▶ **MP & MP* – Option Informatique**
Année 2002–2003, Sixième TP Caml
Vincent Simonet (`http://cristal.inria.fr/~simonet/`)

Compilation d'expressions rationnelles

D'après « Le Langage Caml », Pierre Weis et Xavier Leroy, Dunod (1999)

Nous nous intéressons aujourd'hui au problème de déterminer si une expression rationnelle reconnaît une chaîne de caractères. La traduction naïve de la définition des chaînes reconnues par une expression rationnelle mène à un algorithme par essais et erreurs qui est très inefficace dans certains cas. La manière efficace de déterminer si une expression rationnelle reconnaît une chaîne de caractères est de transformer d'abord l'expression rationnelle en un automate qui reconnaît les mêmes mots, puis d'exécuter l'automate sur ladite chaîne de caractères.

1 Automates non-déterministes et expressions rationnelles

1.1 Expressions rationnelles

On considère un *alphabet* Σ (les éléments de Σ, les *caractères*, sont notés c). Une *expression rationnelle* permet de représenter de manière finie un *langage* sur Σ, c'est-à-dire une partie de l'ensemble des mots sur Σ, usuellement noté Σ^*. Les *expressions rationnelles* r sur Σ sont définies par la grammaire suivante :

$$\begin{array}{rcl} r & ::= & \epsilon \\ & | & c \\ & | & (r \mid r) \\ & | & (rr) \\ & | & r^* \end{array}$$

On définit le langage dénoté par une expression rationnelle de manière récursive. ϵ dénote le langage vide $\varnothing$ tandis que c dénote le langage $\{c\}$. Si r_1 et r_2 dénotent respectivement les langages L_1 et L_2 alors $(r_1 \mid r_2)$ dénote le langage $L_1 \cup L_2$ et $(r_1 r_2)$ dénote le langage $L_1 L_2$. Enfin r_1^* dénote le langage L_1^*.
Dans la suite de cet énoncé, nous supposerons que les éléments de Σ sont représentés en Caml par des *caractères* de type char. Une expression rationnelle (*regular expression* en anglais) sera définie en Caml par un arbre du type suivant :

```
type regexp =
    Empty
  | Char of char
  | Or of regexp * regexp
  | Seq of regexp * regexp
  | Star of regexp
;;
```

1.2 Automates non-déterministes

Nous représentons un *état* d'un *automate fini non-déterministe* (NFA) par un enregistrement à quatre champs du type suivant :

```
type nfa_state =
  { mutable trans: (char * nfa_state) list;
    mutable eps_trans: nfa_state list;
    mutable terminal: bool;
    index: int
  }
;;
```

Le champ terminal est un booléen indiquant si l'état est terminal. Les champs trans et eps_trans contiennent la liste des transitions dont l'état en question est l'origine. Le champ index sert à identifier les états de manière unique : deux états différents portent des numéros différents.

▶ **Question 1** *Écrivez une fonction* nfa_create *qui crée un nouvel état, sans transitions. L'index de l'état créé devra être* unique, *i.e. chaque appel à la fonction doit retourner un état avec un index différent. (Vous pourrez utiliser à cette fin une référence globale.)*

```
value nfa_create: unit -> nfa_state
```

▶ **Question 2** *Écrivez deux fonctions* add_trans *et* add_eps_trans *permettant d'ajouter une transition ou une ϵ-transition entre deux états.*

```
value add_trans: nfa_state -> char -> nfa_state -> unit
value add_eps_trans: nfa_state -> nfa_state -> unit
```

1.3 Construction de Thompson

L'algorithme de transformation d'une expression rationnelle en automate que nous allons employer est connu sous le nom de « construction de Thompson ». Les automates qu'il produit ont la particularité d'avoir un seul état terminal (appelé par symétrie *état final*). De plus aucune transition ne sort de l'état final. Ainsi, un automate de Thompson sera représenté en Caml par un enregistrement du type suivant :

```
type thompson =
  { initial: nfa_state;
    final: nfa_state
  }
;;
```

La construction de Thompson procède par récurrence sur la structure de l'expression rationnelle. Les questions 3 à 7 introduisent quelques fonctions auxiliaires permettant d'effectuer les opérations de base sur les automates, que vous pourrez ensuite utiliser pour réaliser la construction de Thompson à la question 8.

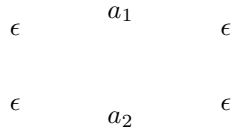
▶ **Question 3** *Écrivez une fonction* thompson_epsilon *telle que* thompson_epsilon () *retourne un automate de Thompson reconnaissant le langage vide.*

```
value thompson_epsilon: unit -> thompson
```

▶ **Question 4** *Écrivez une fonction* thompson_char *telle que* thompson_char c *retourne un automate de Thompson reconnaissant le langage* $\{c\}$.

```
value thompson_char: char -> thompson
```

Étant donnés deux automates de Thompson a_1 et a_2 reconnaissant respectivement les langages L_1 et L_2, on peut construire un automate de Thompson reconnaissant le langage $L_1 \cup L_2$ de la manière suivante :



▶ **Question 5** *Écrivez une fonction* thompson_or *implémentant la construction décrite ci-dessus.*

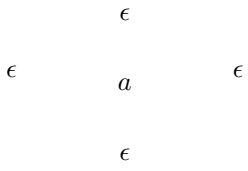
```
value thompson_or: thompson -> thompson -> thompson
```

De même, on peut construire un automate de Thompson reconnaissant le langage $L_1 L_2$ comme suit :

▶ **Question 6** *Écrivez une fonction* thompson_seq *implémentant la construction décrite ci-dessus.*

```
value thompson_seq: thompson -> thompson -> thompson
```

Enfin, étant donné un automate de Thompson a reconnaissant le langage L, l'automate suivant reconnaît le langage L^* :



▶ **Question 7** *Écrivez la fonction* thompson_star *correspondante.*

```
value thompson_star: thompson -> thompson
```

▶ **Question 8** *Déduisez-en une fonction* thompson *prenant en argument un expression rationnelle* r *et retournant l'état initial d'un automate non-déterministe reconnaissant le langage dénoté par* r.

```
value thompson: regexp -> nfa_state
```

2 Déterminisation

Ayant obtenu un automate qui reconnaît les mêmes chaînes que l'expression rationnelle de départ, il nous reste à programmer une fonction qui teste si une chaîne est reconnue ou non par l'automate. La réalisation d'un tel test sur un automate non déterministe est susceptible d'être peu efficace : puisque plusieurs transitions portant le même caractère peuvent sortir d'un même état, il faut quelquefois essayer plusieurs chemins qui épellent la chaîne à reconnaître. C'est la raison pour laquelle il est généralement préférable d'effectuer la reconnaissance sur un automate *déterministe*. L'objet de cette partie est donc de produire un tel automate à partir de celui obtenu grâce à la construction de Thompson.

2.1 Automates déterministes

On rappelle que, en Caml (comme dans beaucoup de langages de programmation), les caractères sont « numérotés » de 0 à 255. Dans cet énoncé, on note c_i le caractère qui porte le numéro i (par exemple c_{97} représente le caractère « a »). Les deux fonctions suivantes de la librairie standard permettent de convertir un caractère en son numéro et réciproquement :

```
value int_of_char: char -> int
value char_of_int: int -> char
```

Nous représentons un état d'un automate déterministe (DFA) par un enregistrement du type suivant :

```
type dfa_state =
  { mutable dtrans: dfa_state vect;
    mutable dterminal: bool
  }
;;
```

Le champ dterminal est un booléen indiquant si l'état est terminal ou non. Un tableau dtrans à 256 cases (une par caractère) décrit les transitions dont l'état est l'origine : étant donnés un état s et un entier i compris entre 0 et 255, $s.dtrans.(i)$ est l'extrémité de la transition d'origine s et étiquetée par c_i. Par souci de simplicité, nous supposons que chaque état est origine d'exactement une transition étiquetée par chaque caractère. Vous avez probablement vu en cours qu'il est toujours possible de se ramener à une telle situation en introduisant un état « poubelle » (i.e. un état s tel que pour tout i, $s.dtrans.(i) = s$).

► **Question 9** *Écrivez une fonction* dfa_create *telle que* dfa_create () *retourne un nouvel état d'un automate déterministe (dont toutes les transitions bouclent sur lui-même).*

Dans notre représentation des automates, chaque état contient directement la liste de ses successeurs. Ainsi, pour désigner un automate, il suffit de donner son état initial.

► **Question 10** *Écrivez une fonction* accept *déterminant si un automate déterministe reconnaît une chaîne de caractères.*

```
value accept: dfa_state -> string -> bool
```

2.2 Construction des sous-ensembles

L'algorithme que nous allons mettre en œuvre pour déterminiser un automate est connu sous le nom de « construction des sous-ensembles » (*subset construction* en anglais). Les états de l'automate déterministe correspondent à des ensembles d'états de l'automate de départ : tous les états qu'on peut atteindre à partir de l'état initial en suivant une certaine chaîne de caractères.

L'état initial de l'automate déterministe est l'ensemble des états qu'on peut atteindre en suivant la chaîne vide, c'est-à-dire l'état initial de l'automate de départ, plus tous les états qu'on peut atteindre à partir de l'état initial en suivant uniquement des ϵ-transitions. L'état correspondant à l'ensemble d'états $\{s_1, \ldots, s_n\}$ est terminal si et seulement si un des états $s_1, \ldots, s_n$ est terminal.

Pour voir où mène la transition sur un caractère c issue de l'ensemble d'états $\{s_1, \ldots, s_m\}$, on regarde toutes les transitions sur c issues des états s_1 à s_m dans l'automate initial. Soient $t_1, \ldots, t_n$ les états auxquelles elles mènent. Soit $\{t_1, \ldots, t_n, t'_1, \ldots, t'_{n'}\}$ la ϵ-fermeture de cet ensemble d'états (c'est-à-dire les états accessibles à partir de $t_1, \ldots, t_n$ en suivant uniquement des ϵ-transitions). On ajoute alors, dans l'automate déterministe produit, une transition sur c depuis l'état $\{s_1, \ldots, s_m\}$ vers l'état $\{t_1, \ldots, t_n, t'_1, \ldots, t'_{n'}\}$. On répète ce procédé jusqu'à ce qu'il soit impossible d'ajouter de nouvelles transitions.

Nous représenterons les ensembles d'états (d'automates non-déterministes) simplement par des listes de type nfa_state list (nous supposerons qu'un même état apparaît au plus une fois dans une telle liste).

► **Question 11** *Écrivez une fonction* mem_state *qui teste si un état (d'un automate non-déterministe) apparaît dans une liste d'états.*

```
value mem_state: nfa_state -> nfa_state list -> bool
```

▷ Indication. Vous ne pouvez pas utiliser l'égalité polymorphe de Caml (=) pour tester l'égalité de deux états : un état peut en effet se contenir récursivement dans sa liste des transitions et ainsi former une structure cyclique sur laquelle la comparaison ne terminerait pas. Pour comparer deux états, le plus simple est d'utiliser leurs index.

La représentation d'un ensemble d'état n'est pas unique : à un même ensemble correspond plusieurs listes ordonnées différemment. C'est pourquoi, pour pouvoir identifier de manière unique un ensemble d'état, nous définissons son *empreinte* comme la liste **triée** des numéros de ses états.

► **Question 12** *Écrivez une fonction* trace *qui calcule l'empreinte d'une liste d'états.*

```
value trace: nfa_state list -> int list
```

▷ Indication. Vous pouvez utiliser les fonctions suivantes de la bibliothèque standard.

value sort__sort : (' a -> 'a -> bool) -> 'a list -> 'a list
sort__sort cmp list retourne la liste list triée en utilisant la relation d'ordre cmp.

value map: ('a -> 'b) -> 'a list -> 'b list
map f [x1 ; ... xn] retourne la liste [f x1 ; ...; f xn].

Étant donné un ensemble S d'états d'un automate non-déterministe, on définit son ϵ-fermeture comme l'ensemble des états accessibles à partir d'un état de S en traversant zéro, une ou plusieurs ϵ-transitions.

► **Question 13** *Définissez une fonction* epsilon_closure *qui calcule la ϵ-fermeture d'une liste d'états (d'un automate non-déterministe).*

```
value epsilon_closure: nfa_state list -> nfa_state list
```

Soit S un ensemble d'états (d'un automate non-déterministe). On appelle *table des transitions* de S le tableau t à 256 cases tel que $t.(i)$ contienne la liste des états accessibles en effectuant une transition étiquetée par le caractère c_i depuis un état de S.

► **Question 14** *Écrivez une fonction* trans_table *qui calcule la table des transitions d'un ensemble d'états.*

```
val trans_table: nfa_state list -> nfa_state list vect
```

▷ Indication. Vous pouvez utiliser la fonction suivante de la bibliothèque standard :

value do_list : (' a -> unit) -> 'a list -> unit
do_list f [x1 ; ... xn] effectue la séquence f x1; ...; f xn.

▶ **Question 15** *Déduisez-en une fonction* determ *qui déterminise un automate. Cette fonction prendra pour argument l'état initial de l'automate non-déterministe et rendra celui de l'automate déterministe obtenu.*

```
value determ: nfa_state -> dfa_state
```

▷ Indication. Vous pourrez mémoriser les états déjà construits dans une référence sur une liste de paires formées d'une empreinte et de l'état de l'automate déterministe correspondant. Pour manipuler une telle *liste d'association*, la bibliothèque standard fournit deux fonctions :

value assoc : 'a -> ('a * 'b) list -> 'b
assoc a list retourne la valeur associée à la clef a dans la liste de paires list. C'est-à-dire, assoc a [...; (a,b); ...] = b si (a,b) est la première occurrence de la clef a dans list.

value mem_assoc : 'a -> ('a * 'b) list -> bool
Identique à assoc mais retourne un booléen indiquant si la clef apparaît dans la liste ou non.

Vous pourrez également utiliser la fonction suivante :

value exists : ('a -> bool) -> 'a list -> bool
exists f list retourne un booléen indiquant si au moins un des éléments x de list satisfait f (i.e. f x = true).

▶ **Question 16** *Déduisez-en une fonction* dfa_of_regexp *qui construit un automate déterministe reconnaissant le langage dénoté par une expression régulière.*

```
value dfa_of_regexp: regexp -> dfa_state
```

▶ **Question 17** *Quelles améliorations relatives aux structures de données suggéreriez-vous pour améliorer l'efficacité de la fonction de déterminisation d'un automate ?*

L'automate obtenu après la déterminisation peut comporter un grand nombre d'états. Il peut donc être souhaitable de chercher à réduire sa taille de manière à obtenir un automate équivalent (c'est-à-dire reconnaissant le même langage) mais comportant un nombre d'état minimal. Ce procédé est appelé *minimisation*. Il consiste à identifier des états *équivalents* : deux états sont équivalents si les suffixes du langage reconnus par l'automate à partir de ces états sont égaux.

▶ **MP & MP* – Option Informatique**
Année 2002–2003, Sixième TP Caml
Vincent Simonet (`http://cristal.inria.fr/~simonet/`)

Compilation d'expressions rationnelles

Un corrigé

▶ **Question 1** On définit tout d'abord une compteur entier « global », dont le contenu initial est 0.

```
let counter = ref 0
;;
```

À chaque appel de la fonction nfa_create, ce compteur est incrémenté d'une unité et un nouvel état est créé.

```
let nfa_create () =
  counter := !counter + 1;
  { trans = [];
    eps_trans = [];
    terminal = false;
    index = !counter
  }
;;
```

▶ **Question 2** Pour ajouter une transition, il suffit de modifier en place une des listes trans ou eps_trans de l'état origine.

```
let add_trans s1 c s2 =
  s1.trans <- (c, s2) :: s1.trans
;;

let add_eps_trans s1 s2 =
  s1.eps_trans <- s2 :: s1.eps_trans
;;
```

▶ **Question 3**

```
let thompson_epsilon () =
    let s1 = nfa_create ()
    and s2 = nfa_create () in
    add_eps_trans s1 s2;
    { initial = s1; final = s2 }
  ;;
```

▶ **Question 4**

```
let thompson_char c =
    let s1 = nfa_create ()
    and s2 = nfa_create () in
    add_trans s1 c s2;
    { initial = s1; final = s2 }
  ;;
```

▶ **Question 5**

```
let thompson_or a1 a2 =
    let s1 = nfa_create ()
    and s2 = nfa_create () in
    add_eps_trans s1 a1.initial;
    add_eps_trans s1 a2.initial;
    add_eps_trans a1.final s2;
    add_eps_trans a2.final s2;
    { initial = s1; final = s2 }
  ;;
```

▶ **Question 6**

```
let thompson_seq a1 a2 =
    add_eps_trans a1.final a2.initial;
    { initial = a1.initial; final = a2.final }
  ;;
```

▶ **Question 7**

```
let thompson_star a =
    let s1 = nfa_create ()
    and s2 = nfa_create () in
    add_eps_trans a.final a.initial;
    add_eps_trans s1 a.initial;
    add_eps_trans a.final s2;
    add_eps_trans s1 s2;
    { initial = s1; final = s2 }
  ;;
```

▶ **Question 8** La fonction thompson_aux construit récursivement l'automate en suivant la structure de l'expression régulière.

```
let rec thompson_aux = function
    Empty ->
      thompson_epsilon ()
  | Char c ->
      thompson_char c
  | Or (re1, re2) ->
      thompson_or (thompson_aux re1) (thompson_aux re2)
  | Seq (re1, re2) ->
      thompson_seq (thompson_aux re1) (thompson_aux re2)
  | Star re ->
      thompson_star (thompson_aux re)
;;
```

Pour obtenir la fonction thompson, il suffit d'appeler la fonction précédente et de marquer l'état final de l'automate obtenu comme terminal.

```
let thompson re =
  let a = thompson_aux re in
  a.final.terminal <- true;
  a.initial
;;
```

► **Question 9** Une écriture naturelle de la fonction dfa_state serait la suivante :

```
let dfa_create () =
  let rec s =
    { dtrans = make_vect 256 s;
      dterminal = false
    }
  in
  s
;;
```

Cependant, la définition récursive de s utilisée n'est pas acceptée par le compilateur Caml. Il faut donc procéder en deux étapes : d'abord créer l'état s avec un vecteur de transitions vide puis modifier en place ce vecteur.

```
let dfa_create () =
  let s =
    { dtrans = [||];
      dterminal = false
    }
  in
  s.dtrans <- make_vect 256 s;
  s
;;
```

► **Question 10** Pour tester si un automate déterministe reconnaît une chaîne, il suffit de lire cette dernière caractère par caractère en effectuant simultanément les transitions sur l'automate. La chaîne est reconnue si le dernier état obtenu est terminal. (La lecture ne peut pas échouer car nous avons supposé que tout état est l'origine d'exactement une transition étiquetée par chaque caractère.)

```
let accept initial string =
  let n = string_length string in
  let rec aux state i =
    if i = n then state.dterminal
    else aux state.dtrans.(int_of_char string.[i]) (i + 1)
  in
  aux initial 0
;;
```

► **Question 11** Notre fonction mem_state est analogue à la fonction usuelle de test d'appartenance à une liste (mem). Cependant, pour comparer deux états, on n'utilise pas directement l'égalité polymorphe de Caml (=) mais on compare leurs index.

```
let rec mem_state s = function
    [] -> false
  | s' :: list ->
      s.index = s'.index || mem_state s list
;;
```

► **Question 12** Il suffit de combiner deux appels aux fonctions de la bibliothèque standard map (afin d'obtenir la liste des index) et sort__sort (pour la trier).

```
let trace list =
  sort__sort (prefix <=) (map (function s -> s.index) list)
;;
```

► **Question 13** Le corps de notre fonction epsilon_closure est une fonction récursive, aux, qui prend deux arguments : (1) la liste accu des états qui font parti de la fermeture et qui ont déjà été traités et (2) la liste des états à traiter. Afin de s'assurer que la fonction termine même en présence de cycles d'ϵ-transitions, on teste si un état est dans l'accumulateur (accu) avant de le traiter.

```
let epsilon_closure list =
  let rec aux accu = function
      [] -> accu
    | s :: q ->
        if mem_state s accu
        then aux accu q
        else aux (s :: accu) (s.eps_trans @ q)
  in
  aux [] list
;;
```

► **Question 14**

```
let trans_table list =
    let t = make_vect 256 [] in
    do_list (function s ->
      do_list (function c, s' ->
        let i = int_of_char c in
        if not (mem_state s' t.(i)) then t.(i) <- s' :: t.(i)
      ) s.trans
    ) list;
    t
  ;;
```

► **Question 15** Le corps de notre fonction de déterminisation est une fonction récursive, transl qui prend pour argument une liste d'états de l'automate non-déterministe et retourne l'état de l'automate déterministe les représentant.
Chaque état de l'automate déterministe est stocké lors de sa création dans la (référence de) liste store, associé à l'empreinte de l'ensemble qu'il représente. Ainsi, lors de chaque appel à la fonction transl, deux cas peuvent se produire :

- L'ensemble d'états passé en argument (list) a déjà été rencontré. L'état associé dans la liste store est alors directement retourné.
- L'ensemble d'états est considéré pour la première fois. Dans ce cas, un nouvel état est créé et stocké dans la liste d'association. Ses transitions sont ensuite calculées (ce qui nécessite des appels récursifs à transl de manière à construire les extrémités des transitions). Enfin, l'état est marqué terminal si et seulement si un des états de l'ensemble qu'il représente est lui-même terminal.

```
let determ initial_state =
  let store = ref [] in

  let rec transl list =
    let tr = trace list in
    if mem_assoc tr !store
    then assoc tr !store
    else begin
      let s = dfa_create () in
      store := (tr, s) :: !store;
      let table = trans_table list in
      for i = 0 to 255 do
        s.dtrans.(i) <-
          transl (epsilon_closure table.(i));
        s.dterminal <-
          exists (fun s' -> s'.terminal) list
      done;
      s
    end
  in

  transl (epsilon_closure [initial_state])
;;
```

► **Question 16** Il suffit d'utiliser les fonctions thompson et determ que nous avons déjà définies.

```
  let dfa_of_regexp re =
    determ (thompsom re).initial
  ;;
```