# Cours A&G – Automates non-déterministes à une pile (AUPnD)

Langages algébriques ≡ Grammaire de type 2 ≡ AUPnD

**Principe** Les AUPnD fonctionnent sur le même principe que les AEF : depuis un état $q$, ils consomment un caractère du mot et effectue la transition correspondante qui les amène dans un nouvel état $q'$. À la différence des AEF, à chaque transition ils mettent à jour une pile et peuvent ainsi enregistrer des informations utiles pour la reconnaissance.

**Condition d'acceptation sur pile vide** Un AUP accepte un mot $w$ s'*il existe* une exécution de l'automate

1. commençant dans un état initial *avec une pile vide*, notée _
2. qui réussit à consommer toutes les lettres de $w$
3. et se terminent dans un état accepteur
4. avec *la pile de départ* (*ie.* une pile vide)

**Variante : condition d'acceptation sur marque spéciale en sommet de pile** En pratique on préfère parfois commencer l'exécution avec une pile quelconque $\gamma$ sur laquelle on empile un symbole spécial, souvent noté $*$. On commence donc avec une pile $\frac{*}{\gamma}$ au lieu d'une pile vide et on accepte si à la fin de l'exécution si on est revenu à la pile de départ, c'est-à-dire $\frac{*}{\gamma}$.

**Les transitions d'un AUPnD** sont de la forme $(q_1) \xrightarrow[\gamma/\gamma']{l} (q_2)$ où $q_1, q_2$ sont des états de l'automate ; $l$ est un symbole de l'alphabet $\Sigma$ du langage à reconnaître ; la partie $\gamma/\gamma'$ indique les modifications de la pile : $\gamma$ désigne la pile avant la transition et $\gamma'$ la pile après la transition.

**Exemples :**

1. $\gamma/\frac{p}{\gamma}$ indique que la transition ajoute un élément $p$ sur la pile
2. $\frac{p}{\gamma}/\gamma$ indique que la transition ne peut se faire que si l'élément $p$ est présent en sommet de pile et cet élément est dépilé.
3. pour transformer un ADEF en AUP il suffit d'ajouter $\gamma/\gamma$ sur chaque transition de l'ADEF.

**Application** Donnez un AUPD qui effectue la vérification de pin sur un téléphone portable. Votre solution doit répondre aux exigences suivantes :
– la longueur du code pin n'est pas connue
– l'utilisateur peut taper autant de chiffres qu'il veut
– la touche [c] permet d'effacer un caractère
– la touche [a] permet d'effacer tous les caractères entrés
– la touche [v] lance la vérification du code entré par l'utilisateur
– on bout de trois échec le téléphone est bloqué

**Indication :** On suppose que le pin est enregistré dans le téléphone dans la variable $pin$. Les transitions qui acceptent le code entrée par l'utilisateur si $code = pin$ s'écrivent

$$q \xrightarrow{\frac{\text{pin}}{\gamma}/\gamma} q' \xrightarrow{*/} q_f$$

où $\frac{pin}{\gamma}$ indique que le haut de la pile correspond à la séquence des digits de $pin$.

**Langages algébriques** Les langages reconnus par les AUPnD sont dits algébriques. Les langages reconnus par les AUPD sont un sous-ensemble stricte du précédent appelés langages algébriques déterministes. Les langages algébriques (déterministes ou non) contiennent les langages réguliers et l'inclusion est stricte puiqu'on peut construire un AUPD qui reconnaît le langage $\{a^n b^n \mid n \in \mathbb{N}\}$.
Mais les AUPnD ne capturent pas tous les langages : il existe des langages qui ne sont pas algébriques (voir l'exemple ci-après).

**Critère de régularité des langages algébriques**

Soit $L$ un langage algébrique alors il existe un rang $n \in \mathbb{N}$ tel que pour tout mot $\omega \in L$ de taille $\geq n$, on peut trouver une décomposition de $\omega$ en cinq parties $d.w_1.m.w_2.f$ avec $w_1, w_2 \neq \epsilon$ et telle que $\forall k \in \mathbb{N},\ d.w_1^k.m.w_2^k.f \in L$

**Application** Le critère précédent permet de montrer que le langage $\{a^n b^n c^n \mid n \in \mathbb{N}\}$ n'est pas algébrique.

***preuve :*** *pour un $n$ donné on construit le mot $w = a^n b^n c^n$ alors quel que soit la décomposition $d.w_1.m.w_2.f$ choisie pour $w$ le mot $d.w_1^k.m.w_2^k.f$ n'appartient pas à $L$ pour tout $k > 1$ . En effet,*

- *si $w_1$ (idem pour $w_2$) contient deux symboles différents ($a$ et $b$ par exemple) alors $w_1^k$ fera apparaître des alternances de $a$ et de $b$.*
- *si $w_1$ et $w_2$ ne contiennent chacun qu'un seul type de symbole (par exemple des $a$ pour $w_1$ et des $b$ pour $w_2$) alors en faisant croître $k$ on modifiera le nombre de $a$ et celui de $b$ dans le mot mais pas le nombre de $c$.*

□

## 0.1 Opérations sur les langages algébriques

- L'union, la concaténation, l'itération non bornée – dite fermeture de Kleene ($L^*$) – d'un langage algébrique est un langage algébrique.
- L'intersection d'un langage algébrique et d'un langage régulier est un langage algébrique.
- on sait construire le complémentaire d'un AUP *déterministe.*

**Résultats négatifs**

- L'intersection de deux langages algébriques n'est pas forcément un langage algébrique.

  ***preuve sous la forme d'un contre-exemple :*** *Le langage $L_1 \stackrel{def}{=} \{a^n b^n c^m \mid n, m \in \mathbb{N}\}$ est algébrique, il est reconnu par une grammaire de type 2. Laquelle ?*

  *Le langage $L_2 \stackrel{def}{=} \{a^m b^n c^n \mid n, m \in \mathbb{N}\}$ est algébrique, il est reconnu par une grammaire de type 2. Laquelle ?*

  *Cependant, le langage $L_1 \cap L_2 = \{a^n b^n c^n \mid n \in \mathbb{N}\}$ n'est pas algébrique, on vient de le montrer avec le lemme de gonflement.*

  □

- Le complémentaire d'un langage algébrique (non déterministe) n'est pas nécessairement un langage algébrique.

  ***preuve par contradiction :*** *On peut définir l'intersection à partir des opérations union et complémentaire : $L_1 \cap L_2 = \overline{\overline{L_1} \cup \overline{L_2}}$. Puisque l'ensemble des langages algébriques est fermé pour l'union, si de plus il était fermé par l'opération de complémentaire alors il le serait aussi pour l'intersection. Ce qui est contredit par le contre-exemple précédent.* □

- On ne peut pas déterminiser les AUPnD : Il n'est pas possible d'associer à chaque AUPnD, un automate déterministe équivalent.

  ***preuve par contradiction :*** *On s'appuie sur les résultats suivants :*

  *(i) Étant donné un langage $L$ reconnu par un AUPD $A$, on sait construire l'AUPD $A^C$ qui reconnaît le langage complémentaire $\overline{L}$. Autrement dit les AUPD sont fermés par l'opération complémentaire.*

  *(ii) Le complémentaire d'un langage algébrique n'est pas forcément un langage algébrique.*

  *Il existe donc des langages algébriques reconnu par un AUPnD pour lesquels il n'existe pas d'AUPD équivalent.*

  *En effet, pour chaque langage algébrique $L$ il existe un AUPnD $A$ qui le reconnaît. Si pour chaque AUPnD $A$ il existait un AUPD $A^D$ équivalent alors on pourrait prendre son complémentaire $A^{DC}$, ce serait un AUPD qui reconnaît le langage $\overline{L}$. Donc $\overline{L}$ serait forcément un langage algébrique, or on a vu que ce n'était pas le cas.*

  □

**La puissance du non-déterminisme** Le langage $L_1 \stackrel{def}{=} \{w.\bullet.R(w) \mid w \in \Sigma^*\}$ des palindromes dont le milieu est marqué par un symbole $\bullet \notin \Sigma$ est irrégulier (cf. TD) mais algébrique puisqu'il est reconnu par une grammaire de type 2 avec pour règles :

$$\mathcal{R} = \left\{ \begin{array}{c} S \to \bullet \\ S \to lSl \text{ pour chaque lettre } l \in \Sigma \end{array} \right\}$$

Le langage des palindromes (de longueur paire) $L_2 \stackrel{def}{=} \{w.R(w) \mid w \in \Sigma^*\}$ est irrégulier (cf. TD) mais algébrique puisqu'il est reconnu par une grammaire de type 2 avec pour règles :

$$\mathcal{R} = \left\{ \begin{array}{c} S \to \epsilon \\ S \to lSl \text{ pour chaque lettre } l \in \Sigma \end{array} \right\}$$

**Application (à faire en TD)**

1. Donnez un AUPD qui reconnaît le langage $L_1$ : on empile les lettres du début du mot jusqu'au symbole $\bullet$ puis on compare la suite du mot avec le contenu de la pile.
2. Exécutez cet automate sur le mot $a \bullet bb$
3. Donnez un AUP qui reconnaît le langage $L_2$ : c'est un AUPnD car on ne sait pas où se trouve le milieu du mot. Il faut donc lancer en parallèle autant de reconnaissance qu'il y a de milieu possible dans le mot. Le non-déterminisme permet de définir un automate simple qui reconnaît les mots de $L_2$. Il est impossible de réaliser cet algorithme de reconnaissance avec un AUPD.

**Exercice (à faire en TD)**

1. Donnez la grammaire $G$ sur $\Sigma = \{a, b\}$ des mots qui contiennent autant de $a$ que de $b$
2. Donnez un AUP *déterministe* qui reconnaît le langage $\mathscr{L}(G)$ générée par la grammaire
3. Exécutez cet automate sur le mot *abba*

   2 états et transitions =

$$\left\{ \begin{array}{c} \xrightarrow[\underline{*}]{\epsilon} 1, \\ 1 \xrightarrow[\underline{*}/\frac{a}{*}]{a} 1, 1 \xrightarrow[\frac{a}{\gamma}/\frac{a}{a\gamma}]{a} 1, 1 \xrightarrow[\frac{b}{\gamma}/\gamma]{a} 1, \\ 1 \xrightarrow[\underline{*}/\frac{b}{*}]{b} 1, 1 \xrightarrow[\frac{b}{\gamma}/\frac{b}{b\gamma}]{b} 1, 1 \xrightarrow[\frac{a}{\gamma}/\gamma]{b} 1, \\ 1 \xrightarrow[\underline{*}/\_]{\epsilon\ \gamma} 2^a \end{array} \right\}$$

**L'impossibilité de déterminiser les AUPnD est très regrettable** L'intérêt d'un algorithme non-déterministe est qu'il est en général facile à écrire. En contre-partie pour l'exécuter efficacement il faudrait disposer d'un nombre arbitrairement grand de machines en parallèle : chacune explorant l'une des exécutions possibles et l'ensemble des machines se concertant à la fin des exécutions pour décider du résultat à rendre.
On perd la magie des automates AEF pour lesquels on pouvait donner une version non déterministe qu'on pouvait ensuite déterminiser et exécuter de manière efficace à l'aide d'une unique machine.
Ce rêve du programmeur « exécuter efficacement des algorithmes non-déterministes » est réduit à néant dès qu'on cherche à résoudre des problèmes qui dépassent la puissance des AEF. Dommage... il va falloir suivre des cours de programmation concurrente et de conception d'applications réparties car l'exécution concurrente introduit du non-déterminisme qu'il va falloir apprendre à gérer.

**Construction d'une grammaire hors contexte équivalente à un automate à une pile** Étant donné $A$ un automate à pile, on doit construire une grammaire qui génère les mots que reconnaît $A$.

> **Principe :** Le germe de la grammaire doit générer les mots qui seront reconnus par une exécution de l'automate $(\langle q_i, \underline{*}\rangle, w) \to_A^* (\langle q_f, \_\rangle, \epsilon)$ où $q_i$ est l'état initial de $A$, $\underline{*}$ est la marque de fond de pile, $q_f$ est l'état accepteur de $A$ et _ représente la pile vide.
>
> Pour traduire cette idée, les non-terminaux de la grammaire sont notés sous la forme de triplets $\langle q, p, q'\rangle$ où $q, q'$ sont des états de $A$ et $p$ est le symbole de sommet de pile actuel dans l'état $q$ ; il devra être consommé lors d'une exécution de l'automate de l'état $q$ vers l'état $q'$.

Ainsi, le germe de la grammaire sera le non-terminal $\langle q_i, \underline{*}, q_f\rangle$ qui indique qu'on s'intéresse aux exécutions de l'état initial $q_i$ jusqu'à l'état final $q_f$, qui finissent par consommer le symbole de fond de pile $*$ pour arriver à la pile vide et ainsi accepter le mot.

**L'idée :** une exécution de l'automate $A$ sur le mot $\omega$ de l'état $(q_i, \underline{*})$ à l'état $(q_f, \_)$. peut se décomposer en deux étapes : une transition de l'état $(q_i, \underline{*})$ à un état $(q, \gamma)$ sur la première lettre du mot $\omega$ suivie d'une exécution de $(q, \gamma)$ à $(q_f, \_)$ qui lit la fin de $\omega$. La règle

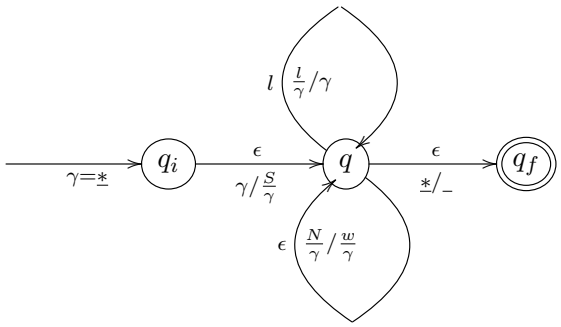
$$\langle q_i, \underline{*}, q_f\rangle \to \langle q_i, \underline{*}, q\rangle.\langle q, \gamma, q_f\rangle$$

traduit cette décomposition. Elle ne convient pas tout à fait : L'algorithme de traduction des transitions de l'automate en règle de grammaire est un peu plus subtile ; mais retenez que le principe est que les règles de la grammaire correspondent à des décompositions possibles des exécutions de l'automate.

Chaque transition génère (en général) plusieurs règles car il faut énumérer les états intermédiaires ($q_?, q'_? \in \mathcal{Q}$) d'une exécution et tous les symboles possibles en sommet de pile ($p_? \in \Gamma$).

| | *germe de* $G : S = \langle \mathbf{q}_i, \underline{*}, \mathbf{q}_f\rangle$ |
|---|---|
| *transitions* | *règles de* $G$ : |
| $\mathbf{q}_1 \xrightarrow{l\ \ \gamma/\gamma} \mathbf{q}_2$ | $\langle \mathbf{q}_1, \gamma, q_?\rangle \to l.\langle \mathbf{q}_2, \gamma, q_?\rangle$ *pour tout* $q_? \in \mathcal{Q}$ |
| $\mathbf{q}_1 \xrightarrow{l\ \ \underline{\mathbf{p}}/\gamma} \mathbf{q}_2$ | $\langle \mathbf{q}_1, \mathbf{p}, \mathbf{q}_2\rangle \to l$ |
| $\mathbf{q}_1 \xrightarrow{l\ \ \gamma/\underline{\mathbf{p}}} \mathbf{q}_2$ | $\langle \mathbf{q}_1, p_?, q_?\rangle \to l.\langle \mathbf{q}_2, \mathbf{p}, q'_?\rangle.\langle q'_?, p_?, q_?\rangle$ *pour tout* $q_?, q'_? \in \mathcal{Q}$ *et* $p_? \in \Gamma$ |

**Construction d'un AUPnD équivalent à une grammaire hors contexte** Considérons une grammaire $G = (\Sigma, \mathcal{N}, S, \mathcal{R})$ de type 2, c'est-à-dire avec des règles de la forme $N \to w$. L'automate qui reconnaît le langage généré par $G$ comporte trois états $q_i, q, q_f$ et à la forme générale suivante :




Cette automate simule les dérivations de la grammaire : le contenu de la pile de l'automate correspond aux étapes successives de dérivations de la grammaire depuis le germe $S$ jusqu'à un mot $\omega$ qui ne comporte plus de non-terminaux.

Pour décider si un mot $\omega$ appartient au langage défini par la grammaire $G$,

- l'automtate commence l'exécution dans l'état initial $q_i$ avec une pile $\gamma$ contenant uniquement le symbole de fond de pile $\underline{*}$
- La transition de $q_i$ à $q$ ne consomme pas de lettre du mot $\omega$ (la transition se fait sur $\epsilon$) et ajoute le germe $S$ dans la pile (à ce moment de l'exécution $\gamma = \frac{S}{\underline{*}}$)
- Lorsque le sommet de pile contient une lettre de $\Sigma$, la transition $q \xrightarrow{l\ \ \underline{\gamma}/\gamma} q$ consomme la lettre $l$ du mot et l'enlève de la pile
- Lorsque le sommet de pile contient un non-terminal $N$, la transition $q \xrightarrow{\epsilon\ \ \underline{N}/\underline{w}} q$ simule l'application de la règle $N \to w$ en remplaçant dans la pile $N$ par sa définition $w$. Cette transition ne consomme pas de lettre puisqu'elle a lieu sur le symbole $\epsilon$.

**Application (à faire en TD)**

a) Donnez la grammaire de type 2 qui reconnaît les palindromes sur l'alphabet $\{a, b\}$ dont le milieu est marqué par le symbole $\bullet$.

b) Construisez l'AUPnD correspondant.

c) Donnez son exécution sur le mot $ab \bullet ba$.

## 1 Implantation en C d'un moteur pour automate à pile déterministe

Les transitions d'un AUPD peuvent être représentées sous la forme d'un tableau :

| **état**$\rightarrow$ | $q_1$ | $q_2$ | $q_3$ | ... |
|---|---|---|---|---|
| **symbole**$\downarrow$ `'a'` | $(s, q_k, \text{EMPILER}, s')$ | $(s, q_k, \text{DEPILER}, s)$ | $(\_, 0, \_, \_)$ | ... |
| `'b'` | $(\_, q_k, \text{EMPILER}, s')$ | $(\_, q_k, \text{INTACT}, \_)$ | ... | ... |
| `'c'` | $\vdots$ | $\vdots$ | $\vdots$ | $\vdots$ |

où

$(s, q_k, \text{EMPILER}, s')$ dans la case `[`$q_1$`][a]` est la réprésentation d'une transition $q_1 \xrightarrow[\frac{s}{\gamma}/\frac{s'}{\gamma}]{\texttt{'a'}} q_k$

$(s, q_k, \text{DEPILER}, s)$ dans la case `[`$q_2$`][a]` est la réprésentation d'une transition $q_2 \xrightarrow[\frac{s}{\gamma}/\gamma]{\texttt{'a'}} q_k$

$(\_, 0, \_, \_)$ dans la case `[`$q_3$`][a]` réprésente l'absence de transition depuis $q_3$ pour le caractère `'a'`.

$(\_, q_k, \text{EMPILER}, s)$ dans la case `[`$q_1$`][b]` est la réprésentation d'une transition (indépendante du sommet de pile) qui empile le symbole $s'$ $q_1 \xrightarrow[\gamma/\frac{s'}{\gamma}]{\texttt{'b'}} q_k$

$(\_, q_k, \text{INTACT}, \_)$ dans la case `[`$q_2$`][b]` est la réprésentation d'une transition (indépendante du sommet de pile) qui ne modifie pas la pile $q_2 \xrightarrow[\gamma/\gamma]{\texttt{'b'}} q_k$

Voici une fonction `step` écrite en C qui implémente l'exécution d'un pas de l'automate. L'appel à `step(&ec,&pc,mot,transition,pile)` modifie l'état courant `ec` de l'automate, la position courante `pc` dans le `mot` à reconnaître et la `pile` en accord avec les `transitions` de l'automate. La fonction `step` retourne 0 si aucune transition n'est possible sinon elle retourne le nouvel état de l'automate.

```
#define NSA ... // nombre de symboles de l'alphabet
#define NSP ... // nombre de symboles de l'alphabet de la pile
#define NE  ... // nombre d'états de l'automate de q_i à q_f

#define EMPILER 0
#define DEPILER 1

typedef etat struct{ int numero ; int action ; char symbole }

char pile[];
pile[0]='*';

int step(int *ec, int *pc, char mot[], etat transition[NE][NSA][NSP], char pile[]){

// ec désinge l'état courant de l'automate
// pc désigne la position courante dans le  mot à reconnaître
// l désigne la lettre à la position pc dans le mot
// sp désinge le symbole en sommet de pile

 sp = sommet(pile);
 l = mot[*pc] ;
 etat = transition[ec][l][sp] ;

 if ( etat.numero !=0){
   *ec = etat.numero ;
   *pc = *pc + 1;
   if (etat.action==DEPILER){ depiler(pile); }
   else
   if (etat.action==EMPILER){ empiler(etat.symbole,pile); }
 }
 return etat.numero ;
}
```