amU Faculté des sciences Aix Marseille Université

Année 2024-25

Domaine Sciences et Technologies
Licence mathématiques-informatique
Compilation : TP 4
Codes UE : SIN6U64L
TP 4 : interpréteur Python

1 Introduction

Dans ce TP, on va implémenter un Interpréteur Python qui permet d'exécuter des scripts Python.

1.1 Fork du projet

Comme pour le premier TP, vous allez devoir forker un dépôt git pour ensuite le cloner en local. Référez-vous aux consignes du premier TP.

Le dépôt à forker se trouve à l'adresse suivante :

https://etulab.univ-amu.fr/alaboure/interpreter-python

1.2 Exécution du projet

Pour compiler et exécuter votre programme, il faut passer par l'onglet *gradle* à droite.

- pour les tests, il faut cliquer deux fois sur `python-interpreter` -> `Tasks` -> `verification` -> `test`.
- pour l'exécution du code, il faut cliquer deux fois sur `python-interpreter` -> `Tasks` -> `application` -> `run` pour exécuter la méthode `main` présente dans la classe `python.interpreter.MainPython`.

2 Présentation interpréteur Python

2.1 Éléments déjà gérés

Le dépôt qui vous est donné contient un interpréteur Python rudimentaire qui gère les éléments suivant du langage Python :

- **Types** : trois types de données sont gérés
  - Entiers
  - Booléens
  - None
- **Structures de contrôle** : une seule structure de contrôle est gérée
  - while
- **Fonction** : une seule fonction peut être appelée
  - `print` limitée à un seul argument
- **Opérateurs** : trois opérateurs sont gérés
  - L'opérateur binaire `+`
  - l'opérateur binaire `<`
  - L'opérateur unaire `not`

— **Gestion des variables** : possibilité d'affecter des valeurs à de variables

2.2 Fonctionnement de l'interpréteur

L'interpréteur fonctionne via un visiteur d'un arbre de dérivation du script. La grammaire est définie par le fichier `PythonParser.g4` qui utilise les *tokens* définis dans le fichier `PythonLexer.g4`. Le visiteur est défini dans la classe `python.interpreter.PythonInterpreterVisitor`. Toutes les méthodes de visite du visiteur renvoient des objets de type `PythonValue` qui correspondent à des valeurs Python gérées par l'interpréteur (de type entier, booléen ou *none*). Le type `PythonValue` est une interface qui définit les méthodes pour les opérations possibles sur les valeurs. Pour les expressions comme `10 + 3`, le visiteur renvoie une valeur correspondant à l'évaluation et donc un objet (de type `PythonInteger`) correspondant à l'entier `13` sur cet exemple. Pour une expression comme `12 < 14`, la valeur renvoyée est un objet (de type `PythonBoolean`) correspondant au booléen `True` sur cet exemple. Pour d'autres instructions ne correspondant pas à une valeur (comme un `while`), la valeur renvoyée est un objet correspondant à `None` (de type `PythonNone`). En plus de calculer la valeur à renvoyer, l'interpréteur exécute le code correspondant. Par exemple, lors de l'évaluation d'un nœud de l'arbre correspondant à une dérivation d'un `while`, l'interpréteur exécute le corps de la boucle `while` tant que la condition s'évalue à `True`.

2.3 Test de l'interpréteur

L'interpréteur est testé via la classe de test `PythonInterpreterTest` présente dans le répertoire `src/test/java/python/interpreter`. Cette classe lance un test pour chaque fichier de script Python (avec l'extension de fichier `.py`) présent dans le répertoire `src/test/ressources` du dépôt. Pour chacun de ces tests et donc chacun de ces scripts, la sortie (affichage via les appels de `print`) de l'exécution du script via le visiteur `PythonInterpreterVisitor` est comparée avec la sortie produite par l'exécution du même script avec l'interpréteur `python3` installé. Les tests comparent donc les sorties avec l'interpréteur officiel de Python.

3 Amélioration de l'interpréteur

Une fois que vous êtes familiarisé avec le code de base de l'interpréteur, votre objectif sera d'ajouter des nouvelles fonctionnalités gérées par `PythonInterpreterVisitor` en faisant en sorte que celui-ci ait le même comportement que l'interpréteur officiel de Python (même sortie standard sur des scripts). Pour cela, vous ajouterez des fichiers de script python (avec l'extension de fichier `.py`) dans le répertoire `src/test/ressources` du dépôt.

3.1 Ajouts d'opérateurs sur les entiers

Modifiez les fichiers `PythonParser.g4`, `PythonValue` (et les classes implémentant cette interface) et `PythonInterpreterVisitor.java` afin de gérer les opérateurs suivants :

— L'opérateur binaire `*`
— L'opérateur binaire `-`
— L'opérateur binaire `//`
— L'opérateur unaire `-`

Testez votre code en rajoutant un fichier `int_operator.py` dans le répertoire `src/test/ressources` avec le contenu suivant :

```
print(2 * 2)
print(23 - 12)
print(16 // 3)
print(-3)
```

Relancez les tests. Normalement, un nouveau test correspondant à ce fichier est lancé. Le test exécute le script avec votre interpréteur et l'interpréteur officiel et compare la sortie produite par les deux exécutions.

3.2 Ajouts d'opérateurs de comparaison

Modifiez les fichiers `PythonParser.g4`, `PythonValue` (et les classes implémentant cette interface) et `PythonInterpreterVisitor.java` afin de gérer les opérateurs de comparaison suivants :

— L'opérateur binaire `<=`
— L'opérateur binaire `>=`
— L'opérateur binaire `>`
— L'opérateur binaire `==`

Testez votre code en rajoutant un fichier `comparators.py` dans le répertoire `src/test/ressources` avec le contenu suivant :

```
print(2 <= 2)
print(1 <= 21)
print(3 <= 2)
print(2 >= 2)
print(1 >= 21)
print(3 >= 2)
print(3 > 2)
print(3 > 3)
print(3 == 2)
print(3 == 3)
```

3.3 Ajouts d'opérateurs sur les booléens

Modifiez les fichiers `PythonParser.g4`, `PyThonValue` (et les classes implémentant) et `PythonInterpreterVisitor.java` afin de gérer les opérateurs de comparaison suivants :

— L'opérateur binaire `and`
— L'opérateur binaire `or`

Testez votre code en rajoutant un fichier `boolean_operations.py` dans le répertoire `src/test/ressources` avec le contenu suivant :

```
print(True and True)
print(False or True)
print(1 < 2 and 2 < 1)
print(11 < 2 or 2 < 1)
```

3.4 Ajouts d'instructions pour les structures de contrôle

Modifiez les fichiers `PythonParser.g4` et `PythonInterpreterVisitor.java` afin de gérer instructions suivantes :

- Instruction `if` sans `else`
- Instruction `if` avec `else`
- Instruction `if` avec `elif`

Testez votre code en rajoutant un fichier `if.py` dans le répertoire `src/test/ressources` avec le contenu suivant :

```
a = 2

if(1 < a):
    print(True)
else:
    print(False)

if(a < 1):
    print(True)
else:
    print(False)
```

3.5 Déclaration de fonction

Pour cette partie, on vous demande d'ajouter le traitement des fonctions. Ces dernières devront être stockées dans la table de fonctions définie par la classe `function.FunctionTable`.

Une fonction devra correspondre à un objet de type `NonNativeFunction` (à définir vous-même) implémentant `PythonFunction`. Vous devez donc créer une classe implémentant l'interface `PythonFunction` qui contiendra :

- le nom de la fonction ;
- la liste des noms des paramètres de la fonction ;
- le nœud de l'arbre de dérivation correspondant aux corps (instructions) de la fonction

Il vous faut donc modifier la grammaire pour prendre en compte la définition de fonction en python avec le mot-clé `def` correspondant au *token* `DEF`. Lorsqu'une fonction est déclarée, il faudra donc construire un objet de type `NonNativeFunction`.

La partie complexe de la gestion d'un appel de fonction est la création de l'environnement local de fonction qui contient les valeurs des paramètres et des variables de la fonction. Pour gérer cela vous devez modifier la classe

`VariableTable` pour rajouter les éléments suivants :

- un attribut `Stack<Map<String, PythonValue>> localVariables` qui contient une pile contenant les associations entre nom et valeurs des variables et paramètres des appels de fonctions. La tête de la pile correspondra à la fonction en cours d'exécution (pile vide si on n'est pas en train d'exécuter une fonction) ;
- une méthode `void pushLocalEnvironment()` qui crée un nouvel environnement local (de type `Map<String, PythonValue>`) et l'empile sur l'attribut `localVariables` (cette méthode sera appelé au début de l'exécution de chaque appel de fonction)
- une méthode `void popLocalEnvironment()` qui dépile un élément de `localVariables` (cette méthode sera appelé à la fin de chaque appel de fonction).

Vous devez aussi faire les modifications suivantes :

- changez la méthode `assignVariable` de sorte qu'elle affecte la valeur à la tête de la pile si celle-ci n'est pas vide ;
- changez la méthode `getVariableValue`de sorte à ce qu'elle renvoie la valeur associée à la tête de la pile si elle existe ou bien la valeur associée à la `globalVariables` dans le cas contraire.

Testez votre code en rajoutant un fichier `function_def_without_return.py` dans le répertoire `src/test/ressources` avec le contenu suivant :

```
a = 10

def print_sum(b):
    print(a+b)

print_sum(15)
```

Dont la sortie doit être :

```
25
```

3.6 Ajout d'instructions

Modifiez les fichiers `PythonParser.g4` et `PythonInterpreterVisitor.java` afin de gérer les instructions suivantes :

- Instruction `return`
- Instruction `break`
- Instruction `continue`

Pour la gestion de ces instructions, on a besoin de pouvoir sortir de l'exécution du corps d'une boucle (`break` et `continue`) ou bien d'une fonction (`return`). Pour cela, le plus simple est de lever avec le mot-clé `throw` une exception héritant de `RuntimeException`. L'utilisation du mot-clé `throw` interrompt l'exécution et pourra être capturé au bon endroit (au niveau de l'exécution de la boucle ou de la fonction) grâce au mot-clé `catch`. Dans le cas d'un `return`, votre exception devra contenir la valeur à retourner par la fonction.

Testez votre code en rajoutant un fichier `function_def_with_return.py` dans le répertoire `src/test/ressources` avec le contenu suivant :

```
def fact(n):
    if n <= 1:
        return 1
    return n * fact(n-1)

print(fact(10))
```

Dont la sortie doit être

```
3628800
```

3.7 Ajouts supplémentaires

— ajout de l'opérateur `%`
— ajout du type float
— ajout de l'opérateur `/`
— ajout du type list
— ajout du type range
— ajout du type tuple
— ajout de l'opérateur `in`
— ajout de l'opérateur `**`
— ajout de la méthode `len`
— ajout du type `str`
— ajout du type `dict`
— ajout de l'instruction `for`
— ajout de la définition de classes avec le mot-clé `class`