Python Biella Group

Serata 2

Obiettivi

- Recap con una grammatica più “complicata”: Calculator
  - Con test automatici
- Nuovi elementi nella grammatica: **nome elementi** e **labels**
- Creare un **traduttore** di formato e usare i **listener**

Un calcolatore (quasi) completo

ANTLR4 - Calculator.g4 - 1/2

Calculator: grammatica che simula il comportamento di un calcolatore

```
// concrete syntax
grammar Calculator;

// non-terminals expressed as context-free grammar (BNF)
expr:   left=expr op=('*'|'/') right=expr  # OpExpr
    |   left=expr op=('+'|'-') right=expr  # OpExpr
    |   atom=INT                           # AtomExpr
    |   '(' expr ')'                       # ParenExpr
    ;

// tokens expressed as regular expressions
INT : [0-9]+ ;
WS  :   [ \t]+ -> skip ;
```

La grammatica ha 1 **regola,** 2 **tokens**
L'**ordine** in cui è espressa la regola **determina** l'ordine di **priorità**, mentre prodotto e divisione hanno stessa priorità

Diamo un **nome agli elementi della regola**, semplificherà il visitor di valutazione
Se si usano, occorre usarli per **tutti** i casi della regola

ANTLR4 - Calculator.g4 - 2/2


```
java org.antlr.v4.Tool -Dlanguage=Python3 -no-listener -visitor Calculator.g4
```

per generare le classi

```
Calculator.interp
Calculator.tokens
CalculatorLexer.interp
CalculatorLexer.py
CalculatorLexer.tokens
CalculatorParser.py
CalculatorVisitor.py
```

Scriviamo il **visitor** custom `EvalVisitor.py`

Verifichiamo la valutazione: `calculator.py`

Prepariamo **unit tests**: `test_calculator.py`

Uno o più traduttori di formato

Un traduttore di formato - 1/3

Semplice traduttore: da array di interi a stringa con Unicode
Esempio: da `{1,2,3} a "b'1'b'2'b'3'"`

Grammatica

```
grammar ArrayInit;
init  : '{' value (',' value)* '}' ;
value : init
      | INT
      ;
INT   : [0-9]+ ;
WS    : [ \t\n]+ -> skip
```

Generiamo le classi helper

```
java org.antlr.v4.Tool -Dlanguage=Python3 ArrayInit.g
```

e anche la classe **Listener**

```
ArrayInit.g             ArrayInit.interp
ArrayInit.tokens        ArrayInitLexer.interp
ArrayInitLexer.py       ArrayInitLexer.tokens
ArrayInitListener.py    ArrayInitParser.py
```

Un traduttore di formato - 2/3

Proviamo la grammatica con la generazione AST
`ast_grammar.py`

Estendiamo il listener implementando gli eventi che ci interessano
`ToUnicodeListener.py`

Scriviamo il traduttore
`translator.py`

Un traduttore di formato - 3/3

Senza cambiare grammatica e files esistenti possiamo con semplicità scrivere un traduttore diverso: per esempio da `{1,2,3,4}`

```
[
1
2
3
4
]
```

Adattiamo **listener**:
`SquaredNewLineListener.py`

Scriviamo il **traduttore**:
`translator_2.py`

Esercizio: trasformare i traduttori in modo che prendano input da file

Walker, Visitor e Listener

Walker, visitor e listener

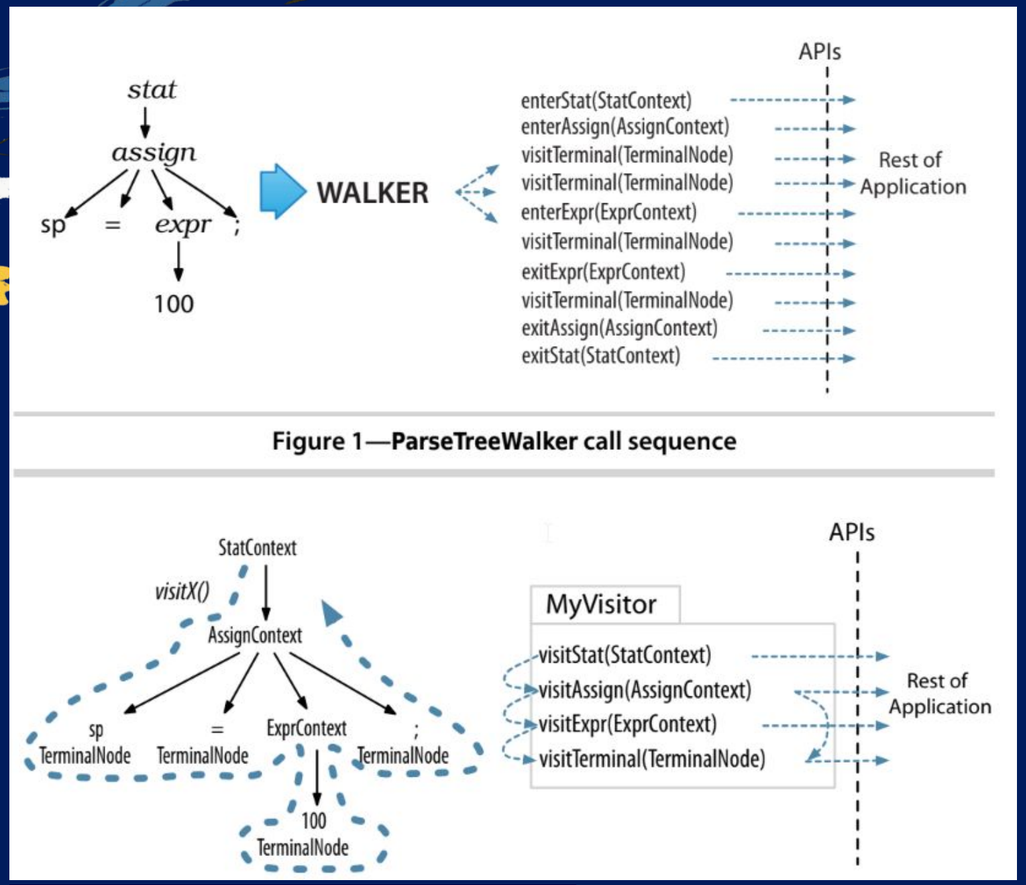
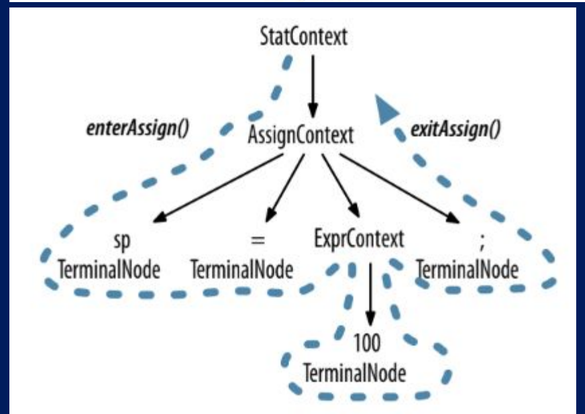


Figure 1—ParseTreeWalker call sequence



Walker

Visitor

Listener
ANTLR genera una sottoclasse Listener (per default) specifica per ogni grammatica con metodi di entrata e uscita per ogni **regola**.

Visitor vs listener

Dipende dal problema e dalle preferenze personali

Visitor
- Offre più controllo ma anche più responsabilità
- Restituisce sempre un valore.
- Marginalmente più lento

Listener
- Particolarmente utili per le traduzioni (meccaniche)
- Azioni di entrata e di uscita
- Utilizza uno stack esplicito allocato nell'heap, gestito da un walker, mentre il visitor utilizza lo stack di chiamate.
- Con input potenzialmente illimitato o alberi molto profondi i visitor potrebbero avere errore di stack nell’albero delle chiamate; i listener possono gestirli senza problemi.

Visitor vs listener

Le differenze principali sono che non puoi né controllare il flusso di un **listener** né restituire nulla dalle sue funzioni, mentre puoi fare entrambe le cose con un **visitor**.

Quindi, se hai bisogno di controllare come vengono inseriti i nodi dell'AST o di raccogliere informazioni da molti di essi, probabilmente vorrai utilizzare un **listener** .

Ciò è utile, ad esempio, con la generazione di codice, in cui alcune informazioni necessarie per creare un nuovo codice sorgente sono distribuite in molte parti.

Sia **listener** che **visitor** utilizzano la ricerca in profondità (depth-first search).

Antlr vs RegEx

Dipende dai requisiti

RegEx è uno strumento di ricerca di testo.
Se tutto ciò che devi fare è solo estrarre stringhe e sostituire stringhe, senza bisogno di un contesto, allora è il tool più adatto.

ANTLR è un generatore di compilatori.
Se hai bisogno di messaggi di errore e azioni di analisi o di una qualsiasi delle cose complicate che vengono fornite con un compilatore durante il parsing, allora è una buona opzione.

(cercare qualche altra citazione)