TP n°1 - Automates et langages réguliers

Notions abordées

- création d'un module en OCaml
- implémentation du type ensemble par des listes
- modélisation d'un automate sans ε-transition en OCaml
- test d'acceptation d'un mot par un automate, même non déterministe
- complétion et déterminisation d'automates
- énumération des mots reconnus par un automate de longueur fixée
- modélisation des expressions régulières en OCaml
- énumération des mots d'une expression régulière de longueur fixée

Exercice 1 : Implémentation d'automates en OCaml

Dans tout cet exercice, les automates ont pour ensemble d'états un intervalle d'entiers de la forme $[\![0, n[\![$ et pour alphabet les m premières lettres de l'alphabet latin pour un certain $m \in \mathbb{N}$, de sorte qu'on l'identifie à $[\![0, m[\![$ ♣.

1. Automates déterministes et complets

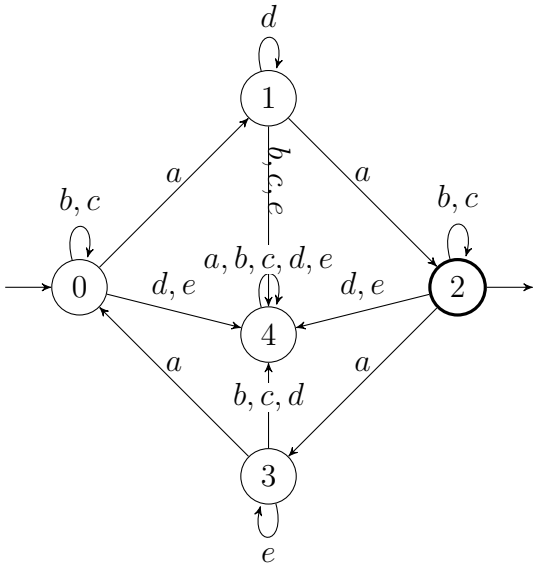
On s'intéresse dans un premier temps à la manipulation d'automates déterministes et complets. De tels automates seront représentés au moyen d'une structure contenant 3 champs.

- Un champ `init` qui est un entier de $[\![0, n[\![$ représentant l'état initial.
- Un champ `fin` qui est une liste d'entiers de $[\![0, n[\![$, sans doublons, représentant les états finaux.
- Un champ `trans` qui est une matrice de dimensions $n \times m$ d'entiers de $[\![0, n[\![$. La case `trans.(`i`).(`j`)` de la matrice représente l'état que l'on atteint en lisant la j-ème lettre de l'alphabet depuis l'état i. Ainsi pour représenter le fait que l'unique transition sortant de l'état 3 étiquetée par la lettre `'a'` conduit à l'état 0, le coefficient `trans.(3).(0)` vaut `0`.

```
type automate_d = {
  init  : int;                (* l'état initial *)
  trans : int array array;    (* la table des transitions *)
  fin   : int list            (* la liste des états finaux *)
}
```

Ainsi l'automate déterministe et complet ci-dessous à gauche est représenté par la valeur OCaml ci-dessous à droite. On remarque que l'état 4 de cet automate est un état puits.

♣. Par exemple pour $m = 3$, l'alphabet est $\{a, b, c\}$




```
let auto_1 = {
  init = 0;
  trans =
    (*  | a | b | c | d | e |*)
    [| [| 1 ; 0 ; 0 ; 4 ; 4 |]; (* état 0 *)
       [| 2 ; 4 ; 4 ; 1 ; 4 |]; (* état 1 *)
       [| 3 ; 2 ; 2 ; 4 ; 4 |]; (* état 2 *)
       [| 0 ; 4 ; 4 ; 4 ; 3 |]; (* état 3 *)
       [| 4 ; 4 ; 4 ; 4 ; 4 |]; (* état 4 *)
    |];
  fin = [2]
}
```

Q. 1 Définir une fonction `int_of_letter : char -> int` prenant en argument un caractère `c` de l'alphabet latin et retournant son numéro. Par exemple, `(int_of_letter 'a')` vaut `0` et `(int_of_letter 'e')` vaut `4`. Si le caractère pris en argument n'est pas une lettre minuscule de l'alphabet latin, cette fonction lèvera une exception.

Indication. *On rappelle que la fonction `int_of_char : char -> int` retourne le code ASCII d'un caractère.*

Q. 2 Définir une fonction `delta : automate_d -> int -> char -> int` prenant en arguments : un automate a, un état q, une lettre l et retournant l'unique état q' tel que (q, l, q') est une transition de l'automate a, autrement dit l'unique état accessible en une étape depuis l'état q en lisant l.

Indications. *On rappelle que pour accéder à un champ d'une structure la syntaxe est `objet.nom` où `objet` est la structure et `nom` le nom du champ de la structure que l'on souhaite lire.*

Q. 3 Définir une fonction `delta_etoile : automate_d -> int -> string -> int` prenant en arguments : un automate a, un état q, un mot w et retournant l'unique état q' de l'automate a qui est accessible depuis l'état q en lisant le mot w.

Q. 4 En déduire une fonction `accepte : automate_d -> string -> bool` prenant en arguments : un automate a, un mot w et retournant si le mot w est accepté par l'automate a.

Indication. *On rappelle que la fonction `List.mem : 'a -> 'a list -> bool` permet de tester si un élément se trouve, ou non, dans une liste.*

Définition récursive. Fixons un automate a et un mot w, dont les lettres sont indicées de 0 à $|w| - 1$. Étant donnés, un état q, et i un indice de $[\![0, |w|]\!]$, on définit alors le prédicat AD(q,i)♣ suivant.

$$\text{AD}(q, i) : \text{“La lecture de } w \text{ à partir de l'indice } i \,^{\heartsuit}\text{, dans l'automate } a,$$
$$\text{en partant de l'état } q \text{ conduit à un état final.”}$$

Q. 5 En s'autorisant la notation $\delta(q, l)$ pour désigner l'unique état atteint en lisant la lettre l depuis l'état q, proposer une définition récursive du prédicat $\text{AD}(q, i)$.

♣. Accepte Depuis

♡. Cela signifie la lecture du mot $w_i w_{i+1} \dots w_{|w|-1}$, soit la lecture de ε si $i = |w|$.

Q. 6 En déduire une fonction `accepte_depuis : automate_d -> int -> string -> int -> bool` prenant en arguments : un automate a, un état q, un mot w un entier i et retournant $\mathrm{AD}(q, i)$ ♣.

Q. 7 En déduire une seconde version de la fonction `accepte : automate -> string -> bool` précédemment définie.

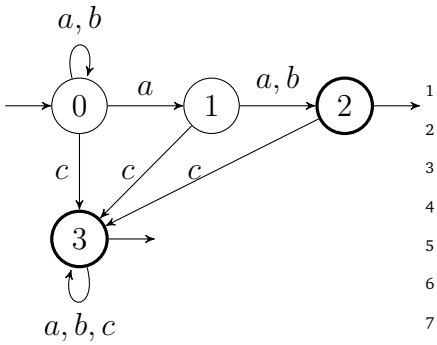
2. Automates non déterministes

On s'intéresse maintenant à la manipulation d'automates non nécessairement déterministes et ni complets. De tels automates seront représentés au moyen d'une structure contenant 3 champs.

- Un champ `init` qui est un entier de $[\![0, n[\![$ représentant l'état initial.
- Un champ `fin` qui est une liste d'entiers de $[\![0, n[\![$, sans doublons, représentant les états finaux.
- Un champ `trans` qui est une matrice de dimensions $n \times m$ de listes d'entiers de $[\![0, n[\![$. La case `trans.(`i`).(`j`)` de la matrice représente l'ensemble des états que l'on atteint en lisant la j-ème lettre de l'alphabet depuis l'état i par une liste sans doublons (possiblement vide). Ainsi pour représenter le fait qu'au automate admet une transition $(0, \texttt{'a'}, 0)$ et une transition $(0, \texttt{'a'}, 1)$, le coefficient `trans.(0).(0)` vaut `[0; 1]`.

```
type automate_nd = {
  init  : int list;             (* les états initiaux *)
  trans : int list array array; (* la table des transitions *)
  fin   : int list              (* les états finaux *)
}
```

Ainsi l'automate ci-dessous à gauche est représenté par la valeur OCaml ci-dessous à droite. On remarque que l'état 3 de cet automate est un état puits.




```
let auto_2 = {
  init = [0];
  trans =
(* lettres |   a   |   b   |   c   |*)
       [| [| [0; 1]; [0]   ; [3]   |];    (* état 0 *)
          [| [2]   ; [2]   ; [3]   |];    (* état 1 *)
          [| []    ; []    ; [3]   |];    (* état 2 *)
          [| [3]   ; [3]   ; [3]   |]     (* état 3 *)
       |];
  fin = [2; 3]
}
```

Q. 8 Définir une fonction `est_deterministe: automate_nd -> bool` prenant en argument un automate et testant si celui-ci est déterministe.

Q. 9 Définir une fonction `enleve_doublons : 'a list -> 'a list` prenant en argument une liste et en retirant les doublons.

On accepte une fonction de complexité quadratique ici.

♣. pour cet automate a et ce mot w

Q. 10 Définir une fonction `delta_singleton : automate_nd -> int -> char -> int list` prenant en arguments un automate a, un état q, une lettre l et retournant l'ensemble[♣] des états q' tels que (q, l, q') est une transition de l'automate a.

Q. 11 En déduire une fonction `delta : automate_nd -> int list -> char -> int list` prenant en arguments un automate a, un ensemble d'états[♡] s et une lettre l et retournant l'ensemble des états[♠] accessibles depuis un état de s en lisant la lettre l dans l'automate a.

Q. 12 Définir une fonction `delta_etoile : automate_nd -> int list -> string -> int list` prenant en arguments : un automate a, un ensemble d'états[♢] s, un mot w et retournant l'ensemble des états[*] accessibles depuis un état de s en lisant le mot w dans l'automate a.

Q. 13 En déduire une fonction `accepte : automate_nd -> string -> bool` prenant en arguments : un automate a, un mot w et retournant si le mot w est accepté par l'automate a.

Définition récursive. Dans un automate non nécessairement déterministe, la définition récursive du prédicat AD se voit complexifiée.

$$\begin{cases} \mathsf{AD}(q,i) = q \in a.\texttt{fin} & \text{si } i = |w| \\ \mathsf{AD}(q,i) = \exists q' \in \delta(q, w_i), \mathsf{AD}(q', i+1) & \text{sinon} \end{cases}$$

Q. 14 En déduire une deuxième définition de la fonction `accepte`.

Q. 15 Exhiber un automate à $\Theta(n)$ états sur l'alphabet $\Sigma = \{a\}$ pour lequel le nombre d'appels récursifs induits par l'appel à la fonction `accepte` sur un mot de taille $\Theta(n)$ est en $\Theta(2^n)$.

3. Déterminisation

Q. 16 ★ Proposer une fonction `determinise : automate_nd -> automate_d` appliquant l'algorithme de déterminisation à l'automate qui lui est passé en argument.

Indications. *On rappelle que les sous-ensembles de $[\![0, n-1]\!]$ peuvent être représentés par n entiers de $\{0, 1\}$, et qu'une telle séquence peut aussi représenter un entier de l'intervalle $[\![0, 2^n - 1]\!]$ grâce à l'écriture en binaire. Ainsi les sous-ensembles de $[\![0, n-1]\!]$ et les entiers de $[\![0, 2^n - 1]\!]$ sont en bijection. On pourra définir deux fonctions `int_of_intlist : int list -> int` et `intlist_of_int : int -> int list` calculant cette bijection et sa réciproque. Par exemple 22 s'écrit $\overline{10110}^2$ en binaire. Ainsi `int_of_intlist [1; 2; 4]` vaut `22` et `intlist_of_int 22` vaut `[1; 2; 4]`.*

♣. représenté par une liste sans doublons
♡. représenté par une liste sans doublons
♠. représenté par une liste **sans doublons**
♢. représenté par une liste sans doublons
*. représenté par une liste **sans doublons**

Exercice 2 : Énumération des mots reconnus par un automate

Dans cet exercice on se propose d'énumérer les mots reconnus par un automate (non nécessairement déterministe). Un tel ensemble pouvant bien sûr être infini, on cherche à fournir un algorithme qui, étant donné un automate $\mathcal{A}$ sur un alphabet Σ et un entier $n \in \mathbb{N}$ produit l'ensemble des mots de Σ^n reconnus par $\mathcal{A}$, ensemble qu'on notera $\mathcal{L}_n(A)$ dans la suite de cet exercice :

$$\mathcal{L}_n(\mathcal{A}) \overset{\text{déf}}{=} \mathcal{L}(\mathcal{A}) \cap \Sigma^n$$

Étant donné un automate $\mathcal{A} = (\Sigma, Q, I, F, \delta)$ et un état $q \in Q$, on note $\mathcal{A}_q$ l'automate $(\Sigma, Q, \{q\}, F, \delta)$. La seule différence entre $\mathcal{A}_q$ et $\mathcal{A}$ se situe au niveau des états initiaux : ceux de $\mathcal{A}$ sont remplacés par le seul état q dans $\mathcal{A}_q$.

Q. 1 Étant donné un automate $\mathcal{A} = (\Sigma, Q, I, F, \delta)$, exprimer $\mathcal{L}_0(\mathcal{A}_q)$ en fonction de $q \in Q$?

Q. 2 Étant donné un automate $\mathcal{A} = (\Sigma, Q, I, F, \delta)$, donner et prouver, pour tout $(n, q) \in \mathbb{N}^\star \times Q$, une relation entre $\mathcal{L}_n(\mathcal{A}_q)$ et les $\mathcal{L}_p(\mathcal{A}_{q'})$ pour $p < n$ et $q' \in Q$.

Q. 3 Donner, pour tout $n \in \mathbb{N}$, une relation entre $\mathcal{L}_n(\mathcal{A})$ et les $\mathcal{L}_n(\mathcal{A}_q)$ pour $q \in Q$.

Les questions précédentes devraient donner une idée d'algorithme récursif permettant d'énumérer les mots d'une longueur donnée reconnus par un automate donné. Toutefois, avant de se jeter dans l'implémentation, on s'interroge sur la complexité de l'algorithme envisagé.

Q. 4 Donner, pour chaque $n \in \mathbb{N}$ un automate $\mathcal{A}_n$ à n états et $\Theta(n)$ transitions♣ tel que l'arbre des appels récursifs pour le calcul de $\mathcal{L}_n(\mathcal{A}_n)$ est de taille exponentielle en n, tandis que le cardinal de $\mathcal{L}_n(\mathcal{A}_n)$ est en $\mathcal{O}(n)$.

Q. 5 Donner, en fonction du nombre d'états M de l'automate $\mathcal{A}$ et de l'entier n une majoration du nombre d'ensembles $\mathcal{L}_p(\mathcal{A}_q)$ qui entrent en jeu dans le calcul récursif de $\mathcal{L}_n(\mathcal{A})$.

Q. 6 Proposer un schéma algorithmique permettant la réconciliation des observations des deux dernières questions.

Q. 7 Proposer une implémentation en OCaml de cet algorithme. On pourra supposer que l'ensemble Q des états de l'automate est de la forme $[\![0, n-1]\!]$, avec $n \in \mathbb{N}$. On accordera une attention particulière au fait que :

- il n'est pas nécessaire d'avoir un espace mémoire en $\Theta(Mn)$, un $\Theta(n)$ suffit.
- si l'on itère sur chaque état, et que pendant chacune de ces itérations, on itère sur ses prédécesseurs, il est peut être envisageable de ne parcourir qu'une fois la liste des transitions.

♣. on entend par là que pour tout $n \in \mathbb{N}$, $\mathcal{A}_n$ a exactement t_n transitions, et que $(t_n)_{n\in\mathbb{N}} \in \Theta(n)$.

Exercice 3 : Les langages reconnaissables sont réguliers

Le but de cet exercice est de redémontrer le résultat de cours : tout langage reconnaissable est régulier. Dans un premier temps on montre le résultat mathématique, on fournit dans une seconde partie de l'exercice un algorithme, inspiré des résultats de la première partie, effectuant le calcul d'une expression régulière équivalente à un automate.

1. Résultats mathématiques

On considère donc un automate fini $\mathcal{A} = (\Sigma, Q, I, F, \delta)$ et le but est de démontrer que $\mathcal{L}(\mathcal{A})$ est régulier. On s'appuie pour cela sur la définition inductive de l'ensemble des langages réguliers, on n'utilisera pas la caractérisation des langages réguliers comme langages associés à des expressions régulières. On note $n = \text{card}(Q)$, et sans perdre en généralité on suppose que $Q = [\![1, n]\!]$.
On définit alors, pour $(a, b) \in [\![1, n]\!]^2$ et $k \in [\![0, n]\!]$, l'ensemble $C^k_{a,b}$ suivant.

$$C^k_{a,b} = \{w_1 w_2 \dots w_l \in \Sigma^\star \mid \exists (q_i)_{i \in [\![1,l-1]\!]} \in [\![1,k]\!]^{l-1} \text{ telle qu'en posant } q_0 = a \text{ et } q_l = b,\\ q_0 \xrightarrow{w_1} q_1 \xrightarrow{w_2} q_2 \dots q_{l-1} \xrightarrow{w_l} q_l \text{ est une suite de transitions de } \mathcal{A}\}$$

Q. 1 Pour $(a, b) \in [\![1, n]\!]^2$ et $k \in [\![0, n]\!]$, décrire en français ce que représente l'ensemble $C^k_{a,b}$.

Q. 2 Pour $(a, b) \in [\![1, n]\!]^2$, que vaut $C^0_{a,b}$?

Q. 3 Pour $(a, b) \in [\![1, n]\!]^2$, et $k \in \mathbb{N}^*$, démontrer que $C^k_{a,b} = C^{k-1}_{a,b} \cup C^{k-1}_{a,k} \cdot (C^{k-1}_{k,k})^\star \cdot C^{k-1}_{k,b}$

Q. 4 Conclure.

2. Implémentation

La preuve mise en avant dans la première partie de l'exercice suggère en fait un algorithme de calcul effectif d'une expression régulière équivalente à un automate. En effet, plutôt que de calculer les langages $(C^k_{a,b})_{(a,b,k) \in [\![1,n]\!]^2 \times [\![0,n]\!]}$, on définir récursivement une famille d'expressions régulières $(e_{a,b,k})_{(a,b,k) \in [\![1,n]\!]^2 \times [\![0,n]\!]}$ de sorte que $\forall (a, b, k) \in [\![1, n]\!]^2 \times [\![0, n]\!], \mathcal{L}(e_{a,b,k}) = C^k_{a,b}$.

`compagnon.ml`. On vous fournit dans le fichier `compagnon.ml` une implémentation d'un type `automaton` permettant la représentation d'automates ainsi que des fonctions permettant l'affichage de ces automates. On fournit aussi une définition inductive d'un type `reg_exp` représentant les expressions régulières, ainsi que la fonction `print_reg_exp` permettant l'affichage de ces expressions régulières. Vous trouverez une fonction `simpl : reg_exp -> reg_exp` qui tente d'effectuer des simplifications de l'expression régulière qui lui est passée en argument. Finalement dans cet exercice nous manipulerons des matrices contenant des expressions régulières, celles-ci pourront être affichées au moyen de la fonction `print_regexp_matrix reg_exp array array -> unit`.

Q. 5 Définir une fonction `find_all_letters (a: automaton) (q: int) (q': int): char list` retournant la liste des lettres se trouvant entre l'état `q` et l'état `q'` dans l'automate `a`.

Q. 6 Définir une fonction `init_matrix (a: automaton): reg_exp array array` calculant la matrice $(e_{a,b,0})_{(a,b) \in [\![1,n]\!]^2}$

Q. 7 Définir une fonction `next_k (r: reg_exp array array): unit` prenant en argument une matrice d'expression régulière que l'on supposera être les $(e_{a,b,k})_{(a,b) \in [\![1,n]\!]^2}$ pour une certaine valeur $k \in [\![0, n]\!]$ et modifiant la matrice pour y faire figurer les $(e_{a,b,k+1})_{(a,b) \in [\![1,n]\!]^2}$.

Q. 8 Conclure en donnant une fonction `regexp_of_automata (a: automaton): reg_exp` calculant une expression régulière équivalente à l'automate passé en argument à la fonction.

Exercice 4 : Gestion d'ensembles en OCaml

En OCaml, le système de module permet de regrouper plusieurs définitions, typiquement la définition d'un type et des fonctions opérant sur ce type. On a déjà utilisé ce système à travers l'utilisation du module `List` ou `String` et des fonctions comme `List.iter` ou `String.make`. La page de la documentation OCaml qui présente ça très bien (en anglais) est disponible à l'adresse suivante, cependant on reprend les points essentiels dans la fiche outil "Créer un module en OCaml ".
`https://v2.ocaml.org/manual/moduleexamples.html#s:module:structures`

Q. 1 Définir un module `SS` (String Set) permettant la représentation d'un ensemble fini de chaînes de caractères en OCaml. Ce module pourra être naïvement implémenté par des listes. Il devra fournir :

- un type `t` permettant la représentation d'un ensemble de chaînes de caractères ;
- une valeur `ens_vide` permettant la représentation de l'ensemble vide ;
- une fonction `singleton : string -> t` calculant un singleton à partir de l'unique chaîne de caractères qu'il contient ;
- une fonction `add : string -> t -> t` calculant l'ensemble de chaînes obtenu en ajoutant une chaîne à un ensemble de chaînes ;
- une fonction `union : t -> t -> t` calculant l'union de deux ensembles de chaînes ;
- **éventuellement** une fonction `fold : (string -> 'b -> 'b) -> t -> 'b -> 'b` permettant d'appliquer un schéma d'accumulation sur un ensemble de chaînes♣.

Q. 2 Pour le TP qui concerne les automates, on aura aussi besoin de gérér des ensembles d'états, mais on ne veut pas fixer a priori le type des états, c'est pourquoi on choisit de créer un type paramétré `'a set` (ou `'state set` pour de tels ensembles. Cette implémentation pourra être naïve et basée sur des listes. Le type `'a set` doit être accompagné des fonctions suivantes :

- une valeur `ens_vide : 'a set` qui représente l'ensemble est vide ;
- une fonction `est_vide : 'a set -> bool` testant si un ensemble est vide ;
- une fonction `singleton : 'a -> 'a set` calculant un singleton à partir de l'unique élément qu'il contient ;
- une fonction `appartient : 'a -> 'a set -> bool` testant si un élément appartient à un ensemble ;
- une fonction `add : 'a -> 'a set -> 'a set` calculant l'ensemble obtenu en ajoutant un élément donné à un ensemble donné ;
- une fonction `union : 'a set -> 'a set -> 'a set` calculant l'union de deux ensembles ;
- une fonction `intersecte : 'a set -> 'a set -> bool` testant si deux ensembles s'intersectent.

Exercice 5 : Énumération des mots d'une expression régulière

Dans cet exercice on souhaite faire énumérer à la machine les mots reconnus par une expression régulière. Un tel ensemble pouvant bien sûr être infini, on souhaite fournir un algorithme qui étant donné une longueur de mot $n \in \mathbb{N}$ et une expression régulière e retourne l'ensemble des mots de longueur n de $\mathcal{L}(e)$, ensemble que l'on notera $\mathcal{L}_n(e)$ dans la suite de cet exercice.

Q. 1 Définir un type `reg_exp` permettant la représentation en OCaml des expressions régulières.

Q. 2 Définir une fonction utilitaire `concat_ss : SS.t -> SS.t -> SS.t` qui calcule la concaténation des langages représentés par les deux objets de type `SS` qu'elle prend en argument. On

♣. Comme `List.fold_left` permet de réaliser l'accumulation sur les éléments d'une liste, sauf qu'ici, pour un ensemble, l'ordre des éléments n'est pas significatif.

pourra remarquer que pour deux langages L_1 et L_2 on a la relation suivante, et au besoin créer une fonction intermédiaire qui calcule la concaténation d'un mot et d'un langage.

$$L_1 \cdot L_2 = \{u \cdot v \mid u \in L_1, v \in L_2\} = \bigcup_{u \in L_1} \{u \cdot v \mid v \in L_2\} = \bigcup_{u \in L_1} \{u\} \cdot L_2$$

Q. 3 Soient e et f deux expressions régulières, $n \in \mathbb{N}$. Démontrer que $\mathcal{L}_n(e \cdot f) = \bigcup_{i=0}^{n} \mathcal{L}_i(e) \cdot \mathcal{L}_{n-i}(f)$.

Q. 4 Soit e une expression régulière.
Démontrer que $\mathcal{L}_0(e^\star) = \{\varepsilon\}$ et que pour tout $n \in \mathbb{N}^*$, $\mathcal{L}_n(e^\star) = \bigcup_{i=1}^{n} \mathcal{L}_i(e) \cdot \mathcal{L}_{n-i}(e^\star)$.

Q. 5 Déduire des remarques précédentes une définition d'une fonction `all_words_of_size (r: reg_exp) (i: int): SS.t` calculant $\mathcal{L}_i(r)$.

Créer un module en OCaml

→ Un module permet de regrouper plusieurs définitions de valeur, de type, de fonction ou d'exception se rapportant à un même objet. Par exemple `List` et `String` sont des modules.

→ Le nom d'un module commence toujours par une majuscule.

→ Un module est défini selon la syntaxe suivante.

```
module Nom_module =
  struct
   définition de type
   définition de fonctions
   définition d'exceptions
  end
```

→ En dehors du corps de la définition du module `Mod`, on accède à un élément (une valeur, un type simple, une fonction ou une exception) `obj` de ce module par `Mod.obj`. De plus on accède à un type paramétré `'a t` défini dans `Mod` par `'a Mod.t` à l'extérieur.

On donne ci-après un exemple de définition module pour les fractions (en supposant que la fonction pgcd est déjà définie), puis quelques exemples d'utilisation dans un interpréteur.

```
module Fraction =
  struct
    type t = int*int
    let un_tiers : t = (1,3)
    exception FractionMalDefinie

    let simplifie ((num,den):t) : t =
      if den = 0 then raise FractionMalDefinie
      else let d = (pgcd num den) in (num/d, den/d)
  end
```

```
# let f1:Fraction.t = 2,6;;
val f1 : Fraction.t = (2, 6)
# Fraction.simplifie f1;;
- : Fraction.t = (1, 3)
# Fraction.simplifie Fraction.un_tiers;;
- : Fraction.t = (1, 3)
# let f0 = 1,0 in Fraction.simplifie f0;;
Exception: Fraction.FractionMalDefinie.
```

Remarque : Bien qu'on l'évite pour améliorer la lisibilité et éviter des erreurs, il est possible d'utiliser la commande `open Nom_module` pour éviter d'avoir à préfixer tous les éléments du module par le nom du module. De plus, il est aussi possible de factoriser le préfixage par un nom de module, ce qui revient à réaliser une ouverture locale du module. Exemple : `fun l -> List.(length (rev l));;`