Analyse récursive descendante

---

1. 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).
2. L'analyse descendante LL(1) est une catégorie d'analyse syntaxique
3. L'analyse récursive descendante désigne une implémentation de l'analyse descendante sous la forme de sous-programmes récursifs.

   Nous opterons ici pour une approche orientée objet : chaque sous-programme sera donc une méthode

   Le principe est de créer **une méthode pour chaque variable** de la grammaire.

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

   Supposons par exemple que pour une variable V on dispose des règles $V \to XY$ et $V \to aS$. La méthode devra définir laquelle des 2 règles doit être appliquée ; ce choix dépend du prochain caractère à lire

   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)

Ce type d'analyse ne peut pas s'appliquer à toutes les grammaires : seules les grammaires vérifiant un propriété particulière (appelée LL(1)) pourront donner lieu à une analyse récursive descendante.
Nous verrons en cours comment déterminer si une grammaire est LL(1) et, si oui, comment construire sa **table d'analyse LL(1).** Retenez d'ores et déjà qu'une grammaire ambigüe ou récursive gauche n'est jamais LL(1).

1.1 Table d'analyse LL(1)

La table d'analyse est à deux entrées : les variables de la grammaire d'un côté, les terminaux de l'autre.

		V	
x		règle ($V \to w$) ou case vide	

Une case de la table contient une et une seule règle, ou est vide. Ceci détermine le comportement à suivre dans chaque circonstance :

- si la variable à dériver est V, la prochaine lettre est x, et si $T[V, x]$ contient $(V \to w)$, alors il faut appliquer la règle $(V \to w)$.
- si la variable à dériver est V, la prochaine lettre est x, et si $T[V, x]$ est vide, alors aucune règle n'est applicable et le mot n'est pas correct.

Exemple : la table LL(1) pour la grammaire $\Sigma = \{a, b, c\}, \mathcal{V} = \{\underline{S}, T\}, \mathcal{R} = \{S \to aSb,\ S \to cT,\ T \to cT, T \to \varepsilon\}$

	S	T
a	$S \to aSb$	
b		$T \to \varepsilon$
c	$S \to cT$	$T \to cT$
#		$T \to \varepsilon$

La table LL(1) définit le schéma d'implémentation de chaque s méthode :

```
Code de S() :
        si le prochain caractère est 'a'
              //appliquer S->aSb
                    passer au caractère suivant
                    S()
                    Vérifier que le prochain caractère est 'b' puis passer au suivant
        sinon si le prochain caractère est 'c'
              //appliquer S->cT
                    passer au caractère suivant
                    T()
         sinon // aucune règle n'est applicable
                 déclencher une erreur «mot incorrect»

Code de T() :
        si le prochain caractère est 'c'
              //appliquer T->cT
                    passer au caractère suivant
                    S()
                    Vérifier que le prochain caractère est 'b' puis passer au suivant
        sinon si le prochain caractère est 'b' ou le marqueur de fin (#)
              //appliquer S->epsilon
                   (ne rien faire)
        sinon // aucune règle n'est applicable
                 déclencher une erreur «mot incorrect»
```

1.2 L'implémentation en python

Les analyseurs seront implémentés en PYTHON. Une classe de base (la classe **Ard**) vous est fournie. Cette classe implémente la gestion de la lecture du texte à analyser ainsi que la méthode principale de déclenchement de l'analyse. Le texte est lu comme une suite de tokens (et non comme une suite de caractères). Voici les composants de la classe Ard :

1. attributs et méthodes protégés (accessibles depuis la classe « fille ») :

   `_current` est un attribut qui contient le token courant

   `_next()` méthode qui passe au token suivant (donc modifie `_current`). La méthode ne produit pas de résultat.

   `_eat(type_de_token)` si le token courant est du type indiqué, passe au token suivant. Déclenche une exception si le token courant n'est pas du type indiqué.

2. méthode publique

   `parse(texte, lexer)` déclenche l'analyse du texte. Le texte est lu au travers d'un analyseur lexical (paramètre lexer) tel que ceux fournis par le paquetage python **sly** (utilisé en ALR).

   Le paramètre lexer est optionnel. À défaut, l'analyseur utilisera un lexer élémentaire (inclus dans le paquet) qui transforme chaque caractère en un token de même nom (p.ex. le caractère 'x' sera transformé en un token de type 'x' et de valeur 'x')

En fin de lecture, la classe Ard ajoute un token supplémentaire (non généré par le lexer), le « marqueur de fin ». Son type est `'EOD'`.

Ce que doit implémenter la classe « fille » :

— les méthodes de l'analyseur récursif, une méthode par variable comme expliqué ci-dessus. Ces méthodes n'ont pas vocation à être appelées directement, elles seront donc protégées (leur nom commencera par un `_`)
— la classe doit définir une méthode `_axiom()`. On peut utiliser un simple alias désignant l'une des méthodes précédentes (voir exemple)

```
import ard
from ard import Ard

class ArdExemple1 (Ard) :
        # S -> aSb | cT
        # T -> cT  | epsilon
    def __init__(self) :
        self._axiom = self._S

    def _S(self) :
        if self._current.type == 'a' :
            # S -> aSb
            print('S->aSb', self._current,  self._current.index)
            self._next()
            self._S()
            self._eat('b')
        elif self._current.type == 'c' :
            # S -> cT
            print('S->cT', self._current,  self._current.index)
            self._next()
            self._T()
        else :
            raise ard.SyntaxError("NO_RULE","S", self._current);

    def _T(self)  :
        if self._current.type == 'c' :
            # T -> cT
            print('T->cT', self._current, self._current.index)
            self._next()
            self._T()
        elif self._current.type in ('b', 'EOD') :
            print('T->epsilon', self._current,  self._current.index)
            # T -> epsilon
            pass
        else :
            raise ard.SyntaxError("NO_RULE","T", self._current);

if __name__ == '__main__' :
    c = ArdExemple1()
    try :
        c.parse('aacbb')
    except ard.SyntaxError as e :
        print (e)
```

2 Exercice 1

2.1 Préparer un analyseur lexical

Écrivez un analyseur lexical en utilisant Lexer (paquet sly). Celui-ci distinguera les tokens suivants
— `ENTIER` : entier décimal non signé. La valeur associée sera le nombre représenté.
— `LETTRE` : ce type de token désigne une lettre au sens usuel
— `OUVRANTE` : parenthèse ouvrante
— `FERMANTE` : parenthèse ouvrante

2.2 La grammaire

Voici une grammaire :

$$S \to E\ R\ S \mid \varepsilon$$
$$E \to \boxed{\text{lettre}} \mid (\ S\ )$$
$$R \to \boxed{\text{entier}} \mid \varepsilon$$

Cette grammaire définit les données de la façon suivante : un mot est une suite d'éléments E, chaque élement pouvant être suivi d'un entier (optionnel). Un élement est une lettre ou une suite d'éléments entre parenthèses.
Voici quelques exemples de mots du langage engendré : aab, $a2b3a2$, $,a(ba)2$, $(a(bc)2)3(ba)2$
et sa table d'analyse LL(1) :

	S	E	R
LETTRE	$S \to ERS$	$E \to$ LETTRE	$R \to \varepsilon$
ENTIER	/	/	$R \to$ ENTIER
OUVRANTE	$S \to ERS$	$E \to$ OUVRANTE S FERMANTE	$R \to \varepsilon$
FERMANTE	$S \to \varepsilon$	/	$R \to \varepsilon$
EOD	$S \to \varepsilon$	/	$R \to \varepsilon$

(`EOD` désigne le marqueur de fin. Ce token est engendré « automatiquement » par la classe Ard)

2.3 Construire un analyseur simple

Implémentez un analyseur récursif descendant pour cette grammaire en étendant la classe `Ard`.
Testez votre analyseur sur différents cas de mots corrects et incorrects.
Cet analyseur vous servira de base pour la question suivante, **mais prenez soin de conserver une copie de cet analyseur simple dans la version que vous venez de terminer**.

2.4 Ajouter des actions sémantiques

On peut voir chacun de ces mots comme une expression condensée décrivant une suite de lettres. Un chiffre indique un nombre de répétitions à appliquer à l'élément précédent. Par exemple

- `ba2` désigne le mot *baa*
- `(ba)2` désigne le mot *baba*
- `(a(bc)2)3(ba)2` désigne le mot *abcbcabcbcabcbcbaba*

Les syntaxe des expressions se décline à partir des règles de la grammaire :

- une expression S est soit vide, soit constituée d'un élément E puis d'un facteur de répétition R puis d'une autre expression S.
- un élément E est soit une lettre L, soit une expression entre parenthèses.
- un facteur de répétition R est soit vide, soit un chiffre C.

Le mot développé correspondant à une expression est le mot équivalent qui ne contient ni parenthèses ni chiffre. Par exemple `baba` est le développé de `(ba)2`.
Vous allez transformer votre analyseur simple pour faire en sorte qu'il calcule le mot développé, au fur et à mesure de l'analyse.
Pour cela chaque méthode va renvoyer (`return`) une valeur qui représente la valeur sémantique de la sous-expression qu'elle vient d'analyser.

1. `_R()` : renvoie le nombre de répétitions représenté.
2. `_E()` : renvoie le mot développé correspondant à cette portion d'expression.
3. `_S()` : renvoie le mot développé correspondant à cette expression.

Notez que si vous avez correctement écrit l'analyseur lexical, le token LETTRE est associé à une valeur chaîne et le token ENTIER à une valeur numérique.
La méthode `parse()` héritée de la classe Ard renvoie le résultat de la méthode definie comme étant l'axiome (`_axiom()`). Une fois les acions sémantiques implémentées, `parse()` renverra donc le mot développé correspondant au mot condensé qui a été lu.