Linguaggi di Programmazione 2022-2023

Prolog e Programmazione Logica V

Marco Antoniotti
Gabriella Pasi
Fabio Sartori

DIPARTIMENTO
DI INFORMATICA
SISTEMISTICA
E COMUNICAZIONE

I/O IN PROLOG

Input e Output in Prolog

- I predicati primitivi principali per la gestione dell'I/O sono essenzialmente due, **`read`** e **`write`**, a cui si aggiungono i vari predicati per la gestione dei files e degli streams: **`open`**, **`close`**, **`seek`**, etc.

- **`read`** e **`write`** sono peculiari: leggono e scrivono ***termini*** Prolog
  - **`write`** è equivalente all'invocazione di un metodo `toString` Java su un oggetto di classe "termine" (il predicato **`write_term`** dà più controllo su come il termine può essere «scritto»)
  - **`read`** di fatto invoca il parser Prolog

Input e Output in Prolog

- **Esempi**

```
?- write(42).
42
true

?- foo(bar) = X, write(X).
foo(bar)
X = foo(bar)

?- read(What).
|: foo(42, Bar).
What = foo(42, _G270).

?- read(What), write('I just read: '), write(What).
|: read(What).
I just read: read(_G301)
What = read(_G301).
```

Input e Output in Prolog

- **Esempi**

```
?- open('some/file/here.txt', write, Out),
   write(Out, foo(bar)), put(Out, 0'.), nl(Out),
   close(Out).
true
 %% But file "some/file/here.txt" now contains the term 'foo(bar).'

?- open('some/file/here.txt', read, In),
   read(In, What) ,
   close(In).
What = foo(bar)
```

- **`open`** e **`close`** servono per leggere e scrivere files; la versione più semplice di **`open`** ha con tre argomenti: un atomo che rappresenta il nome del file, una "modalità" con cui si apre il file ed un terzo argomento a cui si associa l'identificatore del file.
- Vi sono naturalmente molti altri predicati per I/O in Prolog
  - Qui sopra avete visto il predicato **`put/2`**, che emette un carattere sullo stream ed il predicato **`nl/2`** che mette un 'newline' sullo stream
  - Inoltre avete visto che il Prolog usa la notazione **`0'c`** per rappresentare i caratteri come termini

DIPARTIMENTO
DI INFORMATICA
SISTEMISTICA
E COMUNICAZIONE

INTERPRETERS IN PROLOG

“Interpreti” in Prolog

- Il Prolog si presta benissimo alla costruzione di **interpreti** per la manipolazione di linguaggi specializzati (**Domain Specific Languages** – DSLs)

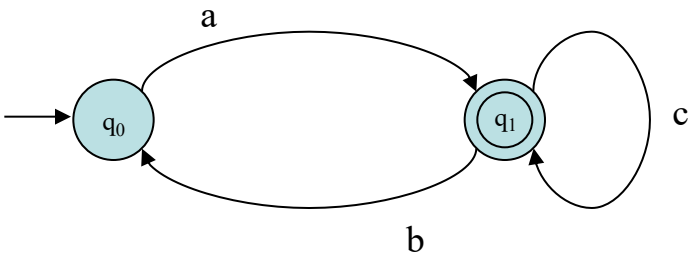
- Esempi tipici sono
  - Intepreti per **Automi** (a Stati Finiti, a Pila, Macchine di Turing)
  - Sistemi per la **deduzione automatica**
  - Sistemi per la manipolazione del **Linguaggio Naturale** (**Natural Language Processing** – NLP)

Interpreti in Prolog: Automi

- Come costruiamo un “interprete” per riconoscere nondeterministicamente dei linguaggi regolari?

```
accept([I | Is], S) :-
     delta(S, I, N),
     accept(Is, N).
accept([], Q) :- final(Q).
```

- Consideriamo l’automa



Interpreti in Prolog: Automi

- L'automa viene codificato con

```
initial(q0).
final(q1).

delta(q0, a, q1).
delta(q1, b, q0).
delta(q1, c, q1).
```

- Per decidere se una certa sequenza di simboli è riconosciuta dall'automa possiamo costruire il seguente predicato

```
recognize(Input) :- initial(S), accept(Input, S).
```

- Esempi

```
?- recognize([a, b, a, c, c, b, a]).
true

?- recognize([a, b, a, c, b]).
false
```

Interpreti in Prolog: Automi a Pila

- Come costruiamo un “interprete” per riconoscere nondeterministicamente dei linguaggi liberi da contesto (context-free languages)?

```
%%% accept(Input, Stato, Pila).

accept([I | Is], Q, S) :-
     delta(Q, I, S, Q1, S1),
     accept(Is, Q1, S1).
accept([], Q, []) :- final(Q).
```

- Consideriamo il linguaggio

$$\boldsymbol{L} = \{wrw^R \mid w \in \{a, b, c\}^n \text{ AND } n \geq 0\}$$

il linguaggio non è regolare (applicate il “Pumping Lemma”)

- Scriviamo le regole necessarie per la codifica della funzione di transizione

Interpreti in Prolog: Automi a Pila

- L'automa viene codificato con

```
initial(q0).
final(q1).

delta(q0, a, P, q0, [a | P]).
delta(q0, b, P, q0, [b | P]).
delta(q0, c, P, q0, [c | P]).
delta(q0, r, P, q1, P).
delta(q1, c, [c | P], q1, P).
delta(q1, b, [b | P], q1, P).
delta(q1, a, [a | P], q1, P).
```

- Come nel caso degli automi a stati finiti, per decidere se una certa sequenza di simboli è riconosciuta dall'automa possiamo costruire il seguente predicato

```
recognize(Input) :- initial(S), accept(Input, S, []).
```

- Esempi

```
?- recognize([a, b, a, c, r, c, a, b, a]).
true

?- recognize([a, b, a, c, r, b]).
false
```

Meta-interpreti

- Il predicato **`call/1`** che abbiamo visto precedentemente è il più semplice **meta-interprete** Prolog

- Possiamo scrivere degli interpreti più complicati e/o specializzati se accettiamo di rappresentare i programmi con una sintassi leggermente diversa

- Considerate la base di dati seguente

```
rule(append([], X, X)).
rule(append([X | Xs], Ys, [X | Zs]), [append(Xs, Ys, Zs)]).
```

Meta-interpreti: variazioni 1

- Considerate ora il seguente programma

```
solve(Goal) :- solve(Goal, []).

solve([], []).
solve([], [G | Goals]) :-
     solve(G, Goals).
solve([A | B], Goals) :-
     append(B, Goals, BGoals),
     solve(A, BGoals).
solve(A, Goals) :-
     rule(A),
     solve(Goals, []).
solve(A, Goals) :-
     rule(A, B),
     solve(B, Goals).
```

- Il programma `solve` è un meta-interprete per i predicati `rule` che compongono il nostro sistema (o programma)

Meta-interpreti: variazioni 2

- Ragioniamo con **incertezza**

```
solve_cf(true, 1) :- !.
solve_cf((A, B), C) :-
     !,
     solve_cf(A, CA),
     solve_cf(B, CB),
     minimum(CA, CB, C).
solve_cf(A, 1) :-
     builtin(A),
     !,
     call(A).
solve_cf(A, C) :-
     rule_cf(A, B, CR),
     solve_cf(B, CB),
     C is CR * CB.
```

- Il programma `solve_cf` è un meta-interprete per stabilire se un goal `G` è vero e quanto siamo certi che sia vero.

Meta-interpreti: variazioni 3

- Ragioniamo con incertezza, ma con una *soglia* (indicata dalla variabile **T**, per “threshold”)

```
solve_cf(true, 1, T) :- !.

solve_cf((A, B), C, T) :-
     !,
     solve_cf(A, CA, T),
     solve_cf(B, CB, T),
     minimum(CA, CB, C).

solve_cf(A, 1, T) :- builtin(A), !, call(A).

solve_cf(A, C, T) :-
     rule_cf(A, B, CR),
     CR > T,
     T1 is T/CR,
     solve_cf(B, CB, T1),
     C is CR * CB.
```

Conclusioni

- Il Prolog è un linguaggio di altissimo livello che ci permette di esprimere problemi, conoscenze e soluzioni in un modo conciso e naturale

- Lo stile di programmazione che meglio si adatta al linguaggio è completamente dichiarativo

- L'uso del Prolog è particolarmente efficace nella programmazione di sistemi di deduzione di vario tipo

- Lo stile di programmazione del Prolog e le idee alla sua base sono i componenti principali di una serie di nozioni utili alla gestione di relazioni semantiche su Web (RDF)

Conclusioni

- Noi abbiamo visto alcuni elementi del **Prolog**, anche sofisticati

- Non abbiamo visto
  - Moduli
  - Uso di Prolog come strumento di *analisi sintattica* (“*parsing*”)
  - Uso della nozione di operatore per modificare il linguaggio
  - Esplorazione più approfondita dell’equivalenza tra programmi e dati