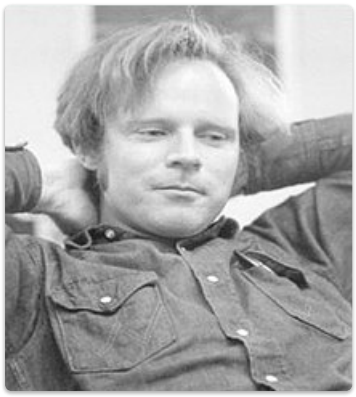
Lezione17

Table of contents

Paradigmi di Programmazione



Robert W. Floyd (1936-2001)

Una volta che abbiamo definito il linguaggio di termini, quello che ci facciamo è un approccio alla programmazione diverso dal solito che conosciamo, molto legato all'AI, perché quasi la totalità della classe di problemi che vi appartengono , passano per questo.

L'idea di **paradigma di programmazione** nasce da Robert W. Floyd, nel discorso presentato alla cerimonia d'inaugurazione al Turing Award, conferitogli il 1978.
La programmazione che conosciamo, e che allora si conosceva, non è l'unico modo di programmare; quello che cambia sono le caratteristiche dell'esecutore che pensiamo di avere in mano: immaginare un esecutore molto legato all'hardware, è una delle classiche metodologie di programmazione, ma non l'unica.

Esecutori più astratti sono meno legati alla realizzazione hardware: il programma non per forza, dato un esecutore diverso, smette di funzionare; tra i passi della compilazione, in mezzo può essere presente un'unità di traduzione o interprete, che guarda l'esecutore come un'unità più astratta. Il linguaggio viene visto come "richiesta verso l'esecutore" e a seconda delle sue caratteristiche e capacità, avremo modi diversi di esprimere le cose.

Vengono introdotti 2 paradigmi di programmazione.

1. Linguaggi di programmazione che seguono il *paradigma imperativo*, in cui ogni esecutore è una macchina di Turing più o meno elaborata, più o meno raffinata, ma che sempre di Turing è. Il programma che viene scritto, non è altro uno che guida la macchina di Turing e quindi un automa a stati (classico programma).
2. Nella programmazione che segue il *paradigma dichiarativo*, quello che si pone all'esecutore è un problema, che deve essere risolto in modo autonomo utilizzando una delle tecniche risolutive generali che ha a disposizione. Anziché dire alla macchina di Turing cosa fare, diciamo al nostro esecutore dichiarativo, quale è il problema che vogliamo affrontare (nuovo approccio).

WolframAlpha

WolframAlpha

Un esempio di risolutore che non si pone altro problema che risolvere con la conoscenza a disposizione è WolframAlpha, sviluppato da Wolfram Research.

WolframAlpha computa risposte a problemi matematici, usando conoscenze di base curate e strutturate, provenienti da siti o libri. WolframAlpha è scritto nel linguaggio Wolfram, implementato in Mathematica, comunemente non usato dai programmatori.

> Declarative programming tells what to do. Imperative programming tells how to do it.

Programmazione imperativa

Nella programmazione imperativa (che già conosciamo), c'è divisione in 2 grandi categorie.

1. La *programmazione procedurale*, in cui si va a costruire una sequenza di comandi che chiamiamo procedure, che vengono assemblate per costruire il programma intero. Ogni procedura è formata da sequenza condizionale di diramazioni di comandi, dove con "comando", s'intende un ordine che diamo all'esecutore. Stiamo ordinando cosa fare, all'esecutore.
2. La *programmazione object-oriented*, dove più esecutori detti oggetti sono presenti e ognuno ha stato di computazione. Gli esecutori interagiscono tra loro, scambiandosi messaggi, che scatenano reazione descritta tramite metodo risolutivo.

Programmazione dichiarativa

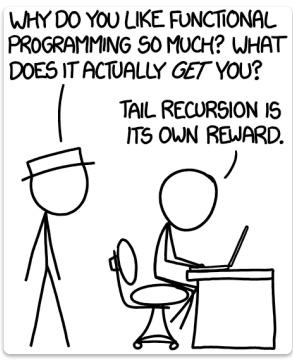
Nella programmazione dichiarativa, le cose sono diverse e nuove.
Delle 2 categorie che esistono, vedremo di più quello logico.

1. Il *paradigma funzionale* serve nella programmazione funzionale, a descrivere i problemi mediante un insieme di oggetti e un insieme di funzioni tra di essi. Un esecutore è in grado di ragionare sulle funzioni e sulla loro composizione per risolvere il problema.
2. Il *paradigma logico* è approccio classico all'AI specialmente in Europa, perché nato da ricerche tra Francia-Irlanda in grossa parte, anche se in realtà lo conoscono tutti. Nella programmazione logica, noi abbiamo a disposizione un esecutore dichiarativo, in cui andiamo a descrivere il problema da risolvere, mediante un insieme di oggetti e di relazioni tra oggetti; una volta fatta la descrizione, i problemi possono essere posti.

La programmazione funzionale è prevalentemente utilizzata negli USA.
L'AI degli Stati usa il paradigma funzionale, di cui Haskell l'esempio più noto; in Europa si è più sull'uso del paradigma logico, come PROLOG (PROgrammation en LOGique), per questioni storiche.

Contrariamente alla programmazione funzionale, di cui numerosi sono i successori di Lisp, PROLOG non ha successori numerosi. Nonostante quest'ultimo sia dove è nata la programmazione logica, è ancora lo strumento più utilizzato per realizzarla: affrontare la programmazione logica ci limita a poche scelte; studieremo PROLOG con funzionalità aggiunte della *programmazione logica con vincoli* (Constraint Logic Programming (CLP)).

Differenze tra i linguaggi



xkcd: Functional

La differenza sostanziale che si pone sui due tipi di linguaggi, è la questione di *assegnazione distruttiva*. Nei primi anni in cui abbiamo imparato a programmare, abbiamo visto la conta fino a n di una variabile i: contare fino a 10 la variabile, vuole significare inizializzare questa a 1 e ogni volta distruggere il vecchio valore, per ogni incremento che si sussegue.

Nella programmazione logica e funzionale, l'assegnamento distruttivo non si può fare, perché motivi seri si pongono per ritenere che l'assegnamento a cui siamo abituati, si uno dei motivi per cui spesso i programmi diventano intricati. Le stesse cose possono essere realizzate senza l'assegnamento distruttivo: quello che siamo abituati a chiamare ciclo `for` da 1 a 10, nella programmazione dichiarativa non lo facciamo; avremo altri modi, come la ricorsione.

Linguaggi di programmazione possono usare più di un paradigma, come per esempio Kotlin, che prende spunto da JAVA per gli oggetti e da Javascript per le funzioni.

La programmazione logica e la programmazione funzionale, la si fanno con linguaggi *Turing completi*, in grado di esprimere qualsiasi computazione, esprimibile da una macchina di Turing. Questo vuole dire che qualsiasi funzione computabile, lo è per definizione e possiamo usarle per qualsiasi cosa

Sintassi di PROLOG$_0$

Il linguaggio PROLOG$_0$ è l'esatto opposto di quello che potremmo considerare Python: estremamente "asciutto", nel senso che pochissime cose sono presenti per la comprensione, ma che ci bastano per mettere in piedi le cose che ci servono. Programmare in PROLOG è molto simile programmare in ASM, con la differenza che l'esecutore non è x64, o x86, ma uno in grado di risolvere problemi.

L'insieme di termini che lo formano sono un insieme di atomi A e un insieme di variabili V. La distinzione dei due deve essere chiara ed evidente; per imporre questo vincolo, usiamo la seguente nomenclatura.

- L'insieme A contiene tutte le parole che iniziano con una lettera minuscola e che proseguono con lettere minuscole, maiuscole, numeri e "_". Qualsiasi cosa che quindi scriviamo con la minuscola, è un atomo.
- Tutto quello che inizia con maiuscola o "_" è invece una variabile dell'insieme V.

La grammatica qui sottostante descrive nel completo PROLOG$_0$.

- $Program \rightarrow Clauses$
  Che cos'è un programma? È una sequenza di clausole. Infatti, un programma è una o più clausole, almeno una la mettiamo.
- $Caluses \rightarrow Clauses \mid Clause\ Clauses$
- $Clause \rightarrow Fact \mid Rule$
  Che cos'è una clausola? È un fatto o una regola. Un fatto come "Oggi piove.", una regola come "Se piove, mi serve un ombrello."

- $Fact \to Head$
  I fatti sono una testa, seguita da un punto. Se il "." in fondo alla frase non c'è, allora la frase non viene terminata; se non terminiamo la frase possiamo aspettarci un errore di sintassi.

- $Rule \to Head$:- $Body$
  Una regola è una testa seguita da `:-` e un corpo; in fondo resterà il punto.
  "Oggi piove." è il fatto, soltanto testa. "Se oggi piove" (corpo), "mi serve l'ombrello."(testa). Attenzione all'ordine di lettura, già preannunciato nelle scorse lezioni.

- $Head \to Atom \mid Atom(Arguments)$
  Se volessi dire "Oggi piove." (testa), ho la possibilità di dire o un atomo, o un atomo seguito da "( )" con dentro degli argomenti, separati da virgole ",". È un modo per costruire termini, dicendo che il fatto/testa, è termine seguito da punto.

- $Body \to Conjuncts$
  Il corpo è 1 o più congiunti. Un congiunto, è un termine.
- $Conjuncts \to Conjunct \mid Conjunct, Conjuncts$
  Se di congiunti ne sono presenti più di uno, allora questi li separiamo con virgola ",".

- $Conjunct \to Atom \mid Atom(Arguments)$
- $Term \to Variable \mid Atom \mid Atom(Arguments)$
- $Arguments \to Term \mid Term, Arguments$
- $Variable \to \{v \in V\}$
- $Atom \to \{a \in A\}$

Vediamo un esempio di PROLOG$_0$, usando la solita nomenclatura $\{a, b, p, q\} \subseteq A, \{X, Y\} \subseteq V$:

```
p(a). % fatto
p(X). % fatto
p(b,X) :- q(Y,X). % regola con congiunto
p(X,b) :- q(X,a), p(a,X) % regola con congiunti
```

Come vediamo non c'è nessuna relazione con la programmazione imperativa, non sembra nemmeno un programma.

Vediamo la nomenclatura del linguaggio.

- L'insieme degli elementi derivabili dalla produzione $Program$, li chiamiamo programmi.
- Tutto quello che possiamo derivare dalla notazione $Clause$, si chiama clausola definita.
- Gli elementi che deriviamo dalla produzione $Fact$, si chiamano fatti.
- Se non sono $Fact$, queste sono $Rules$.
- Il $Conjunct$ è quello che andiamo a derivare dalla produzione congiunto.

Scritto un programma in PROLOG$_0$, non ci basta eseguirlo per vedere il risultato, ma dobbiamo dirgli da dove partire. Per farlo, nel caso della programmazione logica, dobbiamo fornire quello che si chiama *goal*: dobbiamo intendere il programma come una descrizione del mondo, poi fornendo all'esecutore un obbiettivo, o problema, che vogliamo venga risolto. Il risultato del calcolo dopo aver fornito un obbiettivo, è la soluzione del problema che abbiamo posto. Tra l'altro non è detto che il valore per un'equazione sia unico: ci verrà chiesto se continuare per fornire ulteriore risultato. Non è nemmeno detto che una computazione termini perché il risultato non è definito.

Se la regola ha sempre e comunque una testa, allora tutte le clausole sono sempre definite: le *clausole di Horn* sono quelle che ci permettono di esprimere goal; tutte le clausole anche includendo quelle che non hanno testa. Esprimere un goal, ci permette di computare una soluzione.

Per il programma sotto

```
m(a).
m(b).

f(c).
f(d).

c(X) :- m(X).
c(X) :- f(X).
```

un possibile goal è `m(a).`, verificabile tramite la console interattiva al sito Web SWISH https://swish.swi-prolog.org.

```
% Che valore ha il fatto?
:- m(a).
true

:- m(b).
true

:- m(w).
false.

% Per quali valori di X l'esecuzione termina?
:- f(X).
X=c ;
X=d

:- c(X).
X=a ;
X=b ;
X=c ;
X=d
```

Con una variante sintattica, aggiungiamo gli operatori.
Il linguaggio dei termini costruibile con A e V, può essere sintatticamente arricchito aggiungendo cose come liste e operatori. PROLOG_0^+ queste proprietà le ha, in particolare: le liste sono disponibili, intendendo semplicemente dei termini, separando la testa con "|"; un atomo con simbolo "=" come predicato, per costruire termine o testa; variabili dummy (mute) indicate con "_", come se stessimo scrivendo la variabile con nome nuovo mai usato nel programma.

Esempio con liste

```
m(H, [H | X]).
m(H, [Y | R]) :-
        m(H, R).
```

con goal

```
m(X, [a, b, c]).
```

Esempio con $=$ e dummy

```
m(H, [X | _]) :-
        X = H.
m(H, [_ | R]) :-
        m(H, R).
```

09/05/2023