

L'**analyse syntaxique** est le processus qui consiste à vérifier si un mot peut être engendré par une grammaire, en trouvant l'arbre de dérivation qui permet de l'engendrer.

Un **analyseur syntaxique** est un logiciel dont la donnée est un « mot » (texte à analyser) et dont le résultat indique si ce texte est correct (c'est à dire engendré par la grammaire). En plus de ce travail de vérification syntaxique, l'analyseur déclenche le plus souvent des actions **sémantiques** guidées par le texte.

Par exemple un analyseur lit un texte devant correspondre à une expression arithmétique (vérification syntaxique) et calcule en même temps la valeur de l'expression (action sémantique).

L'**analyse récursive descendante** désigne une catégorie d'algorithmes d'analyse syntaxique.

Son principe est de créer un « sous-programme » pour chaque variable de la grammaire. Selon le langage de programmation utilisé, ce sous-programme sera une fonction, une méthode, une procédure... Par la suite nous parlerons de **méthode** puisque nous programmerons en Java.

L'implémentation de chaque méthode découle directement des règles existant pour la variable correspondante.

Supposons par exemple que pour une variable  $S$  on dispose des règles  $S \rightarrow XY$  et  $S \rightarrow aS$ , l'implémentation de  $S()$  ressemblera à

```
S() {
    if (condition à déterminer)
        //appliquer S->XY
        X(); Y();
    else if (condition à déterminer)
        // appliquer S->aS
        Vérifier que la lettre à lire est 'a' et la consommer;
        S();
    else
        déclencher une erreur;
}
```

Comme l'on voit, il nous faut encore déterminer dans quel cas appliquer l'une ou l'autre règle, ou aucune des deux (erreur). Nous verrons plus loin que la réponse sera dans la **table d'analyse LL**.

## 1 Grammaires LL(1)

L'avantage de l'analyse récursive descendante est de fournir un algorithme simple et efficace ne nécessitant qu'une seule lecture du texte à analyser.

Son principal inconvénient est de ne pas pouvoir s'appliquer à toutes les grammaires : seules les grammaires vérifiant une propriété particulière (appelée LL(1)) pourront donner lieu à une analyse récursive descendante.

Nous verrons en cours comment déterminer si une grammaire convient et, si oui, comment construire sa table d'analyse LL(1). Retenez d'ores et déjà qu'une grammaire ambiguë ou récursive gauche n'est jamais LL(1).

### 1.1 Table d'analyse LL(1)

Cette table d'analyse est à deux dimensions. les clés sont les variables pour la première dimension et les symboles terminaux pour la deuxième.

Chaque case  $T[S,x]$  de la table contient

- soit une seule règle, dont  $S$  est la partie gauche.
- soit rien (on note parfois dans ce cas le mot "erreur")

La table détermine quelle règle doit s'appliquer dans chaque circonstance.

Dans le cas de l'implémentation récursive descendante, elle définit implémentation des méthodes : pour chaque méthode  $S()$ , si la prochaine lettre à lire est  $x$ , on doit appliquer la règle figurant dans  $T[S,x]$ . Si  $T[S,x]$  ne contient pas de règle, c'est que le mot ne peut être engendré et l'on déclenche une erreur.

## 2 Exemple

Considérons une grammaire et sa table d'analyse LL(1)

	S	A	B	D
$S \rightarrow AB \mid Da$	$S \rightarrow AB$	$A \rightarrow aAb$	/	/
$A \rightarrow aAb \mid \epsilon$	$S \rightarrow AB$	$A \rightarrow \epsilon$	$B \rightarrow bB$	/
$B \rightarrow bB \mid \epsilon$	$S \rightarrow Da$	/	/	$D \rightarrow dD$
$D \rightarrow dD \mid e$	$S \rightarrow Da$	/	/	$D \rightarrow e$
	$S \rightarrow AB$	$A \rightarrow \epsilon$	$B \rightarrow \epsilon$	/

Le symbole # désigne le marqueur de fin du mot (symbole que l'on ne trouve qu'une seule fois, en fin de mot).

Voici maintenant l'implémentation complète de la méthode  $S()$  :

```
private void S() throws SyntaxException, ParseException {
    switch (current) {
        case 'a':
        case 'b':
        case END_MARKER:
            // S -> AB
            A();
            B();
            break;
        case 'd':
        case 'e':
            // S -> Da
            D();
            eat('a');
            break;
        default:
            // erreur
            throw new SyntaxException(ErrorType.NO_RULE, current);
    }
}
```

Les méthodes de l'analyseur ont accès au texte à analyser, qui est lu lettre par lettre, de gauche à droite :

- `current` : lettre courante
- `eat(caractère)` vérifie que le caractère passé en paramètre est bien le caractère courant. Si oui, la lecture progresse d'un caractère (`current` est donc modifié). Dans le cas contraire une erreur de syntaxe est signalée.

Chacune des méthodes est construite sur le même modèle. Chacune respecte les propriétés suivantes :

- une invocation de la méthode  $V()$  doit analyser une portion de texte pouvant être engendrée par la variable  $V$ .
- avant l'invocation de la méthode  $V()$ , `current` doit contenir la première lettre de la portion de texte à analyser par  $V()$ .
- à l'issue de l'exécution de  $V()$ , `current` contient la première lettre du texte qui SUIT la portion de texte qui vient d'être analysée.

Vous trouverez dans l'archive jointe le code complet d'un analyseur pour cette grammaire (classe `ArdExemple`).



— EOD : marqueur de fin.

Comme vous l'aviez fait lors des séances consacrées aux analyseurs lexicaux, vous créerez les classes nécessaires à la représentation des tokens.

2. dans la grammaire, on remplace les caractères ( et ) par les tokens `Ouvrante` et `Fermante` ; on supprime les variables  $L$  et  $C$  en les remplaçant par des symboles terminaux, respectivement les tokens `Lettre` et `Nombre`.

$$S \rightarrow ERS \mid \epsilon$$

$$E \rightarrow Lettre \mid Ouvr \ S \ Ferm$$

$$R \rightarrow Nombre \mid \epsilon$$

	S	E	R
Lettre	$S \rightarrow ERS$	$E \rightarrow Lettre$	$R \rightarrow \epsilon$
Nombre	/	/	$R \rightarrow Nombre$
Ouvr	$S \rightarrow ERS$	$E \rightarrow OuvrSFerm$	$R \rightarrow \epsilon$
Ferm	$S \rightarrow \epsilon$	/	$R \rightarrow \epsilon$
Eod	$S \rightarrow \epsilon$	/	$R \rightarrow \epsilon$

Créez un nouvel analyseur syntaxique / évaluateur, adapté du précédent mais qui cette fois lit des tokens et non plus directement des caractères.