# Compilation

Anne Grazon – L3 Info Rennes 1

2015 – 2016, S2

## 1 Programmation par automate à états finis

### 1.1 Programmation dirigée par la syntaxe

Les données doivent suivre des règles d'écriture précises (syntaxe), pas toujours simples. Exemple

| Logiciel | Données |
|---|---|
| Compilateur | Programme Java |
| SGBD | Requêtes |

Les logiciels doivent vérifier que les données respectent la syntaxe souhaitée (analyse syntaxique) ; c'est nécessaire mais pas suffisant, on veut ajouter des traitements sur les données (produire un exécutable, des réponses aux requêtes de BDD). On doit donc prévoir des actions pour réaliser ces traitements ; l'analyseur syntaxique « pilote » les actions.

L'ensemble des données satisfaisant les contraintes de syntaxe est appelé langage d'entrée.
*Remarque* — On utilise les outils formels pour décrire un langage (pour éviter les ambiguïtés et pouvoir programmer une solution plus facilement).

Exemple

Langage des fiches de cotation (voir support).

On distingue deux types de contraintes sur les données :
- celles qui définissent l'écriture d'un item (*token*), les contraintes lexicales ;
- celles qui précisent l'organisation des items, les contraintes syntaxiques.

L'analyseur des données est donc constitué d'un analyseur lexical qui découpe la suite de caractères (seules données considérées dans ce cours) en items qui ont un sens pour l'application concernée, et d'un analyseur syntaxique qui vérifie que la séquence de ces items est correcte.

### 1.2 Description formelle d'un langage

La définition d'un langage nécessite la connaissance des entités admises (vocabulaire terminal, noté $V_T$) et des règles indiquant quelles séquences de $V_T^*$ sont admises.

Exemple

Pour le langage support : au niveau lexical $V_T$ = {'a'-'z', '0'-'9', '+', '–', ESP, RC, ';', '/', '.'},
au niveau syntaxique $V_T$ ={ancien, nouveau, par, ident, nbentier, nbreel, '+', '–', ';', '/'}.

**Expressions régulières** On ajoute les notations $w^+ = w.w^*, w^? = w|\epsilon$ ; **[exo 1]** donner les expressions régulières pour définir les items nbentier et nbreel, et l'ensemble des fiches de cotation du langage support.
- nbentier : ('0'-'9')$^+$ ;
- nbreel : ('0'-'9')$^+$($\epsilon$|.('0'-'9')$^*$) ;
- fiches : (ident (par nbentier)$^?$ (ancien nbreel nouveau ('+'|'–')$^?$ nbreel | nouveau nbreel)' ;')$^*$'/'.

Le mécanisme de reconnaissance adapté est l'automate à états finis ; c'est toujours le cas pour l'analyseur lexical, parfois pour l'analyseur syntaxique.

**Grammaires algébriques ou hors-contexte** Pour $G =< \Sigma, V, S, P >$, $L(G) = \{w \in \Sigma^* | S \rightarrow_P^+ w\}$.

## 1.3 Automate fini et actions

**Graphe des transitions** Les notations sont standard. On se limite à des automates déterministes. Pour la numérotation des états, on prend la convention suivante : l'état initial est 0, le final est le plus grand nombre, les autres sont numérotés consécutivement.

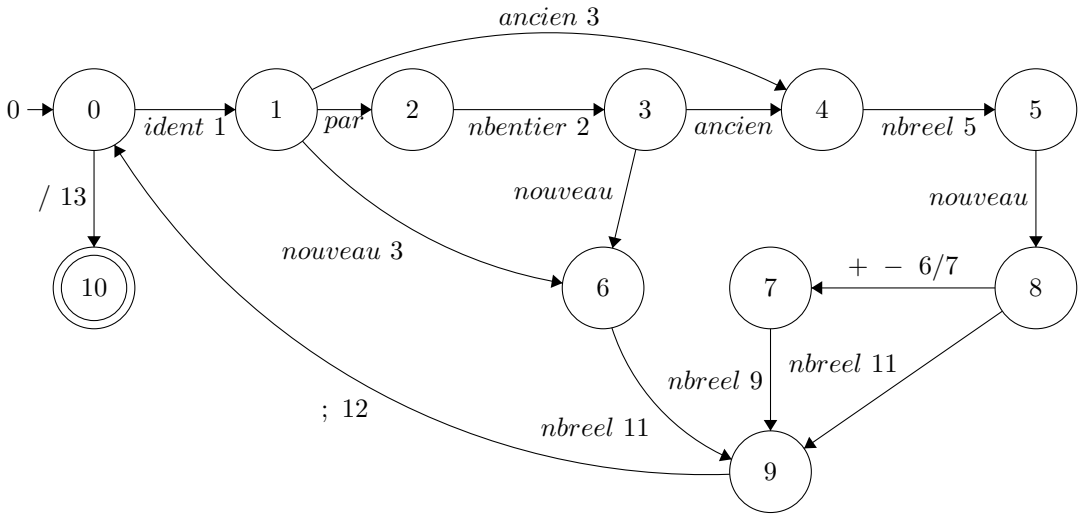


FIGURE 1 – Automate associé aux fiches de cotations. Les numéros qui étiquettent certaines transitions correspondent aux actions (voir paragraphe suivant).

**Actions associées aux transitions** L'objectif est de reconnaître si une donnée appartient au langage et surtout de traiter la donnée. Par exemple, on peut vouloir calculer le cours le plus haut pour chaque monnaie. Pour cela, on exécutera des actions sur certaines transitions ; on leur associe alors un numéro d'action. Par convention, $-1$ désigne l'action vide et 0 l'action d'initialisation.

Pour pouvoir traiter une donnée, on a besoin d'accéder à des informations supplémentaires pour certains items lexicaux (par exemple la valeur associée à un item nbentier ou nbreel). Ces informations sont des attributs lexicaux.

## 1.4 Programmation d'un automate fini déterministe

**Programmation directe** Pour des automates simples comme ceux d'analyse lexicale, on peut programmer directement l'automate. La mise en œuvre utilisée en TD/TP définit une classe abstraite, Lex qui modélise un analyseur lexical abstrait par la définition d'un flot d'entrée, d'une table des identificateurs et les méthodes abstraites d'accès à un item lexical (`abstract int lireSymb()`) ou à un identificateur (`abstract String repId(int nId)`).

Pour les chaînes de lettres, on attend la fin de la chaîne avant de déterminer si c'est un mot réservé ou un identificateur de monnaie.

**Programmation par tables** On utilise cette implémentation pour tout automate plus complexe (analyse syntaxique par exemple). On programme la table des transitions ainsi que la table des actions (pour chacune, une ligne par état, un colonne par item syntaxique possible en sortant de cet état). Ces tables sont ensuite interprétées par au niveau de la classe abstraite Automate, qui effectue l'analyse des données en s'en servant et en utilisant la fonction `lireSymb()` qui fournit l'item lexical courant.

Il ne faut pas oublier une action traitant les erreurs ; l'action d'initialisation, elle, est traitée à part (méthode `initAction` de la classe abstraite Automate).

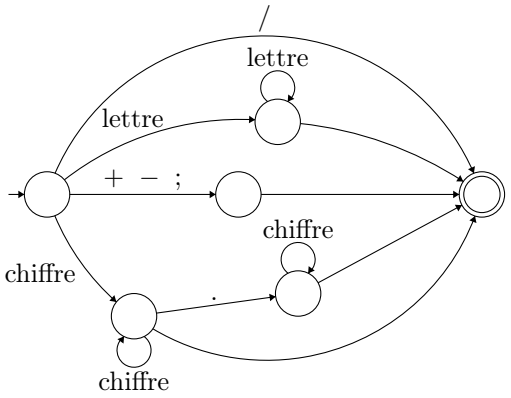


FIGURE 2 – Automate d'analyse lexicale du langage support.

**Traitement des erreurs** Au niveau lexical, on renvoie l'item correspondant à une erreur. Au niveau syntaxique, la table des transitions renvoie vers un état d'erreur. Soit l'erreur est considérée comme fatale et on arrête l'analyse ; soit on veut récupérer la situation et par exemple continuer l'analyse à la prochaine expression.

Dans ce cas, toute éventuelle mise à jour doit être faite uniquement à la fin d'une analyse correcte et on doit prendre garde au cas où l'erreur correspond à une fin d'expression. Une idée pour gérer ce cas plus complexe consiste à établir des symboles de reprise ; si une erreur de syntaxe survient lors de la lecture d'un de ces symboles, on la signale et on reprend l'analyse directement.

# 2 Analyse syntaxique descendante de gauche à droite (LR)

## 2.1 Limite des automates finis

Les automates finis sont adaptés aux langages réguliers ; or les langages des programmations doivent gérer par exemple le parenthésage et la cohérence des accolades. L'analyse de tels langages nécessite un automate à pile, utilisé alors pour l'analyse syntaxique d'une donnée.

## 2.2 Cas particulier : les grammaires d'expressions

On souhaite éviter l'ambiguïté : une grammaire est ambiguë s'il peut exister plusieurs arbres syntaxiques pour une même donnée. Cette situation n'est pas acceptable pour la compilation ; on la résout en évitant la double récursivité dans les règles ($X \to \alpha X$ et $X \to X\alpha$) et en respectant la priorité des opérateurs (en cas de priorité identique, on fera l'évaluation de gauche à droite).

EXEMPLE

Une grammaire non ambiguë pour les expressions arithmétiques :
exp $\to$ exp + terme | exp − terme | terme
terme $\to$ terme $*$ prim | term / prim | prim
prim $\to$ nb | (exp)

## 2.3 Analyseur descendant LR procédural - Points de génération

La machine adaptée à l'analyse de langages algébriques est l'automate à pile (il réalise une analyse descendante LR du langage généré par une grammaire algébrique). En partant de l'axiome, elle essaie d'obtenir la chaîne à analyser par dérivations successives à gauche (on a $L(G) = \{x \in \Sigma^* | S \to_g^+ x\}$).

*Remarque* — L'outil utilisé en TP, ANTLR, génère un analyseur syntaxique LR à partir d'une grammaire donnée (ainsi qu'un analyseur lexical).

L'analyse LR nécessite la notion de pile qui mémorise ce qui reste à faire. Une mise en œuvre consiste à utiliser des procédures pour « programmer » les règles de la grammaire, avec une procédure pour chaque non-terminal (la grammaire doit respecter certaines propriétés, notamment $LL(1)$ pour l'ambiguïté). La pile est alors simulée par les appels et retours de procédure ; quand les appels se terminent sans erreur et que la chaîne est totalement analysée, c'est le win, sinon, c'est la lose.

On peut ajouter des actions dans les règles de la grammaire, pour traiter la donnée en parallèle de sa reconnaissance. Ces actions sont ici appelées « points de génération » (notés par un numéro *as usual*).

# 3 Éléments de compilation

## 3.1 Compilateur considéré

**Structure du compilateur** Le langage d'entrée (langage Projet) est impératif ; il gère :
- les identificateurs de variables et de constantes ;
- les valeurs entières et booléennes ;
- l'affectation (notée `:=`) ;
- les expressions arithmétiques ou booléennes (avec $+, -, *$, div, et, ou, non et comparaisons) ;
- les structures conditionnelles, l'itération et un switch « à la Java » ;
- lecture et écriture ;
- des procédures sans résultat (paramètres modifiables et/ou fixes) ;
- des programmes et modules.

La compilation d'un programme `p.pro` en code objet `p.obj` se fait au moyen du compilateur à réaliser. Ce code objet peut ensuite être exécuté par une machine à pile MAPILE (fournie, avec 30 ordres).

**Réalisation du compilateur** Le langage d'entrée est spécifié par sa syntaxe (règles d'écriture du programme source) et sa sémantique (signification donnée à chaque construction syntaxique). La syntaxe se décompose en une grammaire donnée, et des contraintes non exprimables par une grammaire algébrique (délégation de déclaration, ...). L'analyseur syntaxique est un automate à pile dont les contrôles supplémentaires sont fournis par des points de génération. La sémantique est la séquences d'ordres MAPILE générée à partir du programme source. Le code objet est également produit par les points de génération.

## 3.2 Table des symboles - Compilation des déclarations

Elle permet de traiter les déclarations et l'utilisation des identifiants déclarés. En particulier, elle contient le type et la catégorie des identifiants (variable, constante...) ainsi qu'un champ d'information (adresse d'une variable dans la pile d'exécution, valeur d'une constante).

Le code objet doit en outre fournir un tableau `po` dans lequel sont rangés les ordres MAPILE au fur et à mesure de la compilation. Ce tableau est recopié dans un fichier en fin de compilation. (Les ordres MAPILE sont codés par des entiers auxquels sont associées des constantes à utiliser dans le code du projet.)

## 3.3 Expressions et contrôles de type (TP)

## 3.4 Compilation des instructions

Les instructions `si`, `ttq` et `cond` nécessitent des branchements « en avant » (à des adresses inconnues au moment où on fait le branchement BINCOND ou BSIFAUX) ; il faut mémoriser des emplacements du code objet (indices de `po`) et gérer l'imbrication de ces instructions, au moyen d'une pile nommée `pileRep`.