Python Biella Group

# Serata 1 - Le basi

## Obiettivi

- Capire la struttura generale di un compilatore e le differenze tra interpreti e compilatori
- Sapere dell’esistenza di tools che aiutano per farli, come ANTLR
- Creare un processore di linguaggio:
  - Vocabolario
  - Grammatica
  - AST (Abstract Syntax Tree)
  - Interpretazione lungo il percorso dell’albero


# Overview

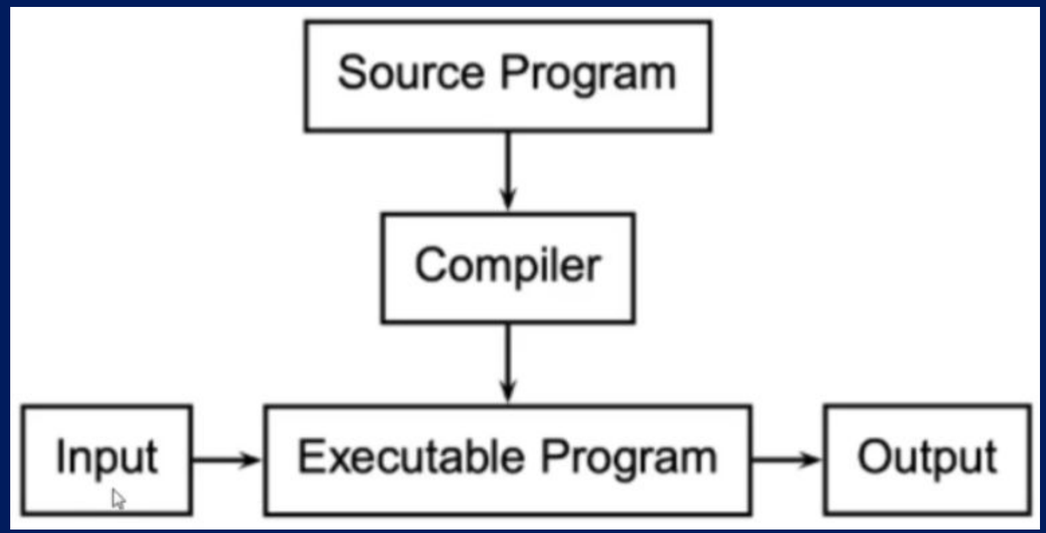
# Compilatori - Overview

Un **compilatore** è un programma che traduce programmi (codice sorgente) scritti in un linguaggio ad alto livello in linguaggio macchina o in generale in un linguaggio a basso livello.

Il sorgente viene letto dal compilatore che effettua il controllo sintattico delle istruzioni (preprocessing) e lo traduce in linguaggio macchina (object file), non ancora eseguibile (non incorpora ancora il codice binario delle librerie).
Il linker incorpora nel file le librerie producendo un file eseguibile

In caso di errore **non** viene prodotto alcun object file

Es: GCC, go



# Compilatori - Esempio GCC

Quando invochi GCC, normalmente esegue preprocessing, compilation, assembly e linking. Le "opzioni" consentono di interrompere questo processo in una fase intermedia o di eseguire step by step: l'opzione -S permette di generare solo codice assembly

Compilation in C

```c
int func(int a, int b) {
    a = a + b;
    return a;
}
```

```
gcc -S test.c
```

```
_func:
    pushq   %rbp
    movq    %rsp, %rbp
    movl    %edi, -4(%rbp)
    movl    %esi, -8(%rbp)
    movl    -4(%rbp), %eax
    addl    -8(%rbp), %eax
    movl    %eax, -4(%rbp)
    movl    -4(%rbp), %eax
    popq    %rbp
    retq
```

Per approfondimenti: https://www.thegeekstuff.com/2011/10/c-program-to-an-executable/

# Interpreti - Overview

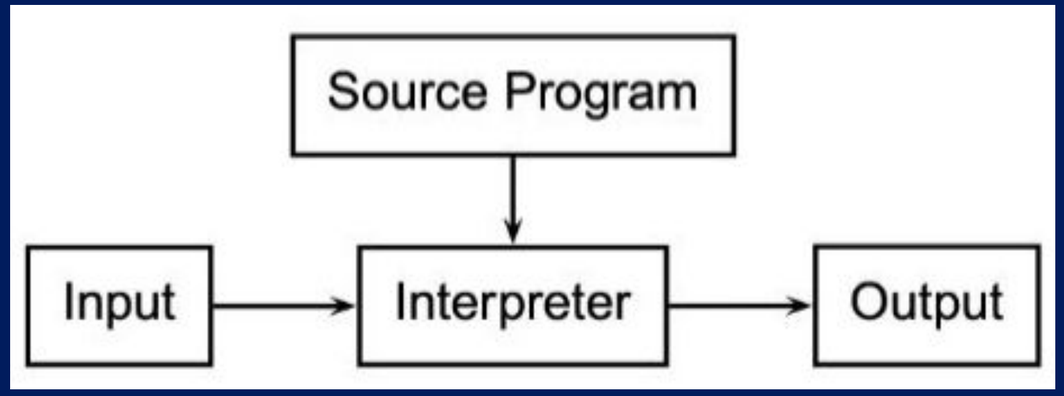
Un **interprete** è un programma che **traduce** le istruzioni del codice sorgente scritto in un linguaggio ad alto livello, **una per una**, in linguaggio macchina.

L'esecuzione può iniziare subito dopo la scrittura del programma sorgente (non serve "compilazione"

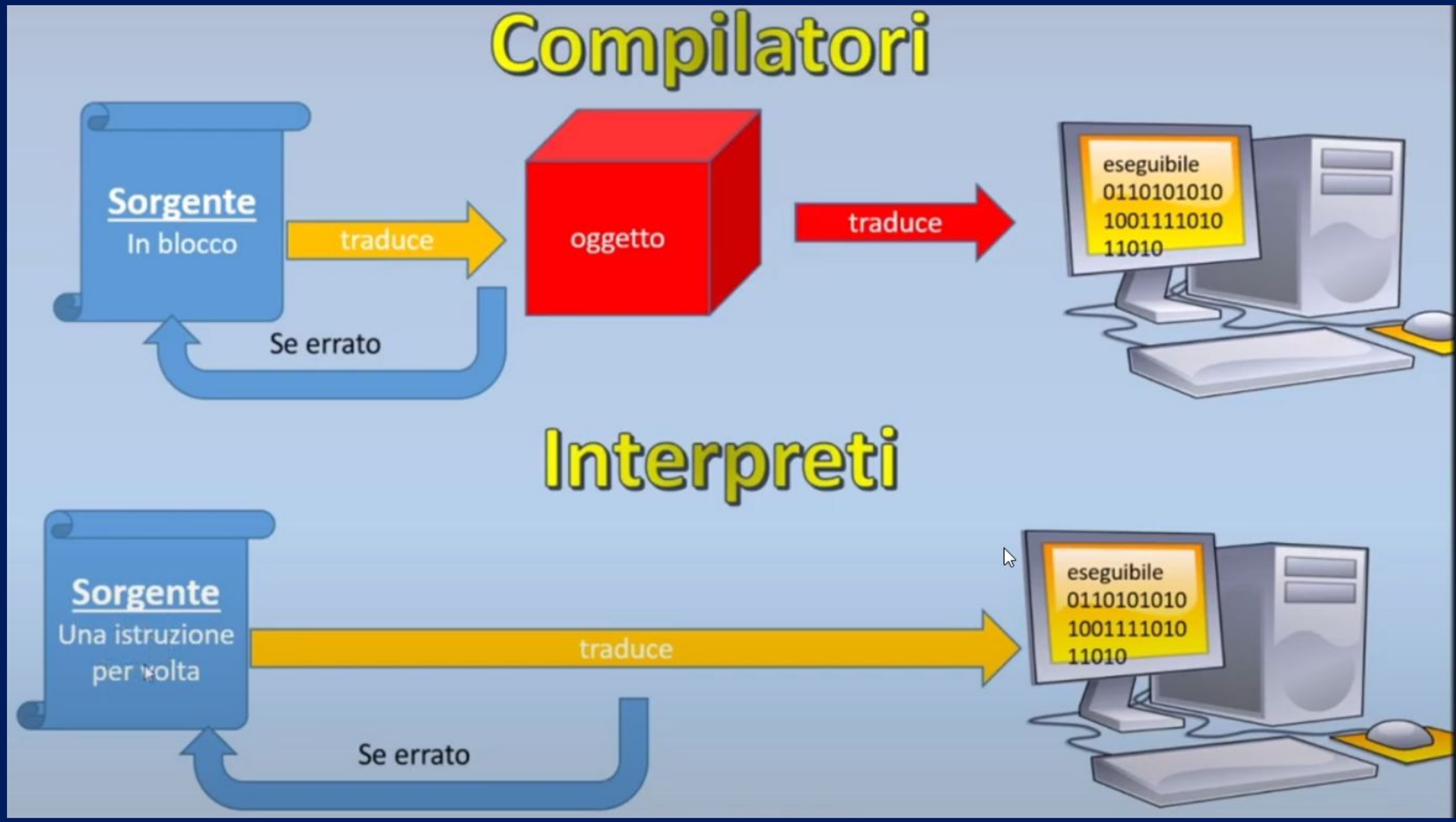
L'interprete non crea un programma oggetto (eseguibile); per essere eseguito interprete e programma devono essere in memoria

Se l'interprete riscontra un errore formale nella scrittura dell'istruzione, segnala l'errore e **interrompe** l'esecuzione del programma

Es: Perl, PHP, ...



# Compilatori vs Interpreti




# E Python?

# E Python? It’s *more* complicated... 1/2

Abbiamo fatto delle semplificazioni; occorre specificare che:

1) interpretato/compilato **non è una proprietà del linguaggio ma una proprietà dell'implementazione** (ci sono parecchi linguaggi con implementazioni utilizzabili di entrambi i tipi, principalmente nell'ambito dei linguaggi funzionali, vedere Haskell e ML)

2) **esiste un tipo di compilazione detta JIT (Just In Time)**, detta anche dynamic translation o run-time compilation, che è un modo per eseguire il codice del computer che prevede la compilazione durante l'esecuzione di un programma (in fase di esecuzione) anziché prima.

# E Python? It’s *more* complicated... 2/2

Python è inizialmente compilato e poi “quasi sempre” interpretato.
Non compilato in linguaggio macchina ma "solo" in bytecode.

Puoi vedere come appare il bytecode con il modulo **dis** della libreria standard.

Interpretare il bytecode è più veloce che interpretare direttamente Python.
L'elaborazione del bytecode dipende da quale implementazione di Python viene utilizzata.

Con **CPython** (l'implementazione standard di python) o **Jython** (mirato per l'integrazione con il linguaggio di programmazione java) viene prima tradotto in bytecode e, a seconda dell'implementazione di python che stai usando, questo bycode viene indirizzato alla corrispondente macchina virtuale per l'interpretazione PVM (Python Virtual Machine) per CPython e JVM (Java Virtual Machine) per Jython.

Con **PyPy** il codice Python viene eseguito attraverso un processo di compilazione JIT, producendo un prodotto finale che funziona più velocemente di CPython in molti casi. Per approfondimenti: http://pypy.org/


# Linguaggi formali

# Sintassi di un linguaggio - Overview

La **sintassi** di un linguaggio di programmazione è l'insieme di regole che definiscono le combinazioni di simboli che sono considerate costrutti corretti.

```
Biella è una città molto bella
```

-> sintatticamente corretta in Italiano ma non in Python

```
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))
```

-> sintatticamente corretta in Python ma non in Java

# Sintassi di un linguaggio - Overview

La **sintassi** di un linguaggio di programmazione è spesso specificata usando una **grammatica context-free** (**CFG**).

Gli elementi base (parole) sono specificate tramite **espressioni regolari**.

Esempio di grammatica per le espressioni algebriche

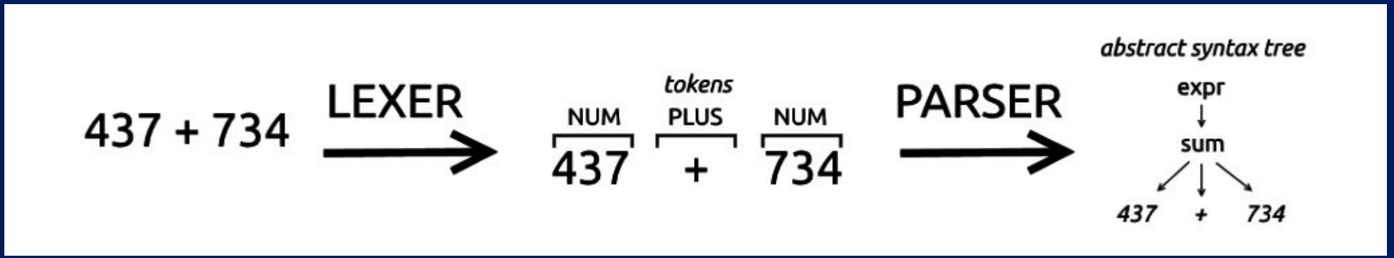
```
expr → NUM
     | '(' expr ')'
     | expr '+' expr
     | expr '-' expr
     | expr '*' expr
     | expr '/' expr

NUM  → [0-9]+ ( '.' [0-9]+ )
```

# Parser e Lexer - Overview

**Lexer**: *analizzatore sintattico, riconosce i tokens*.
Il primo passo in qualsiasi linguaggio di programmazione è leggere i singoli caratteri del codice sorgente di input e capire quali caratteri sono raggruppati.
In un linguaggio naturale, ciò includerebbe guardare le sequenze di lettere adiacenti per identificare le parole.
In un linguaggio di programmazione, gruppi di caratteri formano nomi di variabili, parole riservate o talvolta operatori o segni di punteggiatura composti da più caratteri.
Un lessema è una stringa di caratteri adiacenti che formano una singola entità.
Il pacchetto di informazioni che un linguaggio di programmazione raccoglie per ogni lessema che legge nel codice sorgente è chiamato **token**.

**Parser**: *analizzatore semantico*, prende i tokens, controlla la sintassi delle regole e costruisce l'AST




**Visitor**: attraversa l'AST e implementa la logica di attraversamento


# ANTLR v.4

# ANTLR v. 4 e sua installazione


https://www.antlr.org

**ANTLR (ANother Tool for Language Recognition)** è un potente generatore di parser per leggere, elaborare, eseguire o tradurre testo strutturato o file binari. È ampiamente utilizzato per creare linguaggi, strumenti e framework. Da una grammatica, ANTLR genera un parser in grado di costruire e percorrere alberi di analisi.

Scritto in **Java** (necessita JDK)

Installazione **ANTLR** e aggiunta PATH in Java CLASSPATH

Installazione Python Package **antlr4-python3-runtime**

Verifica installazione

```
java org.antlr.v4.Tool
```

Esempio di chiamata: `java org.antlr.v4.Tool -Dlanguage=Python3 Expr.g`
genera

```
Expr.interp
Expr.tokens
ExprLexer.interp
ExprLexer.py
ExprLexer.tokens
ExprListener.py
```

# ANTLR v. 4 - Storia e motivazioni


Autore: **Terence Parr** (professor of computer science at the University of San Francisco) ci lavora dal 1989

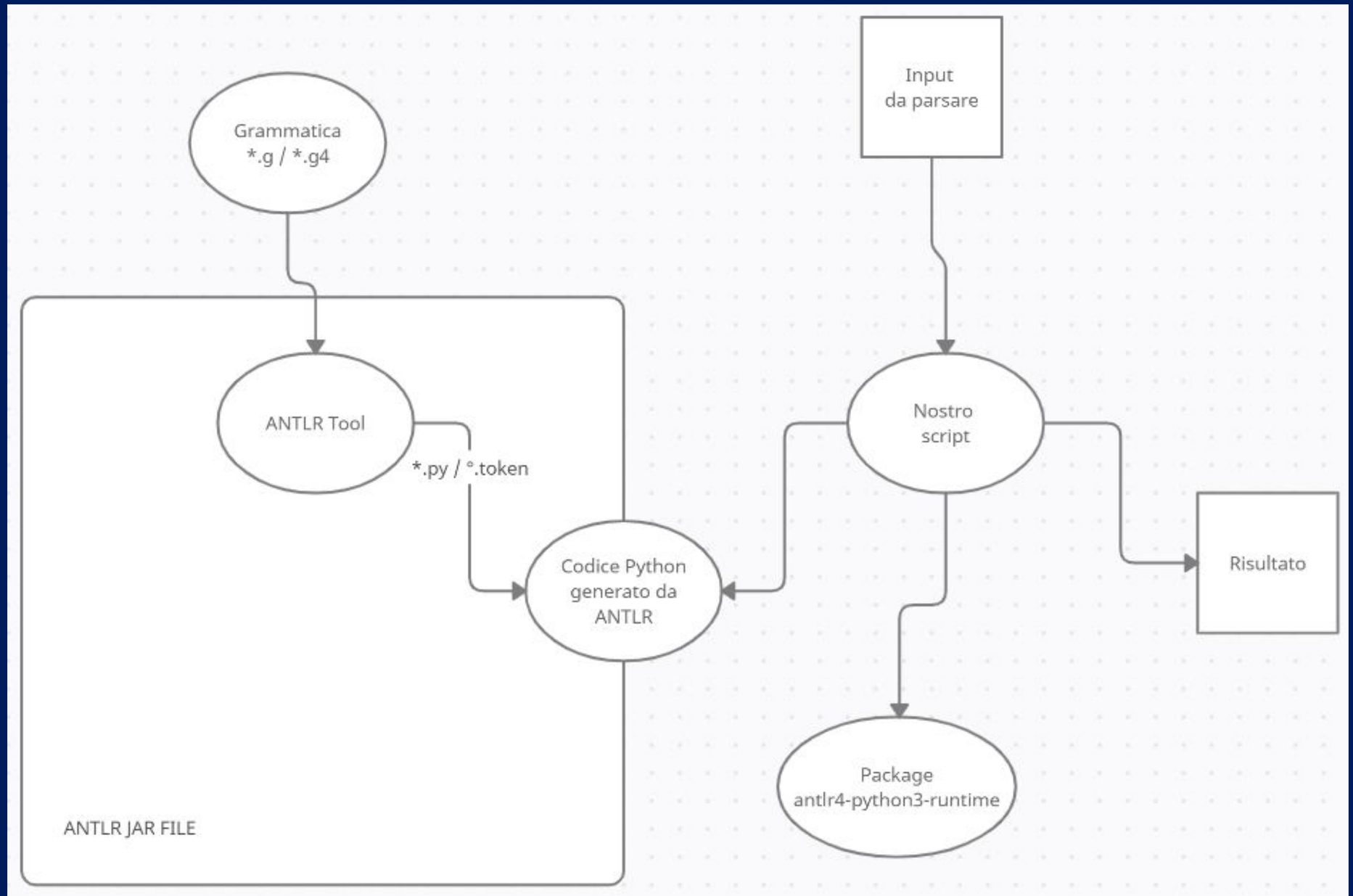
Ampiamente utilizzato nel mondo accademico e industriale (Twitter, Oracle, NetBeans)  per creare linguaggi, strumenti e framework.
*https://www.antlr.org/about.html*

**Da una descrizione formale del linguaggio chiamata grammatica, ANTLR genera un parser per quel linguaggio che può costruire automaticamente alberi di analisi, che sono strutture di dati che rappresentano il modo in cui una grammatica corrisponde all'input.**
**ANTLR genera inoltre automaticamente tree walker (secondo patterns *visitor* o *listener*) che è possibile utilizzare per visitare i nodi di tali alberi per eseguire codice specifico dell'applicazione.**

Puoi creare **strumenti utili** come lettori di file di configurazione, convertitori di codice legacy, renderer di markup wiki e parser JSON.

*“Ho creato piccoli strumenti per mappature di database relazionali a oggetti, descrivendo visualizzazioni 3D, iniettando codice di profilazione nel codice sorgente Java e ho persino realizzato un semplice esempio di corrispondenza di modelli DNA per una lezione.”*

# ANTLR v. 4 - Come funziona




# Scriviamo il primo interprete per sommare due numeri interi

# ANTLR4 - Expr.g - 1/2

**Expr**: semplice grammatica di esempio che gestisce la la somma di due numeri naturali (al momento non gestite virgola e altri segni)

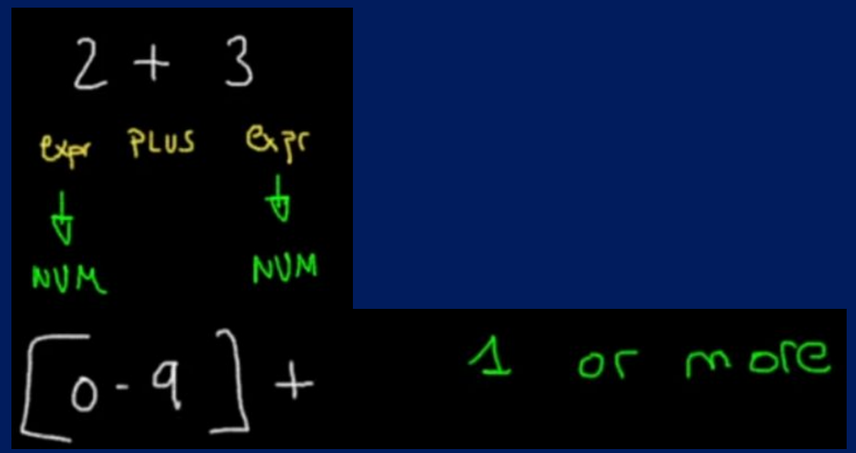
Filename = nome grammatica / skip = non gestione del “token”
root gestisce la fine del file

```
grammar Expr;

root: expr EOF;

expr: expr PLUS expr
    | NUM
    ;

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




La grammatica ha 2 **regole** e definisce 3 **tokens** (definiti in maiuscolo)

# ANTLR4 - Expr.g - 2/2

```
java org.antlr.v4.Tool -Dlanguage=Python3 -no-listener Expr.g
```

genera i seguenti files

```
Expr.interp  Expr.tokens
ExprLexer.interp  ExprLexer.py ExprLexer.tokens
ExprParser.py
```

Scripts per provare la grammatica e **visualizzare l'albero AST** con **diversi tipi di input**

> Una sola riga da consolle: `python expr_ast_1.py`

> N righe da consolle: `python expr_ast_2.py`

> Da File: `python expr_ast_3.py 3.txt`

> Da File con nome accentato: `python expr_ast_4.py 4è.txt`

# ANTLR4 - Visitors 1/2

```
java org.antlr.v4.Tool -Dlanguage=Python3 -no-listener -visitor Expr.g
```

genera il “nuovo file” `ExprVisitor.py` per la nostra grammatica Expr

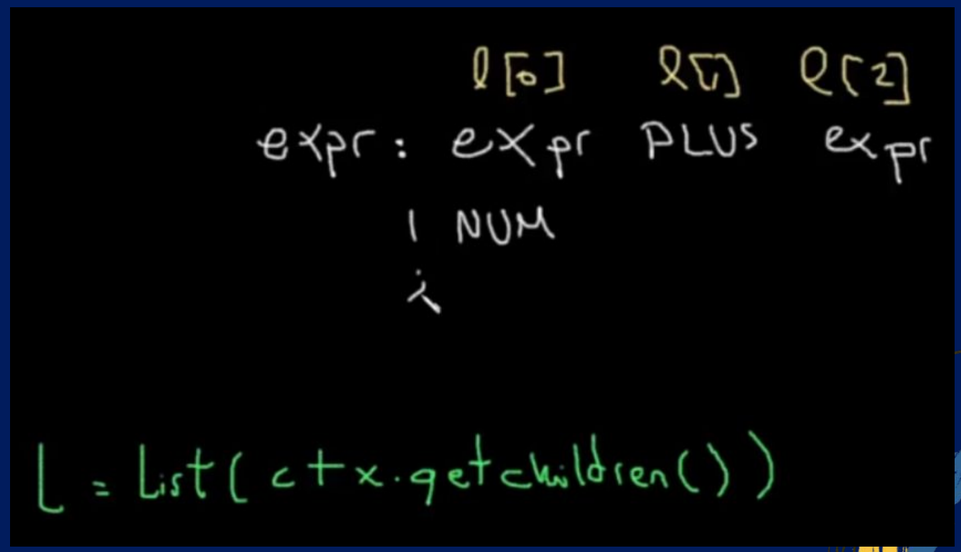
`VisitExpr` è la classe callback associata per visitare il nodo, ovvero specifica cosa fare quando si arriva al nodo.

Non toccheremo il file generato da ANTLR ma creeremo un **nuovo file**, ad esempio `EvalVisitor..py` che implementa una sottoclasse di `ExprVisitor` .

Nella grammatica c’erano 2 regole (`root` e `expr`) e allora implementeremo 2 metodi di visita ( `visitRoot` e `visitExpr` )

Script finale: `expr_eval.py`

Test unit script: `test_expr_eval.py`



# ANTLR4 - Visitors 2/2

## Access to the components on the right side of the rule:

- with the children: `ctx.getChildren()` (it is a generador)
  - `l = list(ctx.getChildren())` o
  - `op1, op, op2 = list(ctx.getChildren())`

## Other interesting methods:

- `n.getText()`: node text
- `ExprParser.symbolicNames[n.getSymbol().type]`: node token in text format
- `ExprParser.PLUS`: internal index of the PLUS token for the parser. It is usually used together with `n.getSymbol().type`

## Additional Information:

- we can exchange information with a visitor through the constructor `__init__` and making the *root rule* return something
  - Example of use: symbol table persistence between different *visitors*
- when a node belongs to the lexical part it contains the attribute `getSymbol` and when it belongs to the syntactic part the attribute `getRuleIndex`.