TP 2 : Expressions rationnelles et automates

25 septembre

Ce TP aborde certaines questions sur les langages reconnaissables : trouver un mot de longueur minimale dans deux situations : si le langage est représenté par une expression régulière mais aussi dans le cas où on connait un automate le reconnaissant. On construira aussi une fonction permettant de déterminer si un mot appartient au langage dénoté par une expression régulière. Enfin, on manipulera un peu la fonction `grep` afin de découvrir une des applications des expressions régulières.

1 Expressions régulières

▷ **Question 1.** Définissez par induction structurelle une fonction des expressions rationnelles qui détermine si un langage est vide. ◁

On définit le type des expressions rationnelles :

```
type expr = Vide
 | Epsilon
 | Lettre of char
 | Union of expr * expr
 | Concat of expr * expr
 | Etoile of expr ;;
```

▷ **Question 2.** Écrire la fonction `vide : expr->bool`. qui calcule la fonction précédente. ◁

▷ **Question 3.** Écrire une fonction `a_eps : expr -> bool` telle que `a_eps e` s'évalue à vrai si et seulement si le langage dénoté par l'expression régulière représentée par `e` contient le mot vide ϵ. ◁

▷ **Question 4.** Écrire une fonction `est_eps : expr -> bool` telle que l'appel `est_eps e` s'évalue à vrai si et seulement si le langage dénoté par l'expression régulière représentée par `e` est exactement $\{\epsilon\}$, le langage formé de l'unique mot vide. ◁

▷ **Question 5.** Définissez par induction structurelle une fonction des expressions rationnelles vers les entiers qui calcule la longueur du plus court mot de L si le langage n'est pas vide, et ∞ sinon. ◁

▷ **Question 6.** Écrire la fonction `longueur_mot_min` qui calcule la fonction précédente. Elle retournera un objet de type `int option`. ◁

▷ **Question 7.** On considère l'expression régulière $e_1 = ab^*a$. Définir cette expression régulière en CAML. Determiner le langage $L(e_1)$ dénoté par cette expression régulière. ◁

Si $L \subset \Sigma^*$ est un langage, son résiduel à gauche (on dira simplement résiduel) pour un mot $u \in \Sigma$ est le langage $u^{-1}L = \{v \in \Sigma, uv \in L\}$. Autrement dit c'est le langage des mots v qui peuvent compléter le mot u pour obtenir un mot de L.

▷ **Question 8.** Montrer que $u \in L$ si et seulement si $\epsilon \in u^{-1}L$. ◁

On va maintenant chercher à écrire une fonction qui détermine si un mot u est dans un langage L. Pour un mot $u \in \Sigma^*$ l'objectif est donc de calculer $u^{-1}L$ et de savoir si ϵ est dans ce langage pour savoir si $u \in L$.

▷ **Question 9.** Pour $u = av$, avec $u, v \in \Sigma^*$ et $a \in \Sigma$, montrer que $u^{-1}L = v^{-1}(a^{-1}L)$. ◁

Il suffit donc de lire les lettres de u une par une et de déterminer successivement les langages résiduels pour aboutir à $u^{-1}L$.
Par exemple, déterminons si aba appartient à $ab^*a = L(e_1)$.

▷ **Question 10.**
Déterminer $a^{-1}L(e_1)$ ainsi qu'une expression régulière e_2 qui dénote ce langage. ◁

▷ **Question 11.** Déterminer $(ab)^{-1}L = b^{-1}L(e_2)$ ainsi qu'une expression regulière e_3 qui dénote ce langage. ◁

▷ **Question 12.** Écrire une fonction `residuel : char -> expr -> expr` qui étant donné un caractère `a` et une expression régulière `e` s'évalue en une expression régulière `ê` qui dénote le langage résiduel $L(\hat{e}) = a^{-1}L(e)$. ◁

▷ **Question 13.** Écrire une fonction `appartient : char list -> expr -> bool` qui vérifie si un mot représenté par une liste de caractères appartient au langage dénoté par une expression régulière. ◁

▷ **Question 14.** Écrire de même une fonction `appartient_bis : string -> expr -> bool` qui a le même comportement que la fonction précédente mais avec une représentation des mots par le type string de CAML. ◁

2 REGEX ET GREP

Il est fortement conseillé de se référer aux explications sur la syntaxe concrète des expressions régulières données à la fin du TP. Dans le cadre du programme, aucune connaissance de la syntaxe concrète n'est exigible : vous êtes seulement censés en avoir entendu parler, et éventuellement savoir vous en servir ***si on vous la rappelle****.*

L'utilitaire `grep` permet de rechercher des occurrences d'une expression régulière dans un fichier. Une occurrence d'une expression régulière est une suite de caractères formant un mot (au sens mathématique du terme) appartenant au langage de l'expression : on utilise souvent l'anglicisme *match*.

- L'utilisation de base est `grep [options] pattern [file]`. Si aucun fichier n'est précisé, la recherche est faite sur l'entrée standard.
- `grep` travaille ligne par ligne : une occurrence à cheval sur deux lignes n'est pas considérée.
- La seule option que nous utiliserons ici est : `-E` qui active le mode " expressions régulières étendues " ;

▷ **Question 15.** Le fichier `francais.txt` (téléchargeable sur cahier de prépa) contient un dictionnaire français assez complet (323422 formes), avec les flexions (pluriel, féminin, formes conjuguées des verbes. . .), à raison d'un mot par ligne. Les caractères ont été convertis en ASCII « standard » pour simplifier (il n'y a plus d'accents, de cédilles. . .), et les traits d'union ont été supprimés (`timbreposte` au lieu de *timbre-poste*).

1. Quels sont les mots qui contiennent au moins un w et au moins un q ? (On trouve 6 mots.)
2. Quels sont les mots qui contiennent au moins 6 fois la lettre i ? (On trouve 4 mots.)
3. Quels sont les mots qui contiennent au moins 10 voyelles ? (On trouve 112 mots de *anticonstitutionnellement* à *technobureaucratiques*.)
4. Quels sont les mots qui commencent par un p et contiennent au moins 9 voyelles ? (On trouve 28 mots, de *panoramiqueraient* à *psychophysiologiques*)
5. Quels sont les mots d'exactement 12 lettres ne contenant ni a ni e ni i ? (On trouve 9 mots, de *boursouflons* à *turcmongols*.)
6. Quels sont les mots dans lesquels chaque groupe consécutif de 2 lettres contient au moins un s ? (On trouve on trouve 37 mots, de *as* à *uses*.)

◁

3 Automates déterministes

Les automates que l'on considère dans cette partie sont déterministes (pour chaque couple (état,lettre), il y a *au plus* une transition).

L'enregistrement `taille` donne le nombre d'états. L'état initial est donné par l'enregistrement `initial`, et les états finaux sont donnés par le tableau de booléens `final`. Le tableau `transitions` contient l'ensemble des transitions.

La taille des tableaux `transitions` et `final` doit être `taille`, mais ceci n'est pas spécifié dans le type.

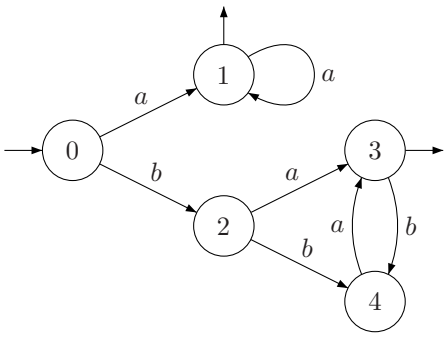
```
type automate =
  { taille :  int ;
    initial :  int ;
    transitions :  (char * int) list array ;
    final :  bool array } ;;
```

▷ **Question 16.** Il existe dans la bibliothèque Caml une fonction `assoc` définie de la manière suivante: `assoc : 'a -> ('a * 'b) list -> 'b` qui gère les listes de couples (appelées aussi listes associatives) : par exemple `assoc 2 [(1,'a'); (2,'b') ; (2,'c') ; (3,'d')]` vaut `'b'`. La fonction `assoc` déclenche l'exception `Not found` en cas d'échec.

Ecrire une telle fonction. ◁

▷ **Question 17.** Écrire une fonction `calcul_det` qui étant donné un mot et un automate supposé déterministe, détermine si l'automate accepte ce mot (on pourra utiliser la fonction `assoc` ou bien la fonction que vous venez de coder). Quelle est sa complexité ?

Définir l'automate 1 représenté ci-dessous, et vérifier sur les exemples aa, aba et bab que la fonction `calcul_det` est correcte. ◁



▷ **Question 18.** Ecrire une fonction qui fait un parcours en profondeur de l'automate depuis l'état initial et se contente d'afficher les noms des états parcourus. ◁

▷ **Question 19.** Écrire une fonction `accessible` qui supprime les états inaccessibles d'un automate. Pour cela, on doit renuméroter les états, on pourra maintenir deux tableaux `tab_conversion` et `tab_inversion` qui gèrent la correspondance entre nouveaux et anciens états. On adaptera le code de la question précédente pour parcourir les sommets depuis l'état initial en les renumérotant. ◁

▷ **Question 20.** Écrire une fonction `est_vide_auto` qui détermine si le langage de l'automate est vide. Écrire une fonction `longueur_mot_min_auto` qui calcule la longueur du mot minimal accepté par l'automate. Elle retournera un objet de type `int option`. ◁

▷ **Question 21.** Écrire une fonction `langage_auto` qui retourne la liste de tous les mots de taille minimale reconnu par l'automate. Quelle est sa complexité? ◁

4 Syntaxe concrète pour les expressions régulières

La syntaxe des expressions régulières n'est pas totalement standardisée dans les différents langages de programmation, et surtout elle diffère significativement de la syntaxe " mathématique ". On donne ici la syntaxe

utilisée par `grep -E`, dite *syntaxe des expressions régulières étendues* et l'on adjoint temporairement à la langue française le verbe *matcher*.

Traduction des expressions régulières mathématiques

- La concaténation est implicite : `"aba"` correspond donc tout simplement à aba ;
- L'étoile de Kleene est traduite par un `*`, et sa priorité est maximale : `ab*c*` correspond à $ab^*c^* = a(b^*)(c^*)$.
- Le | de l'union est traduit par `|`, et sa priorité est minimale : `chat*|chien` correspond simplement à $chat^*|chien$.
- Les parenthèses ont leur rôle habituel.
- Le $^+$ de $a^+ = aa^*$ est traduit par un caractère $+$, sa priorité est la même que celle de *.

Classes de caractères

- `.` (le caractère " point ") accepte un caractère quelconque.
- `[a3ju]` accepte un caractère parmi `a`, `3`, `j` et `u`. Cela équivaut essentiellement à `a|3|j|u` en plus efficace (mais restreint à un choix entre *caractères* et pas entre expressions quelconques).
- `[^a3ju]` accepte un caractère quelconque **sauf** `a`, `3`, `j` ou `u`.
- `[a-eC-Z]` accepte une lettre minuscule entre `a` et `e` ou une lettre majuscule entre `C` et `Z`.
- `\d` ou `[0-9]` accepte un chiffre.
- `\s` accepte un caractère d'espacement (espace, tabulation, retour à la ligne...).
- `\S` accepte tout, sauf un caractère accepté par `\s`.
- `\w` accepte `[a-zA-Z0-9_]` (un *word character*).
- `\W` accepte tout sauf les caractères acceptés par `\w`.
- ...

Autres quantificateurs

- Le `?` accepte 0 ou 1 occurrence de l'expression précédente : `"a(bc)?a"` correspond à $a(bc + \epsilon)a$.
- Un `{n}` accepte exactement n occurrences de l'expression précédente.
- `{m, n}` accepte entre m et n occurrences de l'expression précédente (au sens large).
- `{m,}` accepte au moins m occurrences de l'expression précédente.

Points d'ancrage Pour simplifier, on se limite aux chaînes constituées d'une seule ligne (dans le cas de `grep` c'est justifié car le traitement se fait ligne par ligne). Quand on demande si une expression accepte une ligne, on demande en fait si elle accepte au moins un *facteur* de la ligne : c'est comme si une expression régulière e était implicitement convertie en $\Sigma^* e \Sigma^*$. Les deux caractères suivants servent à modifier ce comportement.

- Le caractère `^` n'accepte que le début d'une ligne (et ne consomme pas de caractère). Autrement dit, il force le match à commencer au début de la ligne (ou, si l'on préfère, il transforme le $\Sigma^* e \Sigma^*$ en $e\Sigma^*$). Ainsi, `^abc` accepte exactement les lignes commençant par `"abc"`.
- Le caractère `$` joue le même rôle pour la fin de la ligne. Ainsi `abc$` n'accepte que les lignes se terminant par ¨`abc`¨ et `^abc$` n'accepte que la ligne réduite à `"abc"`.

Choix du *match* Pour l'instant, on s'est simplement demandé si la ligne était acceptée par l'expression. En réalité (sauf si l'on a utilisé `^` et `$`), l'expression va *a priori* accepter un ou des *morceaux* de la ligne, appelés *occurrences* ou *matchs*. Il y a deux règles à retenir :

- deux occurrences ne peuvent se chevaucher, puisque les caractères sont ¨ consommés par les occurrences ;
- une occurrence sera toujours la plus longue possible, et en cas d'égalité commencera le plus à gauche possible.

Ainsi :

- l'expression `aba` sur la ligne `cabadaba` donnera `c`aba`d`aba` (pas de conflit entre les deux occurrences) ;
- la même expression sur la ligne `ababa` donnera `aba`ba (*match* plus à gauche que `ab`aba`) ;
- l'expression `(ab|bab)*` sur la ligne `cababbab` donnera `ca`babbab` (ce *match* étant plus long que `c`abab`bab, et que tous les autres *matchs* possibles).