TP compilateurs 4 : Partie arrière

Dans ce quatrième et dernier TP sur les compilateurs, nous allons nous intéresser à la partie arrière de notre mini-compilateur. Nous le ferons de trois manières différentes :

1. Un interprète récursif
2. Un interprète itératif
3. Un compilateur vers du bytecode

Nous aurons ainsi survolé chacune des étapes importantes de la création d'un interprète ou compilateur complet [1].

1 Interprète récursif

Il n'est pas très compliqué d'écrire un interprète récursif simple. En gros, il suffit de parcourir récursivement l'arbre syntaxique et d'exécuter les choses au fur et à mesure.

Dans un fichier `recInterpreter.py`, on va commencer par les importations d'usage et préparer quelques variables (que vous connaissez bien) :

```
import AST
from AST import addToClass
from functools import reduce
operations = {
    '+' : lambda x,y: x+y,
    '-' : lambda x,y: x-y,
    '*' : lambda x,y: x*y,
    '/' : lambda x,y: x/y,
}

vars ={}
```

On va ensuite mettre en place le programme principal, qui effectue l'analyse du code, récupère l'AST et lance l'interprétation récursive :

```
if __name__ == "__main__":
    from parser5 import parse
    import sys
    prog = open(sys.argv[1]).read()
    ast = parse(prog)

    ast.execute()
```

1. À noter cependant que notre analyse sémantique reste élémentaire et que nous n'aurons implémenté aucune optimisation de code…

Il ne nous reste plus qu'à implémenter la méthode `execute()` pour chacun des types de noeuds de notre AST. Pour garder les choses bien séparées, nous le ferons à nouveau en utilisant le décorateur `addToClass`.

Par exemple, un noeud de type `Program` exécute simplement ses enfants dans l'ordre :

```
@addToClass(AST.ProgramNode)
def execute(self):
    for c in self.children:
        c.execute()
```

Pour les `TokenNode` on se heurte à un problème : notre AST ne comporte qu'un seul type de noeud pour les nombres et les identificateurs [2]. Nous avons donc deux solutions : changer l'AST ou bricoler avec celui que nous avons. Le temps passe, nous choisirons donc la deuxième. `;-)`

```
@addToClass(AST.TokenNode)
def execute(self):
    if isinstance(self.tok, str):
        try:
            return vars[self.tok]
        except KeyError:
            print ("*** Error: variable %s undefined!" % self.tok)
    return self.tok
```

Pour les opérateurs, la seule difficulté est de tenir compte des opérateurs unaires (du genre `-2`). Nous proposons ici une implémentation qui fait un usage intensif de techniques fonctionnelles :

```
@addToClass(AST.OpNode)
def execute(self):
    args = [c.execute() for c in self.children]
    if len(args) == 1:
        args.insert(0,0)
    return reduce(operations[self.op], args)
```

Pour l'assignation et le `print`, c'est un jeu d'enfant :

```
@addToClass(AST.AssignNode)
def execute(self):
    vars[self.children[0].tok] = self.children[1].execute()

@addToClass(AST.PrintNode)
def execute(self):
    print (self.children[0].execute())
```

... il ne reste plus qu'à s'occuper du while, mais ça je vous laisse le faire par vous même.

Et voilà ! ça n'était pas bien compliqué !

Maintenant, un programme du genre

```
a=0;
while (a-10) {
    print a;
    a = a+1
}
```

devrait s'exécuter sans le moindre problème. Même les boucles imbriquées devraient marcher. Essayez !

2. Une conception propre nous aurait sans doute amené à au moins *trois* types de noeuds différents : un pour représenter un nombre, un pour représenter un identificateur lorsqu'on veut récupérer sa valeur et un pour représenter un identificateur lorsqu'on veut y *stocker* quelque chose. Ainsi, l'AST de l'expression `a=b+2` aurait *trois* feuilles de types différents.

2 Interprète itératif

Dans `itInterpreter.py`, commencez par quelques mises en places :

```
import AST

operations = {
    '+' : lambda x,y: x+y,
    '-' : lambda x,y: x-y,
    '*' : lambda x,y: x*y,
    '/' : lambda x,y: x/y,
}

stack = []
vars = {}
```

De plus, pour pallier aux faiblesses de notre AST, une petite fonction auxiliaire :

```
def valueOfToken(t):
    if isinstance(t, str):
        try:
            return vars[t]
        except KeyError:
            print ("*** Error: variable %s undefined!" % t)
    return t
```

Pendant que nous y sommes, allons-y pour le programme principal. Bien entendu, nous devrons coudre l'AST avant d'appeler notre interprète itératif :

```
if __name__ == "__main__":
    from parser5 import parse
    from threader import thread
    import sys
    prog = open(sys.argv[1]).read()
    ast = parse(prog)
    entry = thread(ast)

    execute(entry)
```

Il ne nous reste plus qu'à mettre en place la fonction d'interprétation proprement dite ; celle-ci parcourra l'AST le long des "fils" de la couture en tenant à jour une pile d'exécution. Voici une première esquisse :

```
def execute(node):
    while node:
        if node.__class__ in [AST.EntryNode, AST.ProgramNode]:
            pass
        elif node.__class__ == AST.TokenNode:
            stack.append(node.tok)
        elif node.__class__ == AST.PrintNode:
            val = stack.pop()
            print valueOfToken(val)
        if node.next:
            node = node.next[0]
        else:
            node = None
```

Ceci nous permettrait déjà d'exécuter des codes qui ne contiendraient que des print et des nombres.

Ajoutons maintenant les opérations ; rien de spécial si ce n'est que les arguments sont empilés sur la pile en sens inverse ; à la suite des `elif` du code précédent, ajoutez :

```
        elif node.__class__ == AST.OpNode:
            arg2 = valueOfToken(stack.pop())
            if node.nbargs == 2:
                arg1 = valueOfToken(stack.pop())
            else:
                arg1 = 0
            stack.append(operations[node.op](arg1,arg2))
```

Pour l'assignation, la seule subtilité est de ne pas prendre la *valeur* de la variable à assigner, mais de garder son *nom* :

```
        elif node.__class__ == AST.AssignNode:
            val = valueOfToken(stack.pop())
            name = stack.pop()
            vars[name] = val
```

Il nous reste le `while`, qui lui va modifier le flux d'exécution (d'où le `continue` qui permet de ne pas passer par l'"incrémentation" de `current`) :

```
        elif node.__class__ == AST.WhileNode:
            cond = valueOfToken(stack.pop())
            if cond:
                node = node.next[0]
            else:
                node = node.next[1]
            continue
```

... et voilà, c'est aussi simple que ça. Expressions arithmétiques, variables, boucles, même imbriquées, `print`, tout cela devrait marcher. Testez !

3 Compilateur

Pour le compilateur, pour nous simplifier la vie, nous compilerons pour une machine virtuelle ultra-simple, conçue pour l'occasion.

Récupérez donc sur le serveur `svm.py` et jetez un coup d'oeil au commentaire en début de fichier.

Vous l'aurez donc compris, il suffit maintenant de transformer un programme tel que celui donné en exemple en page 2 en une sortie du style [3]

```
PUSHC 0.0
SET a
JMP cond1
body1: PUSHV a
PRINT
PUSHV a
PUSHC 1.0
ADD
SET a
cond1: PUSHV a
PUSHC 10.0
SUB
JINZ body1
```

3. La traduction du `while` est celle donnée en cours.

Le programme principal aura une allure très classique :

```
if __name__ == "__main__":
    from parser5 import parse
    import sys, os
    prog = open(sys.argv[1]).read()
    ast = parse(prog)
    compiled = ast.compile()
    name = os.path.splitext(sys.argv[1])[0]+'.vm'
    outfile = open(name, 'w')
    outfile.write(compiled)
    outfile.close()
    print ("Wrote output to", name)
```

Il ne reste donc plus qu'à écrire une méthode `compile` qui s'applique à la racine de l'arbre et retourne le programme compilé sous la forme d'une chaîne de caractères.

Mais maintenant c'est à vous de jouer : écrivez le code nécessaire et testez le "bytecode" obtenu avec `svm.py` (N'oubliez pas de tester que les boucles imbriquées fonctionnent !).

4 Conclusion

Dans cette série de 4 TP's, nous avons rapidement parcouru les grandes étapes nécessaires pour réaliser un interpréteur ou un compilateur simple. De là à réaliser un compilateur complet pour un langage utile, le chemin est encore long, mais vous devriez avoir tous les éléments en mains pour le parcourir. Alors maintenant... à vous de jouer !

Post scriptum (Pour ceux que ça intéresse)

Dans une conception orientée objet, il arrive souvent qu'une fonctionnalité doive être répartie entre plusieurs classes. Dans ces TP's, c'était le cas par exemple de la couture ou de l'interprétation récursive.

Dans un certain sens, ces fonctionnalités sont "orthogonales" – ou "transversales" – par rapport au découpage du problème en classes distinctes[a]. Traditionnellement, le code de ces fonctionnalités sera donc réparti à différents endroits du code source, dans les différentes classes.

Depuis quelques années, différentes recherches étudient la possibilité de regrouper le code de ces "fonctionnalités transversales" pour en faciliter la lecture ou pour pouvoir inclure/exclure le code facilement (comme nous l'avons fait avec la couture par exemple).

Ces différentes approches tentent de développer un nouveau paradigme de programmation qui viendrait compléter la programmation orientée objet traditionnelle avec une couche "transversale". Dans ce contexte, les fonctionnalités transversales sont appelées *aspects* et l'approche correspondante est la *Programmation Orientée Aspects* (*Aspect Oriented Programming*, *AOP*).

On pourrait voir le décorateur `addToClass` que nous avons utilisé comme une forme élémentaire de programmation orientée aspects.

a. En anglais, on les appelle des "crosscutting concerns".