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

Exercice 1

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

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

Exercice 2

Écrire une fonction `langageVide : regex -> bool` qui répond à la question :
"Le langage L[r] *est-il vide ?"*

Exercice 3

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

Exercice 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.*

Exercice 5 - Sans doublons

Proposer une solution pour éviter les doublons et l'implémenter.

I.2 Transformations

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

Exercice 6 - Prefixes

Écrire une fonction `prefixes : regex -> regex` telle que `prefixes r` dénote le langage des préfixes des mots de $L[\mathtt{r}]$.

Exercice 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 $L[\mathtt{r}]$ qui ne contiennent pas a.

Exercice 8 - Mots contenant une lettre

Écrire une fonction `motsAvec : string -> regex -> regex` telle que `motsAvec a r` dénote le langage des mots de $L[\mathtt{r}]$ qui contiennent a.

Exercice 9 - Enlever la première occurrence

Écrire une fonction `oterPrem : string -> regex -> regex` telle que `oterPrem a r` dénote le langage des mots de $L[\mathtt{r}]$ privés de leur premier a.

Exercice 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.

Exercice 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.

Exercice 12 - Ni vide ni mot vide

Écrire une fonction `enlever01 : string -> regex` telle que `enlever01 a r` dénote le même langage que $L[\mathtt{r}]$ mais est de la forme $\varnothing$, ϵ, $\mathtt{r}'$ ou $\mathtt{r}'|\epsilon$ avec $\mathtt{r}'$ réduite.

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
```

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

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, $7\ +\ 11$ s'écrit $7\ 11\ +$.
Dans le cas des expressions régulières, la notation polonaise inversée de $(\texttt{a}|\texttt{b})*(\texttt{ab})$ est $\texttt{ab}|*\texttt{ab}.$. ; 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,

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.

- 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 ♣ a (b ayant été placé avant a dans la pile) ou ♣ a.

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

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, 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.

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

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.

> **Exercice 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`.