

CHAPITRE 5 : ANALYSE LEXICALE

L'analyse lexicale est un autre domaine fondamental d'application des automates finis.

Dans la plupart des langages de programmation, les unités lexicales (identificateurs, mots-clefs du langage, nombres, etc.) sont définies par des expressions régulières (ou rationnelles). L'analyse lexicale consiste à détecter ces unités dans le flot de caractères constitué par le programme. Pour ce faire, on associe des automates finis aux expressions régulières.

Il existe des méthodes efficaces permettant de construire un AFD pour toute expression régulière.

1. Équivalence entre automates et expressions régulières

1.1 Rappel de la définition des expressions régulières (ou rationnelles)

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^* .

1.2 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).

1.3 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).

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

Deuxième étape : Déterminer

- Premier = ensemble des symboles pouvant commencer un mot = $\{x_1, x_2, x_4\}$,
- Dernier = ensemble des symboles pouvant finir un mot = $\{x_1, x_3, x_5\}$,
- 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.

1.4 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 :

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

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

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

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.

1.5 Application à la recherche d'une expression régulière r dans un texte

On construit un AFN qui reconnaît Σ^*r grâce à l'algorithme de Thompson.

Puis deux possibilités :

- soit on le détermine en AFD,
- soit on fait la recherche en simulant l'AFD (sans construire tous les états, mais seulement ceux dont on a besoin au fur et à mesure pour la lecture du texte t).

Voilà un exemple illustrant la *simulation dynamique* de l'AFD, c'est-à-dire la construction de certains états de l'AFD au fur et à mesure où on en a besoin, sans passer par la construction préalable de tous les états.

- Construction de l'AFN reconnaissant l'expression régulière Σ^*r par l'algorithme de Thompson

- Analyse du texte en simulant l'AFD correspondant

Pour P un ensemble d'états de l'AFN, on note

Cloture(P) = les états accessibles à partir de $q \in P$ par ϵ -transition

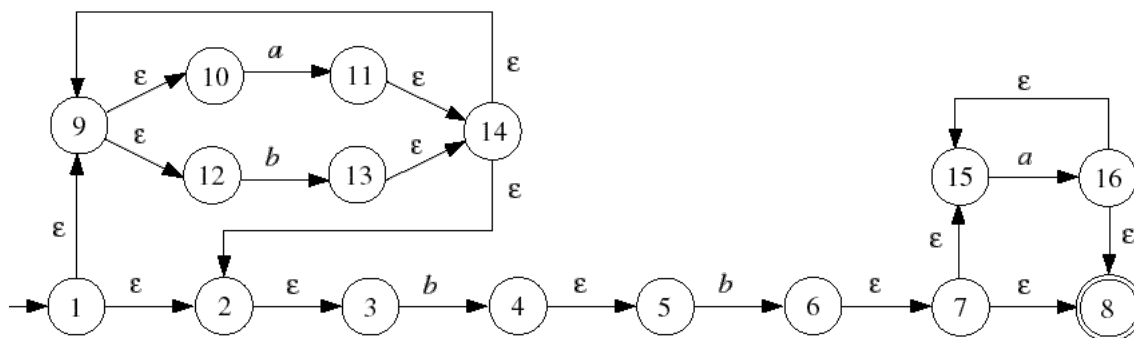
Transition(P, a) = clôturé de l'union des $\delta(q, a)$ pour $q \in P$

L'analyse de t dans l'AFN se fait en partant de Cloture(I) où I est l'ensemble des états initiaux, et en suivant les flèches indiquées par Transition(P, a).

Exemple : recherche du motif $r = bba^*$ dans le texte $t = babbaab$

1) algorithme de Thompson pour l'AFN de $(a + b)^*bba^*$

- pour la concaténation : on sépare les éléments concaténés par des ϵ -transitions
- pour l'union : on regroupe les éléments de la disjonction par des ϵ -transitions qui partent d'un même état de départ et se rejoignent sur un même état d'arrivée
- pour l'étoile : on boucle de l'état d'arrivée vers celui de départ par une ϵ -transition



2) analyse de $t = babbaab$

initialisation : $P = \{1, 2, 3, 9, 10, 12\}$

transition par b : $\{4, 13\}$

clôturé : $\{4, 5, 13, 14, 9, 10, 12, 2, 3\}$

transition par a : $\{11\}$

clôturé : $\{11, 14, 9, 10, 12, 2, 3\}$

transition par b : $\{4, 13\}$

clôturé : $\{4, 5, 13, 14, 9, 10, 12, 2, 3\}$

transition par b : $\{4, 6, 13\}$

clôture : {4, 5, 13, 6, 7, 8, 15, 14, 9, 10, 12, 2, 3}

8 est final, donc une occurrence a été détectée

transition par a : {11, 16}

clôture : {11, 14, 9, 10, 12, 2, 3, 16, 8, 15}

une occurrence a été détectée

transition par a : {11, 16}

clôture : {11, 14, 9, 10, 12, 2, 3, 16, 8, 15}

une occurrence a été détectée

transition par b : {4, 13}

clôture : {4, 5, 13, 14, 9, 10, 12, 2, 3}

Remarque : dans ce cas, la construction de l'AFD par détermination n'aurait pas été trop coûteuse, car l'AFN pour bb^* a 3 états, et on est dans le cas où l'AFD pour Σ^*bb^* a le même nombre d'états.

2. Utilisation des expressions régulières dans la construction des compilateurs

1.1 Principe de la compilation

Un compilateur est un programme qui *traduit* un autre programme dit *source* en un programme dit *cible*.

Le programme source est écrit dans un langage de haut niveau (Java, Lisp, Prolog, etc.).

Le programme cible est écrit dans un langage de bas niveau (assembleur) exécutable par une machine.



1.2 Les phases de la compilation

Exemple : supposons que le programme source contienne le calcul décrit par l'expression

$\text{var1} + 100 * \text{var2}$

Première phase : analyse lexicale

Elle consiste à analyser l'expression (ou en général le programme source) en constituants minimaux appelés *tokens* (ou jetons). Elle produit une chaîne de tokens composés de couples :

(unité lexicale, valeur ou adresse dans la table de symboles)

Dans l'exemple ci-dessus, les unités lexicales sont :

- **id** : identificateurs var1 ou var2 (les identificateurs rencontrés sont placés dans une table de symboles)
- **op** : opérations arithmétiques +, *
- **nb** : constante entière 100

Tokens reconnus : (**id**, var1) (**op**, +) (**nb**, 100) (**op**, *) (**id**, var2)

Cette phase de tokenisation comporte également :

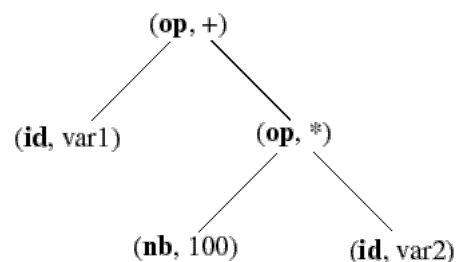
- l'élimination des blancs (et des tabulations, sauts de ligne,...)
- l'élimination des commentaires du programme source.

Les unités lexicales correspondent à des *suites de caractères* du programme source (« var1 », « + », « 100 », « var2 », « + »). On les définit par des expressions régulières.

La tokenisation peut donc être réalisée par un automate fini. On donnera ci-après une présentation détaillée de cette utilisation des automates finis

Deuxième phase : analyse syntaxique

La chaîne de tokens obtenue est ensuite analysée pour construire sa structure syntaxique, qui a la forme d'un arbre :



Cette phase de la compilation est réalisée par une grammaire (voir deuxième partie du cours avec Patrice Enjalbert).

On peut résumer la compilation comme une succession de phases dont les deux premières sont l'analyse lexicale et l'analyse syntaxique :



Cette décomposition en phases permet de rendre plus simple et plus efficace la conception des compilateurs. Le fait que l'analyseur lexical soit la seule phase de la compilation qui traite directement le texte source est important. À partir de la phase d'analyse syntaxique, le compilateur ne traite plus que des unités lexicales indépendantes des particularités du langage source.

1.3 L'analyse lexicale

Les unités lexicales du programme source sont définies par des modèles : *identificateurs*, *nombres*, *opérations arithmétiques*, etc. Les parties du programme source éliminées au cours de cette phase correspondent également à des modèles : *espaces*, *commentaires*.

Ces modèles ont la forme d'expressions régulières (ou rationnelles).

On a vu qu'on peut les représenter de manière équivalente par des automates finis.

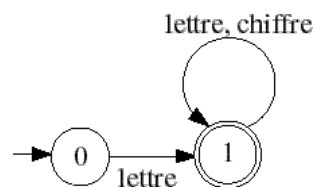
Par exemple, dans un langage comme Pascal, les identificateurs sont définis par le modèle suivant :

lettre → [A-Za-z]

chiffre → [0-9]

id → **lettre** (**lettre** | **chiffre**)*

L'expression rationnelle définissant l'unité lexicale **id** se traduit par un automate :



Dans le fonctionnement d'un tel automate, on associe une action à chaque état final : si on lit par exemple « var1 », on suit le parcours 0111 dans l'automate, qui s'arrête dans l'état final 1. On a alors détecté un identificateur.

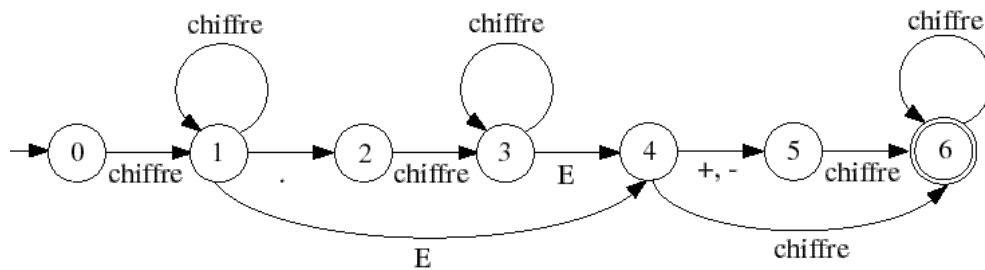
Action : l'analyseur forme l'unité lexicale (**id**, var1) et ajoute le symbole var1 dans la table des symboles.

Les nombres **nb** sont définis de la manière suivante :

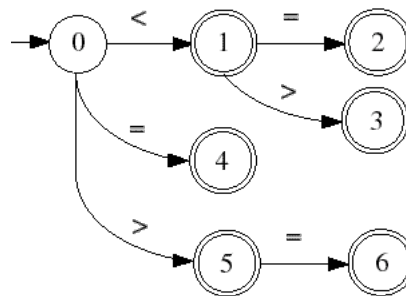
dec → . **chiffre**+

exp → E (+ | -) **chiffre**+

nb → **chiffre**+ (**dec** | ε) (**exp** | ε)



Certains opérateurs qui ont des caractères communs posent problème :



Si le flot de caractères comporte « <= », l'analyseur lit d'abord <. Il se trouve alors dans l'état final 1, mais il ne sait pas s'il doit

- s'arrêter et détecter l'unité lexicale (**op**, <)
- ou continuer et détecter l'unité lexicale (**op**, <=) dans l'état final 2.

(le problème est le même avec les boucles sur l'état final dans les automates précédents).

L'utilisation de ces automates doit donc être améliorée, en incluant un mécanisme de lecture d'un caractère « à l'avance » pour permettre de choisir dans de telles situations.

Pour construire un analyseur lexical :

- on définit les modèles des unités lexicales sous forme d'expressions régulières,
- on construit les automates associés à ces expressions régulières (voir algo de Thompson),
- on réunit tous ces automates en ajoutant un état initial et des ϵ -transitions vers les états initiaux de tous les automates des modèles,
- on détermine l'AFN obtenu pour fabriquer un AFD.

Attention : dans le processus de détermination, il est possible qu'apparaisse un ensemble d'états avec plusieurs états finals de l'AFN. Le problème est que des actions sont associées aux états finals et qu'il faut choisir quelle action exécuter. Pour cela, on instaure un ordre dans les modèles d'unités lexicales, et on choisit l'état final qui correspond au modèle placé en premier.

1.4 La commande `lex` d'Unix

La commande `lex` d'Unix permet

- de définir des modèles d'unités lexicales,

- de fabriquer un automate à partir de leurs expressions rationnelles.

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 (c'est-à-dire un modèle d'unité lexicale), l'instruction C correspondante est exécutée.

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

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

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

`[abc]` pour `abc`

`[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

`[^x]` tout caractère sauf ceux donnés

Exemple : le fichier `essai.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 `essai.c` de code C :

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

Puis on compile le code C obtenu en un exécutable `essai` :

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

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

```
$ essai
aaaaaaaaabcdeeeeeeeee
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     [b-df-hj-np-tv-z]
%%
({consonne}{voyelle})*{consonne}  printf("[%s]", yytext);
%%
int yywrap() { return 1; }
main()      { yylex(); }
```

On obtient :

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

Encore un autre exemple qui montre un mini-analyseur lexical pour un langage de type Pascal, en se limitant aux espacements (blancs, tabulations), identificateurs, nombres, opérations :

```
delim      [ \t]
bl         {delim}+
lettre     [A-Za-z]
chiffre    [0-9]
id         {lettre}+({lettre}|{chiffre})*
nb         {chiffre}+(\.{chiffre}+)?(E[+\-]?{chiffre}+)?
%%
{bl}      { /* pas d action */ }
{id}      printf("[id,%s]", yytext);
{nb}      printf("[nb,%s]", yytext);
[+*]      printf("[op,%s]", yytext);
%%
int yywrap() { return 1; }
main()     { yylex(); }
```

Après compilation du programme `lex`, on peut analyser des expressions et détecter les unités lexicales :

```
$ pasc
var1 + var2
[id,var1][op,+][id,var2]
x1      +    1000  * valeur
[id,x1][op,+][nb,1000][op,*][id,valeur]
```