Llenguatges de Programació

Sessió 3: compiladors

Gerard Escudero i Albert Rubio

UNIVERSITAT POLITÈCNICA DE CATALUNYA
BARCELONATECH
Facultat d'Informàtica de Barcelona

FIB

Contingut

Instal·lació de l'ANTLR4 (per Python)

Requeriments:

- Python 3

Instruccions:

```
pip install antlr4-tools
antlr4

pip install antlr4-python3-runtime
```

Windows: s'ha de fer alguna cosa més, seguiu la referència.

- antlr4-tools reference

Contingut

El primer programa ANTLR

Arxiu de gramàtica `exprs.g4`:

```
// Gramàtica per expressions senzilles

grammar exprs;

root : expr             // l'etiqueta ja és root
     ;

expr : expr '+' expr    # suma
     | NUM              # numero
     ;

NUM : [0-9]+ ;
WS  : [ \t\n\r]+ -> 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.
- `#`: etiqueta per diferenciar branques de les regles (no és un comentari!)

Compilació a Python3

La comanda

```
antlr4 -Dlanguage=Python3 -no-listener exprs.g4      # antlr en MacOS
```

genera els arxius:

- `exprsLexer.py` i `exprsLexer.tokens`
- `exprsParser.py` i `exprs.tokens`

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

Construcció de l'script principal

Script de test `exprs.py`:

```
from antlr4 import *
from exprsLexer import exprsLexer
from exprsParser import exprsParser

input_stream = InputStream(input('? '))
lexer = exprsLexer(input_stream)
token_stream = CommonTokenStream(lexer)
parser = exprsParser(token_stream)
tree = parser.root()
print(tree.toStringTree(recog=parser))
```

Noteu que aquest script processa una única línia d'entrada per consola.

En funcionament:

```
3 + 4
👉
(root (expr (expr 3) + (expr 4)))
```

```
3 +
👉
line 1:3 missing NUM at '<EOF>'
(root (expr (expr 3) +
      (expr <missing NUM>)))
```

Obtenció de l'entrada

una única línia

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

stdin

```
input_stream = StdinStream()
```

un arxiu passat com a paràmetre

```
input_stream = FileStream(nom_fitxer)
```

arxius amb unicode

```
input_stream = FileStream(nom_fitxer, 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
- l'associativitat a la dreta de la potència

Contingut

Visitors

Els *visitors* són *tree walkers*, un mecanisme per recórrer els ASTs. Amb la comanda:

```
antlr4 -Dlanguage=Python3 -no-listener -visitor exprs.g4    # antlr en MacOS
```

compilarem la gramàtica i generarem la class base del visitador (`exprsVisitor.py`):

```
# Generated from exprs.g4 by ANTLR 4.11.1
from antlr4 import *
if __name__ is not None and "." in __name
    from .exprsParser import exprsParser
else:
    from exprsParser import exprsParser

# This class defines a complete generic v

class exprsVisitor(ParseTreeVisitor):

    # Visit a parse tree produced by expr
    def visitRoot(self, ctx:exprsParser.R
        return self.visitChildren(ctx)
    ...
```

`visitRoot` és el *callback* associat a la regla `root` per visitar-la.

Quan hi ha una etiqueta com ara `# suma` , la regle és `visitSuma`.

La crida a `self.visit(node)` visita el visitador associat al tipus de `node`.

Visitor per recórrer l'arbre

Classe *visitor* `TreeVisitor.py` per mostrar l'arbre heretant de la classe base:

```
class TreeVisitor(exprsVisitor):

    def __init__(self):
        self.nivell = 0

    def visitSuma(self, ctx):
        [expressio1, operador, expressio2] = list(ctx.getChildren())
        print('  ' *  self.nivell + '+')
        self.nivell += 1
        self.visit(expressio1)
        self.visit(expressio2)
        self.nivell -= 1

    def visitNumero(self, ctx):
        [numero] = list(ctx.getChildren())
        print("  " * self.nivell + numero.getText())
```

- Cada funció de visita obté els fills del node `ctx` amb `getChildren()`, i:
  - visita els fills sintàctics amb `self.visit(ctx_i)`, o
  - obté algun atribut dels fills lèxics, com ara el seu text amb `ctx_i.getText()`.

Ús del visitor

L'script l'hem de modificar:

```
from antlr4 import *
from exprsLexer import exprsLexer
from exprsParser import exprsParser
from exprsVisitor import exprsVisitor

class TreeVisitor(exprsVisitor):
  ...

input_stream = InputStream(input('? '))
lexer = exprsLexer(input_stream)
token_stream = CommonTokenStream(lexer)
parser = exprsParser(token_stream)
tree = parser.root()

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

Execució

Un exemple de resultat de l'script anterior:

```
2 + 3 + 4
👉
+
  +
    2
    3
  4
```

Exercici 2

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

Avaluació i interpretació d'ASTs

Visitor per avaluar les expressions:

```
class EvalVisitor(exprsVisitor):

    def visitRoot(self, ctx):
        [expressio] = list(ctx.getChildren())
        print(self.visit(expressio))

    def visitSuma(self, ctx):
        [expressio1, operador, expressio2] = list(ctx.getChildren())
        return self.visit(expressio1) + self.visit(expressio2)

    def visitNumero(self, ctx):
        [numero] = list(ctx.getChildren())
        return int(numero.getText())
```

Exemple:

`3 + 4 + 5` 👉 `12`

Nota: podeu utilitzar més d'un visitor en un script.

Contingut

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
```

Contingut

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, per a cada funció, hem de guardar en una estructura:
    - El seu nom (*id*)
    - La seva llista de paràmetres (*ids*)
    - El contexte (node de l'AST) del seu 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
```

Contingut

Errors sintàctics

Antlr té un parell mètodes per tractar-los:

- `getNumberOfSyntaxErrors`: ens indica els errors sintàctics
- `removeErrorListeners`: desactiva els missatges de les excepcions del parser

Exemples:

Amb aquest codi evitem cridar el `visitor` si s'ha produït un error sintàctic.

```
if parser.getNumberOfSyntaxErrors() == 0:
  visitor = TreeVisitor()
  visitor.visit(tree)
else:
  print(parser.getNumberOfSyntaxErrors(), 'errors de sintaxi.')
  print(tree.toStringTree(recog=parser))
```

Amb aquest codi evitem els missatges de les excepcions que llença el parser.

```
parser = exprsParser(token_stream)
parser.removeErrorListeners()
tree = parser.root()
```

Errors lèxics

Es produeixen quan fiquem un símbol no reconegut per la gramàtica.

Per evitar aquests errors hem d'afegir una regla al final de la gramàtica:

```
LEXICAL_ERROR : . ;
```

i desactivar els missatges de les excepcions del lexer:

```
lexer = exprsLexer(input_stream)
lexer.removeErrorListeners()
token_stream = CommonTokenStream(lexer)
```

Contingut

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.