Python Biella Group

Serata 1 - Le basi

Obiettivi

- Cos'è un parser e perché ci interessa
- Come e perché generarne uno con ANTLR4
- Usare un parser ANTLR (prima parte)

Lo speaker

- alessio.stalla@strumenta.com
- Language Engineer
  - ~15yrs xp as developer, ~3yrs in Python

- Company: Strumenta
- "Better tools for better work"
- We help companies in solving complex problems more efficiently by providing specific languages and tools

1. Cos’è un parser e perché ci interessa

Cos’è un parser e perché ci interessa

Tutti/e sappiamo leggere codice.

```
def my_max(a, b):
    if a == b:
        print("I numeri sono identici")
    elif a > b:
        print("Il numero più grande tra i due è " + str(a))
    else:
        print("Il numero più grande tra i due è " + str(b))
```

Cos’è un parser e perché ci interessa

- In realtà, il contenuto di un file Python come il precedente **è una stringa;** il codice è cioè in origine **una sequenza lineare di caratteri, senza alcuna struttura:**

```
def my_max(a, b)\n\tif a == b:\n\t\tprint("I numeri sono identici")\n\telif…
```

- …non è molto leggibile, ma soprattutto, **non è il tipo di dato adatto per ragionare sulla struttura del programma:**
  - Quali variabili utilizza?
  - Se rinomino b in c (refactoring), come si trasforma il codice?
  - Ho chiamato la funzione print correttamente o ho sbagliato tipo/numero di parametri?
  - Qual è il risultato dell’esecuzione del programma (interpretazione/compilazione)?
  - Ecc. ecc.

Cos’è un parser e perché ci interessa

Un parser è una funzione che:

- **In input accetta una stringa,** che si presume del codice in un dato linguaggio;
- **In output, restituisce una rappresentazione logica della struttura del codice,** atta a ragionarci sopra.

Cos’è un parser e perché ci interessa

Più precisamente, **un linguaggio formale** (di programmazione, di configurazione, ecc.) **è definito da una specifica** che comprende:

- **Sintassi:** quali sequenze di caratteri sono valide e quali no.
  - La definizione formale della sintassi si chiama **grammatica.**
- **Semantica:** il significato associato al codice; nel caso di un linguaggio di programmazione, la semantica è “cosa avviene all’esecuzione del programma”, nel caso di un linguaggio di query (come SQL) la semantica è “quali dati vengono restituiti dalla query”, ecc.

Un parser è una funzione che implementa una grammatica. Il risultato del parsing di una stringa è:

- **Successo/fallimento** (il codice è valido secondo la grammatica o no?)
- **Una struttura dati ad albero (parse tree)** che rappresenta:
  - **Il percorso logico attraversato dal parser** per riconoscere la stringa in input;
  - Di conseguenza, **la struttura del codice** riconosciuta dal parser.

NB la specifica di un linguaggio (grammatica e/o semantica) può essere **formale** (linguaggio matematico), **informale** (descritta a parole), o **implicita** (emergente dal comportamento del compilatore/interprete, ma non scritta da nessuna parte).

Cos’è un parser e perché ci interessa

Ci possono interessare i parser perché:

- **I nostri software sono pieni di parser!**
  - **Nelle librerie che utilizziamo,** ad es. parser JSON, YAML, XML
  - **Nei servizi che invochiamo,** ad es. parser SQL
  - **In implementazioni ad hoc,** come quella regexp che causa sempre bachi, la capisce solo il collega che l’ha scritta e dobbiamo tenercelo buono anche se ci sta antipatico.
- Lo stesso linguaggio che utilizziamo (Python) ha bisogno di un parser per essere interpretato o compilato. I tool di sviluppo hanno bisogno di un parser per dare errori e suggerimenti.

Capire il funzionamento dei parser ci aiuta a capire meglio gli strumenti che utilizziamo e le scelte di progettazione che rendono Python confortevole o problematico da utilizzare, a seconda delle situazioni.

2. Scrivere un parser

Scrivere un parser: approccio naif

- Un parser è una funzione come ogni altra
- In particolare possiamo descriverlo come una **macchina a stati.**
  - Ogni stato rappresenta un punto di decisione all'interno del parser
- Possiamo scrivere un parser in modo naif come **discesa ricorsiva:**
  - **Macchina a stati implicita** (catena di if)
  - **Una funzione per ogni costrutto** del linguaggio

Scrivere un parser: approccio naif

```
def parse_module(stream):
    mod = Module()
    if next_char(stream) == 'f':
        if next_char(stream) == 'o':
            if next_char(stream) == 'r':
                if is_whitespace(next_char(stream)):
                    mod.statements.append(parse_for_statement(stream))
            else:
                ...
    return mod

def parse_for_statement(stream):
    var = parse_variable(stream)
    ...
    condition = parse_expression(stream)
    ...
    return ForStatement(var=..., condition=..., body=...)
```

Scrivere un parser: approccio naif

Esempio del risultato di un parser: Python include un parser Python

```
>>> import ast
>>> tree = ast.parse("for x in y: pass")
>>> tree
<_ast.Module object at 0x7ff14cbbe1f0>
>>> ast.dump(tree)
"Module(body=[For(target=Name(id='x', ctx=Store()), iter=Name(id='y',
ctx=Load()), body=[Pass()], orelse=[], type_comment=None)], type_ignores=[])"
```

Scrivere un parser: approccio naif

Requisiti funzionali:

- **Riconoscere codice valido/non valido** ✔
- **Ricostruire la struttura del codice** ✔
- Tolleranza agli errori ✘

Requisiti non funzionali:

- Messaggi di errore utili ✘
- Verifica che il parser rispetta la specifica del linguaggio ✘
- API uniformi/ben note ✘
- Tool di supporto (es. IDE, debugger) ✘
- Prestazioni ✘

Scrivere un parser: approccio naif #2

Perché non usare espressioni regolari?

- Una regexp non è un parser (solo linguaggi regolari)
- Una singola regexp avrebbe complessità enorme
- Tipicamente si scrivono più regexp per estrarre diversi tipi di informazioni
  - Fragilità, difficile da leggere e da manutenere/evolvere
  - Requisiti funzionali e non funzionali ancora meno soddisfatti rispetto al precedente approccio
- Pericolo 80/20

Parser Generator

- Un parser è una **implementazione** di una **grammatica**
- Possiamo specificare la grammatica in un **linguaggio formale**
- Possiamo quindi vedere:
  - La grammatica come **codice sorgente**
  - Il parser come **risultato della compilazione** della grammatica

- Un **parser generator** è un “compilatore” che legge una grammatica e genera un parser
- Anche noto per questo col termine (desueto) “compiler-compiler” ovvero “compilatore di compilatori” perché, tradizionalmente, queste tecniche si sono applicate ai compilatori, di cui un parser è il primo stadio.
- Solitamente, a differenza di un compilatore, l’output di un parser generator non è codice binario ma **codice sorgente** in un dato linguaggio, che quindi **va integrato** in un programma e a sua volta **interpretato o compilato** per poter essere eseguito.

Parser Generator: perché usarli

Un parser generato:

- utilizza **algoritmi avanzati, difficili da scrivere a mano**
  - gestione errori uniforme e generalmente migliore di quella che potremmo scrivere a mano
  - migliori performance
- è per definizione **corrispondente alla grammatica** che può quindi fungere sia da codice sorgente che da documentazione/specifica del linguaggio
- spesso è corredato da strumenti a supporto (che non vedremo)

Nota: solitamente le implementazioni dei linguaggi più diffusi (es. javac per Java, GCC o CLang per C, ecc.) **non utilizzano** un parser generator. Questo perché **un parser attentamente scritto a mano da esperti** utilizzando i migliori algoritmi in letteratura può essere **di qualità superiore** rispetto ad uno generato automaticamente. Tuttavia, **la difficoltà di implementazione è notevolmente maggiore,** fuori dalla portata dello sviluppatore medio (richiede infatti una approfondita specializzazione).

CPython fa eccezione in quanto utilizza un parser generator scritto in C. Vedi https://devguide.python.org/internals/parser/

Lexer/Scanner e Parser

Nota prima di proseguire.

Abbiamo detto che l’implementazione di un parser, con qualsiasi algoritmo, è di fatto una **macchina a stati.**

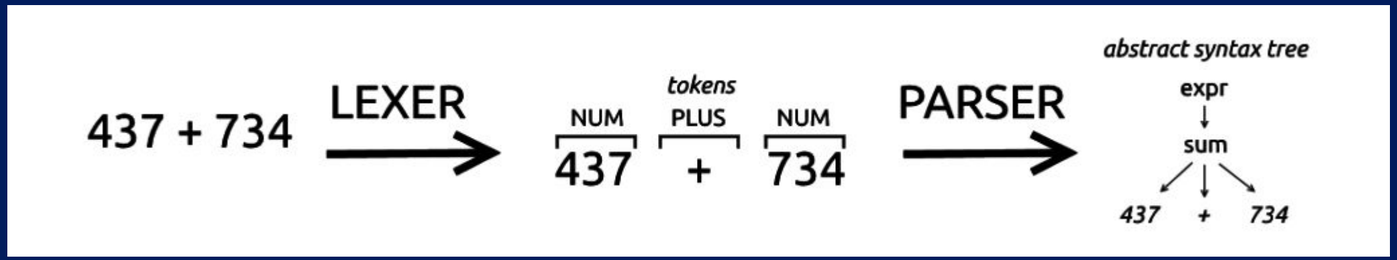
Se la grammatica è complessa, **il numero di stati può esplodere,** e con esso la dimensione del codice del parser. Ricordiamo l’esempio del “for” precedente: ogni stato rappresenta **un punto di decisione** all’interno del parser.

Lexer/Scanner e Parser

Per ridurre il numero di stati, si spezza il parser in due stadi:

1. **Lexing** (o *scanning).* In questa fase, **si suddivide il testo sorgente in una serie di token.** Ad esempio, la stringa “for x in foo:” (13 caratteri) potrebbe essere suddivisa in: [FOR, NAME(x), IN, NAME(foo), COLON] ossia 5 token.
2. **Parsing** vero e proprio. In questo caso il parser vedrà **una sequenza di token** e non di caratteri, con conseguente riduzione del numero di stati dovuta al fatto che più caratteri sono accorpati in un unico token.

Per esempio in Python il lexer si occupa anche di generare **token sintetici** per modellare **l’indentazione significativa** tipica del linguaggio.
Il lexer può anche **eliminare token non significativi** come spazi, a capo e commenti.



3. ANTLR4

ANTLR4 e sua installazione

ANTLR (ANother Tool for Language Recognition) è uno dei generatori di parser più diffusi e maturi, giunto alla versione 4.

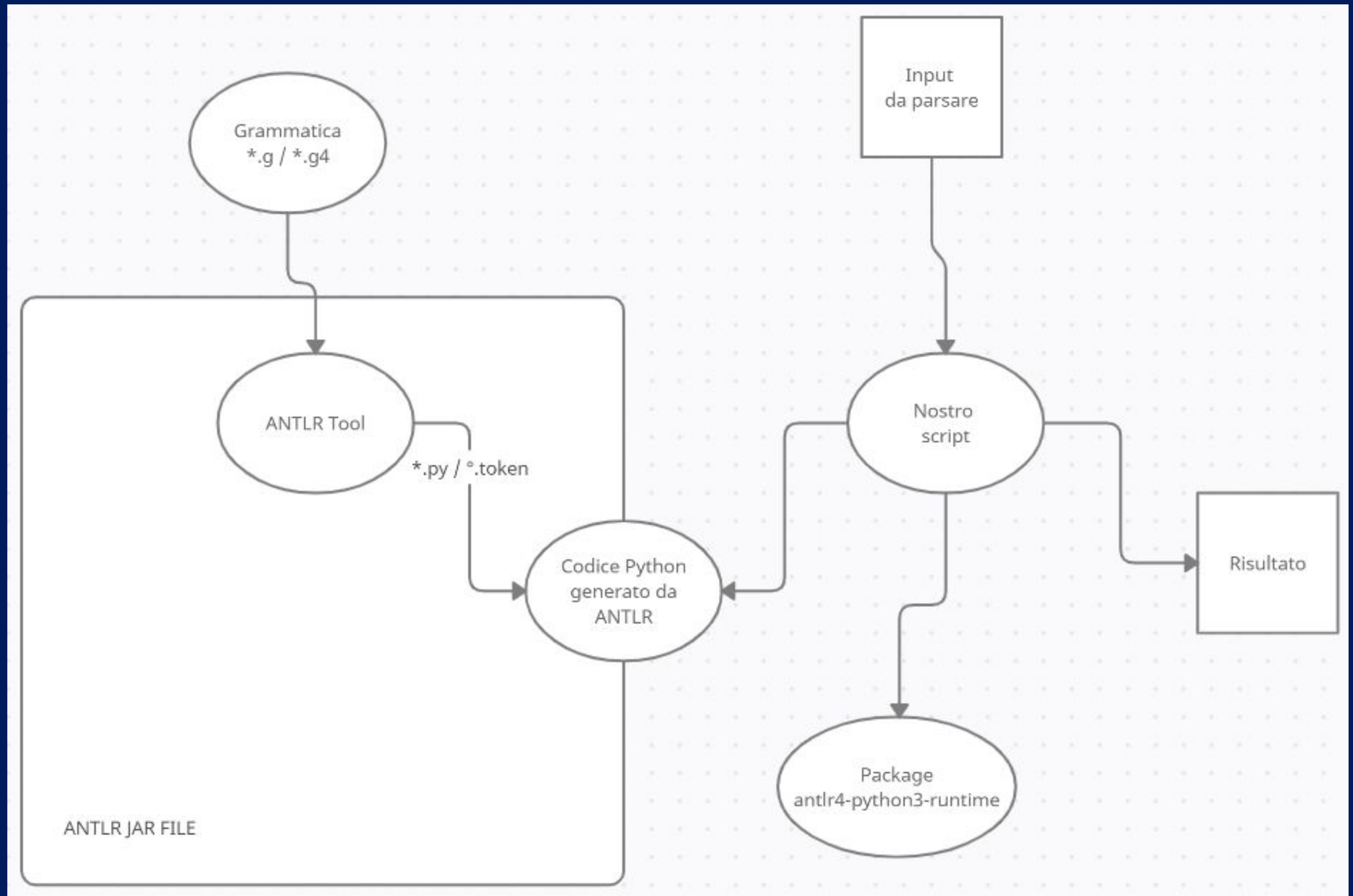
L'autore **Terence Parr** (professor of computer science at the University of San Francisco) ci lavora dal 1989 e ha pubblicato a riguardo diversi articoli scientifici.

Ampiamente utilizzato nel mondo accademico e industriale (Twitter, Oracle, Eclipse, … Strumenta!) per implementare linguaggi, strumenti di sviluppo, editor e altro.
https://www.antlr.org/about.html

Il generatore è scritto in **Java** (necessita quindi di JDK). Il **parser può essere generato in diversi linguaggi,** tra cui ovviamente Python (2 e 3).

Per installare ANTLR **basta scaricare il file jar** e, a scelta, aggiungere alcuni alias alla nostra shell (in modo da poter usare il comando "antlr4" invece che "java -jar …")

ANTLR4 – Come funziona



Esempio: Entity Language

```
module Insurance {
    entity Vehicle {
        license_plate: string;
        year: integer;
        owner: Person;
    }

    entity Person {
        name: string;
        address: string;
    }
}
```

Scriviamo il Lexer

File: `AntlrEntityLexer.g4`

```
lexer grammar AntlrEntityLexer;

options {
    caseInsensitive=true; //Opzione aggiunta recentemente, ignora il case
}

// Token per i tipi
INTEGER: 'integer'; // I nomi di token iniziano con la maiuscola
BOOLEAN: 'boolean'; // Per convenzione si scrivono in ALL_CAPS
STRING: 'string';   // Ogni token ha il suo pattern
// Questi token corrispondono ad una stringa fissa (case insensitive)

// Idem per le keyword (parole chiave)
ENTITY: 'entity';
MODULE: 'module';
```

Scriviamo il Lexer (parte 2)

File: `AntlrEntityLexer.g4` (continua)

```
// Segni di punteggiatura
COLON: ':';
SEMI: ';';
LSQRD: '[';
RSQRD: ']';
LCRLY: '{';
RCRLY: '}';

// Nomi (o identifier)
ID: [A-Z]+; // Notare il pattern tipo espressione regolare

// I caratteri di spaziatura non ci interessano,
// dunque li nascondiamo al parser
WS: [ \r\n\t]+ -> channel(HIDDEN);
```

Generare il Lexer

Possiamo dare un'occhiata al lexer generato da ANTLR anche se non abbiamo ancora un parser.

Ipotizziamo di aver salvato il lexer nella directory **grammar** e di volere l'output (il codice Python) nella directory **entity_parser.**
Supponiamo di avere

```
alias antlr4='java -jar $ANTLR_PATH/antlr-4.11.1-complete.jar'
```

Diamo il seguente comando per generare il lexer:

```
pushd grammar;
antlr4 -Dlanguage=Python3 AntlrEntityLexer.g4 -o ../entity_parser;
popd
```

Di default, ANTLR genera i file accanto alla grammatica. Tuttavia, generandolo in una directory separata, possiamo facilmente **escludere** il lexer dal controllo versione (e da linter, test coverage, ecc.), in quanto ricavabile automaticamente dalla grammatica. Possiamo quindi usare script di build e/o CI (come Github Actions) per rigenerare il lexer ed eseguire i test.

Generare il Lexer

Apriamo il lexer generato (`parser/AntlrEntityLexer.py`):

```
# Generated from AntlrEntityLexer.g4 by ANTLR 4.11.1
from antlr4 import *
from io import StringIO

# Ecc.

class AntlrEntityLexer(Lexer):

# Ecc.
```

Vediamo che:

- **Il Lexer è una classe Python**
- **Necessita di una libreria di runtime** per essere utilizzata

Il runtime per Python 3 è **antlr4-python3-runtime** e si trova su PyPI
https://pypi.org/project/antlr4-python3-runtime/

Nota: **la versione del runtime deve corrispondere a quella del tool** usato per generare il parser.

Generare il Lexer

Possiamo anche vedere che **il lexer generato contiene le dichiarazioni dei token** che useremo nel parser:

```
INTEGER = 1
BOOLEAN = 2
STRING = 3
...
```

E poi:

```
symbolicNames = [ "<INVALID>",
          "INTEGER", "BOOLEAN", "STRING", "ENTITY", "MODULE", "COLON",
          "SEMI", "LSQRD", "RSQRD", "LCRLY", "RCRLY", "ID", "WS" ]
```

Notare il javismo nelle API – purtroppo i parser generati da ANTLR non sono molto Pythonici come regole di nomenclatura.

Scriviamo il Parser

File: `AntlrEntityParser.g4`

```
parser grammar AntlrEntityParser;

options {
   tokenVocab=AntlrEntityLexer; // Riferimento al lexer
}

// Dichiarazione di una regola (o produzione).
// Le regole iniziano con la lettera minuscola.
module:
    MODULE name=ID LCRLY  // Match di 3 token: MODULE, ID e LCRLY "{"
        entities+=entity* // Match di 0 o più sottoregole entity
    RCRLY                 // Match di 1 token RCRLY "}"
    ;

entity:
   ENTITY name=ID LCRLY
      features+=feature*
   RCRLY
   ;
```

Scriviamo il Parser (parte 2)

```
feature:
   name=ID COLON type=type_spec SEMI
   ;

type_spec   // Regola definita per casi
   : INTEGER    #integer_type
   | BOOLEAN    #boolean_type
   | STRING     #string_type
   | target=ID #entity_type
   ;
```

Generare il Parser

Il comando è analogo a quanto visto per il lexer. Normalmente si generano lexer e parser assieme con un unico comando:

```
pushd grammar;
antlr4 -Dlanguage=Python3 AntlrEntityLexer.g4 AntlrEntityParser.g4 -o
../entity_parser;
popd
```

Se tutto va bene, il comando non restituisce output e troveremo nella directory “entity_parser” due file Python, uno per il lexer e uno per il parser.

Generare il Parser

Apriamo il parser generato (`entity_arser/AntlrEntityParser.py`):

```
# Generated from AntlrEntityParser.g4 by ANTLR 4.11.1
# encoding: utf-8
from antlr4 import *

# Ecc.

class AntlrEntityParser(Lexer):

# Ecc.
```

Come per il lexer:

- **Il parser è una classe Python**
- **Necessita di antlr4-python3-runtime**

Generare il Parser

Il parser generato, per ogni regola della grammatica, presenta:

- **Un metodo** chiamato come la regola;
- **Una classe Context.** Il metodo corrispondente restituisce un'istanza della classe.

Esempio:

```
class ModuleContext(ParserRuleContext):
    # Ecc.

def module(self):
    localctx = AntlrEntityParser.ModuleContext(self, self._ctx, self.state)
    # Ecc.
    return localctx
```

Nota: nomenclatura funziona bene per Java ma non per Python (CamelCase vs snake_case).

4. Usare il nostro parser

Come usare il nostro parser

I passi per utilizzare un parser ANTLR sono sempre gli stessi:

1. Costruire lo **stream di caratteri** a partire dal codice in input;
2. Costruire il **lexer** a partire dallo stream
3. Costruire un **TokenStream** a partire dal lexer
4. Costruire il **parser** a partire dal TokenStream
5. **Processare il risultato** del parsing.

Avendo 4 passi separati per costruire il parser, in caso di necessità di elaborazione non standard (ad es. lexer scritto a mano invece che generato), possiamo sostituire uno dei passi standard con una implementazione alternativa.

Vediamo i passi ad uno ad uno.

1. Lo Stream

```
from antlr4 import InputStream

code = '''module Insurance {
   entity Vehicle {
      license_plate: string;
      year: integer;
      owner: Person;
   }

   entity Person {
      name: string;
      address: string;
   }
}'''
input = InputStream(code)
```

1. Lo Stream

InputStream è la classe base che costruisce uno stream a partire da **una stringa.**

Abbiamo anche implementazioni specializzate: **FileStream** e **StdinStream.**

In ogni caso, **l'intero codice sorgente viene mantenuto in memoria** per le particolari caratteristiche dell'algoritmo di parsing utilizzato da ANTLR.

Nota: **gli stream sono stateful** in quanto mantengono l'indice "corrente". Non possono quindi essere usati in maniera concorrente.

2. Il Lexer

```
from entity_parser.AntlrEntityLexer import AntlrEntityLexer
# ...
lexer = AntlrEntityLexer(input)
```

Vediamo come ANTLR è poco Pythonico: non crea automaticamente un package per il nostro parser. Possiamo comunque crearlo noi.

Come gli stream, **anche i lexer sono stateful** in quanto mantengono l'indice "corrente".

3. Il TokenStream

Il lexer supporta una sola operazione: **nextToken()** che ritorna il token successivo (o EOF).

Il parser ha bisogno di “guardare avanti” (lookahead) n token, a seconda della grammatica e dell’input. Occorre quindi costruire un **TokenStream** con tale funzionalità.

```
from antlr4 import CommonTokenStream
# ...
token_stream = CommonTokenStream(lexer)
self.assertEqual(token_stream.LT(4).type, lexer.ENTITY)
```

Ovviamente per quanto già detto anche il token stream è stateful.

4. Il Parser

Finalmente possiamo costruire e invocare il nostro Parser:

```
from entity_parser.AntlrEntityParser import AntlrEntityParser
# ...
parser = AntlrEntityParser(token_stream)
tree = parser.module()
self.assertIsInstance(tree, AntlrEntityParser.ModuleContext)
```

Note:

- **Il parser è stateful** in quanto mantiene un riferimento al token stream e ha anche uno stato proprio interno (lo stato della **macchina a stati** che implementa).
- **Il parser non consuma tutto l'input** di default. La nostra regola (module in questo caso) deve terminare con EOF.
- **Il parser mantiene una cache** per le transizioni della sua macchina a stati. Usandolo più volte (istanze diverse), anche su input diversi, **migliorano le performance** ma aumenta anche l'occupazione di memoria.

5. Processare il risultato

L’output del parser è **un parse tree,** una struttura ad albero che possiamo attraversare con:

- **API generiche** (es. `node.getChild(n)`);
- **API specifiche** del nostro linguaggio (es. `module_node.entities`) parte delle classi *Context generate da ANTLR;
- **visitor o listener** generati da ANTLR.

Noi però suggeriamo di **processare ulteriormente l’albero** invece di attraversarlo tal quale. Questo **sarà l’argomento della prossima serata.**

L’output del parser sono anche **gli eventuali errori** di parsing:

- self.assertEqual(0, **parser.getNumberOfSyntaxErrors()**)
- Possiamo **agganciare un listener** al parser per catturare gli errori, che comprendono **messaggio d’errore e posizione** (linea e colonna) all’interno dell’input.
  - Di default, ANTLR configura un error listener che scrive su stdout.

Concludiamo

Il codice mostrato è su GitHub: https://github.com/PythonBiellaGroup/ANTLR
branch “serata1” (il branch main conterrà il codice completo alla fine delle 3 serate)

Abbiamo lasciato fuori diversi argomenti che si possono approfondire, come:

- Lexical modes
- Error recovery
- Debugging
- Performance tuning
- Ecc. ecc.

Domande e risposte

Perché usare proprio ANTLR?

- Error reporting eccellente
- Prestazioni OK di default
- API consolidate
- Tool di debug e analisi performance
- Community
- Disponibile per più linguaggi
- Librerie StarLasu (Strumenta) – vedremo nelle prossime serate