

CHAPITRE 4 : EQUIVALENCE ENTRE AUTOMATES ET EXPRESSIONS REGULIERES, LEX

1. Rappel de la définition des expressions régulières

Définition. Soit Σ un alphabet. Les expressions rationnelles (ou régulières) sur Σ et les langages correspondants sont définis récursivement :

- 1) • \emptyset est une expression rationnelle, et représente l'ensemble vide,
• ϵ est une expression rationnelle, et représente l'ensemble $\{\epsilon\}$,
• pour toute lettre a de Σ , a est une expression rationnelle, et représente l'ensemble $\{a\}$.
- 2) Si r et s sont des expressions rationnelles, qui représentent les langages R et S , alors $r + s$, rs , et r^* sont des expressions rationnelles qui représentent les langages $R \cup S$, RS et R^* .

2. Équivalence entre automates et expressions régulières (Kleene)

2.1 Théorème de Kleene

Théorème (Kleene). Tout langage reconnu par un AFD peut être représenté par une expression régulière, et réciproquement.

Preuve

- Tout langage représenté par une expression régulière est reconnu par un AFD.

Les langages \emptyset , $\{\epsilon\}$ et $\{a\}$ pour toute lettre a de Σ , sont reconnus par AFD.

De plus, les propriétés de clôture des langages reconnus par AFD montrent que l'union, la concaténation et l'étoile sont aussi reconnus par AFD. Donc les langages représentés par expression régulière sont reconnus par AFD (méthode dite de Thompson).

On verra plus loin une deuxième méthode effective pour construire un AFD reconnaissant le langage décrit par une expression régulière (algo de Glushkov).

- Tout langage reconnu par AFD peut être représenté par une expression régulière.

On peut le montrer directement, ou utiliser une méthode de résolution d'équation (voir plus loin).

1.2 AFN associé à une expression régulière (algorithme de Glushkov)

Exemple : soit une expression rationnelle

$$(a + ab)^*(\epsilon + ab)$$

Première étape : Linéariser l'expression rationnelle, en remplaçant toutes les lettres par des symboles distincts (de la gauche vers la droite).

$$(x_1 + x_2x_3)^*(\varepsilon + x_4x_5)$$

Deuxième étape : Déterminer

- Premier = ensemble des symboles pouvant commencer un mot = $\{x_1, x_2, x_3\}$,
- Dernier = ensemble des symboles pouvant finir un mot = $\{x_4, x_5\}$,
- Pour tout symbole x_i , $\text{Suivant}(x_i)$ = ensemble des symboles pouvant suivre x_i .

Attention : cette fonction Suivant doit tenir compte des étoiles.

	Suivant
x_1	x_1, x_2, x_4
x_2	x_3
x_3	x_1, x_2, x_4
x_4	x_5
x_5	

Proposition. *Un mot $w = w_1w_2 \dots w_n$ appartient au langage décrit par l'expression linéarisée ssi $w_1 \in \text{Premier}$, $w_n \in \text{Dernier}$, et $w_{i+1} \in \text{Suivant}(w_i)$ pour $i = 1, \dots, n-1$.*

Construction de l'AFN pour l'expression linéarisée :

- Q = ensemble des symboles + un état initial i
- F = Dernier, avec en plus l'état initial i si ε appartient au langage
- fonction de transition δ :

$$\delta(i, x_i) = x_i \text{ si } x_i \in \text{Premier},$$

$$\delta(x_i, x_j) = x_j, \text{ pour tout } x_j \in \text{Suivant}(x_i).$$

Puis on remplace les symboles x_i par les lettres d'origine de l'expression rationnelle.

1.3 AFN associé à une expression régulière (algorithme de Thompson)

On construit par induction un AFN avec ε -transition reconnaissant le langage décrit par une expression rationnelle, à partir des opérations de combinaison définissant les expressions rationnelles (union, concaténation, étoile).

- Pour \emptyset , ε , et les lettres, l'automate est évident.
- Pour l'union, la concaténation et l'étoile, on combine les AFN comme cela est indiqué dans la preuve du théorème de Kleene.

Exemple :

$(a + ab)^*(\epsilon + ab)$

La taille d'une expression rationnelle r , notée $|r|$, est son nombre de caractères (sans compter les parenthèses).

Exemple : $|(a + ab)^*(\epsilon + ab)| = 9$.

Complexité. L'algorithme de Thompson construit à partir d'une expression régulière r un AFN avec ϵ -transitions où :

- il y a un état initial et un état final,
- le nombre d'états est borné par $2|r|$,
- de chaque état sort au plus deux transitions

Le temps pour le construire est $O(|r|)$, de même que l'espace pour le représenter.

La construction de l'AFN par l'algo de Glushkov s'effectue en $O(|r|^2)$, avec prétraitement de l'expression régulière.

1.4 La commande lex d'Unix

La commande lex d'Unix fabrique un automate à partir d'une expression rationnelle.

Le format du fichier source traité par la commande lex comporte trois parties séparées par `%%`. La première partie est optionnelle, ainsi que la troisième et les `%%` qui la précèdent.

<définitions>

`%%`

<règles>

`%%`

<instructions C>

Les règles sont du type :

<expression régulière> <instructions C>

Lorsqu'une expression régulière est reconnue, l'instruction C correspondante est exécutée.

Les expressions régulières sous Unix (pour egrep ou lex) s'écrivent avec un certain nombre d'abréviations :

`xly` pour un choix entre `x` ou `y`

`x*` pour une répétition indéfinie de `x`

`[abc]` pour `albc`

`[a-z]` pour `abl...lz`

[a-zA-Z0-9] pour toutes les lettres et chiffres

x+ pour au moins une occurrence de x, c'est-à-dire xx*

x? pour un x optionnel

Exemple : le fichier ex.l contient

```
%%  
abcde      printf("[%s]", yytext);  
%%  
int yywrap() { return 1;}  
main()     { yylex(); }
```

Avec ces instructions C minimales, l'automate lit chaque ligne, en la recopiant, et s'il trouve le mot 'abcde', il l'encadre par des crochets (la variable %s renvoie à yytext, qui contient le motif reconnu).

La commande flex permet de transformer ce fichier en un fichier ex.c de code C :

```
$ flex -o ex.c ex.l
```

Puis on compile le code C obtenu en un exécutable ex :

```
$ cc ex.c -o essai
```

Dans cette version minimale, la fonction ex ne s'arrête pas (faire Ctrl-D pour sortir).

```
$ essai  
aaaaaaaaabcdeeeeeeee  
aaaaaaaa[abcde]eeeeeee
```

Voilà un autre exemple avec des définitions dans la première partie. Les identifiants utilisés dans ces définitions peuvent être repris dans les expressions régulières de la deuxième partie, entre accolades :

```
voyelle      [aeiou]  
consonne     [bcdfghjklmnpqrstvwxyz]  
%%  
( {consonne} {voyelle} ) * {consonne}  printf("[%s]", yytext);  
%%  
int yywrap() { return 1; }  
main()       { yylex(); }
```

On obtient :

\$ essai
consonne
[con][son][ne]
bonjour
[bon][j]ou[r]