TP compilateurs 1 : Prise en main de PLY

1 Introduction

Dans ce TP, nous allons commencer à explorer l'implémentation d'un analyseur lexical et d'un analyseur syntaxique en nous appuyant sur le package python PLY [1].

PLY, pour Python-Lex-Yacc, est composé de deux modules :

1. lex.py est un générateur d'analyseurs lexicaux
2. yacc.py est un générateur d'analyseurs syntaxiques LALR(1)

Ces outils suivent d'assez près l'implémentation traditionnelle des outils Lex et Yacc, mais avec quelques caractéristiques :

- La spécification des grammaires est incluse directement dans le fichier source de l'analyseur.
- Des conventions de nommage et les capacités d'introspection de python permettent à PLY de construire dynamiquement des analyseurs à l'exécution, sans phase de génération séparée.
- Les grammaires sont précisées principalement dans les *docstrings* des fonctions. Ce détournement des *docstrings* peut surprendre, mais il permet une spécification très ramassée des analyseurs.

Ce document va vous guider pas à pas dans la réalisation avec PLY d'un petit interpréteur pour des expressions arithmétiques. Dans les TPs qui suivront, nous étendrons petit à petit cet interpréteur pour illustrer les différentes notions vues au cours.

2 Installation

Sous Linux, il se peut que votre gestionnaire de package vous permette d'installer PLY directement. Par exemple, dans une distribution basée Debian :

```
apt-get install python-ply
```

Pour les autres plateformes, récupérez le fichier zip contenant la dernière version de PLY sur le site http://www.dabeaz.com/ply/. Décompressez-le installez-le :

```
C:\...\ply-x.y> python setup.py install
```

3 Analyseur lexical

Nous allons maintenant réaliser un analyseur lexical pour notre interpréteur. Créez un dossier destiné à contenir les différents fichiers du TP et créez dans ce dossier le document lex1.py.

Bien entendu, on va commencer par importer lex :

```
import ply.lex as lex
```

1. http://www.dabeaz.com/ply/

Il faut ensuite définir les types de lexèmes que nous utiliserons ; pour l'instant :

```
tokens = (
    'NUMBER',
    'PLUS',
    'MINUS'
)
```

Les lexèmes simples peuvent être définis par une simple expression régulière, affectée à une variable portant le nom du lexème précédé de `t_`[2] :

```
t_PLUS    = r'\+'
t_MINUS   = r'-'
```

Pour les nombres, c'est un petit peu plus compliqué, parce qu'on aimerait récupérer leur valeur. On utilise donc une fonction ; vous remarquerez que c'est la *docstring* de la fonction qui contient l'expression régulière définissant le lexème !

```
def t_NUMBER(t):
    r'\d+'
    t.value = int(t.value)
    return t
```

Voilà, notre analyseur lexical est déjà presque prêt ! Il ne reste plus que quelques détails à régler :

1. Garder la trace du numéro de ligne où interviennent nos lexèmes (pour le diagnostic d'erreur) :

   ```
   def t_newline(t):
       r'\n+'
       t.lexer.lineno += len(t.value)
   ```

   Cette fonction ne renvoie rien. Ceci veut dire que le lexème sera ignoré. Si on voulait produire un lexème pour les fins de lignes, il faudrait rajouter `newline` dans la liste des lexèmes et finir la fonction par `return t`.

2. Ignorer les espaces et les tabulations

   ```
   t_ignore  = ' \t'
   ```

   On pourrait aussi les ignorer en utilisant une fonction qui ne renvoie rien (comme pour les fins de lignes), mais `t_ignore` est plus efficace.

3. Enfin, gérer les erreurs en cas de caractère imprévu :

   ```
   def t_error(t):
       print ("Illegal character '%s'" % t.value[0])
       t.lexer.skip(1)
   ```

Il ne nous reste plus qu'à construire notre analyseur...

```
lex.lex()
```

... et à le tester :

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

    lex.input(prog)
```

2. Souvenez-vous que c'est une bonne habitude d'utiliser les *raw strings* pour définir des expressions régulières ; ceci limite les problèmes d'échappement...

```
    while 1:
        tok = lex.token()
        if not tok: break
        print ("line %d: %s(%s)" % (tok.lineno, tok.type, tok.value) ↩
            )
```

Remarque Pour des raisons de simplicité, ce code ne contient pas de gestion des exceptions. Mais il est évident que l'ouverture du fichier devrait se faire un peu plus prudemment...

Sauvez votre programme, écrivez un ou deux fichiers d'entrée et testez le tout !

Maintenant, enregistrez votre programme sous le nom `lex2.py` et apportez-y les modifications suivantes :

Exercice 1 Pour l'instant, un "+" donne un lexème de type PLUS et un "-" de type MINUS. Modifiez votre programme pour que dans les deux cas on produise un lexème de *type* ADD_OP, dont la *valeur* sera "+" ou "-".

Exercice 2 Complétez votre programme pour qu'il accepte aussi les '*' et les '/', tous deux de type MUL_OP.

Exercice 3 Modifiez la définition du lexème NUMBER pour qu'il puisse également prendre en charge des nombres à virgule (du genre `13.47`).

4 Analyseur syntaxique

Avant de se lancer dans la réalisation proprement dite de l'analyseur syntaxique, deux petites remarques :

1. Pour bien séparer les choses, nous allons écrire nos deux analyseurs dans des modules (fichiers) séparés ; mais rien n'empêcherait de les écrire à la suite dans le même fichier.
2. Cet analyseur syntaxique va pour l'instant évaluer les expressions à la volée et fournir directement le résultat. Par la suite, nous le modifierons pour qu'il construise un arbre syntaxique et délègue les autres traitements aux phases ultérieures du compilateur...

Passons maintenant à la réalisation de l'analyseur syntaxique. Nous nous baserons dans un premier temps sur la grammaire BNF simplifiée suivante [3] :

```
expression  →  expression ADD_OP expression
            |  expression MUL_OP expression
            |  NUMBER
```

Créez un fichier `parser1.py` et commencez de manière assez classique :

```
import ply.yacc as yacc
```

Nous aurons aussi besoin de la liste de lexèmes définie dans notre analyseur lexical :

```
from lex2 import tokens
```

Nous pouvons maintenant commencer à décrire notre grammaire. Les règles seront représentées par des fonctions dont le nom doit commencer par `p_` et dont la *docstring* donne la règle de grammaire correspondante. Le nom de la fonction après le `p_` n'a aucune importance, mais nous tenterons bien entendu de garder des noms bien significatifs. Par exemple :

```
def p_expression_num(p):
    'expression : NUMBER'
    p[0] = p[1]
```

3. Notez que cette grammaire est ambiguë ! Nous y reviendrons.

Le paramètre `p` mérite quelques commentaires :
- Pendant l'analyse, on va attribuer à chaque symbole de la règle une valeur
- Ces valeurs sont accessibles par le paramètre `p` : `p[0]` désigne la valeur du symbole à gauche de la règle, `p[1]` la valeur du suivant, etc.
- Ainsi, le code ci-dessus signifie quelque chose comme : "lorsque l'expression se réduit à un nombre, la valeur de l'expression est celle de ce nombre". Logique, non ?[4]

> **Stop !** Vérifiez que vous avez bien compris l'explication du paramètre `p` ci-dessus. Si ce n'est pas le cas, relisez-la une fois, et si vous n'avez toujours pas compris, demandez à un collègue ou au professeur. C'est important, ne passez pas plus loin avant que ce soit bien clair !

On pourrait donc traiter les additions et soustractions par une règle du genre :

```
def p_expression_addop(p):
    "expression : expression ADD_OP expression"
    if p[2]=='+':
        p[0] = p[1] + p[3]
    elif p[2] == "-":
        p[0] = p[1] - p[3]
```

Mais puisque nous sommes dans un cours de programmation avancée, nous utiliserons une autre méthode ! ;-)

Tout d'abord, définissons un dictionnaire de fonctions :

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

Nous pouvons maintenant remplacer la règle/fonction `p_expression_addop` par quelque chose de beaucoup plus général :

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

Et voilà, c'est presque fini ! Ajoutons encore une gestion élémentaire des erreurs :

```
def p_error(p):
    print ("Syntax error in line %d" % p.lineno)
    yacc.errok()
```

La première ligne affiche un message d'erreur et la seconde signale à yacc que l'erreur a été traitée[5].

Il ne nous reste plus qu'à construire l'analyseur et à le tester :

```
yacc.yacc(outputdir='generated')

if __name__ == "__main__":
    import sys
    prog = open(sys.argv[1]).read()
    result = yacc.parse(prog)
    print (result)
```

4. Notez que quand on voudra construire un arbre au lieu d'évaluer l'expression, c'est ici qu'il faudra le faire !
5. Bien entendu, il y aurait bien plus à dire sur la gestion des erreurs ! Il ne s'agit ici que d'une esquisse !

Remarque La construction d'un analyseur syntaxique sur la base d'une grammaire est une opération relativement lourde. Pour gagner du temps lorsque la grammaire n'a pas changé, `PLY` génère des fichiers qui contiennent toutes les informations d'exécution. Ces fichiers sont re-générés automatiquement à chaque changement de la grammaire.

Pour garder ces fichiers séparés de vos fichiers source, créez un répertoire `generated` dans le même répertoire que votre fichier `parser.py` et passez-le en argument à `yacc` comme ci-dessus.

En testant votre nouvelle création, vous ne manquerez pas de noter deux choses :

1. Au premier lancement, `PLY` vous dira quelque chose comme

   ```
   yacc: Generating LALR parsing table...
   yacc: 4 shift/reduce conflicts
   ```

   Vous le savez, les conflits shift/reduce sont dus à l'ambiguïté de la grammaire.
2. Si vous analysez l'expression `3*2+1`, le résultat sera 9, ce qui confirme le problème d'ambiguïté de la grammaire.

Pour lever cette ambiguïté, on peut donner à `PLY` des règles de priorités sur les lexèmes. Dans votre programme, avant de créer l'analyseur, ajoutez :

```
precedence = (
    ('left', 'ADD_OP'),
    ('left', 'MUL_OP'),
)
```

... Ce qui veut dire :

1. ADD_OP et MUL_OP sont associatifs à gauche : par exemple, 1/2/3 doit être compris comme (1/2)/3
2. ADD_OP est *moins prioritaire* que MUL_OP.

Essayez maintenant d'évaluer notre fameux `3*2+1` : le résultat est maintenant correct et `yacc` ne se plaint plus des conflits...

Ouvrez des nouvelles version de vos deux analyseurs sous les noms `lex3.py` et `parser2.py` et...

Exercice 1 Modifiez-les pour que vos expressions puissent contenir des parenthèses.

Exercice 2 `PLY` génère un analyseur syntaxique LALR(1). Pour suivre le déroulement de l'analyse, modifiez l'appel à `yacc.parse()` :

```
result = yacc.parse(prog, debug=1)
```

Vous pourrez ainsi suivre les différentes étapes "shift-reduce" de l'algorithme sur quelques entrées simples. Notez que les numéros d'états se rapportent à la description des tables d'analyse que l'on peut trouver dans `generated/parser.out`.

(Une fois que vous avez exploré cette fonctionnalité, retirez le `debug=1`...)

Remarque si vous utilisez une ancienne version de `PLY`, il se peut que `debug=1` ne produise pas de sortie de debug... remplacez alors cette expression par `debug=2`.

Exercice 3 En vous aidant de la documentation de `PLY` (`ply.html` dans le répertoire `doc` de `ply-x.y`), modifiez votre analyseur pour qu'il puisse traiter les nombres signés (p.ex. `+3--7`). Pour cela, il est plus simple de considérer p.ex. `-2` comme une *expression* formée d'un opérateur unaire et d'une sous-expression. Cherchez donc "unary minus" dans la doc de ply, lisez ce qui y figure et adaptez le code à votre situation.

5 Conclusion

Ce TP vous a permis de prendre en main les principales fonctionnalités de `PLY`. Dans le prochain TP, nous complexifierons le langage que notre mini-compilateur peut traiter et nous construirons un arbre syntaxique plutôt que de tout évaluer directement.

En attendant, pour plus de détails sur `PLY`, vous pouvez vous référer au document `ply.html` dans le répertoire `doc` de `ply-2.3` ainsi qu'aux exemples fournis dans le répertoire `examples`.

Quelques remarques sur la résolution de conflits :

Pour résoudre un conflit **avancer-réduire**
- La méthode utilisée ci-dessus utilisant une prorité sur les lexèmes marche bien dans des cas comme les expressions, mais n'est pas toujours appropriée ;
- on peut aussi modifier la grammaire pour faire disparaître le conflit...
- ... ou, en dernier recours, se convaincre que la règle "préférer avancer plutôt que réduire" fera toujours ce qu'on veut, et *documenter le conflit en expliquant pourquoi il n'est pas dérangeant*.

Les conflits **reduire-réduire** sont presque toujours néfastes et il vaut mieux modifier sa grammaire pour tenter de les éliminer.