Appunti di Programmazione Dichiarativa

Basati sul documento di Dovier–Formisano

Capitolo 1 – Introduzione

- **Contesto storico**
  La scienza del calcolatore è relativamente giovane (circa 40 anni), mentre discipline come l'architettura sono molto più antiche. Questo si ripercuote sulla progettazione del software: errori di preventivazione, difetti di funzionamento e bassa sicurezza sono ancora molto diffusi, mentre in altri settori un errore simile comprometterebbe la sopravvivenza dell'industria.

- **Separatezza tra "cosa" e "come"**
  Il ciclo di vita del software si divide grossolanamente in:

  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. Se si scoprono errori nelle fasi iniziali, bisogna tornare indietro, con costi elevati.

- **Programmazione dichiarativa**
  L'idea è di usare un linguaggio basato sulla logica del primo ordine per specificare il "cosa" in modo formale, conciso, non ambiguo. Mantenere la specifica eseguibile (prototipazione rapida, derivazione di codice) riduce il gap tra la fase di *cosa* e quella di *come*.

- **Prolog e discendenza**
  Prolog nasce negli anni '70: Bob Kowalski mostra che la logica predicativa è un linguaggio di programmazione, Colmerauer e D. Warren implementano un interprete basato su WAM (Warren Abstract Machine). Fondamentali per l'efficienza: algoritmo di unificazione (Paterson–Wegman, Martelli–Montanari).

- **Argomenti del corso/libro**

  1. Semantica operazionale (SLD-risoluzione con clausole definite).
  2. Semantica logica (modellistica, punto fisso).
  3. Prolog "puro" e ricorsione, built-in, costrutti extralogici.

4. Tecniche di programmazione (generate-and-test, metainterpretazione, predicati di secondo ordine, CUT).
5. Negazione (NaF) e stable model semantics.
6. Answer Set Programming (ASP): regole, solver, problemi NP-completi finiti, planning.
7. Constraint Logic Programming (CLP): definizione di vincoli, propagazione, solver, metodologie CP/G.
8. Architetture concorrenti (Concurrent Constraint Programming, CCP).

Capitolo 2 – Richiami di logica del primo ordine

2.1 Sintassi

- **Alfabeto (signature Σ)**
  - Π = insieme di simboli predicati, arità ≥ 0.
  - F = insieme di simboli funzioni (e costanti), arità ≥ 0.
  - V = insieme infinito numerabile di variabili ($X, Y, Z, \ldots$), arità 0.
  - Connettivi logici: $\neg, \wedge, \vee, \rightarrow, \leftrightarrow$; quantificatori: $\forall, \exists$; parentesi e virgole.
- **Termini (Def. 2.2)**
  - Ogni variabile è un termine.
  - Se $f \in F$ con $\mathrm{ar}(f) = n$ e $t_1, \ldots, t_n$ sono termini, allora $f(t_1, \ldots, t_n)$ è un termine.
  - Le costanti sono i simboli di F con arità 0.
  - Esempi: `f(X,f(a,b))`, `+(1,*(3,5))` sono termini; `ab` non lo è.
  - *Sottotermine (Def. 2.3):* un termine s è sottotermine di t se appare come sottostruttura (rispettando la struttura ad albero) e anche s è termine.
- **Forme atomiche e formule (Def. 2.4–2.5)**
  - Atomo: $p(t_1, \ldots, t_n)$ con $p \in \Pi$, $\mathrm{ar}(p) = n$, t_i termini.
  - Atomo ground: tutti i t_i sono ground (nessuna variabile).
  - Formule costruite ricorsivamente:
    - Se φ è atomo, allora φ è formula.
    - Se φ è formula, $\neg\varphi$ è formula.
    - Se φ, ψ sono formule, $\varphi \vee \psi$ è formula.
    - Se φ è formula e $X \in V$, $\exists X\, \varphi$ è formula.
    - Altri connettivi ($\wedge, \rightarrow, \leftrightarrow, \forall$) si riducono a negazione/disgiunzione/esistenziale.
  - Letterale = atomo o $\neg$ atomo; positivo = atomo, negativo = negazione di atomo.
- **Variabili libere/legate (Def. 2.6)**

- Una variabile X è libera in φ se occorre in un atomo $p(t_1, \ldots, t_n)$ in un punto *non* all'interno di un quantificatore che la vincoli.
- $\mathrm{Vars}(\varphi)$ = insieme di variabili libere in φ.
- Se $\mathrm{Vars}(\varphi) = \emptyset$, φ è enunciato (*sentence*).
- Notazione abbreviata: $\forall\varphi$ sta per $\forall X_1 \ldots \forall X_n\, \varphi$, se $\mathrm{Vars}(\varphi) = \{X_1, \ldots, X_n\}$ (e similmente per $\exists\varphi$).

- **Sostituzioni (Def. 2.18–2.24)**
  - Una sostituzione $\sigma : V \to T(F, V)$ con dominio finito, indicate come $[X_1/t_1, \ldots, X_n/t_n]$.
  - *Binding*: X/τ indica $\sigma(X) = \tau$.
  - Tipi di sostituzioni:
    - *variabili*, se tutti i t_i sono variabili;
    - *renaming*, se tutti i t_i sono variabili distinte;
    - *variante/permutazione*, se renaming con $\mathrm{dom}(\sigma) = \mathrm{ran}(\sigma)$;
    - *ground*, se tutti i t_i sono ground.
  - *Applicazione*: $t\sigma$ si definisce ricorsivamente: se $t = X$ variabile, $t\sigma = \sigma(X)$; se $t = f(t_1, \ldots, t_n)$, allora $t\sigma = f(t_1\sigma, \ldots, t_n\sigma)$.
  - s è *istanza* di t se $\exists\sigma : s = t\sigma$; variante se σ è variante.
  - *Composizione*: $(\theta\eta)(X) = (\eta)(X\theta)$. Proprietà di idempotenza, relazione d'equivalenza di "essere variante di", e pre-ordine $\leq$ tra sostituzioni: $\theta \leq \tau$ se $\exists\eta : \tau = \theta\eta$.

2.2 Semantica

- **Interpretazioni (Def. 2.7)**
  Un'interpretazione $\mathcal{A} = \langle A, (\cdot)^{\mathcal{A}} \rangle$:
  - A = dominio non vuoto.
  - Ogni costante $c \in F$ (arità 0) è mappata in un elemento $c^{\mathcal{A}} \in A$.
  - Ogni funzione $f \in F$ (arità $n > 0$) è mappata in una funzione $f^{\mathcal{A}} : A^n \to A$.
  - Ogni predicato $p \in \Pi$ (arità n) è mappato in un sottoinsieme $p^{\mathcal{A}} \subseteq A^n$.

- **Assegnamento di variabili (Def. 2.8)**
  Per valutare termini con variabili serve $\sigma : V \to A$. Si definisce $t\sigma$ in $\mathcal{A}$: se $t = X$, $t\sigma = \sigma(X)$; se $t = c$ costante, $t\sigma = c^{\mathcal{A}}$; se $t = f(t_1, \ldots, t_n)$, $t\sigma = f^{\mathcal{A}}(t_1\sigma, \ldots, t_n\sigma)$.

  Per un atomo $p(t_1, \ldots, t_n)$, si ha $(p(t_1, \ldots, t_n)\sigma)^{\mathcal{A}} = \mathit{true}$ se e solo se $(t_1\sigma, \ldots, t_n\sigma) \in p^{\mathcal{A}}$.

- **Concetti di soddisfacibilità/validità (Def. 2.11–2.13)**
  - $\mathcal{A} \models \varphi$ se $\exists$ assegnamento σ tale che $\varphi\sigma$ valuta `true` in $\mathcal{A}$.
  - φ è soddisfacibile se $\exists\mathcal{A}$ con $\mathcal{A} \models \varphi$; insoddisfacibile se non esiste $\mathcal{A}$; valido se $\forall\mathcal{A}$, $\mathcal{A} \models \varphi$.

  - Per teoria Γ, $\Gamma \models \varphi$ se ogni modello $\mathcal{A}$ di Γ è anche modello di φ. Equivalentemente $\Gamma \cup \{\neg\varphi\}$ è insoddisfacibile.

- **Interpretazioni di Herbrand**

  - Universo di Herbrand $T(F)$: insieme di tutti i termini *ground* costruibili da F.
  - Pre-interpretazione di Herbrand $\mathcal{H} = \langle T(F), (\cdot)^{\mathcal{H}} \rangle$ definita da $c^{\mathcal{H}} = c$ e $f^{\mathcal{H}}(t_1, \ldots, t_n) = f(t_1, \ldots, t_n)$.
  - Interpretazione di Herbrand completa: aggiunge a $\mathcal{H}$ l'interpretazione di ogni predicato p come sottoinsieme di $B_{\Pi,F}$ (base di Herbrand: tutti gli atomi ground $p(t_1, \ldots, t_n)$ con $t_i \in T(F)$).
  - Teorema fondamentale: per un sistema di equazioni ground C, C è soddisfacibile in *qualunque* interpretazione $\iff$ esiste soluzione di unificazione in Herbrand.

2.3 Sostituzioni (continuazione)

- Lemmi su composizione e varianti:
  - $(s\theta)\eta = s(\theta\eta)$.
  - $(\theta\eta)\gamma = \theta(\eta\gamma)$.
  - s è variante di t $\iff$ s è istanza di t e t è istanza di s.
  - Se $s = t\theta$ con θ variante, allora $\mathrm{vars}(\theta) \subseteq \mathrm{vars}(s) \cup \mathrm{vars}(t)$.
- **Pre-ordine su sostituzioni (Def. 2.24)**
  $\theta \leq \tau$ se $\exists \eta : \tau = \theta\eta$.
- Lemma: se $\theta \leq \tau$, allora per ogni termine t, $t\tau$ è istanza di $t\theta$.

Capitolo 3 – Programmazione con clausole definite

3.1 Introduzione a clausole definite

- **Clausola definita (Def. 3.1)**
  Una clausola definita è una regola 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 $n = 0$, $H \leftarrow$ si chiama "fatto" (o clausola *ground*).
  - Un *goal* (query) si scrive $\leftarrow A_1, \ldots, A_n$; il *goal vuoto* è $\leftarrow$!.
  - Un programma P è un insieme finito di clausole definite.
  - Lettura logica: $H \leftarrow A_1 \wedge \ldots \wedge A_n$ è equivalente a $H \vee \neg A_1 \vee \ldots \vee \neg A_n$ (clausola di Horn, un solo letterale positivo).

- **Quantificazione implicita**
  Le variabili in ogni clausola sono considerate universalmente quantificate all'esterno $() \rightarrow \forall$.
- **Esempio 3.1 (nonno/2)**

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

  Lettura "intuitiva": "X è nonno di Y se X è padre di Z e Z è padre di Y". In realtà equivale a

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

3.2 Programmi proposizionali (no variabili, dominio finito)

- **Clausole ground (esempi)**

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

  Tutti i predicati hanno arità 0: $\Pi_P = \{\texttt{estate}, \texttt{caldo}, \texttt{sole}, \texttt{sudato}\}$.
  Query e risposte:
  - `?- estate.` $\rightarrow$ yes
  - `?- inverno.` $\rightarrow$ no
  - `?- caldo.` $\rightarrow$ yes (perché `estate` è *fact*)
  - `?- sudato.` $\rightarrow$ yes (estate e caldo sono veri)

3.3 Programmi con dominio finito (database)

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

- *Fatti* (parte estensionale).
- Query:
  - `?- padre(antonio,bruno).` $\rightarrow$ yes
  - `?- padre(antonio,ettore).` $\rightarrow$ no
  - `?- padre(antonio,Y).` $\rightarrow$ Y = bruno ; Y = carlo ; no.

- `?- padre(X,carlo).` → X = antonio ; no.

- *Parte intensional:*

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

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

- **Ricorsione (antenato/2)**

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

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

3.4 Programmi con dominio infinito (variabili e funzioni)

Numeri naturali "Peano":

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

- Query:
  - `?- num(s(s(0))).` → yes
  - `?- num(Z).` → Z = 0 ; Z = s(0) ; Z = s(s(0)) ; ...

Definizione di predicati su numeri di Peano

- **$\leq$ (leq) e $<$ (lt):**

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

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

- **Somma (plus/3):**

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

  - `?- 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).
```

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

- **Fattoriale (fatt/2):**

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

3.5 Turing-completezza (Teor. 3.1)

- Ogni funzione parzialmente ricorsiva $f : N^n \rightarrow N$ è *definibile* da un programma di clausole definite.
- Si possono implementare:
  1. Funzioni di base:
     - `zero(X_1,...,X_n,0).`
     - `succ(X,s(X)).`
     - `piñ_i(X_1,...,X_n,X_i).`
  2. Operazioni di composizione: se $f = g(h_1(\ldots), \ldots, h_k(\ldots))$, si definiscono clausole che chiamano g e h_i.
  3. Ricorsione primitiva: clausole di *induzione* che usano g per il caso base e f per il passo.
  4. Minimalizzazione (μ-operatore): si scrive un predicato che cerca il minore Y tale che $f(X, \ldots, Y) = 0$, testando linearmente con predicati aggiuntivi (es. "mai_prima").
- Conclusione: i programmi di clausole definite sono almeno potenti quanto le Macchine di Turing (Turing-complete).

Capitolo 4 – Unificazione

4.1 Definizione e pre-ordine

- **Unificatore (Def. 4.1)**
  Dati due termini s, t e una sostituzione θ:
  - θ è *unificatore* se $s\theta \equiv t\theta$ (uguaglianza sintattica).
  - θ è *m.g.u.* (most general unifier) se è unificatore e ogni unificatore σ di s, t soddisfa $\theta \leq \sigma$ (cioè θ è "più generale").
  - Per teorie equazionali E, si definisce E-unificatore se $s\theta =_E t\theta$.
- **Sistema di equazioni (Def. 4.2)**
  Un sistema di equazioni $C \equiv (s_1 = t_1 \wedge \ldots \wedge s_n = t_n)$:
  - θ è unificatore di C se unifica ogni equazione $s_i = t_i$.
  - θ è m.g.u. di C se è unificatore e per ogni unificatore σ di C vale $\theta \leq \sigma$.
- **Relazione $\leq$ e istanza di sostituzione**
  $\theta \leq \tau$ se $\exists \eta : \tau = \theta\eta$.

4.2 Algoritmo di unificazione (Herbrand, Def. 4.4)

- *Obiettivo*: dato un sistema C di equazioni, decidere se esiste un m.g.u.
- Sei regole di riscrittura (riportate in Fig. 4.2 del testo) da applicare finché possibile:
  1. Decomposizione: $f(s_1, \ldots, s_n) = f(t_1, \ldots, t_n) \wedge C \;\rightarrow\; (s_1 = t_1) \wedge \ldots \wedge (s_n = t_n) \wedge C$.
  2. Fail per simboli diversi: $f(\ldots) = g(\ldots) \wedge C$ con $f \neq g$ o arità diverse $\rightarrow$ `false`.
  3. Elimina identità triviale: $X = X \wedge C \;\rightarrow\; C$.
  4. Swap: $t = X \wedge C$ (con t non variabile) $\rightarrow\; X = t \wedge C$.
  5. Binding: $X = t \wedge C$ con $X \notin \text{vars}(t)$, $X \in \text{vars}(C) \;\rightarrow\; C[X/t] \wedge X = t$.
  6. Fail per occur-check: $X = t \wedge C$ con $X \in \text{vars}(t) \;\rightarrow$ `false`.

  In Prolog reale il passo 6 (occur-check) spesso *non* viene eseguito per efficienza.
- **Terminazione/correttezza (Teorema 4.1)**
  L'algoritmo termina sempre (ogni passo riduce una "complessità" lessicografica su $N \times (multisetsinN)$). Restituisce `false` oppure un sistema in *forma risolta*:

  $$X_1 = t_1 \wedge \ldots \wedge X_n = t_n$$

  con X_i variabili distinte e $X_i \notin \text{vars}(t_j)$; da cui il m.g.u. è $\theta = [X_1/t_1, \ldots, X_n/t_n]$. θ unificatore di $C \iff \theta$ unificatore di C'.
- **Unicità dei m.g.u. (Teorema 4.2)**
  Se due risolventi finali in forma risolta si ottengono da C con due scelte diverse, le sostituzioni θ'_1, θ'_2 sono varianti l'una dell'altra (uguali a meno di permutazione di variabili).

4.3 Significato e varianti

- **Significato di unificazione**
  L'unificazione sintattica assicura che $s\theta \equiv t\theta$, quindi in qualunque interpretazione di Herbrand $s\theta$ e $t\theta$ denotino lo stesso termine *ground*. Se si vuole tenere conto di identità più "ricche" (es.: commutatività, associatività, idempotenza per $\cup$), si introduce l'idea di *E-unificazione* (teorie equazionali E). Esempio: con $E = \{ACIsu\cup\}$, $a \cup b$ e $b \cup a$ non unificano sintatticamente, ma unificano modulo E.

- **Teoria di Clark (Def. 4.10)**
  Aggiunge assiomi equazionali per uguaglianza come congruenza ($f(\ldots) = f(\ldots)\ldots$), distinzione di simboli e occur-check (CET). Con CET, `Unify(C)` equivale a C nel contesto di CET.

Capitolo 5 – SLD-risoluzione

5.1 SLD-derivazioni

- **Passo SLD (Def. 5.2)**
  Sia $G = \leftarrow A_1, \ldots, A_i, \ldots, A_n$ un *goal*. Sia $C \equiv H \leftarrow B_1, \ldots, B_m$ una clausola di P, rinominata in $H' \leftarrow B'_1, \ldots, B'_m$ con variabili nuove. Se $\theta = \text{mgu}(A_i, H') \neq \texttt{false}$, allora il risolvente G' si ottiene in due fasi:

  1. Sostituire A_i con $(B'_1 \wedge \ldots \wedge B'_m)$, ottenendo il "risolvente non istanziato":
  $$\leftarrow A_1, \ldots, A_{i-1}, B'_1, \ldots, B'_m, A_{i+1}, \ldots, A_n.$$
  2. Applicare θ a tutta la congiunzione, ottenendo
  $$G' = (\leftarrow A_1, \ldots, A_{i-1}, B'_1, \ldots, B'_m, A_{i+1}, \ldots, A_n)\theta.$$

  Si scrive $G \xrightarrow{\theta,C} G'$. G' è lo *SLD-risolvente* di G tramite C.

- **SLD-derivazione (Def. 5.3)**
  Una sequenza massimale (forse infinita) di passi SLD:
  $$G_0 \xrightarrow{\theta_1,C_1} G_1 \xrightarrow{\theta_2,C_2} G_2 \xrightarrow{\theta_3,C_3} \ldots$$
  con le clausole C_i opportunamente rinominate per evitare collisioni di variabili tra i passi.

  - *Lunghezza* = numero di passi.
  - Se la lunghezza è finita e $G_n = \leftarrow$! (goal vuoto), la derivazione è di *successo* e la risposta calcolata è $\theta_1 \circ \theta_2 \circ \ldots \circ \theta_n \mid \text{vars}(G_0)$.
  - Se finita e $G_n \not\equiv \leftarrow$! ma nessuna clausola è applicabile, la derivazione è di *fallimento*.
  - Se infinita, non è né successo né fallimento.

- **Esempi**

1. Programma "padre/2 + antenato/2" (vedi Esempio 5.2 nel testo).
2. Programma "member/2" su liste (vedi Esempio 5.3 nel testo):

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

   Per `?- member(A,B).` si ottengono infinite risposte, la prima di cui è `[A/X_1, B/[Y_1|[X_1|Z_1]]]` e le successive sono varianti.

5.2 Indipendenza dal non-determinismo nella SLD

5.2.1 Rinomina delle variabili

- Due derivazioni che differiscono solo perché a una clausola si dà una variante differente di nomi di variabili portano a risposte calcolate uguali a meno di permutazione di variabili (*varianti di risposta*).
- **Teorema 5.1 (Unicità modulo varianti)**: se due derivazioni di successo differiscono solo nella scelta degli m.g.u., le due risposte calcolate differiscono unicamente per permutazione/varianti di variabili.

5.2.2 Switching lemma e regola di selezione

- **Regola di selezione (Def. 5.6)**: funzione R che, dato un qualsiasi prefisso di derivazione fino a un goal non vuoto, individua esattamente un atomo nel goal. L'insieme IF (*Initial Fragment*) è l'insieme di tutti i prefissi di tutte le possibili derivazioni finite/infinite per ogni coppia (programma, goal).
- **Switching Lemma (Lemma 5.1)**: dati due passi consecutivi di derivazione che selezionano atomi A_i e A_j $(i < j)$, si può invertire l'ordine di selezione (prima A_j, poi A_i) ottenendo un'altra derivazione identica fuori dai due passi e senza cambiare la sostituzione complessiva dei due step (a meno di variante). In sostanza, la scelta di quale atomo selezionare non influisce (a posteriori) sul risultato, purché si usino le stesse clausole, soltanto in ordine diverso.
- **Conseguenza**: la *regola di selezione* (ad esempio `leftmost`, primo atomo da sinistra) è *completamente legittima*: se esiste qualunque derivazione di successo, allora esiste anche una derivazione di successo che seleziona sempre il primo atomo da sinistra, ottenendo la medesima risposta modulo varianti.
- **Esempio 5.4**: per `?- padre(X,Z), padre(Z,Y).`
  1. Se scelgo `padre(X,Z)` → risposta `[X/antonio, Y/davide, Z/bruno]`.
  2. Se scelgo `padre(Z,Y)` → ottengo la stessa risposta `[X/antonio, Y/davide, Z/bruno]`.

5.3 SLD-alberi e regole di selezione

- **SLD-albero**
  Dato un programma P e un goal iniziale G_0, un SLD-albero è un albero etichettato:
  - Radice: G_0.

- Archi: da un goal G, per ogni clausola $C \in P$ applicabile a ciascun atomo di G, si crea un figlio $G' = risolvente(G, C)$ (con un m.g.u. θ).
- Ogni arco è etichettato con (C, θ).

Un *percorso di derivazione* è un cammino dal nodo radice verso un nodo foglia. Se il nodo foglia è il goal vuoto ($\leftarrow$!), il percorso è di *successo*. Altrimenti, se nessuna clausola è applicabile, il percorso è di *fallimento*. Se il percorso prosegue all'infinito, è "parzialmente infinito". L'insieme delle risposte di $P \cup \{G_0\}$ è l'insieme di tutte le sostituzioni totali $\theta_1 \circ \cdots \circ \theta_n \mid \text{vars}(G_0)$ per ogni percorso di successo.

- **Regole di selezione**
  Specificano quale atomo prendere in ogni goal-nodo. Poiché lo *Switching Lemma* assicura l'indipendenza (modulo varianti), l'implementazione può scegliere sempre lo stesso criterio (ad esempio `first-left`). Un criterio fisso produce un *subalbero* (= R-SLD-albero) che contiene tutte le risposte modulo varianti reperibili nell'albero completo.

5.4 Search rule e backtracking

- **Search Rule (strategia di ricerca)**
  Molti interpreti Prolog adottano:
  - Regola di selezione: `leftmost` (primo atomo da sinistra).
  - Ricerca in profondità (depth-first search) con backtracking:
    1. Per ogni atomo, si scorre la lista di clausole dall'alto verso il basso.
    2. Si applica la prima clausola che unifica. Se successivamente fallisce, si torna indietro (backtracking) e si prova la clausola successiva.
  - Se si trova un goal vuoto ($\leftarrow$!), si riporta la prima risposta. Se l'utente richiede ';' o 'fail/0', l'interprete torna all'ultimo punto di scelta (*choice point*) e continua.

5.5 Esempi di SLD-risoluzione e backtracking

1. **Esempio "padre/2 + antenato/2"**

```
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).
```

- Query: `?- antenato(antonio,Y).`
  (a) *Scelta 1:* atomo = `antenato(antonio,Y)`, unifico con `antenato(X_1,Y_1)` (clausola 1), $\theta_1 = [X_1/antonio, Y_1/Y]$.
  (b) Nuovo goal: $\leftarrow$ `padre(antonio,Y)`; atomo = `padre(antonio,Y)`, uso `padre(antonio,bruno)`, $\theta_2 = [Y/bruno]$.

(c) Nuovo goal: $\leftarrow$!. *Successo*, risposta $\theta_1 \circ \theta_2 = [Y/bruno]$.

- *Backtracking* (';'): si torna all'ultimo *choice point* (clausola 2 per `antenato/2`):
  (a) Riprendo `antenato(antonio,Y)` con clausola 2: $\theta'_1 = [X_2/antonio, Y_2/Y]$.
  (b) Nuovo goal: $\leftarrow$ `padre(antonio,Z), antenato(Z,Y)`. Scelgo `padre(antonio,bruno)`, $\theta'_2 = [Z/bruno]$.
  (c) Nuovo goal: $\leftarrow$ `antenato(bruno,Y)`.
    – Uso clausola (1): $\theta'_3 = [X_3/bruno, Y_3/Y]$, nuovo goal: $\leftarrow$ `padre(bruno,Y)`.
    – Scelgo `padre(bruno,davide)`, $\theta'_4 = [Y/davide]$.
    – Goal: $\leftarrow$!. *Successo*, risposta $\theta'_1 \circ \theta'_2 \circ \theta'_3 \circ \theta'_4 = [Y/davide]$.
- Così si ottengono tutte le risposte in ordine di derivazione:

$$Y = bruno; \quad Y = carlo; \quad Y = davide; \quad Y = ettore; \quad nomoresolutions$$

2. **Backtracking su** `member/2`

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

- Query: `?- member(A,B).`
  (a) Prima derivazione: $B = [A|\_Z_1]$, unità di unificazione $[B/[A|Z_1]]$. *Successo.*
  (b) Backtracking: si scarta la prima clausola per la lista, si usa la seconda; il goal diventa `member(A,Z_1)`, e si ripete ricorsivamente, generando infinite soluzioni di forma $B = [\_, \ldots, [A|\_], \ldots]$.
- Ogni risposta successiva è variante della precedente.

Capitolo 6 – Semantica dei programmi logici

6.1 Semantica osservazionale

- **Semantica operazionale (SLD-risoluzione)**
  Il significato di un programma P rispetto a un goal G è l'insieme (più grande sottoinsieme) di sostituzioni θ tali che esista una derivazione di successo $G \Rightarrow^*$ (goal vuoto) con $\theta = \theta_1 \circ \cdots \circ \theta_n \mid \text{vars}(G)$. Rappresenta ciò che l'interprete Prolog effettivamente calcola.

6.2 Semantica logica (modellistica)

- **Modello minimo di Herbrand (Def. 6.4)**

  – Si considerano tutte le interpretazioni di Herbrand $I \subseteq B_{\Pi,F}$ tali che $I \models P$. Tali I sono modelli di Herbrand di P.
  – L'intersezione di tutti i modelli di Herbrand è il *modello minimo* (*least Herbrand model*, M_1).

- Si dimostra che:
  - M_1 = insieme di tutti gli atomi ground che hanno una derivazione SLD di successo.
  - M_1 = punto fisso dell'operatore T_P (*immediate consequence operator*):

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

- Quindi la semantica osservazionale (SLD) coincide con la semantica denotazionale/fixpoint.

- **Conseguenza logica (Def. 6.1)**

$$P \models A \quad (Aatomoground) \iff A \in M_1 \quad \iff \quad P \cup \{\neg A\} è insoddisfacibile in tutti i modelli di H$$

$$P \models G \quad (Ggoalnon - ground) \iff \exists \theta t.c. P \models G\theta.$$

Capitolo 7 – Programmazione in Prolog

7.1 Liste

- Rappresentazione: una lista `[E1,E2,...,En]` si scrive come `.(E1,.(E2,...(En,[])...))`. Sintassi sugar: `[H|T]` per `.(H,T)`.

- Esempi di predicati di base:

```
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/graph traversals**
  Rappresentare alberi come `node(Val, Left, Right)`; DFS/BFS scritti ricorsivamente.

- **Automi finiti**
  Rappresentati come `trans(State, Symbol, NextState)` e l'accettazione con predicato ricorsivo `accept/1`.

7.3 Liste differenza

- *Lista differenza*: scrivere liste come "differenze" di due liste, ad esempio `[1,2,3|T] - T`. Permette concatenazioni in O(1).

- Esempio: `append_dl(A-B, B-C, A-C)` concatena due liste differenza.

7.4 Predicati built-in

- **Unificabilità:** `=/2` (unificazione), `\==/2`, `==/2` (equivalenza sintattica), `\=/2`, `@</2`, `@>/2` (ordinamento termini).
- **Arithmetic:** `is/2`, `=:=/2`, `=\=/2`, `<`, `>`, `>=`, `=</2`.
- **Gestione liste:** `append/3`, `member/2`, `length/2`, `reverse/2`.
- **I/O:** `read/1`, `write/1`, `nl/0`.
- **Control:** `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)`, `arg(I, T, Arg)`.
- `=../2` (univ operator: `Term =..  [Functor|Args]`).
- `copy_term/2`, `substitute/4` per sostituzioni esplicite.
- `assert/1`, `retract/1` per modifiche dinamiche del programma.

7.6 Predicati metalogici o extralogici

- `call/1` (invoca un termine costruito a runtime), `term_variables/2`.
- `bagof/3`, `setof/3`, `findall/3` per quantificazioni "insiemisttiche" sui risultati.

7.7 Predicati I/O

- `open/3`, `close/1`, `read/2`, `write/1`, `format/2`.

7.8 Predicato FAIL

- `fail/0`: fallisce sempre, forza backtracking.

7.9 Operatori

- Dichiarare nuovi operatori: `:- op(Precedence,Type,Name).`
  - Type può essere `xf`, `yf`, `fx`, `fy`, `xfx`, `xfy`, `yfx`.
  - Esempio: `:- op(100,fy,s).` permette di scrivere `s s 0` al posto di `s(s(0))`.

Capitolo 8 – Tecniche di programmazione dichiarativa

8.1 Programmazione ricorsiva

- *Pattern "base case" e "step case"*:

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

- Alberi, grafi, traversali: implementazioni DFS, BFS con accumulatore.

8.2 Generate and test

- **Schema:**
  1. *Generate* candidate solutions (ricorsivamente).
  2. *Test* condizione, scarta o accetta.

  Esempio: generare permutazioni e testare se 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

- **map/3, filter/3, fold/4**
  - `map(Pred, List, ResultList)`, `filter(Pred, List, FilteredList)`, `fold(Pred, List, Acc, Result)`.
  - Si sfruttano clausole *higher-order* (usano `call/2`).

8.4 Il CUT (`!`)

- **Semantica**
  `!` in un corpo elimina tutti i *choice points* creati prima di esso nella stessa clausola.
  Serve per "committare" su una scelta, evitando backtracking su altre alternative.

- Esempio:

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

8.5 Meta-variable facility e metainterpretazione

- **Termini "meta-variabili"**
  È possibile trattare programmi come dati:

```
cl( Head :- Body ).
run( Query ) :- cl( Query :- Body ), run( Body ).
```

- **Meta-interpreter**

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

  Permette di "interpretare" un programma definito come insieme di fatti `cl/2`. Utile per studi di correttezza, prove, linguaggi derivati.

Capitolo 9 – Negazione e semantica estesa

9.1 Negazione as failure (NaF)

- **SLDNF-risoluzione**
  Se nel corpo compare `not A`, si tenta `A`:
  - Se `A` ha derivazione di successo → `not A` fallisce.
  - Se tutte le derivazioni di `A` (seguendo la search rule) portano a fallimento, allora `not A` ha successo.

  "Negazione per fallimento" assume che se non riesco a provare `A`, allora `A` è *falso*. Problemi: paradossi di autoriferimento, incompletezza in presenza di cicli di negazione (es. `p :- not p.` in Prolog non è gestito direttamente).

9.2 Altre interpretazioni della negazione

- **Semantica a tre valori (True/False/Undefined)**
  - Stable models.
  - Well-Founded Semantics (WFS).

Capitolo 10 – Answer Set Programming (ASP)

10.1 Regole e programmi ASP

- **Regola ASP (Def. 10.1)**

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

con A_i letterali positivi e `not` A_j letterali di negazione "default" (NaF). Alcuni sistemi consentono anche *negazione forte* $\neg A$.

- **Stable Model Semantics (Gelfond & Lifschitz)**
  Dato un'interpretazione I (insieme di atomi assunti veri), si costruisce il *programa ridotto* P^I:

  1. Rimuovere ogni regola che contiene `not` A con $A \in I$.
  2. Da ciascuna regola rimanente, cancellare tutti i letterali `not` A con $A \notin I$.

  I è *modello stabile* se I è modello minimo (al più costruttore) di P^I. Le *answer sets* sono tutti i modelli stabili.

10.2 Negazione esplicita e disgiunzione in ASP

- Alcuni dialetti ASP (e.g. DLV, Clingo) supportano *negazione forte* $\neg A$ e *disgiunzione*:
  $$A \; ; \; B \; :- \text{Body}.$$
- La semantica estesa richiede grounding e risoluzione SAT-like (Gringo & clasp).

10.3 Tecniche di programmazione ASP

- **Cardinality Constraints**
  $$\{ \, X : p(X) \, \} = k, \quad < k, \quad \leq k, \quad \geq k$$
- **Aggregati**
  $$\#\text{count}\{X : p(X)\} = N, \quad \#\text{sum}\{ \, V : \text{cost}(V)\} = C.$$

10.4 ASP-solver e cenni a SAT e Cmodels

- **Workflow tipico:**
  1. *Grounding*: espansione delle variabili su domini finiti (Gringo).
  2. *Solve*: traduzione in SAT, utilizzo di solutore SAT (clasp, Cmodels).
  3. *Traduzione inversa*: associare assegnazioni di proposizioni agli atomi ASP.

Capitolo 11 – Esempi di problemi risolvibili con ASP

- Marriage problem.
- N-regine.
- Zebra puzzle.
- Map coloring (colorazione grafi).
- Circuito hamiltoniano.

- k-clique, vertex cover.
- Task allocation (assegnazione compiti).
- Knapsack.
- Schur numbers.
- Protein structure prediction.
- Altri esercizi vari.

Capitolo 12 – Answer Set Planning

- **Rappresentazione azioni in ASP**
  Atomi: `occ(Azione, Time)`, `holds(Fluent, Time)`.
- **Frame-axioms** gestite con regole di inertia.
- Pianificazione come problema di soddisfacimento bottom-up, generando istanti di tempo fino a un limite T.

- Esecuzione condizionata: es. `A :- Condition, not` $\neg A.(pianicondizionati, rami)$.

Capitolo 13 – Vincoli e risolutori (CLP)

13.1 Vincoli e CSP

- **Vincolo**: relazione booleana su variabili con dominio D.
- **Problema binario**: insieme di variabili X, dominio D, insieme di vincoli $C \subseteq D^n$.
- **Domini**:
  - CLP(FD): dominio finito.
  - CLP(R): dominio continuo.

13.2 Resolver di vincoli

- **Arc consistency, backtracking, forward-checking**
  Algoritmi AC-3, AC-4.
- **Constraint propagation**
  - Esempio: somma, $<$, $\leq$, $\neq$, `all_different`.
  - Filtri globali: `all_different/1`, `cumulative/1`, `element/3`.
- **Esperimenti con solver**
  Esempi con ECLiPSe, SICStus, GNU-Prolog, SICStus CLP(R).

Capitolo 14 – Programmazione logica con vincoli (CLP)

14.1 Sintassi e semantica operazionale

- **Clausola CLP**

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

  dove C_i è congiunzione di vincoli (es.: $Xin1..10$, $X\#/ = Y$), gli A_j sono atomi, e H è la testa.

- **Derivazione CLP**
  Mantiene un insieme di vincoli che si *propagano* e riducono i domini in base ai predicati di propagazione.

- **CLP(FD) in SICStus/GNU-Prolog**

  – Predicati: `#=/2`, `#\=/2`, `#>/2`, `#>=/2`, `#=</2`, `#in/2`, `labeling/2`.
  – Propagazione mediante eventi (domain reduction).

- **Vincoli reificati**
  Trasformano un vincolo in variabile booleana: `B #<=> X #= Y`. Permettono combinazioni logiche di vincoli.

- **CLP(R)**
  Variabili reali, vincoli lineari/non lineari. Predicati: `{}/1` per dimensioni vincoli, `freeze/2` per procrastinare la valutazione.

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

15.1 Schema generale

- **Constrain & Generate**

  1. *Dichiarare vincoli* su variabili (riduzione domini).
  2. *Generare* (labeling) assegnazioni residue.

- **Esempi classici:**

  1. **N-regine:**

```
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).
```

2. **Knapsack:**

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

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

```
coloring(Regions, Colors, Assignment) :-
    length(Assignment, Regions),
    Assignment ins 1..Colors,
    % vincoli per regioni contigue
    all_different([C || (R,C) in Assignment, (R1, C1) in Assignment,
    labeling([], Assignment).
```

4. Altri: Marriage problem, SEND+MORE=MONEY, Schur, allocation tasks, Hamiltonian circuit, ecc.

Appendici

- **Appendice A**: Concetti base di ordini, reticoli, punti fissi (Knaster–Tarski, teoria dei lattice).
- **Appendice B**: Suggerimenti d'uso di Prolog e ASP-solver (differenze, trucchi, parametri).
- **Appendice C**: Soluzioni degli esercizi di ogni capitolo (2–15).
- **Bibliografia**: Elenco di testi e articoli fondamentali (Kowalski, Lloyd, Gelfond, ecc.).