String.sub u (i - 1) 1)
      else
        let w1 = m.(i).(j - 1) ^ (String.sub v (j - 1) 1)
        and w2 = m.(i - 1).(j) ^ (String.sub u (i - 1) 1)
        in
        if String.length w1 < String.length w2 then
          m.(i).(j) <- w1
        else if String.length w2 < String.length w1 then
          m.(i).(j) <- w2
        else (* w1 et w2 sont de même taille *)
        if w1 < w2 then m.(i).(j) <- w1 else m.(i).(j) <- w2
    done
  done ;
  m.(String.length u).(String.length v)
;;
```

À nouveau, les complexités temporelle et spatiale sont en $O\big(|u| \times |v|\big)$, en supposant que que les opérations sur les chaînes de caractères sont en temps constant. De façon plus réaliste, si l'on suppose que ces opérations sont de complexité linéaire en la taille des chaînes de caractères, la complexité de chacune de ces opérations est en $O\big(|u| + |v|\big)$. On obtient encore une complexité polynomiale en $|u| + |v|$.

2 Sous-mots et expressions rationnelles

Question 9 La première expression rationnelle correspond à $((\mathtt{a} + \emptyset) \cdot \mathtt{c})^\star \cdot (\mathtt{b} \cdot (\emptyset \cdot (\mathtt{cc})^\star))$. Or si $L(e) = \emptyset$, alors pour tout e', $L(e \cdot e') = L(e' \cdot e) = \emptyset$. Ainsi, de proche en proche, $L(\emptyset \cdot (\mathtt{cc})^\star) = \emptyset$, puis $L(\mathtt{b} \cdot (\emptyset \cdot (\mathtt{cc})^\star)) = \emptyset$ et enfin $L(e_1) = \emptyset$, donc $L(e_1)$ ne contient aucun mot commençant par `a`, `b` ou `c`.

La deuxième correspond à $(\mathtt{ba})^\star(\epsilon + \mathtt{a})\mathtt{c}^\star$. Ainsi, $L(e_2)$ contient le mot « `ba` » ($(\mathtt{ba})^\star$ capture `ba`, $(\epsilon + \mathtt{a})$ capture ϵ, et $\mathtt{c}^\star$ capture ϵ). $L(e_2)$ contient également « `a` » ($(\mathtt{ba})^\star$ capture ϵ, $(\epsilon + \mathtt{a})$ capture `a`, et $\mathtt{c}^\star$ capture ϵ). Enfin, il contient « `c` » ($(\mathtt{ba})^\star$ capture ϵ, $(\epsilon + \mathtt{a})$ capture ϵ, et $\mathtt{c}^\star$ capture `c`).

Question 10

```
let peut_debuter_par e a =
  let rec analyse e = (* Renvoie un triplet de booléens x y z, où
                         x est vrai ssi e peut commencer par a
                         y est vrai ssi e reconnait le langage vide
                         z est vrai ssi e reconnait epsilon *)
    match e with
    | Epsilon -> false, false, true
    | Empty -> false, true, false
    | Letter ch when ch = a -> true, false, false
    | Letter x -> false, false, false
    | Sum(e1, e2) -> let x1, y1, z1 = analyse e1
                     and x2, y2, z2 = analyse e2 in
                     x1 || x2, y1 && y2, z1 || z2
    | Product(e1, e2) -> let x1, y1, z1 = analyse e1
                         and x2, y2, z2 = analyse e2 in
                     (x1 && not y2) || (z1 && x2),
                     y1 || y2,
                     z1 && z2
    | Star(e1) -> let x, y, z = analyse e1 in
                  x, false, true
  in let x, y, z = analyse e in x ;;
```

Cette fonction a une complexité linéaire en la taille de l'expression donnée en argument, puisque la sous-fonction `analyse` est exécutée une seule fois à chaque « nœud » de l'expression régulière.

On peut aussi séparer les calculs des trois booléens en trois fonctions :

```
let rec langage_vide e = (* Teste si L(e) est vide *)
  match e with
  | Epsilon -> false
  | Empty -> true
  | Letter a -> false
  | Sum(e1, e2) -> langage_vide e1 && langage_vide e2
  | Product(e1, e2) -> langage_vide e1 || langage_vide e2
  | Star(e1) -> false ;;
let rec contient_epsilon e = (* Teste si L(e) contient epsilon *)
  match e with
  | Epsilon -> true
  | Empty -> false
  | Letter a -> false
  | Sum(e1, e2) -> contient_epsilon e1 || contient_epsilon e2
  | Product(e1, e2) -> contient_epsilon e1 && contient_epsilon e2
  | Star(e1) -> true ;;
let rec peut_debuter_par e a =
  match e with
  | Epsilon -> false
  | Empty -> false
  | Letter ch -> ch = a
  | Sum(e1, e2) -> peut_debuter_par e1 a || peut_debuter_par e2 a
  | Product(e1, e2) -> (peut_debuter_par e1 a && not (langage_vide e2))
                       || (contient_epsilon e1 && peut_debuter_par e2 a)
  | Star(e1) -> peut_debuter_par e1 a ;;
```

Question 11

a. Considérons $L = \{\mathtt{aa}\}$. Alors en considérant $v = w = \mathtt{a}$, on a clairement $\epsilon \preccurlyeq w$ et $wv \in L$, donc $\mathtt{a} \in \langle\epsilon\rangle L$, donc l'égalité est fausse. Attention, cette définition de résidus diffère en cela de la définition « classique »[1] des résidus d'un langage.

b. Considérons les langages $L_1 = \{\mathtt{a}\}$, $L_2 = \{\mathtt{bb}\}$. Notons tout d'abord que $L_1 \cdot L_2 = \{\mathtt{abb}\}$.

Ainsi, $\langle\mathtt{a}\rangle(L_1 \cdot L_2)$ contient $\mathtt{b}$ (en prenant dans la définition $w = \mathtt{ab}$, qui est bien un sur-mot de $\mathtt{a}$).

Or $\langle\mathtt{a}\rangle(L_2)$ est vide, puisque L_2 ne contient aucun mot contenant un $\mathtt{a}$. Et de plus, un mot de $(\langle a\rangle L_1) \cdot L_2$ se termine nécessairement par $\mathtt{bb}$, donc cet ensemble ne contient pas $\mathtt{b}$ non plus. Ainsi, cette égalité est également fausse.

Peut-être plus intuitivement, dans le membre de gauche de l'égalité, le sur-mot w de $\mathtt{a}$ pouvait prendre à la fois son $\mathtt{a}$ dans un mot de L_1, mais

[1] mais hors-programme !

aussi prendre des lettres dans un mot de L_2, ce qui n'était pas permis dans le membre de droite.

c. Considérons $u = \mathtt{a}$, $v = \mathtt{b}$ et $L = \{\mathtt{abc}\}$. Alors $\langle ab\rangle\{\mathtt{abc}\}$ contient $\mathtt{c}$ (en prenant $w = \mathtt{ab}$) et ϵ (en prenant $w = \mathtt{abc}$). Avec les mêmes valeurs de w, $\langle \mathtt{b}\rangle L = \{\epsilon, \mathtt{c}\}$ également (et ce sont bien les seules, il n'y a pas d'autre sur-mot de $\mathtt{b}$ qui préfixe un mot de L).

On peut également remarquer que $\langle \mathtt{a}\rangle\{\epsilon, \mathtt{c}\} = \emptyset$: il n'existe aucun sur-mot de $\mathtt{a}$ qui préfixe un mot de cet ensemble. Ainsi, $\langle \mathtt{a}\rangle(\langle \mathtt{b}\rangle\{\mathtt{abc}\}) \neq \langle \mathtt{ab}\rangle\{\mathtt{abc}\}$.

d. Considérons $L = \{a\}$ et $u = \mathtt{aa}$. Alors $\langle u\rangle L = \emptyset$, car L ne contient aucun sur-mot de $\mathtt{aa}$. Donc $(\langle u\rangle L) \cdot L^\star = \emptyset$.

À l'inverse, en prenant $v = w = \mathtt{aa}$, on a bien $u \preccurlyeq w$ et $wv \in L^\star$, donc $v = \mathtt{aa} \in \langle u\rangle(L^\star)$.

Ainsi, aucune de ces égalités n'est vraie.

Question 12

a. L'expression rationnelle e' obtenue doit reconnaitre tous les suffixes de mots de $L(e)$.

```
let rec eps_residu_ratexp e =
  match e with
  | Epsilon -> Epsilon
  | Empty -> Empty
  | Letter a -> Sum(e, Epsilon)
  | Sum(e1, e2) -> Sum(eps_residu_ratexp e1, eps_residu_ratexp e2)
  | Product(e1, e2) ->
     if langage_vide e1 then
       Empty
     else
       Sum(
          Product(eps_residu_ratexp e1, e2),
          eps_residu_ratexp e2)
  | Star(e1) -> Product(eps_residu_ratexp e1, Star(e1)) ;;
```

b. Considérons $f : x \mapsto \dfrac{(x+1)(x+2)}{2}$, et montrons par induction que si e est une expression de taille x, alors l'algorithme ci-dessus ne peut pas produire une expression de taille supérieure à $f(x)$. C'est vrai si $e = \emptyset$, si $e = \epsilon$ car $f(1) = 3 > 1$.

Si e est une lettre, $|e'| = 3 = f(1)$.

Pour la suite, remarquons que $f(x_1+x_2+1) = \dfrac{(x_1+x_2+2)(x_1+x_2+3)}{2} = \dfrac{x_1^2+x_2^2+2x_1x_2+5x_1+5x_2+6}{2}$.

Si e s'écrit $e_1 + e_2$ avec $|e_1| = x_1$ et $|e_2| = x_2$, alors $|e'| \leqslant 1 + f(x_1) + f(x_2)$ par hypothèse d'induction. En utilisant alors que $|e'| \leqslant \dfrac{2 + (x_1+1)(x_1+2) + (x_2+1)(x_2+2)}{2} \leqslant \dfrac{x_1^2 + 3x_1 + x_2^2 + 3x_2 + 6}{2} \leqslant f(x_1 + x_2 + 1)$.

Si e s'écrit $e_1 \cdot e_2$ avec $|e_1| = x_1$ et $|e_2| = x_2$, alors $|e'| \leqslant 2 + f(x_1) + x_2 + f(x_2) \leqslant \dfrac{4 + (x_1+1)(x_1+2) + 2x_2 + (x_2+1)(x_2+2)}{2} = \dfrac{x_1^2 + x_2^2 + 3x_1 + 5x_2 + 8}{2} \leqslant f(x_1 + x_2 + 1)$ (par exemple parce que $x_1 \geq 1$).

Si e s'écrit $e_1^\star$, avec $|e_1| = x_1$, alors $|e'| \leqslant 2 + x_1 + f(x_1) \leqslant \dfrac{4 + 2x_1 + (x_1+1)(x_1+2)}{2} = \dfrac{(x_1+2)(x_1+3)}{2} = f(x_1 + 1)$

Ainsi, dans tous les cas, par induction, la taille de e' est majorée par $f(|e|)$.

Question 13

a. Le langage doit contenir, pour tout mot uav de L, les suffixes de v.

```
let rec char_residu_ratexp a e =
  match e with
  | Epsilon -> Empty
  | Empty -> Empty
  | Letter ch when ch = a -> Epsilon
  | Letter ch -> Empty
  | Sum(e1, e2) -> Sum(char_residu_ratexp a e1, char_residu_ratexp a e2)
  | Product(e1, e2) ->
     if langage_vide e1 then
       Empty
     else let e1_prime = char_residu_ratexp a e1 in
       if langage_vide e1_prime then
         char_residu_ratexp a e2
       else
         Sum(
          Product(char_residu_ratexp a e1, e2),
          eps_residu_ratexp e2)
  | Star(e1) -> if langage_vide (char_residu_ratexp a e1) then Empty
                else eps_residu_ratexp e ;;
```

b. Le même majorant convient, avec la même démonstration.

Question 14

a. Montrons que $\langle au \rangle L = \langle u \rangle \big(\langle a \rangle L \big)$. Pour cela, un mot v appartient à $\langle u \rangle \big(\langle a \rangle L \big)$ si et seulement si il existe w_1 et w_2 tels que

$$a \preccurlyeq w_1, \quad u \preccurlyeq w_2 \quad \text{et} \quad w_1 w_2 v \in L$$

Mais alors, $au \preccurlyeq w_1 w_2$ et donc $v \in \langle au \rangle L$ et donc

$$\langle u \rangle \big(\langle a \rangle L \big) \subseteq \langle au \rangle L$$

Réciproquement, tout mot w tel que $au \preccurlyeq w$ peut s'écrire sous la forme $w = w_1 w_2$ tel que $a \preccurlyeq w_1$ et $u \preccurlyeq w_2$, ce qui assure l'inclusion inverse, et donc l'égalité entre les deux ensembles.

En remarquant que $u \in {\downarrow}L \iff \langle u \rangle L \neq \emptyset$, on en déduit la fonction suivante :

```
let sousmot_de_ratexp u e =
  let n = String.length u in
  let rec aux i e =
    if i = n then not (est_vide (eps_residu_ratexp e))
    else aux (i + 1) (char_residu_ratexp u.[i] e)
  in
  aux 0 e
;;
```

b. Sans simplification des termes, les appels récursifs aux fonctions de calcul de résidu peuvent faire croitre la taille des expressions de façon quadratique à chaque fois, ce qui peut conduire à une augmentation exponentielle de la taille (et même à la puissance $2^{|u|}$).

Question 15 On peut voir une matrice booléenne représentant $FC(u, e)$ comme une matrice d'adjacence, un chemin entre deux positions correspondant à la couverture par e du facteur de u correspondant. En particulier, pour le produit, cela revient à concaténer les chemins et, pour l'étoile, à calculer la cloture transitive du graphe, ce que l'on fait à l'aide de l'algorithme de Floyd-Warshall.

```
let rec facteurs_couverts u e =
  let n = String.length u in
  let m = Array.make_matrix (n + 1) (n + 1) false in
  match e with
  | Epsilon ->
    for i = 0 to n do
      m.(i).(i) <- true
    done;
    m
  | Empty -> m
  | Letter a ->
    for i = 0 to n - 1 do
      if u.[i] = a then m.(i).(i + 1) <- true
    done;
    m
  | Sum (e1, e2) ->
    let m1 = facteurs_couverts u e1
```

```
    and m2 = facteurs_couverts u e2 in
    for i = 0 to n do
      for j = i to n do
        m.(i).(j) <- m1.(i).(j) || m2.(i).(j)
      done
    done;
    m
  | Product(e1, e2) ->
    let m1 = facteurs_couverts u e1
    and m2 = facteurs_couverts u e2 in
    for i = 0 to n do
      for j = i to n do
        for k = j to n do
          m.(i).(k) <- m.(i).(k) || (m1.(i).(j) && m2.(j).(k))
        done
      done
    done ;
    m
  | Star e1 ->
    let m1 = facteurs_couverts u e1 in
    (* on applique un algorithme type Floyd-Warshall *)
    (* on remplit la diagonale *)
    for i = 0 to n do
      m.(i).(i) <- true
    done;
    (* on ajoute les nouveaux chemins à m en passant par m1 *)
    for i = 0 to n do
      for j = i to n do
        for k = j to n do
          m.(i).(k) <- m.(i).(k) || (m.(i).(j) && m1.(j).(k))
        done
      done
    done ;
    m
;;
```

Concernant la complexité, il y a $|e|$ appels récursifs, et chaque appel récursif a une complexité spatiale en $|u|^2$ et temporelle en $|u|^3$. Ainsi, temporellement, la fonction est en $|e| \times |u|^3$.

Question 16 On en déduit une version plus efficace de `sousmot_de_ratexp` en vérifiant si u est entièrement couvert par L, autrement dit si la matrice renvoyée par la fonction précédente possède la valeur `true` en haut à droite.

```
let sousmot_de_ratexp2 u e =
  let n = String.length u
```

```
  and m = facteurs_couverts u e in
  m.(0).(n)
;;
```

Comme pour la fonction précédente, la complexité (temporelle) est en $|e| \times |u|^3$.