Phase 2 : j'utilise mes automates

Dans cette deuxième phase, on va utiliser les automates de Thompson créés par les fonctions de la première phase : lecture de mots dans l'automate, clôtures et déterminisation.

1 Ensemble d'états

Les fonctions de cette phase vont travailler avec des ensembles d'états ; il est donc indispensable d'avoir une manière de les représenter. On rappelle que dans les structures `automate` utilisée, le `numero` d'un état doit correspondre à son indice dans le tableau `etats`. Deux approches sont possibles :

1.1 représentation par tableau

En supposant qu'on a n états, on fait un tableau `tab` de n booléens (ou entier, c'est comme vous voulez) et on prend la convention pour représenter un ensemble donné E :

$$\begin{cases} \texttt{tab[i]=true} & \Leftrightarrow i \in E \\ \texttt{tab[i]=false} & \Leftrightarrow i \notin E \end{cases}$$

Pour "empaqueter" les variables associées, l'utilisation de la structure suivante est conseillé :

```
typedef struct EnsembleEtats {
  automate    *auto;  // l'automate contient nbEtat
  bool        *tabEtats;
} EnsembleEtats;
```

1.2 représentation par liste

Cette représentation est plus économe en mémoire. Un ensemble d'état est représenté par une liste chaînée (simplement ou doublement, c'est vous qui voyez), et chaque élément de la liste contient un état (référencé par son numéro ou par un pointeur sur lui). D'où la structure :

```
typedef struct ListeEtat {
  etat       *state; // l'état concerne
  ListeEtat  *suiv;
} ListeEtat;

typedef ListeEtat EnsembleEtats;
```

1.3 Idées de fonctions à faire pour se faciliter la tâche

Manipuler les ensembles d'états va impliquer de faire les oprérations suivantes : créer un ensemble d'états vide, ajouter et enlever un état, tester si un état est dedans. D'où des fonctions :

```
EnsembleEtats*   creerEnsemble(automate* autom);
void             ajouterEtat(EnsembleEtats* ensemb,etat* iState);
void             enleverEtat(EnsembleEtats* ensemb,etat* iState);
bool             appartientA(EnsembleEtats* ensemb,etat* iState);

bool             contient(EnsembleEtats* A,EnsembleEtats* B);
                    //teste si B est un sous-ensemble de A

bool             estEgal(EnsembleEtats* A,EnsembleEtats* B);
                    //teste si A=B

void             afficheEnsemble(EnsembleEtats* ensemb); //pour débugguer
```

2 Clôtures

Ce sont les fonctions à faire en premier, pusique toutes les suivantes les utiliseront. Leur fonctionnement est expliqué dans le cours : elles calculent l'ensemble d'états qu'on peut atteindre à partir d'un état donné (ou d'un ensemble d'états) par des ϵ-transition.

Il faut :

```
EnsembleEtats*  ajouterCloture(etat*  s,EnsembleEtats* D);
```

pour ajouter la clôture d'un état dans l'ensemble d´états `D`. On peut s'épargner de retourner l'ensemble d'états résultat en passant `D` par référence.

```
EnsembleEtats*  cloture(EnsembleEtats* D);
```

pour calculer la clôture d'un ensemble d'états (cf. cours).

```
EnsembleEtats*  delta(automate* autom,EnsembleEtats* C,char x);
```

qui calcule l'ensemble d'états atteignables par une transition étiquetée par x à partir de l'ensemble d'états `C` (utilisée dans la fonction `reconnait` du cours).

3 Fonctions sur les AFND$_\epsilon$

3.1 Lecture de mots

Pour tester l'automate :

```
bool      reconnait(automate* A,char* mot);
```

fonctionnant comme dans le cours.

3.2 AFND sans ϵ

La structure pour les automates de Thompson ne convient pas pour les automates non déterministes généraux. Voici une suggestion de structure pour ces derniers : stocker pour chaque état e et chaque lettre x l'ensemble d'états $\delta(e, x)$. Cela donne :

```
typedef struct automateNonDet {
  int              nbEtats,nbLettres;
  char*            alphabet; // optionel
  int              etatInitial;
  bool*            estFinal;
  EnsembleEtats**  delta;
} automateNonDet;
```

`nbEtats` et `nbLettres` sont explicites, et correspondent aux dimensions de la matrice `delta`. `etatInitial` est le numéro de l'unique état initial, `estFinal` est un tableau de taille `nbEtats` tel que `estFinal[i]` vaut `true` si l'état `i` est final et `false` sinon.

Pour s'aider dans les conversion lettre↔ indice , on peut faire les fonctions :

```
int      lettreVersIndice(char x,char* alphabet);
char*    indiceVersLettre(int ind,char* alphabet);
```

Notez qu'il existe une fonction `strchr` qui recherche un caractère dans une chaîne de caractères.

Les automates non déterministes sans ϵ sont un intermédiaire de calcul ; le but est de les déterminiser pour obtenir un automate déterministe, sur lequel les calculs sont faciles. Pour les contruire, il faut une fonction éliminant les ϵ-transitions, *ie* calculant $\delta_{\mathcal{B}}(s, x)$ pour tous les états s de l'AFND$_\epsilon$ de départ possibles (et toutes les lettres x possibles), par la méthode de votre choix.

```
automateNonDet*  elimineEpsilon(automate* A,char* alphabet);
```

3.3 Déterminisation

Les automates déterministes peuvent se stocker comme des automates non déterministes (après tout ce ne sont que des automates non déterministes un peu particulier) ou de manière plus efficace en stockant pour chaque état la liste des transitions qui en sortent. Par exemple :

```
typedef struct transitionDet {
  char   lettre;
  int    destination; // état dans lequel arrive la transition
} transitionDet;
```

stockera une transition, et :

```
typedef struct etatDet {
  int             nbTransitions;
  transitionDet*  transitions;
} etatDet;
```

stockera les transitions sortant d'un état. Cela conduirait à utiliser pour l'automate déterministe :

```
typedef struct automateDet {
  int             nbEtats;
  etatDet*        etats;
  int             etatInitial;// comme dans automateNonDet
  bool*           estFinal;   // comme dans automateNonDet
} automateDet;
```

On peut envisager un certain nombre de fonctions annexes pour manipuler ces structures :

```
automateDet* creeAutomateDet(void); //cree un automate vide(sans états)
int          ajouteEtat(automateDet* A); //ajoute un état et renvoie son numéro
void         ajouteTransition(automateDet* A,int de,char x,int vers);
```

Et pour créer ces automates déterministes :

```
automateDet*  determinise(automateNonDet* A,char* alphabet);
```

Bien remarquer que la déterminisation impose de conserver les ensembles d'états associés à chacun des états de l'automate déterministe en cours de construction.