

Des expressions régulières aux automates

OPTION INFORMATIQUE - TP n° 4.2 - Olivier Reynet

À la fin de ce chapitre, je sais :

- ✎ coder la linéarisation d'une expressions régulière
- ✎ déterminer les composantes P,S et F relatives à une expression régulière linéaire
- ✎ coder l'algorithme de Berry-Sethi et trouver l'automate de Glushkov associé à une expression régulière

A Linéarisation d'une expression régulière

On souhaite réaliser la linéarisation d'une expression régulière dans le but d'implémenter l'algorithme de Berry-Sethi. On dispose du type regexp algébrique suivant :

```
type regexp =  
    EmptySet  
    | Epsilon  
    | Letter of char  
    | Sum of regexp * regexp  
    | Concat of regexp * regexp  
    | Kleene of regexp ;;
```

On se donne le type algébrique lregexp qui représente une expression régulière linéarisée :

```
type lregexp =  
    Letter_ind of char * int  
    | SumL of lregexp * lregexp  
    | ConcatL of lregexp * lregexp  
    | KleeneL of lregexp ;;
```

Un littéral est donc codé par une lettre associée à un numéro qui code l'ordre d'apparition de la lettre dans l'expression régulière.

A1. Écrire une fonction récursive de signature

`linearize_and_count : regexp -> int -> lregexp * int`

qui linéarise une expression régulière. Le paramètre de type int est le compteur de variable : on l'incrémente à chaque fois qu'on découvre un littéral ou une occurrence d'un littéral. La fonction renvoie l'expression linéarisée ainsi que l'état du compteur de variable. On choisira des caractères arbitraires¹ pour l'ensemble vide et le mot vide. La fonction s'utilise ainsi :

`linearize_and_count e 1`

en initialisant le compteur à 1. Pour l'expression régulière $(a|b)^*c$, la fonction renvoie :

1. Par exemple `char_of_int 0xD8` et `char_of_int 0x80`

```
(ConcatL (KleeneL (SumL (Letter_ind ('a', 1), Letter_ind ('b', 2))), Letter_ind ('c', 3)), 4)
```

Solution :

```
let rec linearize_and_count e counter =
  match e with
  | EmptySet -> Letter_ind(char_of_int 0xD8, counter), counter + 1
  | Epsilon -> Letter_ind(char_of_int 0x80, counter), counter + 1
  | Letter(a) -> Letter_ind(a, counter), counter + 1
  | Sum(e1,e2) -> let (e3,c) = linearize_and_count e1 counter in let (e4,
    c2) = linearize_and_count e2 c in SumL(e3,e4), c2
  | Concat(e1,e2) -> let (e3,c) = linearize_and_count e1 counter in let (
    e4,c2) = linearize_and_count e2 c in ConcatL(e3,e4), c2
  | Kleene(e) -> let (e1,c) = linearize_and_count e counter in KleeneL(e1)
    ,c
;;
```

- A2. Proposer une fonction de signature `epsilon_is_in : lregexp -> bool` qui teste si une expression régulière linéarisée contient le mot vide.

Solution :

```
let rec epsilon_is_in e =
  match e with
  | SumL (e1,e2) -> epsilon_is_in e1 || epsilon_is_in e2
  | ConcatL (e1,e2) -> epsilon_is_in e1 && epsilon_is_in e2
  | KleeneL _ -> true
  | Letter_ind (c,_) -> let eps = char_of_int 0x80 in eps = c ;;
```

- A3. Écrire une fonction «wrapper» de signature `linearize : regexp -> lregexp` qui permette de ne récupérer que l'expression régulière linéarisée, sans le compteur.

Solution :

```
let linearize e = fst(linearize_and_count e 1);;
```

B Calcul des composantes P, S et F associées à une expression régulière

Toujours dans l'optique d'implémenter l'algorithme de Glushkov, on cherche maintenant à caractériser les ensembles P (préfixes à une lettre), S (suffixes à une lettre) et F (facteurs possibles de deux lettres) d'une expression régulière linéarisée.

- B1. Pour les expressions régulières suivantes, trouver les ensembles P, S et F tels que définis dans le cours : $(a|b)^*c$ et $((a|b)^*c)|b$.

Solution : Pour $(a|b)^*c = (a_1|b_2)^*c_3$:

- $P = \{(a_1, b_2, c_3),$
- $S = \{c_3\},$
- $F = \{a_1 a_1, a_1 b_2, b_2 a_1, b_2 b_2, a_1 c_3, b_2 c_3\}.$

Pour $((a|b)^*c)|b = ((a_1|b_2)^*c_3)|b_4$:

- $P = \{(a_1, b_2, c_3, b_4),$
- $S = \{c_3, b_4\},$
- $F = \{a_1 a_1, a_1 b_2, b_2 a_1, b_2 b_2, a_1 c_3, b_2 c_3\}.$

B2. Ensemble P. Écrire une fonction récursive de signature

`first_letter_prefix : lregex -> (char * int)list`

qui renvoie la liste des préfixes à une lettre d'une expression régulière linéarisée. Par exemple pour $(a|b)^*c$ linéarisée, la fonction renvoie `(char * int)list = [('a', 1); ('b', 2); ('c', 3)]`. On utilisera la concaténation de liste @ et, si besoin, la fonction `epsilon_is_in`.

Solution :

```
let rec first_letter_prefix e =
  match e with
  | Letter_ind (a,i) -> [a,i]
  | SumL(e1, e2) -> (first_letter_prefix e1)@(first_letter_prefix e2)
  | ConcatL(e1, e2) when epsilon_is_in e1 ->
    (first_letter_prefix e1)@(first_letter_prefix e2)
  | ConcatL(e1, _) -> first_letter_prefix e1
  | KleeneL(e) -> first_letter_prefix e
;;
```

B3. Ensemble S. Écrire une fonction récursive de signature

`last_letter_suffix : lregex -> (char * int)list`

qui renvoie la liste des suffixes à une lettre d'une expression régulière linéarisée. Par exemple pour $(a|b)^*c$ linéarisée, la fonction renvoie `(char * int)list = [('c', 3)]`. On utilisera la concaténation de liste @ et, si besoin, la fonction `epsilon_is_in`.

Solution :

```
let rec last_letter_suffix e =
  match e with
  | Letter_ind(a,i) -> [a,i]
  | SumL(e1, e2) -> (last_letter_suffix e1)@(last_letter_suffix e2)
  | ConcatL(e1, e2) when epsilon_is_in e2 ->
    (last_letter_suffix e1)@(last_letter_suffix e2)
  | ConcatL(_, e2) -> last_letter_suffix e2
  | KleeneL(e) -> last_letter_suffix e
;;
```

B4. Écrire une fonction de signature

`cartesian_product : 'a list -> 'b list -> ('a * 'b)list`
 qui renvoie le produit cartésien de deux listes d'entiers. Par exemple, `cartesian_product [1;3] [2;4;6;8]` renvoie `[(1, 2); (1, 4); (1, 6); (1, 8); (3, 2); (3, 4); (3, 6); (3, 8)]`.

Solution :

```
let cartesian_product set1 set2 =
  List.fold_left (fun acc e -> acc@ (List.map (fun e' -> (e,e')) set2)) []
  set1;;
```

B5. **Ensemble F.** Écrire une fonction récursive de signature `two_factors : lregexp -> ((char * int) * (char * int))list` qui renvoie les facteurs possibles de longueur 2 d'une expression régulière linéarisée. On utilisera la fonction `cartesian_product` et la concaténation de listes `@`.

Solution :

```
let rec two_factors e =
  match e with
  | Letter_ind(_,_) -> []
  | SumL(e1, e2) -> (two_factors e1)@(two_factors e2)
  | ConcatL(e1, e2) -> let l = (two_factors e1)@(two_factors e2)
  in l@(cartesian_product (last_letter_suffix e1) (first_letter_prefix
    e2))
  | KleeneL(e) -> (two_factors e)@(cartesian_product (
    last_letter_suffix e) (first_letter_prefix e))
  ;;
```

C Algorithme de Berry-Sethi

L'algorithme de Berry-Sethi permet d'obtenir l'automate de Glushkov qui n'est pas déterministe a priori. C'est pourquoi on choisit de modéliser l'automate comme suit :

```
type ndfsm = { states : int list;
  alphabet : char list;
  initial : int list;
  transitions : (int * char * int) list;
  accepting : int list};;
```

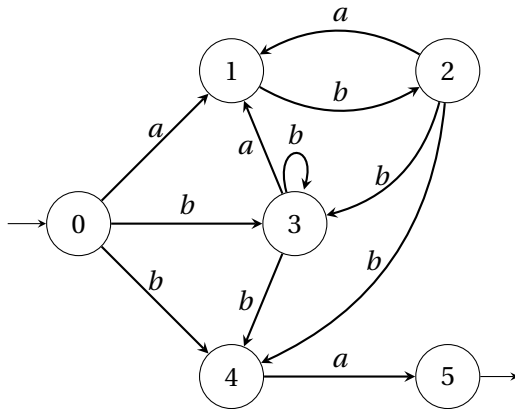
On choisit de représenter les états par un numéro. **Le zéro est l'état initial. Les états sont ensuite numérotés d'après l'indice des lettres de l'expression régulière linéarisée.** On s'appuie par ailleurs sur toutes les fonctions précédemment écrites.

C1. Linéariser à la main l'expression régulière $(ab|b)^*ba$ telle que l'effectue la fonction `linearize` déjà programmée.

Solution : $(a_1b_2|b_3)^*b_4a_5$

C2. Déterminer à la main l'automate de Glushkov associé à l'expression régulière $(ab|b)^*ba$.

Solution : cf. cours. C'est un automate local non déterministe.



C3. Écrire une fonction de signature `all_states : regexp -> int list` qui renvoie la liste de tous les états de l'automate de Glushkov associés à une expression régulière. On utilisera la fonction `linearize`. Par exemple, pour $(a|b)^*c$, cette fonction renvoie `[0; 1; 2; 3]`.

Solution :

```
let all_states e = let (_,c) = linearize_and_count e 1 in List.init c (fun i
-> i);;
```

C4. Les états accepteurs de l'automate de Glushkov sont déterminés par l'ensemble `S` obtenu grâce à la fonction `last_letter_suffix`. Écrire une fonction de signature `accepting_states : ('a * 'b)list -> 'b list` qui prend comme paramètre un ensemble `S` lié à une expression régulière linéarisée et qui renvoie l'ensemble des états accepteurs de l'automate de Glushkov.

Solution :

```
let rec accepting_states s =
  match s with
  | [] -> []
  | (_,n)::t -> n::(accepting_states t);;
```

C5. Écrire une fonction récursive de signature `initial_transitions : ('a * 'b)list -> (int * 'a * 'b)list` dont le paramètre est un ensemble `P` et qui renvoie la liste des transitions depuis l'état initial de l'automate de Glushkov.

Solution :

```
let rec initial_transitions p =
```

```

match p with
| [] -> []
| (c,n)::t -> (0,c,n)::(initial_transitions t);;

```

- C6. Écrire une fonction récursive de signature `inner_transitions : (('a * 'b) * ('c * 'd))list -> ('b * 'c * 'd)list` dont le paramètre est un ensemble F et qui renvoie la liste des transitions internes de l'automate de Glushkov.

Solution :

```

let rec inner_transitions factors =
  match factors with
  | [] -> []
  | ((_,n1),(c2,n2))::t -> (n1,c2,n2)::(inner_transitions t);;

```

- C7. Écrire une fonction de signature `all_transitions : lregex -> (int * char * int)list` qui renvoie la liste des transitions de l'automate de Glushkov.

Solution :

```

let all_transitions e =
  (initial_transitions (first_letter_prefix e)) @ (inner_transitions (
    two_factors e));;

```

- C8. Écrire une fonction de signature `rm_dup : 'a list -> 'a list` qui élimine les doublons dans une liste.

Solution :

```

let rm_dup s = List.fold_left (fun acc x -> if List.mem x acc then acc else
  x :: acc) [] s;;

```

- C9. Écrire une fonction de signature `get_alphabet_from_trans : ('a * 'b * 'c)list -> 'b list` qui renvoie l'alphabet de l'automate de Glushkov d'après ses transitions.

Solution :

```

let get_alphabet_from_trans trans = rm_dup (List.map (fun (_,a,_) -> a)
  trans);;

```

- C10. Écrire une fonction de signature `glushkov : regexp -> ndfsm` qui renvoie l'automate de Glushkov associé à une expression régulière.

Solution :

```

let glushkov rexp =
  let (e,c) = (linearize_and_count rexp 1) in
  let t = all_transitions e in
  { states = List.init c (fun i -> i);
    alphabet = get_alphabet_from_trans t;
    initial = [0] ;
    transitions = t;
    accepting = accepting_states (last_letter_suffix e) };;

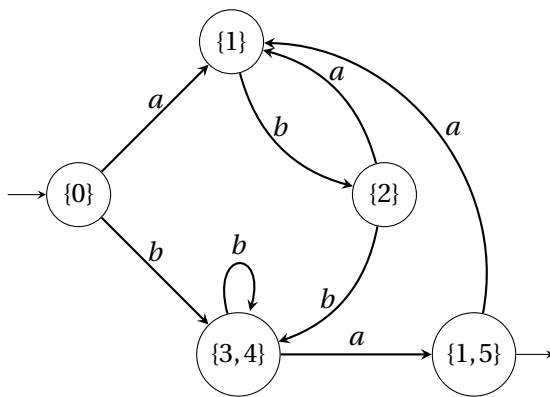
```

C11. Déterminer à la main l'automate de Glushkov obtenu grâce la fonction précédente à partir de l'expression régulière $(ab|b)^*ba$.

Solution :

	$\downarrow\{0\}$	$\{1\}$	$\{2\}$	$\{3,4\}$	$\uparrow\{1,5\}$
a	$\{1\}$		$\{1\}$	$\{1,5\}$	
b	$\{3,4\}$	$\{2\}$	$\{3,4\}$	$\{3,4\}$	$\{2\}$

ce qui se traduit par l'AFD :

**D Entraînement**

D1. En utilisant l'algorithme de Berry-Sethi, trouver l'automate associé aux expressions régulières suivantes :

(a) aab^*ab

Solution :



(b) $a(ab)^* | b * a$

Solution :

1. Linéarisation : $a_1(a_2b_3)^* | b_4^*a_5$

2. Ensembles P, S, F :

- $P = \{a_1, b_4, a_5\}$
- $S = \{a_1, b_3, a_5\}$
- $F = \{a_1a_2, a_2b_3, b_3a_2, b_4b_4, b_4a_5\}$

3. Automate = $(Q, \Sigma, q_0, \delta, F)$:

- $Q = \{q_0, q_{a_1}, q_{a_2}, q_{b_3}, q_{b_4}, q_{a_5}\}$
- $F = \{q_{a_1}, q_{b_3}, q_{a_5}\}$



(c) $(b|ab)^*(\epsilon|ab)$

D2. En utilisant l'algorithme de Thompson, trouver l'automate associé aux expressions régulières suivantes :

- (a) a^*b
- (b) aab^*ab

(c) $(a|b)^* a^* b^*$

(d) $(b|ab)^* (\epsilon|ab)$

(e) $a(ab)^* | b^* a$

Solution : cf. cours : automates associés aux expressions régulières élémentaires et élimination des transitions spontanées. On doit retrouver le même qu'avec Berry-Sethi!