Python Biella Group

Serata 3 – Code Generation

Obiettivi

- Da interprete a compilatore
- Generazione con templating
- Generazione da AST target

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. Da Interprete a Compilatore

Da Interprete a Compilatore

La scorsa serata abbiamo visto la realizzazione di un **interprete.**

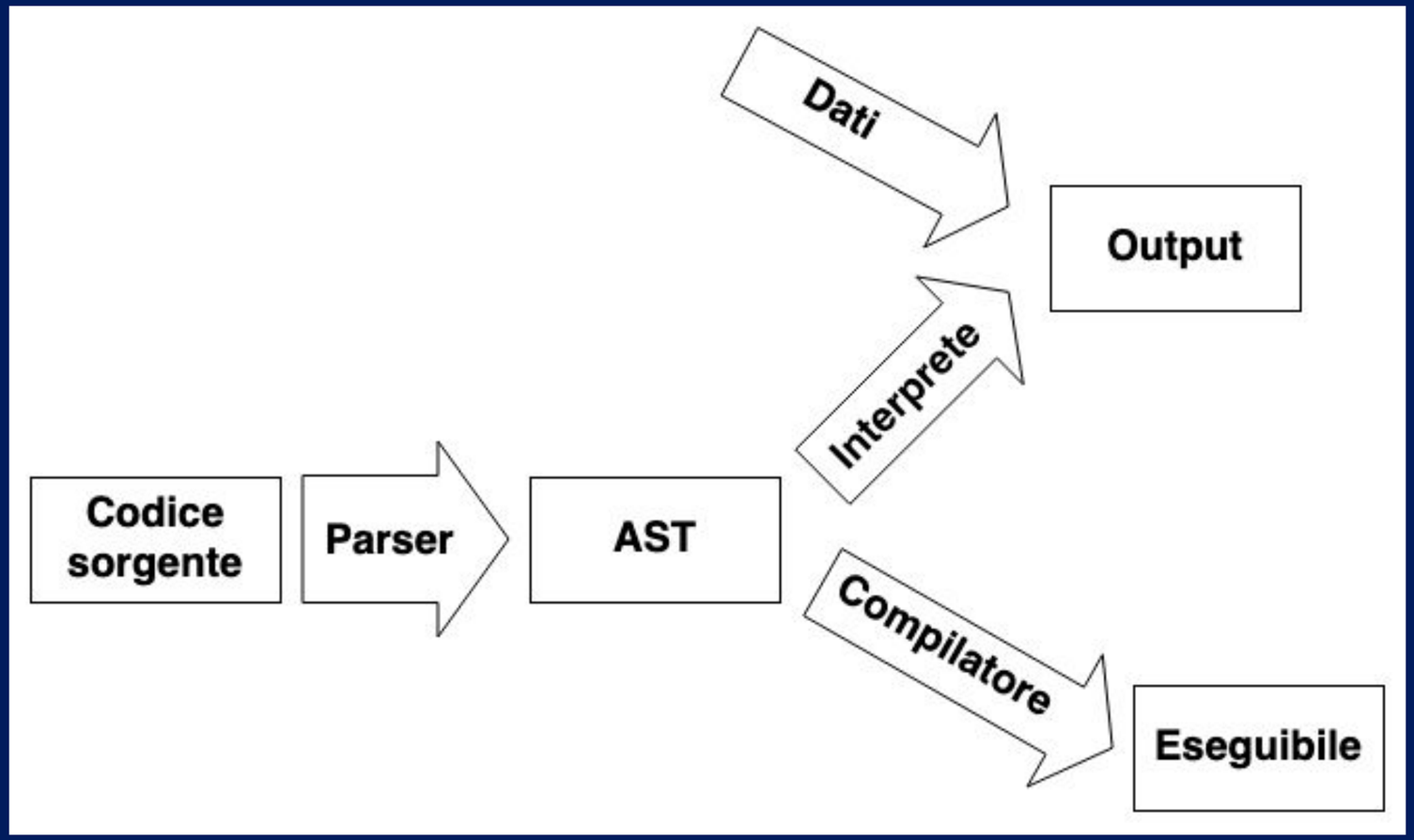
Tradizionalmente **interpreti** e **compilatori** sono due categorie di applicazioni ben distinte:

- Un interprete **esegue il programma “immediatamente”**
- Un compilatore **traduce il programma** in una forma che può essere **successivamente eseguita dal sistema operativo**

- **L’interprete è presente a runtime** insieme al programma
- Il compilatore **non accompagna il programma** durante la sua esecuzione

Radici comuni

In realtà, interpreti e compilatori hanno almeno alcune componenti in comune, banalmente il parser.



Non è tutto bianco o nero

Oltre al parser, **anche altre componenti** possono essere presenti sia in un interprete che in un compilatore.

Interprete

- Tradizionalmente, l'interprete è un processo leggero, con pochi passi di elaborazione
- Raramente interpreti "reali" si limitano ad attraversare l'AST in un unico passo per produrre un risultato nel minor tempo possibile
- Infatti, è possibile raffinare un interprete precalcolando alcune informazioni
- Più trasformazioni, controlli e ottimizzazioni avvengono in un interprete, più questo finisce per assomigliare ad un compilatore.

Compilatore

- Un compilatore può produrre **bytecode** ossia codice per una Virtual Machine
- Molti compilatori moderni sono "JIT" (just-in-time compiler) ossia **operano a runtime** a fianco del programma in esecuzione
- ⇒ molte ottimizzazioni possono avvenire a runtime invece che ahead-of-time

⇒ La distinzione interprete/compilatore non è nettissima.

Il caso di Python

Python è tradizionalmente considerato “un linguaggio interpretato”. In realtà, interpretato/compilato **non è una proprietà del linguaggio ma una proprietà dell'implementazione,** anche se il design del linguaggio influisce sulla facilità di interpretazione/compilazione.

Solitamente Python viene prima compilato in bytecode e poi immediatamente interpretato da una VM. Possiamo vedere come appare il bytecode con il modulo **dis** della libreria standard.

Con **CPython** o **Jython**, il bytecode viene indirizzato alla corrispondente macchina virtuale per l'interpretazione: PVM (Python Virtual Machine) o JVM (Java Virtual Machine).

Con **PyPy** invece 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/

Generazione di codice

Tradizionalmente la **code generation** è l'ultimo passo di un compilatore, nel quale si produce il codice eseguibile. In realtà può essere **assembly code** testuale o **machine code** binario. Come abbiamo detto il codice prodotto può essere anche **bytecode,** non immediatamente eseguibile.

Un compilatore può anche essere "source-to-source", anche detto **transpiler.**
Un transpiler non genera assembly, bytecode, o machine code, ma **codice sorgente in un linguaggio ad alto livello.**

Tradizionalmente abbiamo avuto numerosi compilatori che generavano codice C. Questo per la grande diffusione di buoni compilatori C.

Oggi abbiamo anche transpiler che generano codice che poi verrà interpretato.
Es. TypeScript → JavaScript.

Da Interprete a Transpiler

Obiettivo: **trasformare il nostro interprete** della scorsa serata **in un transpiler.**

L’output del transpiler sarà **codice Python equivalente** al comportamento dell’interprete.

Da Interprete a Transpiler

Primo passo: **rifattorizzare** l'interprete in modo da estrarre le **parti comuni** che ci possono servire anche nel compilatore.

In particolare:

- **Symbol resolution** associare ogni nome ad una definizione
- **Type computation** associare ad ogni espressione un tipo e verificare che i tipi siano corretti.

Nel ns interprete queste parti sono implementate in **metodi privati.** Possiamo **estrarre i metodi** e trasformarli in **funzioni in un modulo dedicato,** insieme alle classi che rappresentano i tipi.

Da Interprete a Transpiler

Il nostro interprete è composto da **due parti.**

Una si occupa di caricare la definizione delle entità. È un “caso degenere” di interprete, poco più di un parser, perché il linguaggio che definisce le entità è puramente descrittivo, senza comportamento associato.

La componente principale, l’interprete vero e proprio, **esegue** le istruzioni di uno script, **istanzia** le nostre entità e mantiene un **registro** delle istanze (lo **stato** della nostra applicazione).

Utilizzeremo le 2 componenti per mostrare due diverse tecniche di code generation.

2. Generazione di codice con template

Generazione di codice con template

Partiamo con un **approccio semplice** per generare codice Python per le nostre **definizioni di entità.** Per ogni entità genereremo una **dataclass** in Python.

Così come il nostro interprete si limita ad attraversare l'AST **eseguendo un comportamento** per ogni nodo, il nostro mini-transpiler attraverserà l'AST **applicando un template Jinja2** ad ogni nodo del linguaggio "entities". I template sono nel modulo "code_generators.templates".

```
env = Environment(
    loader=PackageLoader("code_generators"),
    autoescape=select_autoescape())
```

Generazione di codice con template

Possiamo utilizzare nuovamente gli **extension methods.** In questo modo arricchiamo la gerarchia delle classi esistenti con **nuovi metodi** per la generazione di codice. Uno “di default” per un generico Node che va a recuperare e renderizzare il template corrispondente:

```
@extension_method(Node)
def to_python(self: Node) -> str:
    template = env.get_template(f"{self.node_type.__name__.lower()}.txt")
    return template.render(node=self)
```

Ed uno per ogni **caso particolare** che devia dal default, come ad esempio:

```
@extension_method(EntityRefType)
def to_python(self: EntityRefType) -> str:
    return f'"{self.entity.name}"'
```

Fine. Il grosso della “logica” è nei template.

Vantaggi

- Facile “vedere” il codice generato dai template
- Possibilità di editare o sostituire i template anche da non esperti
- Utilizzo di tecnologie ben note con poca curva di apprendimento

Limiti nell’approccio

Applicare direttamente un template per ogni nodo dell’AST funziona in un caso semplice come questo. Tuttavia non consente spazio per elaborazioni più complesse. Per esempio:

- richiede una marcata corrispondenza tra linguaggio sorgente e linguaggio generato (es. una entità → una classe Python)
- richiede una traduzione lineare, con poche possibilità di “guardare indietro”. Come si vede per esempio nelle dichiarazioni di tipo (Person vs “Person”)

Possiamo applicare i template come **ultimo passo** dopo una serie di **trasformazioni dell’AST** per ricondurlo ad una forma sufficientemente semplice da poter applicare i template.

Nel seguito vedremo però un **approccio differente** anche se rimane il principio cardine di **attraversare e trasformare l’AST.**

3. Generazione di codice mediante trasformazione in AST esistente

3. Generazione di codice mediante trasformazione in AST esistente

AST Transformations

Uno dei vantaggi di avere metodologie e librerie standard per la trasformazione degli AST è che possiamo facilmente **integrare ed utilizzare componenti esistenti** nella nostra pipeline. Possono essere come in questo caso **componenti di terze parti** oppure componenti che possiamo riutilizzare da progetti precedenti (economia di scala di Strumenta).

Fortunatamente per Python **esistono diverse librerie** che consentono di **costruire e manipolare AST** che rappresentano codice Python e, da questi, **generare il codice** corrispondente.

Quindi, per tradurre il linguaggio “script” di istanziazione delle entità”, useremo il seguente approccio:

Source ⇒ AST per “script” ⇒ AST per Python ⇒ Target (codice Python)

Come avviene l’ultima fase della generazione **non ci interessa,** fa parte della libreria che andremo ad utilizzare. Può essere programmatico o tramite template.

AST per Python

Python include un modulo “ast” https://docs.python.org/3/library/ast.html

Possiamo usarlo per “parsare” codice Python, da stringa ad AST.

Possiamo usarlo anche per fare “unparsing” ossia da AST a stringa, in altre parole code generation!

Tuttavia, essendo un AST, **non include informazioni sintattiche** come commenti, linee spezzate ecc.

AST per Python/2

Abbiamo però a disposizione librerie open source che implementano un “CST” o concrete-syntax tree: un AST arricchito con informazioni di sintassi.

Nota CST =/= parse tree! La struttura del CST è la stessa struttura “logica” dell’AST.

Due esempi di CST:
- RedBaron https://github.com/PyCQA/redbaron
- Instagram’s libcst https://libcst.readthedocs.io/en/latest/

Hanno alcune differenze:
- Mutability
- Smart collections
- Ovviamente, API un po’ diverse
- **naming** di nodi e proprietà
- Supporto

Scegliamo libcst (ma invitiamo a provare anche RedBaron)

Generator con libcst

- **Manteniamo la struttura** del nostro interprete
- Ai fini di una demo, implementiamo solo il create statement
  - gli altri sono lasciati come esercizio :)
- Abbiamo bisogno di un **runtime:** codice a supporto (ad es. per il registro delle entità).
  - Nell'interprete esso era **implicito:** ero lo stato dell'interprete.
  - Nel code generator, il runtime è una **dipendenza** del codice generato
  - Nel nostro caso semplice, fa parte dello stesso progetto
    - Possiamo usare la dinamicità di Python a nostro vantaggio per caricare runtime e codice generato nello stesso processo Python
  - Quanta logica mettere nel runtime e quanta nel codice generato?

Show me the Code!

Concludiamo

Il codice mostrato è su GitHub: https://github.com/PythonBiellaGroup/ANTLR
branch “serata3” (il branch main conterrà il codice completo alla fine delle 3 serate)

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

- Pulizia (linting) del codice generato
- “Source mapping”: tracciatura della corrispondenza tra codice sorgente e codice generato
- Sistemi tipo macro o AST transformations user-defined

Domande?

Grazie!