Interpréteur Serpent

Documentation

Version 1

Auteur

Yanis Nebbaki (approfondissement, code de base HowCode)

Réalisé dans le cadre du cours Interprétation et Compilation d'une L3 Informatique au sein de l'Institut d'Enseignement à Distance de l'Université Paris 8.

Description

Le langage *Serpent* est un langage basique personnalisé ayant une syntaxe très proche du français. Il se spécialise dans les calculs trigonométriques pour la classe de troisième au collège. Il incorpore également quelques autres fonctionnalités simples. Il offre une interface en ligne de commande pour saisir les instructions. L'interpréteur est écrit en Python et utilise la bibliothèque *SLY* pour l'analyse lexicale et syntaxique.
La structure du code provient de *HowCode* (https://github.com/howCodeORG/2018-Programming-Language-Series)

Structure générale du code

Le programme comporte 3 classes et une boucle principale.
La première classe est BasicLexer. Elle hérite de la classe Lexer de sly (https://sly.readthedocs.io/en/latest/sly.html#writing-a-lexer). On y retrouve les tokens ainsi que les expressions régulières qui leur sont associées. Suite à ça se trouve la gestion des nombres avec la conversion en flottant. La règle suivante définit les lignes commençant par # comme des commentaires. Enfin on trouve la gestion des lignes et tabulations du lexer.

```
class BasicLexer(Lexer):
    tokens = { NAME, STRING, SI, FAIRE, SINON, POUR, FONC, JSQ, ARROW, EQEQ, ECRIS, EXEMPLE, RAPPEL, SOH, OSH, HOS, SIN, AS
    ignore = '\t '  # ignore les tabulations et les espaces

    literals = { '=', '+', '-', '/', '*', '(', ')', ',', ';' }

    # Define tokens
    #S : sinus
    #O : opposé
    #H : hypoténuse
    #C : cosinus
    #T : tangente
    #SIN : sinus ASIN : arcsinus | COS : cosinus ACOS : arcsinus | TAN : tangente ATAN : arctangente
    # Token pour les calculs avec le sinus, 1ère lettre => ce qu'on veut calculer, 2è et 3è lettre les opérandes
    SOH = r'SOH'
    OSH = r'OSH'
    HOS = r'HOS'
    SIN = r'SIN'      # SIN se trouve au dessus du token SI pour éviter les conflits
    ASIN = r'ASIN'
    # Token pour les calculs avec le cosinus, 1ère lettre => ce qu'on veut calculer, 2è et 3è lettre les opérandes
    CAH = r'CAH'
    ACH= r'ACH'
    HAC = r'HAC'
    COS = r'COS'
    ACOS = r'ACOS'
    # Token pour les calculs avec la tangente, 1ère lettre => ce qu'on veut calculer, 2è et 3è lettre les opérandes
```

Class BasicLexer(Lexer)

La seconde classe est *BasicParser*, elle hérite de classe *Parser* de *sly* et des tokens de la classe précédente *BasicLexer*.
Cette classe *BasicParser* comporte les règles de grammaire pour notre langage (boucle *POUR*, test *SI*, fonction, opérations mathématiques etc…). Chaque règle va créer et renvoyer une structure de données comportant toutes les informations relatives à la règle. Par exemple pour une affectation de variable, on a :

```
@_('NAME "=" expr')
def var_assign(self, p):
    return ('var_assign', p.NAME, p.expr)
```

Règles pour une affectation de variable avec des nombres

Dans le *return* se trouve :

- *var_assign* qui indique le type d'instruction
- *p.NAME* le nom de la variable obtenu à partir du token *NAME*
- *p.expr* extrait à partir du token *expr,* le contenu de la variable

On retrouve également la priorité des opérateurs ainsi que leur associativité en cas d'ambiguïté.

```
precedence = (              # priorité et associativté des opérateurs
    ('left', '+', '-'),
    ('left', '*', '/'),
    ('right', 'UMINUS'),
    )
```

Priorité et associativité des opérateurs

La dernière classe est *BasicExecute*. Elle comporte le parcours de l'arbre syntaxique construit précédemment et exécute les instructions correspondantes.

```
if node[0] == 'add':                                                    # noe
    return self.walkTree(node[1]) + self.walkTree(node[2])
elif node[0] == 'sub':
    return self.walkTree(node[1]) - self.walkTree(node[2])
elif node[0] == 'mul':
    return self.walkTree(node[1]) * self.walkTree(node[2])
elif node[0] == 'div':
    return self.walkTree(node[1]) / self.walkTree(node[2])
```

Noeud pour les opérations mathématiques

(Voir les commentaires du code pour les explications de chaque noeud)

Finalement, la boucle principale *__main__* crée une instance du lexer et du parser puis entre dans une boucle où l'utilisateur peut saisir du code à interpréter. Le lexer *tokenize* le code, le parser génère un arbre syntaxique et enfin l'arbre est parcouru et les instructions sont exécutées.

Ajout de nouvelle fonctionnalité

Pour ajouter une nouvelle fonctionnalité, les étapes suivantes sont à prendre en compte :

- Identifier et créer les tokens nécessaires, si nécessaire modifier les tokens existants (par exemple le token *NOMBRE_FLOTTANT* si il y a besoin de nombres négatifs)
- Créer la règle adéquate en renvoyant les données nécessaires et le nom du noeud
- Ajouter dans le parcours de l'arbre les actions à faire pour ce nouveau noeud