Llenguatges de Programació

Python 3: compiladors

Gerard Escudero

Universitat Politècnica de Catalunya, 2019

Instal·lació del ANTLR4 (per Python)

Requeriments:

- Python 3

Instruccions:

- Download the ANTLR4 jar file:
  - jar file
  - Getting started
- Install python runtime:
  - `pip3 install antlr4-python3-runtime` or
  - `pip install antlr4-python3-runtime`

El primer programa ANTLR

Arxiu de gramàtica `Expr.g`:

```
grammar Expr;

root : expr EOF ;

expr : expr MES expr
     | NUM
     ;

NUM : [0-9]+ ;
MES : '+' ;
WS : [ \n]+ -> skip ;
```

⚠️ Noteu que el nom de l'arxiu ha de concordar amb el de la gramàtica.

- `expr`: definició de la gramàtica per la suma de nombres naturals.
- `skip`: indica a l'escàner que el token WS no ha d'arribar al parser.
- `root`: per processar el final d'arxiu.

Compilació a Python3

La línia de comandes:

```
antlr4 -Dlanguage=Python3 -no-listener Expr.g
```

genera els arxius:

- `ExprLexer.py` i `ExprLexer.tokens`
- `ExprParser.py` i `Expr.tokens`

⚠️ Noteu que els arxius anteriors comencen pel nom de la gramàtica.

Construcció de l'script principal

Script de test:

```
from antlr4 import *
from ExprLexer import ExprLexer
from ExprParser import ExprParser

input_stream = InputStream(input('? '))

lexer = ExprLexer(input_stream)
token_stream = CommonTokenStream(lexer)
parser = ExprParser(token_stream)
tree = parser.root()
print(tree.toStringTree(recog=parser))
```

Noteu que aquest script processa una única linia entrada per consola.

Test: `3 + 4` 👉 `(root (expr (expr 3) + (expr 4)) <EOF>)`

Què passa amb: `3 + +`, `3 3` o `3 + 4 + 5`?

Notes sobre l'entrada

una única linia

```
input_stream = InputStream(input('? '))
```

stdin

```
input_stream = StdinStream()
```

un arxiu passat com a paràmetre

```
input_stream = FileStream(sys.argv[1])
```

arxius amb accents

```
input_stream = FileStream(sys.argv[1], encoding='utf-8')
```

En aquest cas haurem d'incloure a la gramàtica aquest tipus de caràcters:

```
WORD : [a-zA-Z\u0080-\u00FF]+ ;
```

Notes sobre gramàtiques

Recursivitat per l'esquerra:

Amb les versions anteriors no es podia afegir una regla de l'estil:
`expr : expr '*' expr`

Per solucionar això s'afegien regles tipus `expr : NUM '*' expr`

Precedència d'operadors:

Amb l'ordre d'escriptura:

```
expr : expr '*' expr
     | expr '+' expr
     | INT
     ;
```

Associativitat:

L'associativitat com la potència queda com:

```
expr : <assoc=right> expr '^' expr
     | INT
     ;
```

Exercici 1

Afegiu a la gramàtica els operadors de:

- resta
- multiplicació
- divisió
- potència

Tingueu en compte:

- la precedència d'operadors
- la associativitat a la dreta de la potència

Visitors

Els *visitors* són *tree walkers*, un mecanisme per recòrrer els ASTs.

Amb la comanda:

```
antlr4 -Dlanguage=Python3 -no-listener -visitor Expr.g
```

compilarem la gramàtica i generarem la plantilla del visitor (`ExprVisitor.py`):

```
# Generated from Expr.g by ANTLR 4.5.1
from antlr4 import *
if __name__ is not None and "." in __name__:
    from .ExprParser import ExprParser
else:
    from ExprParser import ExprParser

# This class defines a complete generic visitor ...
class ExprVisitor(ParseTreeVisitor):

    # Visit a parse tree produced by ExprParser#expr.
    def visitExpr(self, ctx:ExprParser.ExprContext):
        return self.visitChildren(ctx)

del ExprParser
```

`visitExpr` és el *callback* associat a la regla `Expr` per visitar-la.

Visitor per recorrer l'arbre

Codi d'un *visitor* `TreeVisitor.py` per mostrar l'arbre heredant de la plantilla:

```
if __name__ is not None and "." in __name__:
    from .ExprParser import ExprParser
    from .ExprVisitor import ExprVisitor
else:
    from ExprParser import ExprParser
    from ExprVisitor import ExprVisitor

class TreeVisitor(ExprVisitor):
    def __init__(self):
        self.nivell = 0

    def visitExpr(self, ctx):
        l = list(ctx.getChildren())
        if len(l) == 1:
            print("  " * self.nivell +
                  ExprParser.symbolicNames[l[0].getSymbol().type] +
                  '(' +l[0].getText() + ')')
        else:  # len(l) == 3
            print('  ' *  self.nivell + 'MES(+)')
            self.nivell += 1
            self.visit(l[0])
            self.visit(l[2])
            self.nivell -= 1
```

Informació per crear *visitors*

Accés als components de la part dreta de la regla:

- amb els fills: `ctx.getChildren()` (és un generador)
  - `l = list(ctx.getChildren())` o
  - `op1, op, op2 = list(ctx.getChildren())`

Altres mètodes interessants:

- `n.getText()`: text del node
- `ExprParser.symbolicNames[n.getSymbol().type]`: token del node en format text
- `ExprParser.MES`: índex intern del token MES per al parser. Es sol utilitzar junt amb `n.getSymbol().type`

Informació adicional:

- podem intercanviar informació amb un visitor mitjançant el constructor `__init__` i fent que el mètode de la *regla arrel* torni quelcom
  - Exemple d'ús: persistència de taula de símbols entre diferents *visitors*
- quan un node pertany a la part lèxica conté l'atribut `getSymbol` i quan pertany a la part sintàctica l'atribut `getRuleIndex`.

Arxiu de test

L'arxiu de test l'hem de modificar:

```
from antlr4 import *
from ExprLexer import ExprLexer
from ExprParser import ExprParser
from TreeVisitor import TreeVisitor

input_stream = InputStream(input('? '))

lexer = ExprLexer(input_stream)
token_stream = CommonTokenStream(lexer)
parser = ExprParser(token_stream)
tree = parser.root()

visitor = TreeVisitor()
visitor.visit(tree)
```

Execució

Un exemple de resultat de l'script anterior:

```
3 + 4
👉
MES(+)
  NUM(3)
  NUM(4)
```

Exercici 2

Afegiu el mecanisme per mostrar l'arbre generat a la gramàtica de l'exercici 1.

Ús d'etiquetes en les gramàtiques ANTLR4

Les etiquetes són un mecanisme que ens ajuden a clarificar el codi.

Donada la gramàtica següent:

```
grammar Expr;

root : expr EOF ;

expr : expr MES expr # Suma
     | NUM           # Valor
     ;

NUM : [0-9]+ ;
MES : '+' ;
WS  : [ \n]+ -> skip ;
```

L'ANTLR ens generarà un mètode per la producció `Suma` i un altre per `Valor` en el *visitor*.

Nota: en una regla de la gramàtica han de ser totes les produccions amb etiquetes o cap.

Visitor amb etiquetes

```
if __name__ is not None and "." in __name__:
    from .ExprParser import ExprParser
    from .ExprVisitor import ExprVisitor
else:
    from ExprParser import ExprParser
    from ExprVisitor import ExprVisitor

class TreeVisitor(ExprVisitor):
    def __init__(self):
        self.nivell = 0

    def visitSuma(self, ctx):
        l = list(ctx.getChildren())
        print('  ' *  self.nivell + 'MES(+)')
        self.nivell += 1
        self.visit(l[0])
        self.visit(l[2])
        self.nivell -= 1

    def visitValor(self, ctx):
        l = list(ctx.getChildren())
        print("  " * self.nivell +
              ExprParser.symbolicNames[l[0].getSymbol().type] +
              '(' +l[0].getText() + ')')
```

Avaluació i interpretació d'ASTs

Visitor per avaluar les expressions:

```
if __name__ is not None and "." in __name__:
    from .ExprParser import ExprParser
    from .ExprVisitor import ExprVisitor
else:
    from ExprParser import ExprParser
    from ExprVisitor import ExprVisitor

class EvalVisitor(ExprVisitor):
    def visitRoot(self, ctx):
        l = list(ctx.getChildren())
        print(self.visit(l[0]))

    def visitExpr(self, ctx):
        l = list(ctx.getChildren())
        if len(l) == 1:
            return int(l[0].getText())
        else:  # len(l) == 3
            return self.visit(l[0]) + self.visit(l[2])
```

Exemple:

```
3 + 4 + 5 👉 12
```

Exercici 3

Afegiu el tractament d'avaluació per la resta d'operadors de l'exercici 3.

Exercici 4

Definiu una gramàtica i el seu mecanisme d'avaluació/execució per a quelcom tipus:

```
x := 3 + 5
write x
y := 3 + x + 5
write y
```

Nota: es pot utilitzar un diccionari com a taula de símbols.

Exercici 5

Amplieu l'exercici anterior per a que tracti quelcom com el següent:

```
c := 0
b := c + 5
if c = 0 then
    write b
end
```

Exercici 6

Exploreu que passa si realitzem l'exercici anterior sense el token `end`.

Exercici 7

Amplieu l'exercici anterior per a que tracti l'estructura `while`:

```
i := 1
while i <> 11 do
    write i * 2
    i := i + 1
end
```

Què passa amb les funcions?

Imagineu una llenguatge tipus:

```
function sm(x, y)
    return x + y
end

main
    a := 1 + 2
    b := a * 2
    write sm(a, b)
end
```

amb només:

- variables locals
- paràmetres per valor

Qüestions a tenir en compte:

1. La taula de símbols pot ser una *pila de diccionaris*.
2. En *visitar* la declaració de funcions hem de guardar en una estructura per a cada funció:
   - Nom (*id*)
   - Llista de paràmetres (*ids*)
   - El contexte del bloc de codi (per a poder fer un `self.visit(bloc)` en trobar la crida)
3. S'ha de gestionar el `return` en cascada.

Exercici 8

Amplieu l'exercici anterior per a incloure funcions d'aquest tipus.

Exercici 9

Comproveu que el vostre programa funciona amb recursivitat:

```
function fibo(n)
    if n = 0 then
        return 0
    end
    if n = 1 then
        return 1
    end
    return fibo(n-1) + fibo(n-2)
end

main
    a := 1
    while a <> 7 do
        write fibo(a)
        a := a + 1
    end
end
```

Referències

1. Terence Parr. *The Definitive ANTLR 4 Reference*, 2nd Edition. Pragmatic Bookshelf, 2013.

2. Alan Hohn. *ANTLR4 Python Example*. Últim accés: 26/1/2019. https://github.com/AlanHohn/antlr4-python

3. Guillem Godoy i Ramón Ferrer. *Parsing and AST cosntruction with PCCTS*. Materials d'LP, 2011.