TP 1

Manipulation d'expressions régulières

MPI/MPI*, lycée Faidherbe

Résumé

Dans ce T.P. on manipule des expressions régulière.

Dans un premier temps celles-ci seront représentées par un arbre puis on passera d'une écriture sous forme de chaîne de caractères aux arbres.

I Expressions régulières en arbres

On codera en OCaml une expression régulière par l'arbre qui lui est associé. Le type est

```
type regex =  Vide
            |Epsilon
            |Lettre of string
            |Union of regex*regex
            |Produit of regex*regex
            |Etoile of regex
```

Une chaîne de caractères est encadrée par des guillemets double, on se restreindra pour l'alphabet à des chaînes à un seul caractère qui est une lettre minuscule.
Par exemple l'expression régulière `(a|b)*aba(a|b)*` peut être représentée par

```
let r0 = Produit(
           Produit(Etoile (Union (Lettre "a", Lettre "b")),
                   Produit(Lettre "a", Lettre "b")),
           Produit(Lettre "a",
                   Etoile (Union (Lettre "a", Lettre "b"))))
```

I.1 Premières fonctions

Question 1 Écrire une fonction `ecrire : regex -> string` qui renvoie l'expression régulière associée à un arbre en utilisant le parcours infixe parenthésé.

```
let rec ecrire r =
  match r with
  |Vide -> "0"
  |Epsilon -> "1"
  |Lettre c -> c
  |Union (r1, r2) -> "(" ^ (ecrire r1) ^ "|" ^ (ecrire r2) ^ ")"
  |Produit (r1, r2) -> "(" ^ (ecrire r1) ^ (ecrire r2) ^ ")"
  |Etoile r ->  "(" ^ (ecrire r) ^ "*)"
```

Le symbole pour le vide sera `0` et celui pour ε sera `1`.

Question 2
Écrire une fonction `langageVide : regex -> bool` qui répond à la question :

"Le langage L[r] est-il vide ?"

```
let rec langageVide = function
  |Vide -> true
  |Epsilon -> false
  |Lettre _ -> false
  |Union (r1, r2) -> langageVide r1 && langageVide r2
  |Produit (r1, r2) -> langageVide r1 || langageVide r2
  |Etoile _ -> false
```

Question 3

Écrire une fonction `motVide : regex -> bool` qui répond à la question :
"Le langage L[r] contient-il le mot vide ?"

```
let rec motVide  = function
  |Vide -> false
  |Epsilon -> true
  |Lettre _ -> false
  |Union (r1, r2) -> motVide r1 || motVide r2
  |Produit (r1, r2) -> motVide r1 && motVide r2
  |Etoile _ -> true
```

Question 4 - Préfixes, suffixes et facteurs Écrire des fonctions `prefixes1 : regex -> string list`, `suffixes1 : regex -> string list` et `facteurs2 : regex -> string list` qui calculent les ensembles des préfixes de taille 1, des suffixes de taille 1 et des facteurs de taille 2 d'un langage dénoté par une expression régulière. *On ne se préoccupera pas, d'éviter les doublons.*

```
let rec prefixes1 = function
  |Vide -> []
  |Epsilon -> []
  |Lettre x -> [x]
  |Union (r1, r2) -> (prefixes1 r1) @ (prefixes1 r2)
  |Produit (r1, r2)
    -> if (langageVide r1) || (langage vide r2)
       then []
       else if motVide r1
            then (prefixes1 r1)@(prefixes1 r2)
            else prefixes1 r1
  |Etoile r -> prefixes1 r
```

```
let rec suffixes1 = function
  |Vide -> []
  |Epsilon -> []
  |Lettre x -> [x]
  |Union (r1, r2) -> (suffixes1 r1) @ (suffixes1 r2)
  |Produit (r1, r2)
    -> if (langageVide r1) || (langage vide r2)
       then []
       else if motVide r2
            then (suffixes1 r1)@(suffixes1 r2)
            else suffixes1 r2
  |Etoile r1 -> suffixes1 r1
```

On a besoin de recoller suffixes et préfixes pour obtenir des facteurs.

```
let rec assembler l1 l2 =
   let ajouter x = List.map (fun y -> x^y) l2 in
   match l1 with
   |[] -> []
   |t::q -> (ajouter t)@(assembler q l2)
```

```
let rec facteurs2 = function
  |Vide -> []
  |Epsilon -> []
  |Lettre x -> []
  |Union (r1, r2) -> (facteurs2 r1) @ (facteurs2 r2)
  |Produit (r1, r2)
    -> if (langageVide r1) || (langage vide r2)
       then []
       else  ((facteurs2 r1) @ (facteurs2 r2))
           @ (assembler (suffixes1 r1) (prefixes1 r2))
  |Etoile r
    ->  (facteurs2 r) @ (assembler (suffixes1 r) (prefixes1 r))
```

Question 5 - Sans doublons Proposer une solution pour éviter les doublons et l'implémenter.

On peut insérer en vérifiant l'appartenance, maintenir des listes triées ou supprimer les doublons à la fin.
Pour la deuxième solution.

```
let rec fusion l1 l2 =
   match l1, l2 with
   |[], _ -> l2
   |_, [] -> l1
   |t1 :: q1, t2 :: q2 when t1 = t2 -> t1 :: (fusion q1 q2)
   |t1 :: q1, t2 :: q2 -> if t1 < t2
                          then t1 :: (fusion q1 l2)
                          else t2 :: (fusion l1 q2);;
```

La troisième solution est très efficace si on utilise une table de hachage pour tester les éléments déjà présents.

I.2 Transformations

On implémente ici la traduction algorithmique d'exercices du TD01.

Question 6 - Prefixes Écrire une fonction `prefixes : regex -> regex` telle que `prefixes r` dénote le langage des préfixes des mots de L[r].

```
let rec prefixes r =
  match r with
  |Vide -> Vide
  |Epsilon -> Epsilon
  |Lettre x -> Union (Epsilon, Lettre x)
  |Union (r1, r2) -> Union (prefixes r1, prefixes r2)
  |Produit (r1, r2)
      -> if (langageVide r1) || (langage vide r2)
         then []
         else  Union (prefixes r1, Produit (r1, prefixes r2))
  |Etoile r -> Produit (Etoile r, prefixes r)
```

Question 7 - Mots ne contenant pas une lettre Écrire une fonction de type `motsSans : string -> regex -> regex` telle que `motsSans a r` dénote le langage des mots de L[r] qui ne contiennent pas a.

```
let rec motsSans a = function
  |Vide -> Vide
  |Epsilon -> Epsilon
  |Lettre x when x = a -> Vide
  |Lettre x -> Lettre x
  |Union(r1, r2) -> Union(motsSans a r1, motsSans a r2)
  |Produit (r1, r2)
      ->  Produit (motsSans a r1, motsSans a r2)
  |Etoile r -> Etoile (motsSans a r)
```

Question 8 - Mots contenant une lettre Écrire une fonction `motsAvec : string -> regex -> regex` telle que `motsAvec a r` dénote le langage des mots de L[r] qui contiennent a.

```
let rec motsAvec a = function
  |Vide -> Vide
  |Epsilon -> Vide
  |Lettre x when x = a -> Lettre a
  |Lettre x -> Vide
  |Union(r1, r2) -> Union(motsAvec a r1, motsAvec a r2)
  |Produit (r1, r2) -> Union(Produit (motsAvec a r1, r2),
                             Produit (r1, motsAvec a r2))
  |Etoile r -> Produit (Produit (Etoile r, motsAvec a r),
                        Etoile r)
```

Question 9 - Enlever la première occurrence Écrire une fonction `oterPrem : string -> regex -> regex` telle que `oterPrem a r` dénote le langage des mots de L[r] privés de leur premier a.

```
let rec oterPrem a = function
  |Vide -> Vide
  |Epsilon -> Vide
  |Lettre x when x = a -> Epsilon
  |Lettre x -> Vide
  |Union(r1, r2) -> Union (oterPrem a r1, oterPrem a r2)
  |Produit (r1, r2)
    ->  Union (Produit (oterPrem a r1, r2),
               Produit (motsSans a r1, oterPrem a r2))
  |Etoile r -> Produit (Etoile(motsSans a r),
                        Produit (oterPrem a r), Etoile r))
```

Question 10 - Enlever une occurence Écrire une fonction `oterUne : string -> regex -> regex` telle que `oterUne a r` dénote le langage des mots de $L[\mathtt{r}]$ privés d'une des occurences de a.

```
let rec oterUne a = function
  |Vide -> Vide
  |Epsilon -> Vide
  |Lettre x when x = a -> Epsilon
  |Lettre x -> Vide
  |Union(r1, r2) -> Union(oterUne a r1, oterUne a r2)
  |Produit (r1, r2) ->  Union(Produit (oterUne a r1, r2),
                              Produit (r1, oterUne a r2))
  |Etoile r -> Produit (Produit (Etoile r, oterUne a r),
                        Etoile r)
```

Question 11 - Sans vide Écrire une fonction `enlever0 : regex -> regex` telle que `enlever0 a r` dénote le même langage que $L[\mathtt{r}]$ mais ne contient pas $\varnothing$ si L est non vide.

On évite les pattern-matching imbriqués, on écrit des fonctions spécialisées.

```
let etoile0 = function
   |Vide -> Epsilon
   |r -> Etoile r
```

```
let union0 r1 r2 =
   match r1, r2 with
   |Vide, _ -> r2
   |_, Vide -> r1
   |_ -> Union (r1, r2)
```

```
let produit0 r1 r2 =
   match r1, r2 with
   |Vide, _ -> Vide
   |_, Vide -> Vide
   |_ -> Produit (r1, r2);;
```

La fonction elle-même est alors simple

```
let rec enlever0 = function
  |Union(r1, r2) -> union0 (enlever0 r1) (enlever0 r2)
  |Produit (r1, r2) -> produit0 (enlever0 r1) (enlever0 r2)
  |Etoile r -> etoile0 (enlever0 r)
  |r -> r;;
```

Question 12 - Ni vide ni mot vide Écrire une fonction `enlever01 : string -> regex` telle que `enlever01 a r` dénote le même langage que L[r] mais est de la forme $\varnothing$, ϵ, r′ ou r′|ϵ avec r′ réduite.

```
let etoile_reduite = function
   |Vide -> Epsilon
   |Epsilon -> Epsilon
   |Union(r1, Epsilon) -> Etoile r1
   |r -> Etoile r
```

```
let union_reduite r1 r2 =
   match r1, r2 with
   |Vide, _ -> r2
   |_, Vide -> r1
   |Epsilon, Epsilon -> Epsilon
   |Epsilon, Union(r4, Epsilon) -> Union(r4, Epsilon)
   |Epsilon, _ -> Union(r2, Epsilon)
   |Union(r3, Epsilon), Epsilon -> Union(r3, Epsilon)
   |_, Epsilon -> Union(r1, Epsilon)
   |Union(r3, Epsilon), Union(r4, Epsilon)
       -> Union(Union(r3, r4), Epsilon)
   |Union(r3, Epsilon), _ -> Union(Union(r3, r2), Epsilon)
   |_, Union(r4, Epsilon) -> Union(Union(r1, r4), Epsilon)
   |_ -> Union(r1, r2)
```

```
let produit_reduit r1 r2 =
   match r1, r2 with
   |Vide, _ -> Vide
   |_, Vide -> Vide
   |Epsilon, _ -> r2
   |_, Epsilon -> r1
   |Union(r3, Epsilon), Union(r4, Epsilon)
      -> Union(Union(Union(Produit (r3, r4), r3),
                     r4),
               Epsilon)
   |Union(r3, Epsilon), _ -> Union(Produit (r3, r2), r2)
   |_, Union(r4, Epsilon) -> Union(Produit (r1, r4), r1)
   |_ -> Produit (r1, r2);;
```

```
let rec enlever01 = function
  |Union(r1, r2)
      -> union_reduite (enlever01 r1) (enlever01 r2)
  |Produit (r1, r2)
      -> produit_reduit (enlever01 r1) (enlever01 r2)
  |Etoile r1 -> etoile_reduite (enlever01 r1)
  |r -> r;;
```

II Expressions régulières en chaînes de caractères

On va maintenant traduire en arbre les chaîne de caractères formant une expression régulière.

II.1 Expressions régulières bien parenthésées

On commence par le cos où les expressions régulières sont écrites sous forme complète, avec toutes les parenthèses, le produit marqué par un point et sans espace.
Une expression régulière est de la forme `0`, `1`, `x` , $(\texttt{r}*)$, $(\texttt{r}_1|\texttt{r}_2)$ ou $(\texttt{r}_1.\texttt{r}_2)$
avec $x \in \{a, b, c, \ldots, z\}$ et `r` , $\texttt{r}_1$ et $\texttt{r}_2$ expressions régulières.
Pour traduire une chaîne de caractère on va utiliser une fonction auxiliaire qui lit une expression régulière depuis une position i dans la chaîne `r` et renvoie le sous-arbre correspondant et la position caractère suivant.

- Si `r`.$[i]$ est `'0'`, la fonction renvoie `Vide` et `i` + `1`.
- Si `r`.$[i]$ est `'1'`, la fonction renvoie `Epsilon` et `i` + `1`.
- Si `r`.$[i]$ est un lettre minuscule $'\pi'$, la fonction renvoie `Lettre` "π" et `i` + `1`.
- Si `r`.$[i]$ est `'('`, on doit lire l'expression régulière à partir de la position $i+1$ qui renvoie un arbre `a` et un entier j. Le caractère suivant (à la position j) peut être
  `'*'` qui doit être suivi de `')'`,
  `'|'` suivi d'une autre expression régulière puis `')'`,
  ou `'.'` suivi d'une autre expression régulière puis `')'`.
- Si `r`.$[i]$ est un autre caractère, l'entrée n'est pas valide.

Pour gérer les entrées invalides on définit une exception qui renvoie la position du problème.

```
exception ChaineFautive of int;;
```

Pour tester si un caractère est une minuscule on compare son code :

```
if Char.code 'a' <= Char.code x && Char.code x <= Char.code 'z'
```

Pour convertir un caractère en chaîne :

```
String.make 1 x
```

Question 13 Écrire une fonction de traitement d'une chaîne de caractère représentant une expression régulière : `lire : string -> regex`.

```
let lire r =
  let rec sousER i =
    match r.[i] with
    |'0' -> Vide, i
    |'1' -> Eps, i
    |'(' -> begin
              let rg, j = sousER (i+1) in
              match r.[j] with
              |'*' -> if r.[j+1] = ')'
                      then Etoile rg, j+2
                      else raise (ChaineFautive (j+1))
              |'+' -> let rd, k = sousER (j+1) in
                      if r.[k] =  ')'
                      then Union (rg, rd), k+1
                      else raise (ChaineFautive k)
              |'.' -> let rd, k = sousER (j+1) in
                      if r.[k] =  ')'
                      then Produit (rg, rd), k+1
              |_ -> raise (ChaineFautive j)
            end
    |c -> Feuille c, i+1 in
  let arbre, n = aux 0 in
    if n = String.length chaine
    then arbre
    else raise (ChaineFautive n);;
```

II.2 Notation polonaise inversée

En 1920 le mathématicien polonais Jan Lukasiewicz propose la notation pré-fixé (ou polonaise) qui consiste à considérer les opérations comme des opérateurs à 2 variables, $7+11$ s'écrit $+\ 7\ 11$. L'avantage est que les parenthèses deviennent inutiles.
Dans la notation polonaise inversée l'opérateur est **après** ses arguments, $7\ 11\ +$.
Dans le cas des expressions régulières, la notation polonaise inversée de $(a|b)*(ab)$ est $ab|*ab..$; on notera qu'il est nécessaire de réintroduire le point.t
Cette notation permet d'effectuer les calculs en utilisant simplement une pile [1].

- On lit l'expression terme-à-terme :
- si on lit une valeur numérique, elle est empilée,
- si on lit une opération ♣, on dépile les deux derniers opérandes ou le dernier a(et b) et on empile le résultat du calcul $b \clubsuit a$ (b ayant été placé avant a dans la pile) ou $\clubsuit\ a$.

Un exemple pour "ab|*ab..", on simplifie la notation des arbres.

1. C'est pour cette raison qu'elle a été utilisée comme interface utilisateur avec les calculatrices de bureau de Hewlett-Packard (HP-9100), puis avec la calculatrice scientifique HP-35 en 1972.

Lecture	Action	Pile
	On crée une pile vide	
a	On empile	a
b	On empile	a b
|	On dépile 2 arbres, on empile l'union	U(a, b)
|	On dépile 1 arbre, on empile l'étoile	E(U(a, b))
a	on empile	E(U(a, b)) a
b	on empile	E(U(a, b)) a b
.	On dépile 2 arbres,	E(U(a, b))
	on empile lle produit	E(U(a, b)) P(a, b)
.	On dépile 2 arbres, on empile lle produit	P(E(U(a, b)),P(a, b))

On peut alors dépiler le résultat :

```
Produit (Etoile (Union (Lettre "a", Lettre "b")),
         Produit (Lettre "a", Lettre "b"))
```

On pourra utiliser le module `Stack` (`create, push, pop, is_empty, top`) pour utiliser une pile, mais vous pouvez créer vous-même votre pile si cela vous semble utile.

Question 14 Écrire une fonction de traitement d'une chaîne de caractère représentant une expression régulière en notation NPI : `versNPI : string -> regex`.

```
let lireNPI r =
 let n = String.length r in
 for i = 0 to (n-1) do
   match r.[i] with
   |'0' -> Stack.push Vide pile
   |'1' -> Stack.push Epsilon pile
   |'|' -> let rg, rd = depiler2 pile in
           Stack.push (Union (rg, rd)) pile
   |'.' -> let rg, rd = depiler2 pile in
           Stack.push (Produit (rg, rd)) pile
   |'*' -> let r = Stack.pop pile in
           Stack.push (Etoile r) pile
   |x -> if   Char.code 'a' <= Char.code x
           && Char.code x <= Char.code 'z'
         then Stack.push (Lettre (String.make 1 x)) pile
         else raise (ChaineFautive i)
  done;
  let r = Stack.pop pile in
    if Stack.is_empty pile
    then r
    else raise (ChaineFautive n)
```

II.3 Expressions régulières simplifiées

Pour traduire une entrée simplifiée en une forme NPI, on sort les lettres quand on les lit et on ajoute les opérations quand les arguments sont passés. Pour cela on empile les opérateurs et les parenthèses ouvrantes dans une pile et on les sort quand apparaît une fermeture : ce peut être une parenthèse fermante, on dépile toutes les opérations jusqu'à une parenthèse ouvrante, ou un opérateur | qui fait dépiler les produits.

En raison de sa priorité maximale et de son absence de second argument, l'étoile est considérée comme une lettre.
On notera qu'il faut gérer la création du produit.

Question 15 Écrire une fonction de transformation d'une chaîne de caractère représentant une expression régulière vers cette même exoression régulière en notation NPI : `versNPI : string -> string`.

```
let versNPI r =
 let n = String.length r in
 let pile = Stack.create () in
 let out = ref "" in
 let attendOp = ref false in
 for i = 0 to (n-1) do
   match r.[i] with
   |'0' -> out := !out ^ "0";
           if !attendOp then Stack.push "." pile;
           attendOp := true
   |'1' -> out := !out ^ "1";
           if !attendOp then Stack.push "." pile;
           attendOp := true
   |'.' -> Stack.push "." pile;
           attendOp := false
   |'|' -> while     not (Stack.is_empty pile)
                 && (Stack.top pile = ".")
             do out := !out ^ (Stack.pop pile) done;
           Stack.push "|" pile;
           attendOp := false
   |'*' -> out := !out ^ "*";
           attendOp := true
   |'(' -> if !attendOp then Stack.push "." pile;
           Stack.push "(" pile;
           attendOp := false
   |')' -> while not (Stack.is_empty pile)
                 && (   (Stack.top pile = ".")
                     || (Stack.top pile = "|"))
             do out := !out ^ (Stack.pop pile) done;
           let _ = Stack.pop pile in attendOp := true
           (* On enlève la parenthèse ouvrante *)
   |x -> if !attendOp then Stack.push "." pile;
         if   Char.code 'a' <= Char.code x
           && Char.code x <= Char.code 'z'
         then begin
           out := !out ^ (String.make 1 x);
           attendOp := true end
         else raise (ChaineFautive i)
 done;
 while not (Stack.is_empty pile)
   do out := !out ^ (Stack.pop pile) done;
 !out
```

Solutions