Appunti di Programmazione Dichiarativa

Basati sul documento di Dovier–Formisano

Capitolo 1 – Introduzione

La programmazione dichiarativa nasce dall'idea di separare con chiarezza il *cosa* (ovvero la descrizione del problema) dal *come* (ovvero la concreta implementazione e i dettagli di esecuzione). Storicamente, mentre altre discipline ingegneristiche, come l'architettura, hanno radici antiche, l'informatica è relativamente giovane (circa 40 anni). Questa differenza si riflette anche nella qualità del software prodotto: errori di specificazione e difetti di funzionamento sono frequenti e costosi, e la necessità di tornare indietro nelle fasi di progettazione per correggere problemi può comportare sforzi enormi.

Nel ciclo di vita del software, si distinguono tipicamente sei fasi:

1. Analisi dei requisiti (cosa deve fare il sistema).
2. Specifiche di sistema (utenti, funzioni, prestazioni).
3. Progetto ad alto livello ("meta-linguaggio" astratto, logico/insiemistico).
4. Implementazione (linguaggio imperativo, codifica del "come").
5. Test/integrazione.
6. Assistenza/miglioramento.

Le fasi 1–3 si occupano di *cosa* definire, le fasi 4–6 di *come* realizzarlo. Errori nella fase di analisi possono richiedere costose revisioni successive.

La programmazione dichiarativa propone di esprimere le specifiche del problema in un linguaggio basato sulla logica del primo ordine, in modo formale, conciso e privo di ambiguità. Una caratteristica cruciale è che la specifica rimane *eseguibile*: ciò permette di prototipare rapidamente e, in alcuni casi, di derivare automaticamente parte del codice di implementazione, riducendo il divario tra la fase di *cosa* e quella di *come*.

Tra i linguaggi dichiarativi, Prolog è emblematico. Sviluppato negli anni Settanta, nasce dall'intuizione di Bob Kowalski, che mostrò come la logica predicativa potesse diventare un linguaggio di programmazione. Colmerauer e David H. D. Warren realizzarono un interprete basato sulla *Warren Abstract Machine* (WAM), rendendo Prolog efficiente per applicazioni pratiche. Un ruolo centrale è giocato dall'algoritmo di unificazione, con contributi di Paterson–Wegman e Martelli–Montanari.

Gli argomenti principali affrontati nel testo sono:

- Semantica operazionale (risoluzione SLD con clausole definite).
- Semantica logica (modello di Herbrand e teoria dei punti fissi).
- Prolog "puro", con ricorsione, costrutti built-in e predicati extralogici.

- Tecniche di programmazione: *generate-and-test*, metainterpretazione, predicati di secondo ordine, CUT.
- Gestione della negazione: *Negazione as Failure* (NaF) e *stable model semantics*.
- Answer Set Programming (ASP): regole, solver, problemi NP-completi, pianificazione.
- Constraint Logic Programming (CLP): definizione e propagazione di vincoli, solver, metodologie CP/G.
- Architetture concorrenti con CLP (Concurrent Constraint Programming, CCP).

Capitolo 2 – Richiami di logica del primo ordine

Nel secondo capitolo si richiamano i fondamenti della logica del primo ordine, indispensabili per formalizzare la semantica dei linguaggi dichiarativi.

Alfabeto (signature Σ) Un alfabeto Σ è composto da:

- Π: insieme di simboli predicati (con arità ≥ 0).
- F: insieme di simboli funzione (e costanti), ciascuno con arità ≥ 0.
- V: insieme infinito numerabile di variabili ($X, Y, Z, \ldots$).
- Connettivi logici: $\neg, \wedge, \vee, \rightarrow, \leftrightarrow$ e quantificatori $\forall, \exists$.

Termini

- Ogni variabile $X \in V$ è un termine.
- Se $f \in F$ con $\text{ar}(f) = n$ e $t_1, \ldots, t_n$ sono termini, allora $f(t_1, \ldots, t_n)$ è un termine.
- Le costanti sono simboli in F di arità 0.
- Ogni termine si rappresenta come un albero: i nodi sono simboli funzione o costanti, i figli sono i sottotermini.
- Un *sottotermine* di t è ogni termine che appare come sottoalbero di t.

Forme atomiche e formule

- Atomo: $p(t_1, \ldots, t_n)$ con $p \in \Pi$ di arità n, t_i termini. Atomo *ground* se tutti i t_i sono senza variabili.
- Formule:
  - Ogni atomo è formula.
  - Se φ è formula, allora $\neg\varphi$ è formula.
  - Se φ, ψ sono formule, allora $\varphi \vee \psi$ è formula.

  - Se $X \in V$ e φ è formula, allora $\exists X\, \varphi$ è formula.
  - Altri connettivi ($\wedge, \rightarrow, \leftrightarrow, \forall$) si riducono a negazione e disgiunzione.

- Una *letterale* è un atomo o la negazione di un atomo. Letterale positivo = atomo; letterale negativo = $\neg$atomo.

Variabili libere/legate

- Una variabile X è *libera* in φ se appare in φ al di fuori di un quantificatore che la vincola. Altrimenti è *legata*.
- $\mathrm{Vars}(\varphi)$ è l'insieme delle variabili libere in φ. Se $\mathrm{Vars}(\varphi) = \emptyset$, φ è *enunciato* (sentence).
- Notazione abbreviata: se $\mathrm{Vars}(\varphi) = \{X_1, \ldots, X_n\}$, scriviamo $\forall\varphi$ per $\forall X_1 \ldots \forall X_n\, \varphi$, e analogamente per $\exists\varphi$.

Sostituzioni Una *sostituzione* $\sigma : V \to T(F, V)$ con dominio finito si rappresenta come $[X_1/t_1, \ldots, X_n/t_n]$, dove ciascun X_i mappa nel termine t_i.

- *Binding*: X/τ indica che $\sigma(X) = \tau$.
- Tipi di sostituzioni:
  - **variabili** (ogni t_i è variabile);
  - **renaming** (ogni t_i è variabile distinta);
  - **variante/permutazione** (renaming con dominio = codominio);
  - **ground** (tutti i t_i sono privi di variabili).
- Applicazione: per un termine t,

$$t\sigma = \{\, \sigma\,(X), t \equiv X \in V, f(t_1\sigma, \ldots, t_n\sigma), t \equiv f(t_1, \ldots, t_n).$$

- s è *istanza* di t se $\exists\sigma : t\sigma = s$. Se σ è un renaming, s è *variante* di t.
- Pre-ordine tra sostituzioni: $\theta \leq \tau$ se esiste η tale che $\tau = \theta\,\eta$ (allora θ è più generale di τ).

Semantica

- Un'interpretazione $\mathcal{A} = \langle A, (\cdot)^{\mathcal{A}} \rangle$ assegna:
  - a ogni costante $c \in F$ un elemento $c^{\mathcal{A}} \in A$;
  - a ogni funzione $f \in F$ di arità n una funzione $f^{\mathcal{A}} : A^n \to A$;
  - a ogni predicato $p \in \Pi$ di arità n un sottoinsieme $p^{\mathcal{A}} \subseteq A^n$.
- Con un'assegnazione $\sigma : V \to A$ si valuta un termine t come

$$t\sigma = \{\, \sigma\,(X), t \equiv X \in V, c^{\mathcal{A}}, t \equiv c \in F,\ \mathrm{ar}(c) = 0, f^{\mathcal{A}}(t_1\sigma, \ldots, t_n\sigma), t \equiv f(t_1, \ldots, t_n).$$

- Un atomo $p(t_1, \ldots, t_n)$ è *vero* in $\mathcal{A}$ rispetto a σ se

$$(t_1\sigma, \ldots, t_n\sigma) \in p^{\mathcal{A}}.$$

- Una formula φ è *soddisfacibile* se esistono $\mathcal{A}, \sigma$ tali che $\mathcal{A} \models \varphi\sigma$. È *insoddisfacibile* se non esiste interpretazione che la renda vera, *valida* se è vera in ogni interpretazione.
- Dato un insieme di formule Γ, $\Gamma \models \psi$ se ogni modello $\mathcal{A}$ di Γ soddisfa anche ψ. Equivalente a: $\Gamma \cup \{\neg\psi\}$ è insoddisfacibile.

Interpretazioni di Herbrand

- *Universo di Herbrand* $T(F)$: insieme di tutti i termini *ground* costruibili da F.
- Una *pre-interpretazione di Herbrand* $\mathcal{H} = \langle T(F), (\cdot)^{\mathcal{H}} \rangle$ è data da:

$$c^{\mathcal{H}} = c, \quad f^{\mathcal{H}}(t_1, \ldots, t_n) = f(t_1, \ldots, t_n).$$

- Un modello di Herbrand completa stabilisce, per ogni predicato p, quali atomi ground $p(t_1, \ldots, t_n)$ sono veri.
- Teorema: un sistema di equazioni ground è soddisfacibile in qualsiasi interpretazione (in particolare in un'interpretazione di Herbrand) se e solo se ammette un'unificazione in senso di Herbrand.

Capitolo 3 – Programmazione con clausole definite

Il nucleo di Prolog e dei linguaggi a clausole definite è costituito dalle *clausole di Horn*. Una *clausola definita* è della forma

$$H \leftarrow A_1, \ldots, A_n,$$

dove H è un atomo (testa) e $A_1, \ldots, A_n$ sono atomi (corpo). Se $n = 0$, si ha un *fatto*, cioè una clausola *ground*. Un *goal* (o query) si scrive come

$$\leftarrow A_1, \ldots, A_n;$$

il *goal vuoto* è $\leftarrow$! e indica successo della derivazione. Un *programma* P è un insieme finito di clausole definite. Dal punto di vista logico,

$$H \leftarrow A_1 \wedge \ldots \wedge A_n \;\equiv\; H \vee \neg A_1 \vee \ldots \vee \neg A_n$$

(funziona come clausola di Horn con un solo letterale positivo). Nella notazione Prolog, le variabili sono implicitamente quantificate in modo universale su tutta la clausola. Ad esempio:

```
nonno(X,Y) :- padre(X,Z), padre(Z,Y).
```

equivale logicamente a

$$\forall X\, \forall Y\ (nonno(X, Y) \leftarrow \exists Z\, [\, padre(X, Z) \wedge padre(Z, Y)\,]).$$

3.2 Programmi proposizionali

Se si considerano solo predicati di arità 0 (nessun argomento), si ottiene un *programma proposizionale* fatto di clausole *ground*. Esempio:

```
estate.
caldo :- estate.
caldo :- sole.
sudato :- estate, caldo.
```

Qui `estate`, `caldo`, `sole`, `sudato` sono atomi proposizionali. Le query:

- `?- estate.` restituisce `yes`.
- `?- inverno.` restituisce `no`.
- `?- caldo.` restituisce `yes` (perché `estate` è vero).
- `?- sudato.` restituisce `yes` (perché `estate` e `caldo` sono veri).

3.3 Programmi con dominio finito

Un esempio di albero genealogico con variabili e fatti è:

```
padre(antonio, bruno).
padre(antonio, carlo).
padre(bruno, davide).
padre(bruno, ettore).
```

Questi sono *fatti* (parte estensionale). Le query possibili includono:

- `?- padre(antonio, bruno).` → `yes`.
- `?- padre(antonio, ettore).` → `no`.
- `?- padre(antonio, Y).` → `Y = bruno; Y = carlo; no`.
- `?- padre(X, carlo).` → `X = antonio; no`.

Si aggiungono poi clausole *intensionali*:

```
figlio(X, Y) :- padre(Y, X).
nonno(X, Y) :- padre(X, Z), padre(Z, Y).
```

Quindi:

- `?- figlio(Y, bruno).` → `Y = davide; Y = ettore; no`.
- `?- nonno(antonio, Y).` → `Y = davide; Y = ettore; no`.

Un esempio di *ricorsione* è il predicato `antenato/2`:

```
antenato(X, Y) :- padre(X, Y).
antenato(X, Y) :- padre(X, Z), antenato(Z, Y).
```

La query `?- antenato(antonio, Y).` restituisce `Y = bruno; Y = carlo; Y = davide; Y = ettore; no`.

3.4 Programmi con dominio infinito

Introducendo simboli funzione, si può codificare, ad esempio, l'insieme dei numeri naturali in notazione di Peano:

```
num(0).
num(s(X)) :- num(X).
```

Il predicato `num/1` afferma che `0` è un numero e che se `X` è numero, allora `s(X)` (successore di `X`) è numero. Di conseguenza:

- `?- num(s(s(0))).` → `yes`.
- `?- num(Z).` genera infinite soluzioni: `Z = 0; Z = s(0); Z = s(s(0)); ...`

Su questa base si definiscono predicati aritmetici:

- *Leq (≤)*:

  ```
  leq(0, _).
  leq(s(X), s(Y)) :- leq(X, Y).
  ```

- *Somma (plus/3)*:

  ```
  plus(X, 0, X).
  plus(X, s(Y), s(Z)) :- plus(X, Y, Z).
  ```

  Ad esempio: `?- plus(s(0), X, s(s(s(0)))).` → `X = s(s(0))`.

- *Prodotto (times/3)*:

  ```
  times(_, 0, 0).
  times(X, s(Y), Z) :- times(X, Y, V), plus(V, X, Z).
  ```

- *Esponenziale (exp/3)*:

  ```
  exp(_, 0, s(0)).
  exp(X, s(Y), Z) :- exp(X, Y, V), times(V, X, Z).
  ```

  Esempi:

  - `?- exp(X, s(s(s(0))), s(s(s(s(s(s(s(s(s(0)))))))))).` → `X = s(s(0))` (radice cubica di 8).
  - `?- exp(s(s(0)), Y, s(s(s(s(s(s(s(s(0)))))))).` → `Y = s(s(s(0)))` ($\log_2$ di 8).

- *Fattoriale (fatt/2)*:

```
fatt(0, s(0)).
fatt(s(X), Y) :- fatt(X, V), times(s(X), V, Y).
```

Questi esempi mostrano come i programmi a clausole definite siano *Turing-completi*. Infatti, ogni funzione parzialmente ricorsiva $f : N^n \to N$ si può implementare con un programma di clausole definite, sfruttando le funzioni base (zero, succ, proiezioni), la composizione, la ricorsione primitiva e la minimalizzazione (μ-operatore).

Capitolo 4 – Unificazione

L'unificazione è fondamentale nell'esecuzione di Prolog. Dati due termini s e t, si ricerca una sostituzione θ tale che $s\theta$ e $t\theta$ risultino identici. Tale θ è un *unificatore*. Tra gli unificatori, interessa il *m.g.u.* (most general unifier), ovvero un'unica θ che sia "più generale" di ogni altro unificatore σ, nel senso che $\theta \leq \sigma$ (esiste η con $\sigma = \theta\,\eta$).

L'*algoritmo di unificazione* lavora su un sistema di equazioni $C = \{\, s_1 = t_1, \ldots, s_n = t_n \,\}$ mediante queste regole di riscrittura:

1. **Decomposizione:** se $f(s_1, \ldots, s_k) = f(t_1, \ldots, t_k)$ compare in C, sostituire con $s_1 = t_1, \ldots, s_k = t_k$ e il resto di C.
2. **Fail per simboli diversi:** se $f(\ldots) = g(\ldots)$ con $f \neq g$, ritorna `false`.
3. **Elimina identità triviale:** se compare $X = X$, si rimuove.
4. **Swap:** se compare $t = X$ con t non variabile, si scambia in $X = t$.
5. **Binding:** se compare $X = t$ con $X \notin \mathrm{vars}(t)$, si sostituisce nel resto $X \mapsto t$.
6. **Fail per occur-check:** se $X = t$ con $X \in \mathrm{vars}(t)$, fallisce.

Nella pratica Prolog spesso omette il controllo di occur-check per efficienza, ma il principio rimane lo stesso. L'algoritmo termina restituendo `false` (nessun unificatore) oppure un *sistema in forma risolta*

$$X_1 = t_1,\ X_2 = t_2,\ \ldots,\ X_n = t_n,$$

con X_i variabili distinte che non compaiono nei termini t_j di binding; la sostituzione $\theta = [X_1/t_1, \ldots, X_n/t_n]$ è il m.g.u. Un teorema assicura che, se si segue qualsiasi sequenza di regole, il risultato (in caso di successo) è unico a meno di rinominazione di variabili.

Capitolo 5 – SLD-risoluzione

L'esecuzione di Prolog si basa sulla *SLD-risoluzione*. Dato un programma P (insieme di clausole definite) e un goal iniziale

$$G_0 = \leftarrow A_1, A_2, \ldots, A_n,$$

un *passo SLD* consiste nel:

1. Scegliere un atomo A_i in G_0 e una clausola $C \in P$ la cui testa H (rinominata con variabili nuove in H') può unificare con A_i tramite il m.g.u. θ.

2. Sostituire A_i con il corpo di C (rinominato in $B'_1, \ldots, B'_m$), ottenendo il *risolvente non istanziato*:
$$\leftarrow A_1, \ldots, A_{i-1}, B'_1, \ldots, B'_m, A_{i+1}, \ldots, A_n.$$

3. Applicare θ a tutte le variabili del goal, ottenendo il nuovo goal
$$G_1 = (\leftarrow A_1, \ldots, A_{i-1}, B'_1, \ldots, B'_m, A_{i+1}, \ldots, A_n)\theta.$$

Si nota che ogni passo riduce il "numero di atomi rimasti" (almeno in senso lessicografico), garantendo termine finito o fallimento. La sequenza di passi
$$G_0\theta_1, C_1G_1\theta_2, C_2G_2\theta_3, C_3 \ldots$$
è una *derivazione SLD*. Se dopo un numero finito di passi si ottiene il goal vuoto $\leftarrow!$, si raggiunge una derivazione di *successo* e la *risposta calcolata* è la composizione delle sostituzioni $\theta_1 \circ \theta_2 \circ \ldots \circ \theta_n$, ristretta alle variabili di G_0. Se si fallisce (nessuna clausola applicabile) senza arrivare a $\leftarrow!$, si ottiene una derivazione di *fallimento*. Se la derivazione continua all'infinito senza mai arrivare al goal vuoto, non è né successo né fallimento.

Il *Switching Lemma* garantisce che, se esiste *qualunque* derivazione di successo, esiste anche una derivazione che segue la medesima sequenza di clausole (a meno di rinominazioni di variabili), indipendentemente dall'ordine in cui si scelgono gli atomi del goal. In pratica, Prolog adotta la *regola di selezione leftmost* (primo atomo da sinistra) e una ricerca in profondità con backtracking:

- Per ogni goal, si seleziona sempre il primo atomo da sinistra.
- Si tenta di unificare con la prima clausola della lista di P che unifica. Se il ramo porta a fallimento, Prolog fa backtracking all'ultimo *choice point* (l'ultimo punto in cui c'erano alternative, ossia una clausola successiva da provare) e prosegue.
- Quando si raggiunge il goal vuoto, si restituisce la prima risposta. Se l'utente chiede ';', Prolog riprende il backtracking per trovare risposte successive fino a esaurire tutte le alternative.

Esempi

Padre/Antennato

```
padre(antonio, bruno).
padre(antonio, carlo).
padre(bruno, davide).
padre(bruno, ettore).

antenato(X, Y) :- padre(X, Y).
antenato(X, Y) :- padre(X, Z), antenato(Z, Y).
```

La query `?- antenato(antonio, Y).` viene risolta così:

1. **Scelta 1:** atomo = `antenato(antonio, Y)`, unifico con `antenato(X_1, Y_1)` (clausola 1), $\theta_1 = [X_1/antonio, Y_1/Y]$. Nuovo goal: `?- padre(antonio, Y).`.
2. Unifico con `padre(antonio, bruno).`, $\theta_2 = [Y/bruno]$. Nuovo goal: `?-\ !.`. Successo → risposta $\theta_1 \circ \theta_2 = [Y/bruno]$.
3. *Backtracking*: esco dall'ultimo choice point (clausola 2 per `antenato/2`). Uso `antenato(X_2,Y_2) :- padre(X_2, Z), antenato(Z, Y_2).` con $\theta'_1 = [X_2/antonio, Y_2/Y]$. Nuovo goal: `?- padre(antonio, Z), antenato(Z, Y).`.
4. Unifico `padre(antonio, bruno).` con $\theta'_2 = [Z/bruno]$. Nuovo goal: `?- antenato(bruno, Y).`.
   - Uso clausola 1 di `antenato`: $\theta'_3 = [X_3/bruno, Y_3/Y]$. Nuovo goal: `?- padre(bruno, Y).`.
   - Unifico con `padre(bruno, davide).`, $\theta'_4 = [Y/davide]$. Nuovo goal: `?-\ !.`. Successo → risposta $[Y/davide]$.
5. Ancora backtracking → trova `Y=carlo` (dalla clausola `padre(antonio, carlo).`) e infine `Y=ettore`.

L'ordine delle risposte è: `Y = bruno; Y = carlo; Y = davide; Y = ettore; no`.

Member/2

```
member(X, [X|_]).
member(X, [_|T]) :- member(X, T).
```

La query `?- member(A, B).` genera infinite soluzioni:

1. Primo passo: unifica `member(A,B)` con `member(X,[X|_])`, ottenendo `B = [A|`$_{Z1}$`].`*Successo.Backtrackin*
provalasecondaclausola`member(X,[`$|T$`])`, *riducea*`member(A,T)`, *generandoricorsivamentesoluzionidif*
Ogni soluzione è variante (stessa forma, variando nomi di variabili).

Capitolo 6 – Semantica dei programmi logici

Qui si definiscono due visioni della semantica:

2• *Semantica operazionale*: basata sulla SLD-risoluzione, descrive ciò che Prolog calcola effettivamente. Un programma P risponde a un goal G con una sostituzione θ se esiste una derivazione SLD in P che raggiunge il goal vuoto $\leftarrow !$ e θ è la composizione dei m.g.u. incontrati, ristretta alle variabili di G.

- *Semantica logica (modellistica)*: utilizza il *modello minimo di Herbrand*. Si definisce l'operatore T_P sugli insiemi di atomi ground:

$$T_P(I) \;=\; \{\, H\sigma \mid (H \leftarrow B_1, \ldots, B_n) \in P,\; B_i\sigma \in I \;\forall i\}.$$

  L'insieme vuoto $\emptyset$ è punto di partenza, e si considerano le iterazioni successive $T_P^0(\emptyset) = \emptyset$, $T_P^{i+1}(\emptyset) = T_P(T_P^i(\emptyset))$. Il punto fisso minimo (ossia $\bigcup_{i=0}^{\infty} T_P^i(\emptyset)$) è il *modello minimo di Herbrand* M_1, e contiene esattamente tutti gli atomi ground che ammettono una derivazione SLD di successo. Quindi la semantica operazionale (risposte di Prolog) coincide con la semantica denotazionale (modello minimo).

Di conseguenza, $P \models A$ (per atomo ground A) se e solo se $A \in M_1$; per un goal non-ground G, $P \models G$ se esiste θ tale che $P \models G\theta$.

Capitolo 7 – Programmazione in Prolog

Questo capitolo descrive le strutture dati e i predicati built-in di Prolog.

7.1 Liste

Le liste in Prolog sono rappresentate come termini dell'operatore `.[/2]`:

$$[E_1, E_2, \ldots, E_n] \;=\; .(E_1, .(E_2, \ldots .(E_n, [])\ldots)).$$

Si usa la sintassi sugar `[H|T]` per `.(H, T)`. Esempi di predicati comuni:

```
append([], L, L).
append([H|T], L2, [H|R]) :- append(T, L2, R).

member(X, [X|_]).
member(X, [_|T]) :- member(X, T).

length([], 0).
length([_|T], N) :- length(T, N1), N is N1 + 1.
```

7.2 Alberi, grafi, automi

- **Alberi** si rappresentano come `node(Value, Left, Right)`. Traversamenti DFS/BFS si scrivono ricorsivamente.
- **Automi finiti** si codificano con fatti `trans(State, Symbol, NextState)` e un predicato ricorsivo `accept/1` che simula l'automa sui simboli.

7.3 Liste differenza

Una *lista differenza* è una coppia `Front-Back`, dove `Back` è la coda variabile di `Front`. Permette concatenazioni in O(1). Ad esempio:

```
append_dl(A-B, B-C, A-C).
```

7.4 Predicati built-in

Prolog fornisce molte primitive per manipolare termini, liste e fare I/O:

- **Unificazione e comparazione sintattica:** `=/2` (unifica), `==/2` (uguaglianza sintattica),
  `==/2` (negazione di ==), `@</2` e `@>/2` (ordinamento lessicografico tra termini).
- **Operazioni aritmetiche:** `is/2`, `=:=/2`, `=\=/2`, `<`, `>`, `>=`, `=<`.
- **Gestione di liste:** `append/3`, `member/2`, `length/2`, `reverse/2`.
- **I/O:** `read/1`, `write/1`, `nl/0`.
- **Controllo:** `!` (cut), `fail/0`, `true/0`.

7.5 Predicati di tipo e manipolazione termini

- `var(X)`, `nonvar(X)`, `atomic(X)`, `compound(X)`.
- `functor(T, N, A)`: estrae functor e arità di `T`.
- `arg(I, T, Arg)`: restituisce l'I-esimo argomento di `T`.
- `=../2` (univ operator): `Term =.. [Functor|Args]` decompone o costruisce un termine.
- `copy_term/2`, `substitute/4` per sostituzioni esplicite.
- `assert/1`, `retract/1` per modificare dinamicamente la base di conoscenza.

7.6 Predicati metalogici o extralogici

- `call/1`: invoca un termine costruito a runtime.
- `term_variables/2`: restituisce le variabili di un termine.
- `bagof/3`, `setof/3`, `findall/3`: raccolgono rispettivamente tutte le soluzioni in una lista, con o senza ordinamento e rimozione duplicati.

7.7 Predicati di I/O

- `open/3`, `close/1` per file/stream.
- `read/2`, `write/1`.
- `format/2` per output formattato.

7.8 Il predicato `fail/0`

`fail/0` fa fallire sempre, costringendo Prolog a tornare all'ultimo choice point e proseguire col backtracking.

7.9 Definizione di operatori

Si possono definire nuovi operatori con:

```
:- op(Precedence, Type, Name).
```

Ad esempio:

```
:- op(100, fy, s).
```

permette di scrivere `s s 0` al posto di `s(s(0))`.

Capitolo 8 – Tecniche di programmazione dichiarativa

Il capitolo otto illustra tecniche di programmazione comuni in Prolog.

8.1 Programmazione ricorsiva

Si adotta il classico schema "caso base" + "caso ricorsivo".

```
sum_list([], 0).
sum_list([H|T], N) :-
    sum_list(T, N1),
    N is H + N1.
```

Questo schema serve per liste, alberi, grafi, ecc., definendo condizioni terminali e riducendo a sottoproblemi.

8.2 Schema "Generate and Test"

Consiste nel generare *tutte* le soluzioni candidate (generate) e poi testarle (test) per filtrare quelle valide. Esempio: generare permutazioni e verificare se sono ordinate.

```
permutation([], []).
permutation(L, [H|P]) :-
    select(H, L, R),
    permutation(R, P).

is_sorted([]).
is_sorted([_]).
is_sorted([X, Y|T]) :-
    X =< Y,
    is_sorted([Y|T]).

sorted_perm(L, S) :-
    permutation(L, S),
    is_sorted(S).
```

8.3 Predicati di secondo ordine

Mediante `call/2` si implementano predicati più generali come `map/3`, `filter/3`, `fold/4`:

```
map(_, [], []).
map(Pred, [X|Xs], [Y|Ys]) :-
    call(Pred, X, Y),
    map(Pred, Xs, Ys).

filter(_, [], []).
filter(Pred, [X|Xs], [X|Zs]) :-
    call(Pred, X), !,
    filter(Pred, Xs, Zs).
filter(Pred, [_|Xs], Ys) :-
    filter(Pred, Xs, Ys).

fold(_, [], Acc, Acc).
fold(Pred, [X|Xs], Acc, Res) :-
    call(Pred, Acc, X, Acc1),
    fold(Pred, Xs, Acc1, Res).
```

8.4 Il CUT (`!`)

Il *cut* taglia tutti i choice point creati *prima* di esso nella stessa clausola:

```
max(X, Y, X) :- X >= Y, !.
max(_, Y, Y).
```

Qui, se `X >= Y` è vero, il cut impedisce di provare la seconda clausola per `max/3`.

8.5 Metainterpretazione

Si può rappresentare un programma come fatti `cl(Head:-Body)` e definirne un meta-interprete:

```
cl(Head :- Body).

solve(true).
solve((A, B)) :-
    solve(A),
    solve(B).
solve(A) :-
    cl(A, B),
    solve(B).
```

Questo `solve/1` interpreta a runtime le clausole definite in `cl/2`.

Capitolo 9 – Negazione e semantica estesa

9.1 Negazione as Failure (NaF)

In Prolog, `not A` si intende "*fallisco se trovo una prova di A; altrimenti considero A falso*". Più precisamente:

- si tenta di risolvere `A`;
- se esiste un ramo di successo per `A`, allora `not A` fallisce;
- se tutte le derivazioni di `A` falliscono, allora `not A` ha successo.

Questa *negazione per fallimento* non corrisponde alla negazione classica e in presenza di ricorsioni o cicli porta a casi di indeterminazione, ad esempio `p :- not p.` non ha un comportamento ben definito in Prolog puro.

9.2 Semantiche a tre valori

Per ovviare ai limiti di NaF si introducono semantiche a tre valori:

- *Well-Founded Semantics* (WFS): assegna a ogni proposizione uno dei valori {*vero*, *falso*, *indefinito*}.
- *Stable Model Semantics*: fissa un insieme di atomi "assunti veri" e valuta il programma ridotto ottenuto eliminando le regole con negazioni insoddisfatte; se l'interpretazione risultante è coerente, è un modello stabile.

Capitolo 10 – Answer Set Programming (ASP)

L'Answer Set Programming (ASP) è un metodo dichiarativo per risolvere problemi combinatori. Un programma ASP è costituito da regole del tipo:

$$A_0 \ :- \ A_1, \ldots, A_m, \ notA_{m+1}, \ldots, notA_n.$$

Qui A_i (per $i \leq m$) sono letterali positivi e `not` A_j (per $j > m$) sono letterali di negazione "default" (NaF). Si può anche usare la negazione forte $\neg A$ e la disgiunzione in testa (`A ; B :- Body`).

Stable Model Semantics (Gelfond–Lifschitz) Dato un'interpretazione I (insieme di atomi considerati veri), si costruisce il *programma ridotto* P^I:

1. Si rimuove ogni regola contenente `not` A con $A \in I$.
2. Si cancellano i letterali `not` A in tutte le regole rimanenti.

Un insieme di atomi I è un *modello stabile* (o *answer set*) di P se I è il modello (minimo) del programma ridotto P^I. Le soluzioni di un problema codificato in ASP corrispondono agli *answer sets* di P.

10.1 Cardinality Constraints e Aggregati

ASP permette di esprimere vincoli di cardinalità in modo compatto:

$$\{ \ X : p(X) \ \} = \ k, \quad < k, \quad \leq k, \quad \geq k.$$

E aggregati come:

$$\#\text{count}\{ \ X : p(X)\} = N, \quad \#\text{sum}\{ \ V : cost(V)\} = C.$$

Questi costrutti semplificano la scrittura di vincoli che altrimenti richiederebbero molte regole.

10.2 Workflow con ASP-solver

Il flusso tipico è:

1. **Grounding**: si eliminano le variabili instanziandole ai domini finiti (`Gringo`).
2. **Solve**: si traduce il programma ground in SAT-like e lo si risolve con un solutore SAT (`clasp`, `Cmodels`).
3. **Traduzione inversa**: si interpreta l'assegnazione SAT come atomi ASP per ottenere l'answer set.

Capitolo 11 – Esempi di problemi risolvibili con ASP

Tra i problemi classici che si modellano in ASP troviamo:

- Stable Marriage (problema del matrimonio stabile).
- N-Queens (problema delle N-regine).
- Zebra Puzzle (Five Houses Puzzle).
- Map Coloring (colorazione di grafi).
- Hamiltonian Circuit (circuito hamiltoniano).
- k-Clique, Vertex Cover.
- Task Allocation (assegnazione di compiti).
- Knapsack.
- Calcolo dei numeri di Schur.
- Protein Structure Prediction (predizione di struttura proteica).

Per ciascuno di questi si definiscono regole ASP che generano *tutte* le soluzioni candidate e poi vincoli'imporranno solo quelle corrette (answer sets).

Capitolo 12 – Answer Set Planning

ASP si applica anche al planning, rappresentando stati e azioni come atomi:

$$occ(Azione, Time), \quad holds(Fluent, Time).$$

Le *frame-axioms* (coniugano l'invarianza dei fluenti se non esiste azione che li modifica) si esprimono con regole ASP di tipo inertia. Si genera uno spazio di tempo fino a un limite T, si impongono vincoli sugli stati iniziali e finali, e si recuperano, dagli answer sets, le sequenze di azioni (piani) che soddisfano gli obiettivi.

Capitolo 13 – Vincoli e risolutori (CLP)

CLP (Constraint Logic Programming) unisce Prolog con tecniche di risoluzione di vincoli su variabili:

- **CLP(FD)**: domini finiti. Variabili che assumono valori in un insieme finito. Vincoli comuni: $X\# = Y$, $X\# \neq Y$, $X\# < Y$, $X\# \leq Y$, $X\# \in [L..U]$.
- **CLP(R)**: domini reali. Variabili che assumono valori reali, vincoli lineari e non lineari su R.
- Algoritmi di base: *Arc Consistency* (es. AC-3, AC-4), *Forward Checking*, *Constraint Propagation*.

- *Vincoli globali*: `all_different/1`, `cumulative/1`, `element/3`, usano algoritmi specializzati per ridurre i domini in modo più efficiente.

Sono disponibili solver come ECLiPSe, SICStus Prolog (CLP(FD), CLP(R)), GNU Prolog con supporto CLP(FD), che implementano predicati per dichiarare e propagare vincoli, e meccanismi di *labeling* per la ricerca sul dominio residuale.

Capitolo 14 – Programmazione logica con vincoli (CLP)

Una *clausola CLP* ha la forma

$$C \ \wedge \ A_1 \ \wedge \ \ldots \ \wedge \ A_n \ \rightarrow \ H,$$

dove C è una congiunzione di *vincoli* (es. $X \in 1..10$, $X\# \neq Y$), gli A_i sono atomi e H è la testa. Durante la derivazione, si mantiene un insieme di vincoli che viene continuamente *propagato* per ridurre i domini delle variabili fino a individuare inconsistenze o assegnazioni uniche.

CLP(FD) (domini finiti):

- Predicati principali: `#=/2`, `#\=/2`, `#>/2`, `#>=/2`, `#=</2`, `#in/2`, `labeling/2`.
- Quando si aggiunge un vincolo, il solver riduce i domini delle variabili coinvolte (propagazione) e segnala conflitti se un dominio diventa vuoto.
- Vincoli reificati: ad es. `B #<=> X #= Y` trasforma il vincolo in una variabile booleana `B`, utile per combinazioni logiche di vincoli.

CLP(R) (domini reali):

- Variabili reali, vincoli lineari/non lineari su R.
- Predicati: `{}/1` per esprimere un insieme di equazioni/inequazioni reali, `freeze/2` per ritardare l'applicazione di vincoli fino a quando una certa variabile non è istanziata.
- Si mantengono intervalli reali per ogni variabile e si applicano tecniche di propagazione su di essi.

Capitolo 15 – CLP(FD): metodologia "Constrain & Generate"

La metodologia "Constrain & Generate" consiste in due fasi:

1. **Constrain**: si definiscono *tutti* i vincoli sul problema. Il solver riduce i domini delle variabili via propagazione.
2. **Generate (Labeling)**: si assegnano ai restanti intervalli valori specifici, solitamente con euristiche che scelgono prima le variabili più ristrette o più vincolate, fino a trovare soluzioni complete.

Esempi classici

1. **N-Queens** (`n_queens/2`):

```
n_queens(N, Qs) :-
    length(Qs, N),
    Qs ins 1..N,
    all_different(Qs),
    all_different([Q+I || I,Q in enumerate(Qs)]),
    all_different([Q-I || I,Q in enumerate(Qs)]),
    labeling([], Qs).
```

   - `Qs` è una lista di variabili lunghezza N (ogni variabile indica la colonna della regina nella riga corrispondente).
   - `all_different(Qs)` impone che non ci siano due regine sulla stessa colonna.
   - I due `all_different` successivi impongono i vincoli diagonali (somma e differenza di indice riga e colonna).
   - `labeling([], Qs)` esplora gli assegnamenti rimanenti finché non trova soluzioni.

2. **Knapsack** (`knapsack/4`):

```
knapsack(WeightLimit, Weights, Values, BestValue) :-
    length(Items, N),
    Items ins 0..1,
    scalar_product(Weights, Items, #=, WeightSum),
    WeightSum #=< WeightLimit,
    scalar_product(Values, Items, #=, ValueSum),
    labeling([maximize(ValueSum)], Items),
    BestValue = ValueSum.
```

   - `Items` è una lista di N variabili in $\{0, 1\}$ che indicano se includere o meno ogni oggetto.
   - `scalar`$_p roduct(Weights, Items, =, WeightSum)$`calcolailpesototaleselezionato.Siimpone`We
   - La fase di `labeling` con `[maximize(ValueSum)]` cerca la combinazione che massimizza il valore totale.

3. **Map Coloring** (`coloring/3`):

```
coloring(NumRegions, NumColors, Assignment) :-
    length(Assignment, NumRegions),
    Assignment ins 1..NumColors,
    findall((R1, R2), adjacent(R1, R2), Edges),
    impose_edges(Edges, Assignment),
    labeling([], Assignment).

impose_edges([], _).
```

```
impose_edges([(R1,R2)|T], Assignment) :-
    nth1(R1, Assignment, C1),
    nth1(R2, Assignment, C2),
    C1 #\= C2,
    impose_edges(T, Assignment).
```

- `Assignment` è una lista di variabili di lunghezza `NumRegions`, ciascuna in $[1..NumColors]$.
- `adjacent(R1,R2)` è un fatto che indica che le regioni $R1$ e $R2$ sono contigue.
- `impose_edges/2` impone che per ogni coppia di regioni contigue i colori siano diversi (`C1 #`$\bar{\texttt{C}}$`2`).
- `labeling([], Assignment)` esplora gli assegnamenti rimanenti.

4. Altri problemi tipici: *Stable Marriage*, `SEND + MORE = MONEY`, *Schur numbers*, *Task Allocation*, *Hamiltonian Circuit*, e molti altri. Tutti si modellano definendo vincoli su variabili discrete e poi applicando `labeling` per enumerare soluzioni.

Appendici

- **Appendice A**: nozioni di teoria degli ordini e dei reticoli, teoremi di Knaster–Tarski e fisso dei punti, utili per comprendere il punto fisso di T_P.
- **Appendice B**: suggerimenti pratici per l'uso di Prolog e ASP-solver (Gringo, Clingo, DLV), parametri, ottimizzazioni e trucchi.
- **Appendice C**: soluzioni commentate agli esercizi proposti nei capitoli 2–15, per esercitarsi sui temi affrontati.
- **Bibliografia**: elenco di testi e articoli fondamentali (Kowalski, Lloyd, Gelfond, e altri).