Un analyseur ascendant en Python : sly.Parser

Le paquetage python **sly**, déjà utilisé pour l'analyse lexicale, fournit également un analyseur syntaxique : `sly.Parser`.

1. `sly.Parser` génère lui-même tout ce qui est nécessaire à l'analyse syntaxique (en analyse récursive descendante (ARD), nous devions calculer d'abord « à la main » la table LL(1)). Techniquement, `sly.Parser` est un **générateur** d'analyseur syntaxique.
2. ici l'utilisateur « se contente » de coder les règles de la grammaire, sous une forme attendue par le paquetage **sly**. Lors de l'exécution, `sly.Parser` va d'abord compiler les règles de la grammaire et fabriquer les tables internes nécessaires à l'analyse.
3. `sly.Parser` utilise une méthode d'analyse **ascendante** (et non descendante comme LL(1)) appelée LALR(1). Même si les principes de l'analyse ascendante n'ont pas encore été vus en cours, `sly.Parser` peut être utilisé par un utilisateur néophyte.

   Il faut néanmoins savoir que toutes les grammaires algébriques ne sont pas compatibles avec l'analyse LALR(1). Quand la grammaire codée n'est pas LALR(1) des messages d'erreurs ou des warning sont émis à l'exécution par `sly.Parser`. Ces messages font état de « conflits » de règles.
4. on peut associer aux règles des actions sémantiques et des calculs d'attributs synthétisés.

1 Mode d'emploi simplifié

- pour définir un parser, il faut créer une classe qui étend `sly.Parser`
- **alphabet terminal** : `sly.Parser` s'utilise conjointement avec `sly.Lexer`. L'alphabet terminal est donc constitué des tokens renvoyés par le lexer. Ils sont à déclarer par un attribut `tokens`
- **variables** : chaque variable est implémentée par une méthode (ou « fonction ») du parser. En réalité (et même si cela paraît bizarre) chaque fonction sera définie autant de fois qu'il y a de règles la concernant (ne pas trop s'étonner ... sly gère !). La fonction doit posséder un unique paramètre (en plus de `self`)
- **convention d'écriture** : les tokens s'écrivant en majuscules, on écrira les **variables en minuscules** pour les différencier des tokens ... donc l'inverse de ce qu'on fait habituellement sur papier !
- l'**axiome** est défini par l'attribut `start='nom de la variable axiome'`
- **règles** : chaque règle $V \rightarrow$ *partie droite* est traduite par une définition de la fonction V précédée du décorateur `@_('partie droite')`
- le corps des fonctions ne sert qu'aux actions sémantiques. Pour une simple analyse les fonctions peuvent donc ne rien faire.

1.1 Exemple très simple, sans action sémantique

Pour le langage $\{a^n b^n | n \geq 0\}$, on dispose de la grammaire $S \rightarrow aSb \mid \varepsilon$
les lettres a et b seront implémentées par les tokens `A` et `B`. La grammaire devient $s \rightarrow AsB \mid \varepsilon$
Voici ci-dessous sa traduction avec **sly**. Notez que la méthode `parse()` du parser renvoie le résultat de la fonction axiome (c'est à dire `None` dans notre exemple) si la chaîne saisie est correcte vis à vis du parser et déclenche une exception en cas contraire.

```
from sly import Parser
from sly import Lexer
class MiniLexer(Lexer) :
    tokens = {A, B}
    A = 'a'
    B = 'b'

    ignore = ' \t'

class AnBnParser(Parser) :
    tokens = MiniLexer.tokens
    start = 's'
    # s -> A s B  :
    @_('A s B')
    def s(self, p):
        return None
    # s -> epsilon  :
    @_('')
    def s(self, p):
        return None

if __name__ == '__main__':
    lexer = MiniLexer()
    parser = AnBnParser()
    while True:
        try:
            text = input('test a^n b^n > ')
            result = parser.parse(lexer.tokenize(text))
            print(result)
        except EOFError:
            break
```

Le code de ce fichier vous est fourni dans l'archive. Testez-le en incluant dans vos essais plusieurs cas de chaînes incorrectes.

2 Calculs d'attributs

`sly.Parser` permet d'associer un calcul d'attributs **synthétisés**. À chaque variable et chaque token est associé un attribut. On rappelle qu'un calcul d'attribut synthétisé consiste à calculer la valeur qui sera associée à la variable de la partie gauche, en utilisant éventuellement les valeurs d'attributs des variables ou tokens de la partie droite.

- l'attribut associé à un token est la **valeur** du token, celle que renvoie le lexer.
- l'attribut associé à une variable est la valeur `renvoyée` (le `return`) par la fonction qui porte son nom (dans l'exemple précédent l'attribut de chaque variable valait donc `None`)
- on accède aux attributs des variables/tokens de la partie droite en utilisant le nom du paramètre de la fonction (on l'a toujours nommé `p` dans notre exemple, on continuera à faire ainsi)
  Prenons l'exemple d'une règle $v \to x\ y\ z\ y\ x$ implémentée par

```
@_('x y z x y x')
function v(self, p) :
    ...
```

  Plusieurs notations sont possibles. La première consiste à suffixer p par le nom de la variable : `p.z` est l'attribut associé à `z` par exemple. Pour `x` et `y`, nous avons plusieurs occurrences, il faut alors ajouter un numéro (on commence à numéroter à partir de 0, de gauche à droite, en ne prenant en compte QUE la partie droite). Par exemple `p.x0` est la valeur d'attribut du premier `x`, `p.x1` celle du deuxième `x`
  On peut aussi utiliser un index `[i]` (une numérotation unique de la partie droite).
  En résumé, voici les 2 écritures possibles :

x	y	z	y	x
p.x0	p.y0	p.z0 p.z	p.y1	p.z1
p[0]	p[1]	p[2]	p[3]	p[4]

2.1 Exemple : calcul du nombre de a

Dans l'analyseur précédent nous allons associer à s un attribut qui représente le nombre de a (ou de b) du mot analysé.

- règle $s \to \varepsilon$: l'attribut de s vaut 0 (aucun a ni b)
- règle $s \to A\ s\ B$: la valeur associée au s (partie gauche) est égale à 1 plus celle du s de la partie droite.

```
class AnBnParser(Parser) :
    tokens = MiniLexer.tokens
    start = 's'
    # s -> A s B  :
    @_('A s B')
    def s(self, p):
        return 1 + p.s
    # s -> epsilon  :
    @_('')
    def s(self, p):
        return 0
```

Réalisez cette modification dans le code fourni et testez.

2.2 Attributs multiples et type des attributs

Selon le modèle présenté en cours on peut associer plusieurs attributs à une variable.
SLY ne permet d'associer qu'un seul attribut mais cet attribut est de type quelconque. Si l'on doit associer plusieurs attributs, on choisira donc en réalité un attribut de type **tuple**, ou **liste**, ou un **objet**.
Vous serez vigilant à ce que les attributs associés à une même variable soient de même type (toutes les fonctions de même nom doivent renvoyer des valeurs de même type).

3 Exercices

Exercice 1 :
Nous reprenons le langage déja présenté dans la fiche de TD3, exercice 3. Il est composé des expressions faisant intervenir des constantes (symbolisées par la lettre c) des appels d'une fonction g à un seul paramètre et d'une fonction f à 2 paramètres. Les appels de fonctions peuvent être imbriqués sans limite. Voici quelques exemples de mots corrects :

```
c   g(c)   f(c,c)   f(c,g(c))   g(g(c))   g(f(c,c))  f(g(f(c,c)),f(c,c))
```

et voici une grammaire pour ce langage : $E \to c \mid g(E) \mid f(E,E)$ avec les terminaux $\{c, f, g, (,), ,\}$
Vous trouverez dans l'archive fournie un lexer définissant les 6 symboles terminaux (les constantes sont des entiers décimaux et la valeur qui leur est associée est de type `int`)
Q 1 . Définissez un parser pour le langage, sans action sémantique à cette étape. Testez.
Q 2 . On souhaite afficher le nombre de constantes figurant dans l'expression. Associez à la variable un attribut qui désigne ce nombre et ajoutez les calculs d'attributs nécessaires. Conservez une copie de cette version du parser.
Q 3 . Si l'on suppose que les constantes sont des nombres (c'est le cas avec notre lexer), que la fonction g consiste à multiplier par 3 sonparamètre et la fonction f à faire la somme de ses 2 paramètres, faites en sorte qu'à la fin de l'analyse on connaisse le résultat de l'expression. NB : ce calcul sémantique remplace celui de la question précédente. Conservez une copie de cette version du parser.
Q 4 . On souhaite cette fois connaître le nombre maximal de niveaux d'imbrication des appels de fonctions. Par exemple 0 pour c, 1 pour $g(c)$ ou $f(c,c)$, 2 pour $f(g(c),g(c))$ ou $g(g(c))$ ou $f(c,g(c))$ (etc ..). Un dernier exemple : 3 pour $f(g(f(c,c)),f(c,c))$
NB : ce calcul sémantique remplace celui de la question précédente. Conservez une copie de cette version du parser.
Q 5 . Réalisez une version du parser qui calcule simultanément les résultats des 3 questions précédentes. Indication : l'attribut synthétisé associé à la variable peut être un triplet, par exemple. L'attribut associé au token `CONST` reste un entier simple.

Exercice 2 :
Nous reprenons ici le thème des expressions booléennes (cf TDM2). Vous utiliserez la grammaire :

$$\begin{array}{lcl} E & \longrightarrow & E \texttt{ || } T \mid T \\ T & \longrightarrow & T \texttt{ \&\& } F \mid F \\ F & \longrightarrow & \texttt{! } F \mid \texttt{ (} E \texttt{) } \mid \texttt{c} \mid \texttt{i} \end{array}$$

> Pour le TDM2 il nous fallait une grammaire $LL(1)$: la grammaire avait donc été transformée pour en supprimer la récursivité gauche.
> Avec l'analyse $LALR$ implémentée par `sly.Parser`, la récursivité gauche n'est plus un problème, on peut revenir à la version originelle de la grammaire.

1. Vous utiliserez le lexer issu du TDM2. Traduisez la grammaire en vous fondant sur les tokens définis dans ce lexer et implémentez-la dans un parser.
2. En utilisant le calcul d'attributs synthétisés, implémentez le calcul de la valeur de l'expression (le calcul de la valeur booléenne, comme dans la première partie du TDM2, pas la réduction d'expression)