Linguaggi di programmazione

Autori
Silvio Peroni – silvio.peroni@unibo.it
Dipartimento di Filologia Classica e Italianistica, Università di Bologna, Bologna, Italia

Aldo Gangemi – aldo.gangemi@unibo.it
Dipartimento di Filologia Classica e Italianistica, Università di Bologna, Bologna, Italia

Avviso sul copyright
Questo lavoro è rilasciato con licenza Creative Commons Attribution 4.0 International License. Tu sei libero di condividere (riprodurre, distribuire, comunicare al pubblico, esporre in pubblico, rappresentare, eseguire e recitare questo materiale con qualsiasi mezzo e formato) e modificare (remixare, trasformare il materiale e basarti su di esso per le tue opere per qualsiasi fine, anche commerciale) questo lavoro alle seguenti condizioni: attribuzione, ovvero devi riconoscere una menzione di paternità adeguata, fornire un link alla licenza e indicare se sono state effettuate delle modifiche. Puoi fare ciò in qualsiasi maniera ragionevole possibile, ma non con modalità tali da suggerire che il licenziante avalli te o il tuo utilizzo del materiale. Il licenziante non può revocare questi diritti fintanto che tu rispetti i termini della licenza.

Sommario

In questo capitolo verrà fornita una definizione generale di *linguaggio di programmazione*, per poi mostrare degli esempi di linguaggi visuali che possono essere usati per implementare veri e propri programmi.

Che cos'è un linguaggio?

Un linguaggio "naturale" è un linguaggio comune, come l'italiano, che può essere scritto o orale, e che si è evoluto in maniera naturale all'interno di una comunità. Da come li conosciamo, i linguaggi naturali hanno il vantaggio (e, contemporaneamente, lo svantaggio) di essere così tanto espressivi che specifiche istruzioni comunicate attraverso il loro uso possono sembrare ambigue. Consideriamo, per esempio, la frase "la vecchia porta la sbarra". Questa frase significa che c'è una vecchia porta che blocca qualcuno o che c'è una signora anziana che trasporta una sbarra? Tuttavia, spesso, noi riusciamo a disambiguare il significato di una frase mediante l'uso di convenzioni sociali o analizzando il contesto in cui l'azione avviene, in modo da restringere il possibile significato di un pezzo di informazione. Mentre i linguaggi naturali non sono formali per definizione, molti studi in linguistica cercano di fornirne una formalizzazione mediante l'uso di strumenti matematici [Bernardi, 2002].

Una delle figure più importanti dietro alla formalizzazione del linguaggio naturale è Noam Chomsky (mostrato in Figura 1). Chomsky è uno dei più importanti accademici degli ultimi cento anni e uno dei padri della linguistica moderna, insieme a Ferdinand de Saussure, Lucien Tesnière, Luis Hjelmslev, Zellig Harris, Charles Fillmore, etc. Le sue ipotesi di ricerca principali sul linguaggio umano sono: (1) la sua struttura sintattica di base è rappresentabile mediante una teoria matematica e (2) tale struttura è determinata biologicamente in tutti gli umani – ovvero, è già presente in noi sin dalla nascita – e, come tale, è una caratteristica unica che si è evoluta nel tempo e che è condivisa soltanto dagli umani e non da altri animali.

Figura 1. Una fotografia di Chomsky scattata nel 2011.
Foto di Andrew Rusk, sorgente:
https://en.wikipedia.org/wiki/Noam_Chomsky#/media/File:Noam_Chomsky_Toronto_2011.jpg.

Questa sua visione del linguaggio umano è molto dibattuta, ma ha avuto il merito di fondare su basi rigorose la ricerca sulle relazioni tra competenza e produzione linguistica, sulle dinamiche e le interazioni tra capacità "innate" e sviluppo cognitivo e sociale delle abilità linguistiche. Ha anche permesso di chiarire in parte distinzioni come sintassi vs. semantica, lessico vs. discorso, tra rappresentazione logica e rappresentazioni mentali, etc.

Tra la sua produzione accademica troviamo la classificazione delle grammatiche formali in una gerarchia di crescente potere espressivo, che rappresenta uno dei suoi più importanti contributi, considerando il suo impatto anche al di fuori della linguistica tradizionale – per esempio nel dominio dell'informatica teorica e nei linguaggi di programmazione, come vedremo in seguito. Una grammatica formale è uno strumento matematico usato per definire la sintassi di un linguaggio (sia lingue naturali come l'italiano, sia linguaggi artificiali) attraverso l'uso di un insieme finito di regole di produzione, che permettono di costruire una qualunque frase valida in quello specifico linguaggio.

Una grammatica formale è composta da un insieme di regole di produzione di forma `premessa ::= espressione` (in notazione Backus–Naur, or BNF), dove la premessa e l'espressione possono contenere uno o più simboli delle seguenti tipologie:

- simbolo *terminale* (specificato tra virgolette in BNF), che identifica tutti i simboli elementari del linguaggio in considerazione (come nomi, verbi, etc.);
- simbolo *non terminale* (specificato tra parentesi angolari in BNF), che identifica tutti i simboli di una grammatica formale che possono essere sostituiti da una combinazione di simboli terminali e non terminali.

In linea di principio, l'applicazione di una regola di produzione riguarda la sostituzione dei simboli nella premessa con quelli specificati nell'espressione finché non si raggiunge una sequenza che include soltanto simboli terminali. Per esempio, le regole di produzione <frase> ::= "Io" <verbo>, <verbo> ::= "scrivo" e <verbo> ::= "leggo" permette la creazione di tutte le frasi di due parole che hanno il pronome di prima persona singolare accompagnato da uno dei verbi possibili (ad esempio "Io scrivo"). In più, ogni grammatica formale deve specificare un *simbolo di inizio*, che deve essere non terminale.

La gerarchia proposta da Chomsky mette a disposizione un modo per descrivere formalmente le relazioni che possono esistere tra diverse grammatiche in termini delle possibili strutture sintattiche che sono in grado di generare. In pratica, queste tipologie sono caratterizzate dal tipo di simboli che possono essere usati nella premessa e nell'espressione delle regole di produzione. Queste tipologie di grammatica sono elencate di seguito, dalla meno espressiva alla più espressiva[1]:

- *grammatiche regolari* – forma delle regole di produzione:
  - <non-terminale> ::= "terminale"
  - <non-terminale> ::= "terminale" <non-terminale>
  - Esempio:
    ```
    <frase> ::= "Io" <verbo>
    <verbo> ::= "scrivo"
    <verbo> ::= "leggo"
    ```
- *grammatiche libere dal contesto* – forma delle regole di produzione:
  - <non-terminale> ::= γ
  - Esempio:
    ```
    <frase> ::= <pronome> <sintagma-verbale>
    <pronome> ::= "Io"
    <nome> ::= "libro"
    <nome> ::= "documento"
    <sintagma-verbale> ::= <verbo>
    <sintagma-verbale> ::= <verbo> "un" <nome>
    <verbo> ::= "scrivo"
    <verbo> ::= "leggo"
    ```

[1] Vengono usate le lettere dell'alfabeto greco per indicare una qualsivoglia possibile combinazione di simboli terminali e non terminali, incluso il simbolo terminale vuoto, solitamente rappresentato con ε.

- *grammatiche dipendenti dal contesto* – forma delle regole di produzione:
    - α <non-terminale> β ::= α γ β
    - Esempio:

```
<frase> ::= <pronome-soggetto> <sintagma-verbale>
"Io" <pronome-oggetto> <verbo> ::= "Io" <pronome-oggetto> "amo"
"Io" <verbo> <nome> ::= "Io" "leggo" "un" <nome>
<sintagma-verbale> ::= <verbo> <nome>
<sintagma-verbale> ::=  <pronome-oggetto> <verbo>
<pronome-soggetto> ::= "Io"
<pronome-oggetto> ::= "ti"
<pronome-oggetto> ::= "vi"
<nome> ::= "libro"
<nome> ::= "documento"
```

- *grammatiche ricorsivamente enumerabili* – forma delle regole di produzione:
    - α ::= β (nessuna restrizione)
    - Esempio:

```
<frase> ::= <pronome-soggetto> <sintagma-verbale>
"Io" <pronome-oggetto> <verbo> ::= "Io" "ci" "vedo"
"Io" <verbo> <nome> ::= "Io" "leggo" "un" "libro"
<sintagma-verbale> ::= <verbo> <nome>
<sintagma-verbale> ::= <pronome-oggetto> <verbo>
<pronome-soggetto> ::= "Io"
<pronome-oggetto> ::= "mi"
<pronome-oggetto> ::= "ti"
<verbo> ::= "amo"
<verbo> ::= "odio"
```

I linguaggi di programmazione

C'è un particolare aspetto di ogni computer (sia esso umano o macchina) che non è stato ancora affrontato direttamente, ovvero: quale meccanismo possiamo usare per chiedere a un computer di eseguire una particolare attività? La modalità per affrontare questo problema è estremamente connessa con il particolare canale comunicativo che vogliamo adottare. Se consideriamo un computer umano, possiamo usare un linguaggio naturale (ad esempio l'italiano) o i diagrammi di flusso per istruirlo sui passi algoritmici che deve compiere. Invece, per comunicare efficacemente con un computer elettronico si usano i *linguaggi di programmazione*.

Un linguaggio di programmazione è un linguaggio formale che obbliga l'uso di specifiche regole sintattiche sviluppate in modo tale da evitare possibili istruzioni ambigue – solitamente restringendo l'espressività del linguaggio – cosicché tutte le "frasi" componibili possano trasmettere un solo possibile significato. I linguaggi di programmazione sono solitamente basati

su grammatiche libere dal contesto, in conformità con la classificazione Chomskiana fornita nella sezione precedente – e possono distinguersi per un basso o elevato livello di astrazione dal linguaggio propriamente in uso da un elaboratore elettronico per eseguire le operazioni. In particolare, possiamo raggruppare i linguaggi di programmazione in tre macro insiemi:

- il linguaggio macchina è un insieme di istruzioni che possono essere eseguite direttamente dalla CPU (*central processing unit*, o *processore*) di un computer elettronico[2]. Per esempio, il codice seguente è un codice binario eseguibile (composto da una sequenza di *0* e *1*) che definisce una funzione (ovvero, uno strumento che prende qualche input e produce un qualche output, e che di fatto implementa un algoritmo in un certo linguaggio di programmazione) per calcolare l'n-esimo numero di Fibonacci:
```
1000101101010100001001000000010001000001111111010000000000111011
1000001101011100000000000000000000000000000000000110000011100000
1111111010000000100111011100000110101110000000000001000000000000
0000000000011000011010100111011101100000001000000000000000000000
0000101110010000000100000000000000000000000001000110100000100000
1100110000011111110100000001101110110000001111000101111101100110
001001110000010100101011101011111100010101101111000011
```
- *i linguaggi di programmazione a basso livello* sono linguaggi che forniscono un livello di astrazione sopra il linguaggio macchina, e che permettono di scrivere programmi in modo che siano un pochino più intellegibili dagli umani. Il più famoso linguaggio di questo tipo è l'Assembly. Anche se introduce simboli più comprensibili, di solito una linea di codice in Assembly rappresenta una specifica istruzione in linguaggio macchina. Per esempio, la funzione per calcolare l'n-esimo numero di Fibonacci introdotta precedentemente può essere definita in Assembly come segue:

```
fib:
    mov edx, [esp+8]
    cmp edx, 0
    ja @f
    mov eax, 0
    ret

    @@:
    cmp edx, 2
    ja @f
    mov eax, 1
    ret
```

[2] La CPU rappresenta il cuore di funzionamento di un computer elettronico, ed è quel componente che esegue esplicitamente le istruzioni specificate in un qualsiasi programma in esecuzione.

```
@@:
push ebx
mov ebx, 1
mov ecx, 1

@@:
    lea eax, [ebx+ecx]
    cmp edx, 3
    jbe @f
    mov ebx, ecx
    mov ecx, eax
    dec edx
jmp @b

@@:
pop ebx
ret
```

- *i linguaggi di programmazione ad alto livello* sono quei linguaggi caratterizzati da un forte livello di astrazione dal linguaggio macchina. In particolare, possono usare parole proprie del linguaggio naturale per definire costrutti specifici, così da essere di più facile comprensione per un umano. A livello generale, più astrazione da un linguaggio di programmazione a basso livello è fornita, più comprensibile è il linguaggio. Per esempio, nell'estratto che segue mostriamo come usare il linguaggio di programmazione Python per implementare la stessa funzione introdotta in precedenza:

```
def fib(n):
    if n <= 0:
        return 0
    elif n <= 2:
        return 1
    else:
        a = 1
        b = 1
        while True:
            c = a + b
            if n <= 3:
                return c
            a = b
            b = c
            n = n - 1
```

Grace Brewster Murray Hopper (mostrata in Figura 2) è tra i più grandi pionieri dei linguaggi di programmazione. È stata un'informatica e il primo programmatore dell'Harvard Mark I, che era un computer elettromeccanico *general-purpose* usato durante la seconda guerra mondiale e

interamente ispirato alla macchina analitica di Babbage. Grace Hopper era fermamente convinta della necessità di avere linguaggi di programmazione che fossero indipendenti dalle macchine su cui erano utilizzati, che l'ha portata allo sviluppo del COBOL, uno dei primi linguaggi di programmazione ad alto livello che è tuttora usato per alcune applicazioni industriali.

Figura 2. Ritratto di Grace Hopper.
Foto di James S. Davis, sorgente:
https://en.wikipedia.org/wiki/File:Commodore_Grace_M._Hopper,_USN_(covered).jpg.

COBOL (*common business-oriented language*) è un linguaggio di programmazione sviluppato per uso industriale che utilizza largamente termini propri alla lingua inglese per descrivere le varie istruzioni dei programmi. L'idea di adottare, per la prima volta, comandi in lingua inglese ha reso il linguaggio un po' più verboso ma anche molto più leggibile e chiaro. Per fare un esempio, nei linguaggi di programmazione odierni se vogliamo definire la condizione di un costrutto condizionale (ovvero, l'equivalente di un un oggetto grafico decisionale nei diagrammi di flusso), ad esempio controllare se il valore assegnato ad una certa variabile `x` è maggiore di

quello assegnato alla variabile `y`, si usano i simboli matematici per le uguaglianze, come `x > y`. In COBOL, la stessa condizione è esprimibile dall'istruzione `x IS GREATER THAN y`.

Dopo la seconda guerra mondiale, sono stati creati diversi linguaggi di programmazione, con caratteristiche specifiche a seconda dei principi di sviluppo e dell'uso che se ne doveva fare – ad esempio, dipendentemente dai problemi computazionali per cui venivano usati. Mentre tutti questi linguaggi, in principio, permettono lo sviluppo di soluzioni per un qualsiasi problema computazionale, alcuni di questi sono più appropriati per determinati domini rispetto ad altri. Per esempio, il COBOL, come già anticipato, è stato sviluppato per la creazione di applicazioni industriali, mentre il FORTRAN è stato sviluppato per gestire al meglio la computazione di dati scientifici.

Mentre un'analisi accurata di tutti i linguaggi di programmazione è fuori degli obiettivi del corso, in Figura 3 sono mostrati i principali linguaggi di programmazione dal 1954 al 2017.

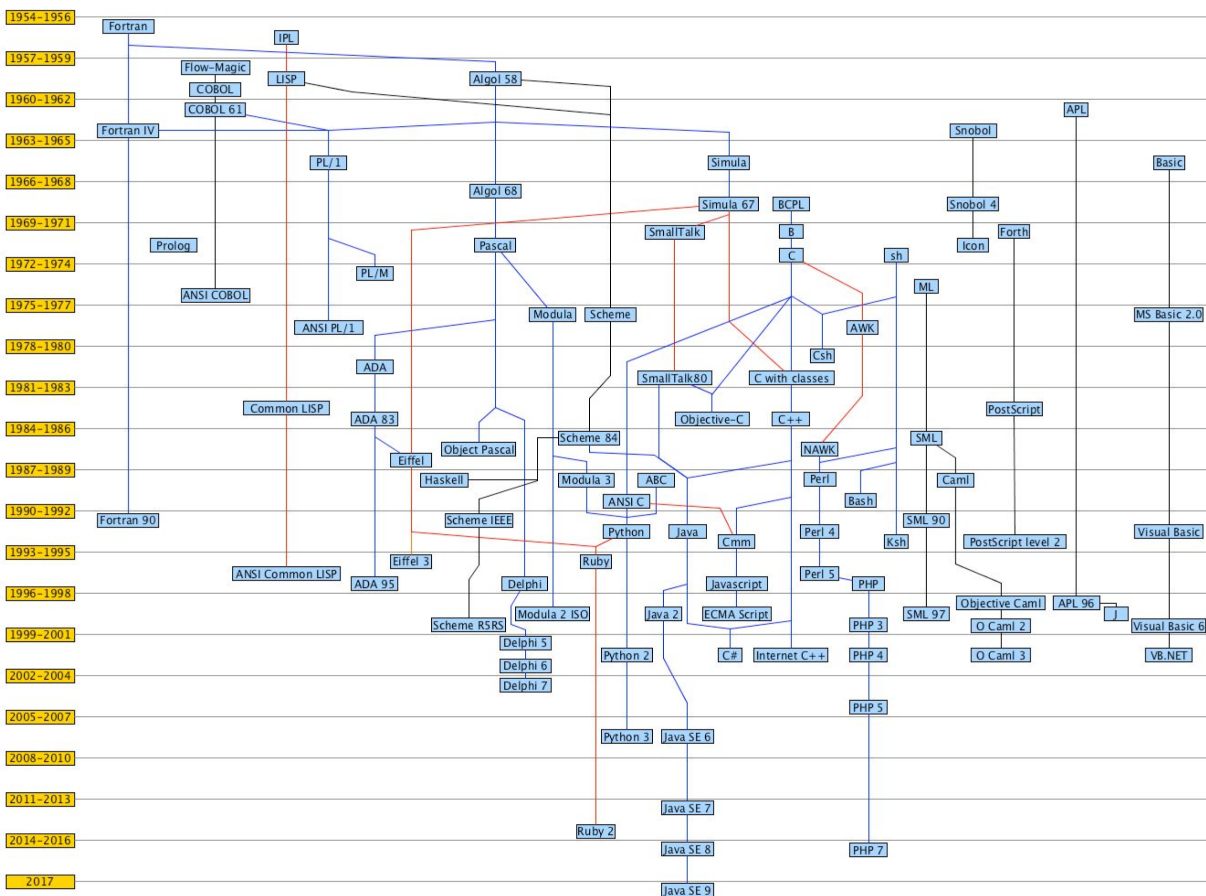

Figura 3. Una *timeline* grafica che riassume i principali linguaggi di programmazione creati dal 1954 al 2017. Mentre le linee delineano le relazioni di parentela tra un linguaggio e un altro, i colori usati sono stati scelti per garantire una più efficace leggibilità e non hanno alcun significato particolare.

Linguaggi visuali

In modo da facilitare l'avvicinamento all'uso dei linguaggi di programmazione tradizionali, negli ultimi anni sono stati sviluppati diversi linguaggi di programmazione visuali che permettono lo sviluppo di piccoli programmi per risolvere problemi computazionali specifici. Alcuni di questi, proposti sotto forma di gioco, permettono l'introduzione dei costrutti principali propri dei linguaggi di programmazione mediante l'utilizzo di oggetti grafici.

Uno delle proposte più interessanti degli ultimi anni è stata avanzata da Google in occasione del cinquantennale dell'introduzione dei primi linguaggi di programmazione per bambini. Il 4 dicembre 2017, Google ha messo a disposizione un *doodle* in cui si deve istruire un coniglio in modo che riesca a mangiare tutte le carote posizionate su un determinato percorso. Le azioni che si possono far compiere al coniglio riguardano attività di movimento, ad esempio "vai avanti" o "gira a destra", e devono essere disposte in una specifica sequenza in modo che il problema computazionale *mangiare tutte le carote presenti sul percorso* venga risolto efficacemente. Ad esempio, in Figura 4, viene mostrata una delle schermate iniziali del gioco proposto. Seppur possa sembrare semplice ad una prima analisi, il quantitativo di nozioni che il gioco veicola e permette di apprendere è estremamente allineato con quelle che tipicamente contraddistinguono i linguaggi di programmazione. Per cui, giocando a questo gioco, si imparano implicitamente e, da un certo punto di vista, in modo completamente inconsapevole, dei concetti chiave propri alla programmazione e al pensiero computazionale.

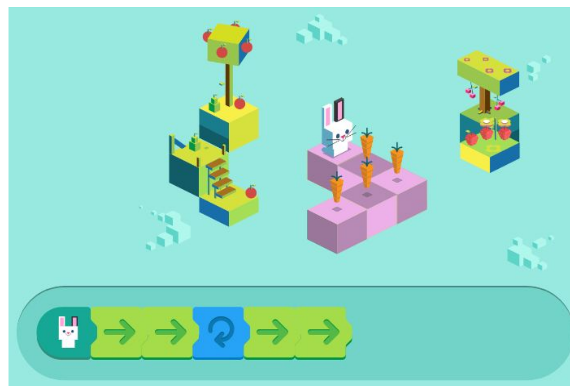

Figura 4. Uno screenshot del doodle rilasciato da Google in cui si deve istruire un coniglio di modo che possa mangiare tutte le carote presenti nel percorso. La sequenza di azioni da compiere per risolvere questo problema computazionale è qui limitata alle sole azioni "vai avanti" e "gira a destra", esplicitate da due tipologie di oggetti grafici distinti, rispettivamente una freccia dritta verde e una freccia arcuata blu.

La sequenza di istruzioni specificate per risolvere il problema computazionale proposto di fatto implementano uno specifico algoritmo. Utilizzando i diagrammi di flusso, introdotti nel capitolo

precedente, è possibile specificare lo stesso insieme di istruzioni usando un linguaggio più formale, come mostrato in Figura 5.

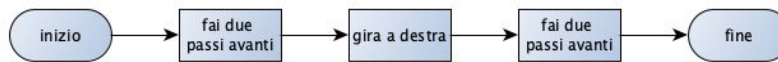

Figura 5. Un diagramma di flusso che rappresenta la sequenza di istruzioni indicate in Figura 4 per permettere al coniglio di mangiare tutte le carote. In questo caso, azioni simili successive sono raggruppate nel medesimo blocco di processo (ad esempio “fai due passi avanti” invece che “vai avanti” e “vai avanti”).

Mano a mano che si va avanti nel gioco, il numero di azioni che si devono far compiere al coniglio diventa più elevato. Per esempio, in Figura 6 la sequenza di azioni da compiere non è visualizzata interamente nella finestra di gioco.

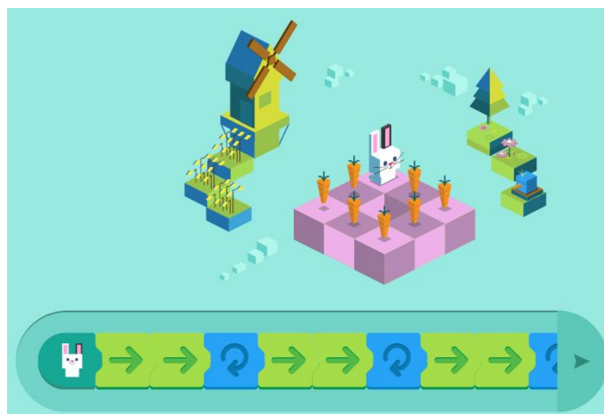

Figura 6. Uno scenario più complesso di quello mostrato in Figura 4. In particolare, la sequenza di operazioni che permettono al coniglio di mangiare tutte le carote presenti sul percorso è estremamente più estesa, seppur abbastanza ripetitiva.

In modo da semplificare la specificazione di queste lunghe sequenze di azioni, il gioco propone costrutti addizionali mano a mano che si va avanti nei vari livelli di difficoltà. In particolare, in Figura 7, viene mostrata come eseguire una sequenza di azioni, per risolvere il problema computazionale, mediante l’utilizzo di un nuovo oggetto grafico (o costrutto) che permette di raggruppare sequenze ripetitive di azioni e di eseguirle un numero definito di volte, ovviando al problema di indicarle tutte per esteso come fatto in Figura 6.

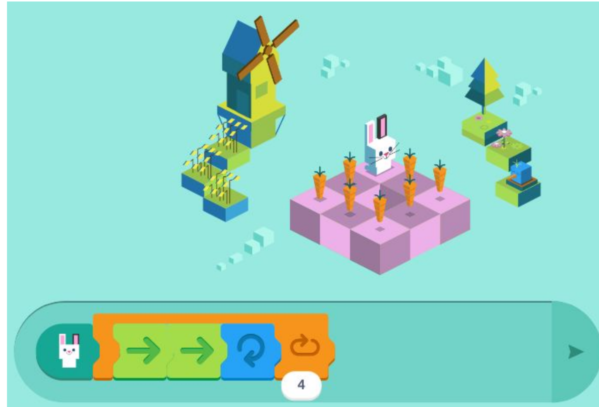

Figura 7. L'utilizzo di un nuovo oggetto grafico, indicato dalla freccia circolare arancione, che permette di eseguire una particolare sequenza di operazioni in esso contenute (ovvero "vai avanti", "vai avanti", "gira a destra") tante volte quanto è indicato dal numero che lo accompagna, in questo caso 4.

Anche questo algoritmo prettamente grafico può essere rappresentato mediante l'utilizzo di un diagramma di flusso, come mostrato in Figura 8. In questo caso, la ripetizione è realizzata attraverso l'utilizzo di un oggetto grafico decisionale che controlla se la sequenza di azioni "fai due passi avanti" e "gira a destra" è stata eseguita il numero necessario di volte (4) per risolvere il problema computazionale, e così concludere l'algoritmo.

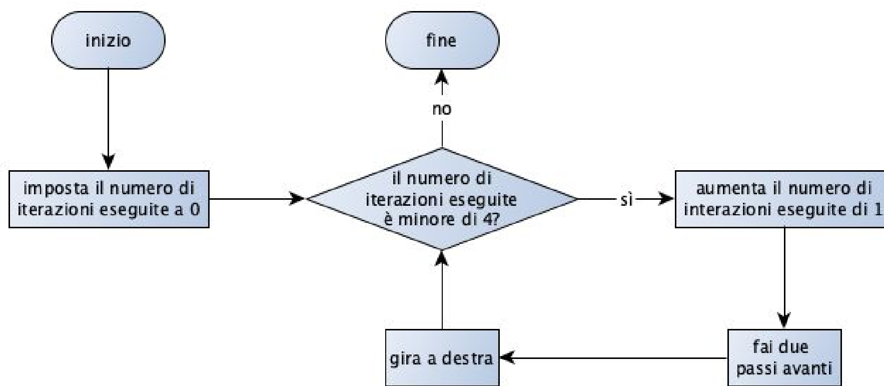

Figura 8. Il diagramma di flusso che descrive i passi necessari per descrivere la sequenza di operazioni, con ciclo iterativo, descritta in Figura 7.

Il linguaggio di programmazione visuale proposto finora è stato sviluppato specificamente per risolvere un particolare problema computazionale, far mangiare tutte le carote presenti su un percorso ad un coniglio. Ovviamente, esistono anche altri linguaggi di programmazione visuale che sono più *general-purpose*, ovvero che permettono di sviluppare algoritmi per risolvere problemi computazionali di vario genere. Uno di questi è Blockly, creato da Google. Questo linguaggio mette a disposizione diversi costrutti propri dei linguaggi di programmazione tradizionali, ma li propone sotto forma di blocchetti che si possono incastrare uno sull'altro o uno dentro l'altro per definire le sequenze di operazioni di un algoritmo.

Figura 9. L'implementazione in Blocky della sequenza di azioni mostrate in Figura 4. In questo caso, le azioni sono organizzate all'interno di una lista, una particolare struttura dati introdotta nel primo capitolo del corso.

In Figura 9 è introdotto un algoritmo in Blocky che costruisce la sequenza di azioni descritte in Figura 4, inserendole dentro una lista nell'ordine in cui vanno fatte eseguire al coniglio. In questo caso, viene inizializzata una variabile `istruzioni` con una lista vuota che, nelle operazioni successive, viene riempita con le operazioni che devono essere eseguite dal coniglio. In modo da mantenere il giusto ordine delle azioni della sequenza, ogni nuova istruzione viene inserita alla fine della lista.

In modo del tutto simile all'algoritmo introdotto nel diagramma di flusso in Figura 8, che a sua volta implementa la sequenza di azioni presentate in Figura 7, è possibile utilizzare specifici costrutti di Blocky per permettere ripetizioni sistematiche di un gruppo di istruzioni. Per esempio, in Figura 10 viene mostrato l'utilizzo del costrutto `repeat while <condition> do`, che permette l'esecuzione di una sequenza di operazioni in esso contenute fintanto che la condizione del costrutto è vera.

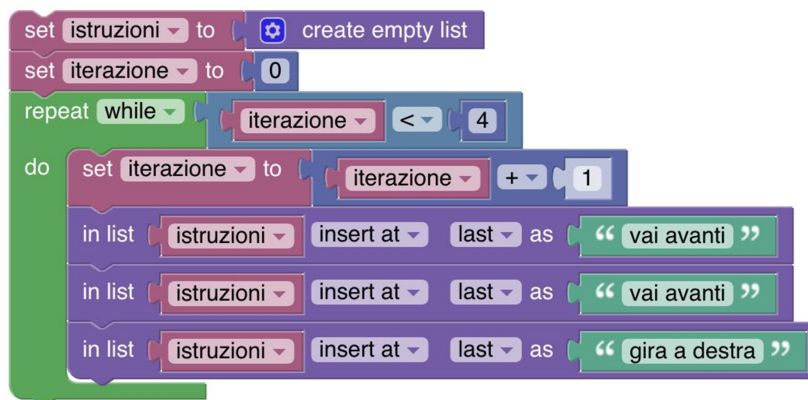

Figura 10. L'implementazione in Blocky della sequenza di azioni mostrate in Figura 7. In questo caso, le azioni vengono inserite all'interno della lista mediante un costrutto che permette la ripetizione di tre specifiche operazioni fintanto che la condizione `iterazione < 4` è vera.

Oltre alla possibilità di creare graficamente un algoritmo, Blocky mette anche a disposizione una sorta di traduttore che permette di descrivere i vari passi dell'algoritmo implementato in uno tra

cinque diversi linguaggi di programmazione. Per esempio, il Listato 1 mostra la traduzione in Python delle istruzioni presentate in Figura 10.

```
istruzioni = list()
iterazione = 0
while iterazione < 4:
    iterazione = iterazione + 1
    istruzioni.append("avanti")
    istruzioni.append("avanti")
    istruzioni.append("gira a destra")
```

Listato 1. L'implementazione in Python dell'algoritmo descritto in Blocky mostrato in Figura 10.

Python è un linguaggio di programmazione ad alto livello *general-purpose*, ed è correntemente uno dei linguaggi più usati per il Web e per attività di analisi automatiche di dati e del linguaggio naturale. Tra i vari vantaggi, Python è uno dei linguaggi più semplici con cui iniziare a studiare come programmare e creare applicazioni. Seppur in questo corso non verranno approfonditi i principi della programmazione in Python, sul Web ci sono a disposizione tantissime risorse gratuite (principalmente in inglese) che permettono uno studio approfondito del linguaggio, in particolare:

- Il libro introduttivo *Dive into Python 3* [Pilgrim, 2009];
- La documentazione ufficiale del linguaggio;
- Una piattaforma online per giocare con Python 3 senza installare alcun software sul proprio computer;
- Un corso interattivo per imparare Python da zero;
- Un altro libro interamente dedicato alla risoluzione di problemi e algoritmi sviluppati in Python [Miller and Ranum, 2011];
- Un libro digitale che contiene un'introduzione a Python per le Scienze Umane.

Ringraziamenti

Gli autori ringraziano Eugenia Galli per le preziose correzioni proposte al testo.

Bibliografia

Bernardi, R. (2002). The Logical Approach in Linguistics. In Reasoning with Polarity in Categorial Type Logic. Ph. D. Thesis, Utrecht University. http://disi.unitn.it/~bernardi/Papers/thesis-chapter1.pdf (ultimo accesso 15 Febbraio 2019)

Miller, B. N., Ranum, D. L. (2011). Problem Solving with Algorithms and Data Structures using Python. ISBN: 978-1590282571. Liberamente disponibile all'URL

https://runestone.academy/runestone/static/pythonds/index.html (ultimo accesso 15 Febbraio 2019)

Pilgrim, M. (2009). Dive into Python 3. ISBN: 978-1430224150. Liberamente disponibile all'URL http://histo.ucsf.edu/BMS270/diveintopython3-r802.pdf (ultimo accesso 15 Febbraio 2019)