TD 10

Langages

Correction

Exercice 1 : Un langage L est dit :

- récursivement énumérable s'il existe un algorithme qui énumère tous les mots de L ;
- récursif s'il existe un algorithme qui, prenant un mot u de A^*, retourne true si u est dans L et false sinon.

Pour les langages suivants, écrire la fonction `test_i` qui prend en argument un `mot` et vérifie l'appartenance au langage L_i.

```
test_i : string -> bool
```

(a) $L_1 = \{w \in A^* \quad | \quad 3 | |w|\}$

Correction

Listing 1 – La fonction `test_1`

```
let test_1 w = String.length w mod 3 = 0;;
```

(b) $L_2 = \{w \in A^* \quad | \quad aabb$ est un sous-mot$\}$

Correction

Listing 2 – La fonction `test_2`

```
let test_2 w =
  let mot = "aabb" in
  let n = String.length w in
  let i = ref 0 in
  let nb = ref 0 in
  while (!i < n) && (!nb != 4) do
    if mot.[!nb] = w.[!i] then incr nb ;
    incr i ;
  done; !nb = 4;;
```

(c) $L_3 = \{w \in A^* \quad | \quad aabb$ est un facteur$\}$

Correction

Listing 3 – La fonction `test_3`

```
let test_3 w =
  let mot = "aabb" in
  let n = String.length w in
  let i = ref 0 in
  let nb = ref 0 in
```

```
let booleen = ref false in
while (!i < n) && (not !booleen ) do
  nb := 0;
  let temp = !i in
      while (!i < n)&& (!nb != 4)&&  (mot.[!nb] = w.[!i]) do
        incr i ; incr nb
      done ;
  booleen := ( !nb = 4) ;
  i := temp +1 ;
done; !booleen;;
```

(d) $L_4 = \{w \in A^* \quad | \quad agaga \text{ est un préfixe}\}$

Correction

Listing 4 – La fonction `test_4`

```
let test_4 w =
  let mot = "agaga" in
  let n = String.length w in
  let i = ref 0 in
  while (!i < n) && (!i != 5  ) && w.[!i] = mot.[!i] do
    incr i
  done;
  !i = 5;;
```

(e) $L_5 = \{w \in A^* \quad | \quad \text{est un paladrome}\}$

Correction

Listing 5 – La fonction `test_5`

```
let test_5 w =
  let n = String.length w in
  let i = ref 0 in
  while (!i < n/2) && w.[!i] =w.[n-1- !i] do
      incr i
  done ;
  !i = n/2 ;;
```

(f) $L_6 = \{a^n b^n \quad | \quad n \in \mathbb{N}\}$

Correction

Listing 6 – La fonction `test_6`

```
let test_6 w =
  let n = String.length w in
```

```
  ( n mod 2 = 0 ) &&
  begin
  let n = String.length w in
  n=0 ||
  let i = ref 0 in
  while (!i < n/2) && w.[!i]='a' && w.[n-1- !i]='b' do
      incr i
  done ;
  !i = n/2
  end ;;
```

Exercice 2 : On définit le type expression rationnelle par :

```
type 'a exp_rat =
  |Vide
  |Epsilon
  |Lettre of 'a
  |Plus of ('a exp_rat )*('a exp_rat )
  |Concat of ('a exp_rat )*('a exp_rat )
  |Etoile of 'a exp_rat
;;
```

Définir les expressions rationnelles suivantes :

$$e_1 = a.b + b.a$$

$$e_2 = (a+b)^*(aabb)$$

On souhaite construire une fonction qui vérifie si un mot donné est issu d'une expression rationnelle donnée.

1. Écrire une fonction `test_1` qui renvoie vrai si le mot vérifie une expression rationnelle simple (soit une lettre, soit une somme)
   Par exemple pour l'expression

   ```
   let e3 = Plus ( Lettre 'a',Lettre 'b');;
   ```

   Correction

Listing 7 – La fonction `test_1`

```
let rec test_1 w e  =
  let n = String.length w in
     match e with
  | Vide -> false
  | Epsilon -> (n=0)
  | Lettre(a) -> (n=1)&&(w.[0]=a)
  | Plus ( e1,e2) -> ( test1 w e1 )||(test1 w e2)
  | _ -> failwith "␣pas␣encore"
;;
```

2. Écrire une fonction `test_2` qui renvoie vrai si le mot vérifie une expression rationnelle issue d'une concaténation. (on rappelle la fonction : `String.sub w n p` qui renvoie le sous-mot extrait de w)

```
let e4 = Concat( Concat( Concat ( Lettre 'a' , Lettre 'b'),Lettre 'a'),
Lettre 'a' );;
```

Correction

Listing 8 – La fonction `test_2`

```
let rec test_2 w e =
  let n = String.length w in
    match e with
  |Vide -> false
  |Epsilon -> (n=0)
  |Lettre(a) -> ( n=1)&&(w.[0]=a)
  |Concat(e1,e2) ->(let r = ref false and i = ref 0 in
    begin
    while (!i<=n)&& (not !r) do
      r:=(test2 (String.sub w 0 !i) e1)&&(test2 (String.sub w !i (n- !i))
 e2);
      i:= !i+1
     done;
    !r
    end)
  | _ -> failwith "␣pas␣encore"
;;
```

3. Écrire une fonction `test` qui renvoie vrai si le mot vérifie une expression rationnelle quelconque.

Par exemple :

```
let e5 = Concat( Etoile( Plus( Lettre 'a' , Lettre 'b')),
Concat( Concat( Concat ( Lettre 'a' , Lettre 'a'),Lettre 'b'), Lettre 'b'
 ));;
```

Correction

Listing 9 – La fonction `test`

```
let rec test w e=
  let n= String.length w in
  match e with
    | Vide -> false
    | Epsilon -> n=0
```

```
    | Lettre(a)-> (n=1) && (w.[0]=a)
    | Plus(e1,e2)->(test w e1)||(test w e2)
    | Concat(e1,e2)->
      (let r = ref false and i = ref 0 in
      begin
      while (!i<=n)&& (not !r) do
        r:=(test (String.sub w 0 !i) e1)&&(test (String.sub w !i (n- !i))
 e2);
        i:= !i+1
       done;
      !r
      end)
    | Etoile(e1)->
      (let r=ref (n=0) and i=ref 1 in
      begin
      while (!i<=n)&& (not !r) do
        r:=(test (String.sub w 0 !i) e1)&&(test (String.sub w !i (n- !i))
 e);
        i:= !i+1
      done;
      !r
      end)
;;
```

Exercice 3 : **Expressions rationnelles linéaires**

On conserve le type précédente.

On pourra travailler avec l'exemple suivant :

```
let e = Concat(Etoile ( Plus (Lettre 'a', Lettre 'b')),Concat(
Concat(Concat(Lettre 'c', Lettre 'd'), Lettre 'e'), Lettre 'f'));;
```

1. Écrire une fonction `est_lineaire` qui renvoie vrai si l'expression rationnelle entrée en argument est linéaire. (on pourra écrire au préalable la fonction `appartient`, et la fonction `lineaire` qui renvoie le couple de type `bool` $*'$ `a list`, la liste étant la liste des lettres composant l'expression.)

   ```
   appartient : 'a -> 'a list -> bool
   lineaire : 'a exp_rat -> bool * 'a list
   est_lineaire : 'a exp_rat -> bool
   ```

   Correction

Listing 10 – La fonction `est_lineaire`

```
let rec appartient a l = match l with
    | [] -> false
    | b::reste -> (a=b) || appartient a reste;;
```

```
let rec lineaire e =
    let rec aux e l =  match e with
  |Vide -> ( true , [])
  |Epsilon ->  ( true , [])
  |Lettre(a) -> ( not (appartient a l) , a::l)
  | Plus(e1,e2)-> let (b1,l1) = aux e1 l in let (b2,l2)= aux e2 l1 in (b1
 && b2 , l2)
  |Concat(e1,e2) ->   let (b1,l1) = aux e1 l in let (b2,l2)= aux e2 l1 in
 (b1 && b2 , l2)
  | Etoile(e1)-> aux e1 l
in aux e []
;;

let est_lineaire e = let (b,l) =lineaire e in b;;
```

2. Écrire une fonction `mot_vide` qui renvoie vrai si le mot vide appartient au langage dénotée par une expression rationnelle entrée en argument.

   `mot_vide : 'a exp_rat -> bool`

   Correction

Listing 11 – La fonction `mot_vide`

```
let rec mot_vide e = match e with
  |Vide -> false
  |Epsilon -> true
  |Lettre(a) -> false
  | Plus(e1,e2)->(mot_vide e1)||(mot_vide e2)
  |Concat(e1,e2) -> mot_vide e1 &&  mot_vide e2
  | Etoile(e1)-> true
;;
```

3. Écrire une fonction `prefixe` qui renvoie la liste de première lettres possibles pour une expression rationnelle linéaire entrée en argument.

   `prefixe : 'a exp_rat -> 'a list`

   Correction

Listing 12 – La fonction `prefixe`

```
let rec prefixe e =match e with
  | Vide -> []
  | Epsilon -> []
  | Lettre(a) -> [a]
  | Plus (e1, e2) ->  (prefixe e1) @  (prefixe e2)
```

```
  | Concat (e1, e2)   when mot_vide e1 ->  (prefixe e1)@ (prefixe e2)
  | Concat (e1, e2) -> prefixe e1
  | Etoile e -> prefixe e ;;
```

4. Écrire une fonction `suffixe` qui renvoie la liste de dernières lettres possibles pour une expression rationnelle linéaire entrée en argument.

   `suffixe : 'a exp_rat -> 'a list`

   Correction

Listing 13 – La fonction `suffixe`

```
let rec suffixe e =match e with
  |Vide -> []
  | Epsilon -> []
    |Lettre(a) -> [a]
  | Plus (e1, e2) ->  (suffixe e1) @  (suffixe e2)
  | Concat (e1, e2)   when mot_vide e2 ->  (suffixe e1)@ (suffixe e2)
  | Concat (e1, e2) -> suffixe e2
    | Etoile e -> suffixe e ;;
```

5. Écrire la fonction `produit` qui renvoie le produit cartésien de deux listes.

   `produit : 'a list -> 'b list -> ('a * 'b) list`

   Correction

Listing 14 – La fonction `produit`

```
let rec produit l_1 l_2 = match (l_1,l_2) with
  | (_ ,[]) | ( [] , _ ) -> []
  | ( a::reste , l_2)  ->
    let rec aux a l = match l with
      | [] -> []
      | b::reste -> (a,b)::( aux a reste  ) in
      (aux a l_2) @ (produit reste l_2 );;
```

6. Écrire la fonction `facteur` qui renvoie la liste des facteurs de deux lettres sous la forme d'une liste de couples.

   ```
   facteur : 'a exp_rat -> ('a * 'a) list
   ```

   Correction

Listing 15 – La fonction `facteur`

```
let rec facteur  e =match e with
  |Vide -> []
  | Epsilon -> []
                |Lettre(a) -> []
  | Plus (e1, e2) ->  (facteur e1) @  (facteur e2)
  | Concat (e1, e2) ->  let l = facteur e1 @ facteur e2 in l@ ( produit
( suffixe e1)  (prefixe e2))
    | Etoile e -> (facteur e) @ ( produit  ( suffixe e)  (prefixe e));;
```