TP compilateurs 2 : Perfectionnement de `PLY`

Dans ce TP, nous allons reprendre et développer l'esquisse de mini-compilateur du TP compilateurs 1.

Dans un premier temps, nous allons enrichir le langage que nos analyseurs peuvent reconnaître, puis nous nous attaquerons à la contruction d'un arbre syntaxique.

1 Enrichissement du langage

1.1 Plusieurs expressions

Jusqu'à maintenant, notre "compilateur" est seulement capable d'analyser des fichiers qui ne contiennent qu'une expression.

Ouvrez des nouvelles versions de vos analyseurs sous les noms `lex4.py` et `parser3.py` et modifiez-les pour qu'ils puissent reconnaître plusieurs expressions de suite, séparées par des points-virgules.

Chaque expression sera évaluée mais les résultats "intermédiaires" seront (pour l'instant) perdus et seul le dernier résultat sera écrit à la console.

Par exemple, le fichier

```
12*3+4;
1+1; 2*3
```

sera analysé correctement mais seul le nombre 6 sera écrit à la console.

- Commencez par enrichir votre analyseur lexical pour qu'il reconnaisse le point-virgule
- Ensuite, ajoutez une règle *au début*[1] de votre analyseur syntaxique lui disant qu'un programme est une suite d'expressions séparées par des virgules. PLY ne supportant pas les grammaires EBNF, on devra donc utiliser la récursivité. Cela donnera donc quelque chose comme

```
programme  →  expression
           |  expression ';' programme
```

1.2 Utilisation de variables

Le but maintenant est d'ajouter la gestion de variables, c'est-à-dire de pouvoir évaluer un fichier du genre

1. Par défaut, le premier symbole défini dans votre grammaire en sera le symbole de départ. C'est pourquoi vous devrez ajouter votre règle au début.

```
toto = 12*3+4;
a = toto+1; a*2
```

Notre “compilateur” devra alors afficher 82.

- Complétez `lex4.py` pour qu'il puisse reconnaître des identificateurs et le signe '='. Nos identificateurs seront des suites de lettres (majuscules ou minuscules), de chiffres et de soulignements ne commençant pas par un chiffre.
- Modifiez la grammaire de `parser3.py` pour qu'elle définisse les non-terminaux suivants :
  - Un programme est une suite de *statements*
  - Un statement est soit une assignation, soit une expression
  - Une assignation est de la forme
    ```
    identifier = expression
    ```
    Lors de l'analyse d'une assignation, on stockera dans un dictionnaire nommé `vars` une entrée dont la clé est le nom de l'indentificateur et la valeur celle de l'expression.
- Enfin, complétez les règles définissant une expression pour qu'une expression puisse contenir des variables.

2 Construction de l'arbre

2.1 Motivation

Avec plusieurs expressions et affectation de variables, notre langage commence à ressembler à un mini-langage de programmation. Mais l'approche actuelle, avec évaluation des expressions au fur et à mesure de leur analyse, nous empêche d'aller plus loin.

Imaginons par exemple que nous voulions étendre notre langage pour inclure des programmes du type

```
toto = 0;
repeat 10 {
    toto = toto + 1
};
print toto
```

Il est impossible dans ce cas d'évaluer la valeur de l'expression de la ligne 3 au moment de l'analyse. On est obligé de commencer par analyser tout le programme, *puis* de l'exécuter.

De manière générale, toutes les structures conditionnelles ou répétitives présentent ce “problème”.

Pour pouvoir aller plus loin, nous serons donc obligés de compliquer un peu la structure de notre compilateur :

- L'analyse syntaxique se contentera de construire un arbre d'analyse...
- ... et c'est les phases suivantes du compilateur qui s'occuperont des autres traitements, notamment de l'exécution ou de la génération de code.

2.2 Préparatifs

Construire un arbre syntaxique, c'est bien ; pouvoir voir ce qu'on a construit, c'est encore mieux !

Pour pouvoir visualiser facilement les arbres de plus en plus complexes auxquels nous serons confrontés, nous allons utiliser

- `Graphviz`[2], qui est un logiciel de génération de représentation graphiques de graphes en tout genres ; `Graphviz` excelle lorsqu'il s'agit de trouver un placement adéquat des noeuds d'un graphe complexe pour obtenir une représentation lisible ;

2. `http://www.graphviz.org/`

– `PyDot` [3], qui est un module python permettant d'utiliser `Graphviz` ;

Récupérez et installez ces deux logiciels [4].

Enfin, pour vous gagner du temps, une partie du code permettant de construire l'arbre syntaxique et de générer ses représentations graphiques vous est fournie. Récupérez sur le serveur le fichier `AST.py` et copiez-le dans votre répertoire de travail (ou n'importe où d'autre dans votre `PYTHONPATH`)

C'est fait ? alors vous êtes maintenant prêt(e) pour continuer ce TP.

2.3 Construction de l'arbre

Lors de chaque réduction à l'aide d'une règle, PLY exécute le code correspondant. Jusqu'à maintenant, nous nous sommes contentés de faire des calculs directement. Par exemple :

```
def p_expression_op(p):
    '''expression : expression ADD_OP expression
                  | expression MUL_OP expression'''
    p[0] = operations[p[2]](p[1],p[3])
```

Nous allons maintenant remplacer ce code par du code de contruction d'arbre. Pour gagner du temps, les classes permettant de construire (et de représenter graphiquement) l'arbre vous sont fournies. Il suffit donc des les importer :

```
import AST
```

On peut maintenant modifier l'extrait de code ci-dessus pour contruire un "bout d'arbre" à la place :

```
def p_expression_op(p):
    '''expression : expression ADD_OP expression
                  | expression MUL_OP expression'''
    p[0] = AST.OpNode(p[2], [p[1], p[3]])
```

Ainsi, au lieu de contenir le résultat du calcul, `p[0]` contiendra un arbre dont la racine est l'opération à effectuer et les enfants les sous-arbres correspondant aux sous-expressions droite et gauche.

Comme l'analyse LALR(1) est une analyse ascendante, cet arbre deviendra un sous-arbre au cours de l'analyse du contexte de cette expression.

Il est à noter que nous cherchons à construire un *arbre abstrait* et non un arbre d'analyse complet. Ainsi, par exemple pour les expressions parenthésées, on pourra se satisfaire de "remonter" le sous-arbre sans inclure de noeud "expression" intermédiaire.

```
def p_expression_paren(p):
    '''expression : '(' expression ')' '''
    p[0] = p[2]
```

Il ne vous reste plus qu'à compléter la construction de l'arbre à l'aide des classes de `AST.py` :

AST.OpNode(op,children) comme vu ci-dessus, un noeud représentant une opération arithmétique.

AST.TokenNode(token) noeud correspondant à un lexème. Prend en paramètre la valeur du lexème (chaîne de caractère, nombre, ...)

AST.AssignNode(children) noeud correspondant à une assignation ; prend en paramètre une *liste* dont le premier élément correspond à l'identificateur à assigner et le second à l'expression assignée.

Exemple : `a = 2` ⇝ `AST.AssignNode([aTokenNode, anExprNode])`

3. `http://code.google.com/p/pydot/`

4. Officiellement, `pydot` nécessite `pyparsing`. Dans les faits nous n'en aurons pas besoin. Vous pouvez donc vous passez de l'installer si vous le désirez, au prix d'un *warning* chaque fois que vous utiliserez `pydot`.

ProgramNode(children) noeud racine d'un programme. Prend en paramètre une liste d'enfants.

Par exemple, pour analyser un programme, vous devriez avoir des règles du genre

```
def p_programme_statement(p):
    ''' programme : statement '''
    p[0] = AST.ProgramNode(p[1])

def p_programme_recursive(p):
    ''' programme : statement ';' programme '''
    p[0] = AST.ProgramNode([p[1]]+p[3].children)
```

Exercice À vous de jouer maintenant ! Dans le fichier `parser4.py`, complétez le code de construction de l'arbre pour les règles encore non traitées.

Une fois ceci fait, le code "`main`" déjà existant marchera toujours, mais au lieu de vous afficher le résultat, il vous affichera une saisissante représentation en "ASCII-art" de votre arbre :

```
if __name__ == "__main__":
    import sys

    prog = open(sys.argv[1]).read()
    result = yacc.parse(prog)
    print (result)
```

Ainsi, un fichier d'entrée comme

```
toto = 12*-3+4;
a = toto+1; a*2
```

devrait donner

```
Program
| =
| | 'toto'
| | + (2)
| | | * (2)
| | | | 12.0
| | | | - (1)
| | | | | 3.0
| | | 4.0
| =
| | 'a'
| | + (2)
| | | 'toto'
| | | 1.0
| * (2)
| | 'a'
| | 2.0
```

Pour un arbre simple, ceci est déjà bien... mais si l'arbre se complique, cela devient assez illisible !

C'est ici qu'entrent en jeu `Graphviz` et `pydot`, par l'intermédiaire du module `AST`.

On va donc ajouter dans notre "`main`" les lignes suivantes :

```
    import os
    graph = result.makegraphicaltree()
    name = os.path.splitext(sys.argv[1])[0]+'-ast.pdf'
    graph.write_pdf(name)
    print ("wrote ast to", name)
```

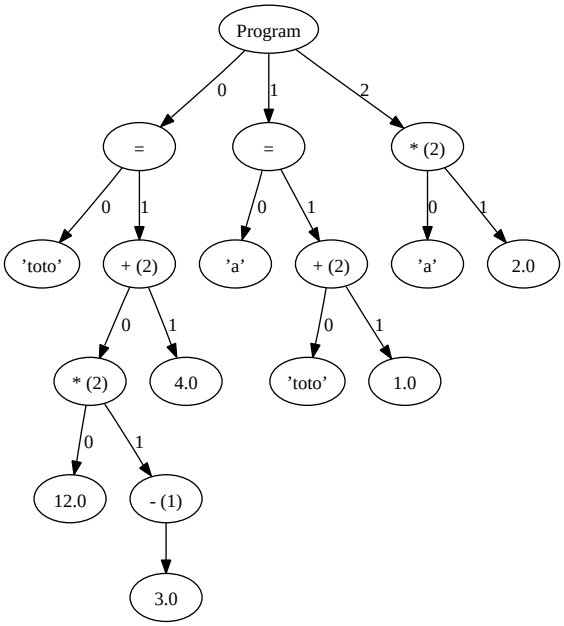

FIGURE 1: Un arbre abstrait

Si votre fichier d'entrée s'appelle `input.txt`, vous devriez maintenant avoir, dans le même répertoire, un `input-ast.pdf` [5] qui contient plus ou moins la même chose que la figure 1.

Remarque Si plutôt que de générer bien sagement l'arbre attendu, l'exécution se termine sur une erreur du type

```
pydot.InvocationException: GraphViz's executable
""C:\Program Files\Graphviz2.16 \Bin\dot.exe""
is not a file or doesn't exist
```

c'est que vous êtes confronté à un bug de certaines versions de `pydot` sous Windows. Dans ce cas, dé-commentez la ligne 63 de `AST.py` et adaptez-là à votre installation :

```
dot.set_graphviz_executables({'dot': r'C:\Program Files\Graphviz2 ↩
    .16\bin\dot.exe'})
```

3 Conclusion

Ce TP nous a permis d'ajouter à notre "mini-compilateur" :
- La possibilité d'avoir plusieurs expressions
- la gestion de variables
- la construction d'un arbre syntaxique abstrait.

Pour avoir un "vrai" langage, il faudrait encore des structures de contrôle (`for`, `if`, ...) et pour avoir un "vrai" compilateur, il faudrait traiter l'arbre syntaxique pour produire du code.

Ces tâches – ainsi que quelques autres – feront l'objet des prochains TP's.

5. Vous pouvez d'ailleurs à votre convenance remplacer le pdf par n'importe quel autre format contenu dans l'attribut `formats` des objets de type `pydot.Dot`. Certaines anciennes versions de `pydot` sous Linux ne permettent pas de générer du pdf. On le remplacera donc par du jpg, du ps, ou tout autre format disponible.