TDM 4 - Automates et Langages Réguliers

octobre 2021

Analyse lexicale

Un **analyseur lexical** est un composant logiciel qui découpe un flux de données composé de caractères en une suite d'entités de niveau supérieur : les **unités lexicales** (ou « **tokens** »). Une unité lexicale est spécifiée par une expression régulière.
Un analyseur lexical est aussi appelé « **lexical parser** », « **tokenizer** », ou « **scanner** ».
Nous utiliserons un générateur d'analyseur lexical fourni par un paquetPython : **SLY** .

1 Analyse lexicale : principe et exemple

1.1 Les tokens

Un **token** (« unité lexicale ») est une portion du texte qui correspond à un motif préalablement spécifié par une expression régulière.
Par exemple si l'on définit 3 motifs :

- entier : `[0-9]+`,
- identificateur : `[A-Za-z][A-Za-z0-9]*`
- opérateur : `[-+*/]`

le texte `alpha+321*x5` sera découpé en une suite de 5 tokens :

IDENT	OPE	ENTIER	OPE	IDENT
alpha	+	321	*	x5

IDENT, OPE, ENTIER désignent le type du token, alors que `alpha + 321 * x5` désigne sa valeur.

1.2 L'analyseur lexical

L'analyseur lexical est un composant logiciel intégré à une application. Il offre aux autres composants une méthode permettant de lire des données token par token plutôt que caractère par caractère.
Avantages :

- le reste de l'application n'a pas à se soucier de la syntaxe des tokens. Une éventuelle modification de la syntaxe des tokens ne concernera donc que l'analyseur lexical .
- un même analyseur lexical peut être utilisé par plusieurs applications.

1.3 Générateur d'analyseur lexical

Un analyseur lexical peut être développé directement, mais il est préférable d'utiliser un outil appelé **générateur d'analyseur lexical**.
Le plus connu est l'utilitaire historique nommé **lex** qui génère un analyseur lexical écrit en langage C.

2 SLY, générateur d'analyseur lexical

Le paquet SLY fournit un générateur d'analyseur lexical : Lexer que nous utiliserons ici. Notez, pour mémoire, que SLY ournit un générateur d'analyseur syntaxique (Parser) pour des langages algébriques (donc non réguliers, donc hors programme ALR).
La documentation de SLY est là : `https://sly.readthedocs.io/en/latest/sly.html`

2.1 Un premier exemple simple

La spécification de l'analyseur est écrite dans une classe Python Voici la spécification d'un analyseur pour l'exemple précédent :

```
from sly import Lexer
from sly.lex import LexError

class ExampleLexer(Lexer):
    # token types :
    tokens = {IDENT,OPE,ENTIER}
    # token specifications :
    ENTIER = r'[0-9]+'
    IDENT =  r'[A-Za-z][A-Za-z0-9]*'
    OPE =    r'[-+*/]'
```

— la spécification figure dans une classe python qui étend la classe `Lexer`. Elle doit donc être déclarée par :
`class MonAnalyseur(Lexer) :`

— les types de token doivent être déclarés dans un ensemble associé à la variable `tokens`
`tokens = {IDENT,OPE,ENTIER}`

— la spécification de l'expression régulière de chaque token est ensuite définie dans une variable du nom du token :
```
ENTIER = r'[0-9]+'
IDENT =  r'[A-Za-z][A-Za-z0-9]*'
OPE =    r'[-+*/]'
```

2.2 Utilisation de l'analyseur

```
analyseur = ExampleLexer()
tokenIterator = analyseur.tokenize('alpha+321*x5')
for tok in tokenIterator :
    print(f'token -> type: {tok.type}, valeur: {tok.value}')
```

affiche :

```
token -> type: IDENT, valeur: alpha
token -> type: OPE, valeur: +
token -> type: ENTIER, valeur: 321
token -> type: OPE, valeur: *
token -> type: IDENT, valeur: x5
```

Explication du code :
— `analyseur` est une instance de la classe `ExempleLexer`. Cest donc un objet capable de découper un texte en tokens.
— sa méthode `.tokenize` découpe en tokens le texte fourni en paramètre. Le résultat (ici rangé dans la variable `tokenIterator`) est un objet qui va permettre de parcourir la liste des tokens.
— chaque token possède un attribut `.type` et un attribut `.value`

2.3 Recherche de token : principe

La méthode tokenize progresse dans le texte en cherchant à quelle expression régulière correspond le début du texte non encore découpé. Si une expression est trouvée, le token correspondant est produit. Dans notre exemple (`alpha+321*x5`) :
— le texte à analyser est `alpha+321*x5` : son début correspond à l'expression `[A-Za-z][A-Za-z0-9]*` : le token IDENT a été trouvé avec pour valeur `alpha`
— le texte restant à analyser est `+321*x5` son début correspond à l'expression `[+-*/]` : le token OP a été trouvé avec pour valeur `+`
— ...etc ...

Si le texte ne correspond à aucune des expressions régulières, un erreur se déclenche et le programme s'arrête en déclenchant une exception `LexError`.

Et si plusieurs expressions régulières conviennent ?
Sur notre exemple, cela ne peut pas se produire, mais ajoutons une définition de token :

```
SI = r'if'
```

Le texte `if` peut correspondre à la fois au token IDENT et au token SI.
L'analyseur généré par SLY donne priorité au token dont la spécification d'expression a été définie en premier.
Donc, avec la déclaration :

```
SI = r'if'
IDENT = r'[A-Za-z][A-Za-z0-9]*'
```

le texte `ifx+2` sera découpé en SI (if) IDENT (x) OP (+) ENTIER (2). alors qu'avec la déclaration

```
IDENT =  r'[A-Za-z][A-Za-z0-9]*'
SI = r'if'
```

le texte `ifx+2` sera découpé en IDENT (ifx) OP (+) ENTIER (2).

2.4 Des spécifications plus élaborées

2.4.1 Des traitements spécifiques à certains tokens

Quand on veut appliquer un traitement particulier à un type de token, on peut fournir une fonction (ou plus exactement une **méthode**) du nom du type de token.
Exemple :

```
def ENTIER(self, t) :
    t.value = int(t.value)
    return t
```

Retenir
- la méthode reçoit en argument (dans notre exemple, l'argument `t`) le token détecté, avant sa transmission définitive.
- la méthode renvoie le token qui sera réellement pris en compte. Par exemple ici on change l'attribut `value` du token, pour faire en sorte que ce soit un entier et non une chaîne.

2.4.2 Ignorer certaines parties de texte

En principe la présence d'un texte qui ne correspond à aucun token déclenche une exception `LexError`.

On entend par « ignorer » le fait de tolérer sans produire d'erreur certains caractères ou portions de texte qui ne correspondent à aucun token. L'analyseur fera « comme s'ils n'étaient pas là ».
Ce comportement est très utilisé pour autoriser des espaces entre les tokens ou encore des commentaires.
Il y a 2 possibilités pour ignorer une portion de texte.

Les variables `ignore`
On peut définir dans la classe :
- une variable nommée `ignore` : une chaîne dont tous les caractères seront ignorés lors de l'analyse lexicale.
  Par exemple si l'on définit `ignore=' _'` les espaces et les underscores seront ignorés.
- une ou plusieurs variables `ignore_`*quelquechose* : leur contenu doit être une expression régulière qui indique un motif à ignorer.
  Par exemple si l'on définit `ignore_spaces=r'\s+'` les suites de caractères « espace » seront ignorées.
  On peut définir autant de variables `ignore_`*qqchose* que l'on veut.

Des méthodes déclenchées par une regExp
Un exemple :

```
@_(r'\n')
def increment_newline(self, t):
    self.lineno += 1
```

La méthode sera déclenchée à la rencontre d'un caractère de fin de ligne. Celui-ci ne sera pas considéré comme un caractère erroné. De plus la méthode incrémente la variable `lineno` de façon à ce qu'elle contienne le nombre de lignes rencontrées.
Notez bien que l'expression régulière qui déclenche la méthode figure juste avant la déclaration de méthode entre `@_(  )` (un « décorateur python »)

2.4.3 Définir des variables dans la classe

Dans la classe de spécification, on peut défiunir ses propres variables mais avec un nom commençant par un underscore. Par exemple pour définir une expression régulière de façon modulaire

```
_entier_base_10=r'[1-9][0-9]*'
_entier_base_8=r'0[0-7]*
ENTIER =rf'{_entier_base_8}|{_entier_base_10}'
```

3 Exercices

Un fichier python vous est fourni. Copiez le dans votre espace personnel.

Exercice 1 :
Commencez par examiner le contenu du fichier `exemple_simple.py` puis testez-le.
Q 1 .
En python, un entier écrit en décimal est formé de chiffres décimaux. Il est écrit en octal quand il est formé de chiffres entre 0 et 7 précédés de `0o` ou `0O`; Il est écrit en hexadécimal quand il est formé de chiffres décimaux et de lettres entre A et F (minuscules ou majuscules) précédés par `0x` ou `0X`. Les underscores sont autorisés mais pas au début du nombre ou en fin de nombre ni à l'intérieur du préfixe et sans se succéder.
Modifiez l'expression régulière du token `ENTIER` pour adopter la syntaxe Python. Attention à la conversion en `int`. Pour les chaînes représentant un nombre non décimal, il faut préciser la base. Ex : `int(s,8)` pour une chaîne « en octal ».
Q 2 .
Autorisez la présence de commentaires (ils seront ignorés). Autorisez les 3 types de commentaires :

1. commentaire commençant par `#` et se terminant en fin de ligne
2. commentaire commençant par `{` et se terminant par `}`. Ces commentaires peuvent s'étendre sur plusieurs lignes. Par contre on ne peut les emboîter (le commentaire s'arrête au premier `}` rencontré).
3. commentaire commençant par `<!-` et se terminant par `->`. Mêmes règles que le type précédent (multiligne et sans emboîtement)

Q 3 . Autorisez pour le token `OP2` les chaînes `add sub mul div` synonymes de `+ - * /`
Modifiez la valeur associée aux tokens `OP2`. La valeur associée sera la fonction correspondant à l'opérateur. En python les fonctions prédéfinies correspondant aux opérateurs sont `operator.add` `operator.suboperator.mul` `operator.floordiv` (il faut importer le paquetage `operator`)