

Politecnico di Milano  
Facoltà di ingegneria Milano Leonardo  
Corso di Ingegneria della conoscenza e sistemi esperti

# **Appunti dalle lezioni**

## **Parte II: capitoli 4–7**

Marco Colombetti

### **Avvertenza**

Queste dispense sono disponibili sul sito web

<http://www.elet.polimi.it/Users/DEI/Sections/Compeng/Marco.Colombetti/ICSE.html>. La proprietà intellettuale è dell'autore. È consentita la riproduzione e la distribuzione con qualunque mezzo e sotto qualsiasi forma purché non a fini di lucro.

## **Indice (parte II)**

<b>4.</b>	<b>La conoscenza come teoria del mondo</b>	<b>3</b>
4.1	Le teorie del primo ordine	
4.2	Caratterizzazione di un modello finito	
4.3	Le assunzioni di chiusura	
4.4	Le teorie generali	
<b>5.</b>	<b>I sistemi di produzioni</b>	<b>11</b>
5.1	L'architettura di un sistema di produzioni	
5.2	Produzioni e logica del primo ordine	
<b>6.</b>	<b>Interazioni fra regole</b>	<b>22</b>
6.1	Conflitti fra regole	
6.2	Confronto fra strategie	
6.3	I contesti	
<b>7.</b>	<b>La rappresentazione del cambiamento</b>	<b>32</b>
7.1	Rappresentazione logica del cambiamento	
7.2	Rappresentazione del cambiamento in CLIPS	
	<b>Riferimenti bibliografici (parte II)</b>	<b>37</b>

## 4 La conoscenza come teoria del mondo

### 4.1 Le teorie del primo ordine

Consideriamo la seguente argomentazione:

Anna è più alta di Bruno.  
 Bruno è più alto di Chiara.  
 Quindi Anna è più alta di Chiara.

L'argomentazione sembra intuitivamente valida. Proviamo a rappresentarla in una logica del primo ordine per verificare questa intuizione. Scegliamo le tre costanti  $a$ ,  $b$  e  $c$  per denotare rispettivamente Anna, Bruno e Chiara, e il predicato binario  $A^+(-,-)$  per esprimere la proprietà "più alto di". L'argomentazione diventa allora

$$A^+(a,b), A^+(b,c), \therefore A^+(a,c).$$

L'argomentazione è valida se e solo se

$$A^+(a,b), A^+(b,c) \models A^+(a,c).$$

Applicando il calcolo degli alberi semantici otteniamo:

1.  $A^+(a,b)$       premessa       $a, b$
2.  $A^+(b,c)$       premessa       $c$
3.  $\neg A^+(a,c)$      $\neg$ conclusione

Qui il calcolo si ferma, mostrando che l'argomentazione *non* è valida. Perché?

La risposta è molto semplice: la validità intuitiva dell'argomentazione si basa anche su una premessa talmente ovvia che è stata lasciata sottintesa. Ovvero: la relazione "più alto di" gode della proprietà transitiva. Proviamo a riformulare l'argomentazione esplicitando anche questa premessa:

$$\forall x \forall y \forall z (A^+(x,y) \wedge A^+(y,z) \rightarrow A^+(x,z)), A^+(a,b), A^+(b,c) \models A^+(a,c).$$

Ora è facile verificare la validità:

1.  $\forall x \forall y \forall z (A^+(x,y) \wedge A^+(y,z) \rightarrow A^+(x,z))$       premessa
2.  $A^+(a,b)$       premessa       $a, b$
3.  $A^+(b,c)$       premessa       $c$
4.  $\neg A^+(a,c)$        $\neg$ conclusione
5.  $A^+(a,b) \wedge A^+(b,c) \rightarrow A^+(a,c)$        $\forall(/a) \forall(/b) \forall(/c)$
6.  $A^+(a,b) \wedge A^+(b,c)$        $+\wedge(2,3)$
7.  $A^+(a,c)$       MP(5,6)
8.  $\perp$        $\perp(7,4)$

dove abbiamo utilizzato la regola derivata di *introduzione della congiunzione*,

**rule**  $+ \wedge$   
**if**  $j, y$   
**then**  $j \wedge y$ .

Ora, c'è una differenza fra la prima premessa e le altre due: mentre queste riguardano uno stato di cose particolare (le altezze relative di Anna, Bruno e Chiara), la prima premessa esprime una proprietà generale della relazione “più alto di”, indipendente dai particolari individui considerati. Questa premessa può essere giustamente considerata come elemento di una *teoria generale* della relazione “più alto di”. Più precisamente, diremo che è un *assioma* di questa teoria.

In generale, un insieme di assiomi è un insieme arbitrario di enunciati,  $\mathbf{G}$ , che consideriamo come premesse date una volta per tutte e che ci riserviamo di utilizzare in una dimostrazione ovunque ve ne sia il bisogno. L'insieme dei teoremi dimostrabili a partire dagli assiomi,  $Th(\mathbf{G})$ , è detto la *teoria* derivabile da  $\mathbf{G}$ .

Intuitivamente, una teoria può rappresentare le conoscenze che un agente possiede su una determinata porzione della realtà (in questo caso, sulla relazione “più alto di”). L'aspetto interessante delle teorie assiomatiche è che le conoscenze dell'agente sono ricondotte a un insieme di enunciati di partenza, gli assiomi, da cui ogni altro enunciato della teoria è derivabile.

Fino a qui abbiamo esaminato una teoria dal punto di vista della derivabilità, e quindi del calcolo. Possiamo però fare considerazioni analoghe sul piano semantico, e quindi della conseguenza logica. Considerando le proprietà di correttezza e completezza del calcolo, è chiaro che  $Th(\mathbf{G})$  è costituita esattamente dagli enunciati che sono veri in tutti i modelli che verificano  $\mathbf{G}$ . In altre parole, invece di concentrarci sugli enunciati *validi in assoluto* (cioè veri in tutti i possibili modelli) abbiamo spostato l'attenzione sugli enunciati *validi relativamente agli assiomi* (cioè veri in tutti i modelli che verificano gli assiomi). Ciò significa che, dal punto di vista semantico, l'effetto degli assiomi è di restringere la classe dei modelli di riferimento: dalla totalità dei modelli passiamo a considerare soltanto i modelli che verificano gli assiomi. L'argomentazione

$$A^+(a,b), A^+(b,c), \therefore A^+(a,c).$$

non è certo valida se prendiamo in considerazione tutti i possibili modelli; ma lo diventa se restringiamo la nostra attenzione alla classe di modelli che verificano

$$\forall x \forall y \forall z (A^+(x,y) \wedge A^+(y,z) \rightarrow A^+(x,z));$$

ovvero, alla classe di modelli in cui l'estensione del predicato  $A^+(-,-)$  è una relazione transitiva.

In base a queste considerazioni formuliamo un'ipotesi di lavoro: le conoscenze di un agente possono essere viste come una teoria assiomatica che l'agente possiede attorno al mondo con cui interagisce. In altre parole, la conoscenza è una “teoria del mondo”. Nei prossimi paragrafi cercheremo di sviluppare questa ipotesi.

## 4.2 Caratterizzazione di un modello finito

Cominciamo con un esempio molto semplice. Il mondo del nostro agente è costituito da un certo numero di cubetti (come quelli con cui giocano i bambini), che possono essere posti su un ripiano e impilati l'uno sull'altro. La figura 4.1 mostra un mondo di questo genere contenente tre cubetti.

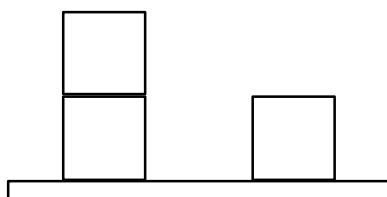


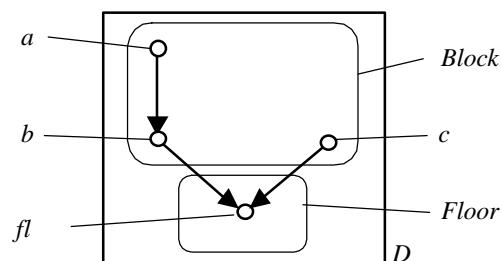
Figura 4.1 *Un mondo di cubetti.*

Ci chiediamo ora come si possa costruire una teoria assiomatica che descriva questo mondo. Ora, come sappiamo una teoria logica non descrive mondi concreti, bensì *modelli*. Il primo passo quindi è rappresentare il mondo della figura 4.1 con un modello del primo ordine.

Definiamo quindi un modello  $\mathbf{M}$  basato su un dominio  $\mathbf{D}$  di quattro individui, tre cubetti e un ripiano. Per prima cosa definiamo un linguaggio del primo ordine che contiene:

- tre costanti ( $a$ ,  $b$  e  $c$ ) per denotare i tre cubetti e una costante ( $fl$ ) per denotare il ripiano;
- un predicato unario,  $Block(-)$ , la cui estensione è l'insieme dei tre cubetti
- un predicato unario,  $Floor(-)$ , la cui estensione è l'unico ripiano
- un predicato binario,  $On(-,-)$ , la cui estensione è data dalle coppie ordinate di individui che stanno l'uno sull'altro.

Il modello risultante è descritto graficamente nella figura 4.2.

Figura 4.2 *Rappresentazione grafica del modello M.*

Vogliamo ora caratterizzare il modello  $\mathbf{M}$  “direttamente”, ovvero senza costruire una teoria generale dei mondi costituiti di cubetti. Per fare questo, dobbiamo fornire tre insiemi di assiomi:

- Gli *assiomi di dominio* di  $\mathbf{M}$ , che indicheremo con  $D(\mathbf{M})$ . La funzione degli assiomi di dominio è di caratterizzare il dominio  $D$  di  $\mathbf{M}$  come insieme costituito da esattamente quattro individui distinti.
- Il *diagramma positivo* di  $\mathbf{M}$ , che indicheremo con  $\Delta^+(\mathbf{M})$ . La funzione del diagramma positivo è di elencare i fatti esprimibili nel linguaggio formale che sussistono nel modello.
- Il *diagramma negativo* di  $\mathbf{M}$ , che indicheremo con  $\Delta^-(\mathbf{M})$ . La funzione del diagramma negativo è di elencare i fatti esprimibili nel linguaggio formale che *non* sussistono nel modello.

#### *Gli assiomi di dominio*

Il dominio del modello può essere caratterizzato nel modo seguente. Per prima cosa imponiamo che non esistano altri individui oltre a quelli denotati dalle costanti  $a$ ,  $b$ ,  $c$  e  $fl$ :

$$D_0. \quad \forall x (x = a \vee x = b \vee x = c \vee x = fl).$$

Poi imponiamo che gli individui denotati dalle costanti  $a$ ,  $b$ ,  $c$  e  $fl$  siano tutti distinti:

$$D_1. \quad A \neq b,$$

$$D_2. \quad A \neq c,$$

$$D_3. \quad A \neq fl,$$

$$D_4. \quad B \neq c,$$

$$D_5. \quad B \neq fl,$$

$$D_6. \quad C \neq fl.$$

In generale, se il dominio è costituito da  $n$  individui distinti gli assiomi di dominio sono esattamente  $1 + n(n-1)/2$ . Supponiamo infatti di denotare gli individui con le costanti  $a_1, \dots, a_n$ . Gli assiomi di dominio saranno allora:

$$D_0. \quad \forall x (x = a_1 \vee \dots \vee x = a_n),$$

$$D_{ij}. \quad A_i \neq a_j, \quad \text{per } 1 \leq i < j \leq n.$$

Questi assiomi impongono l'esistenza di una corrispondenza biunivoca fra le costanti del linguaggio formale e gli individui del dominio. Si noti che questo metodo può essere applicato soltanto ai domini finiti: se il dominio ha cardinalità infinita, infatti, non è possibile scrivere l'assioma  $D_0$ .

In realtà, le cose sono un po' più complesse, perché è sempre possibile interpretare il simbolo '=' come una relazione binaria diversa dall'identità fra gli individui del dominio. Infatti, le regole che abbiamo introdotto per l'uguaglianza valgono altrettanto bene se il simbolo '=' è interpretato come una relazione di congruenza nel modello, ovvero come una relazione di equivalenza nel dominio compatibile con tutte le proprietà e relazioni definite nel modello. Tuttavia, soltanto i modelli in cui il simbolo '=' è interpretato come l'identità sul dominio corrispondono al significato intuitivo che attribuiamo all'uguaglianza. Tali modelli sono detti *normali*. Gli assiomi di dominio, quindi, garantiscono l'esistenza di una corrispondenza biunivoca fra le costanti del linguaggio formale e gli individui del dominio di ogni modello normale.

#### Diagramma positivo

Il diagramma positivo è costituito da tutti gli *enunciati fattuali* (ovvero, enunciati atomici privi di variabili) che sono esprimibili nel linguaggio e veri nel dominio. Nel nostro caso si tratta di sette assiomi, che disponiamo secondo uno schema a tabella che motiveremo fra poco:

$\Delta^+_1.$	$Block(a),$	$\Delta^+_2.$	$Block(b),$	$\Delta^+_3.$	$Block(c),$		
	–		–		–		$\Delta^+_4.$
			$\Delta^+_5.$		–		–
	–				–		$\Delta^+_6.$
	–		–		–		$\Delta^+_7.$
	–		–		–		–

#### Diagramma negativo

Il diagramma negativo è costituito dalla negazione di tutti gli enunciati fattuali esprimibili nel linguaggio e falsi nel dominio. Nel nostro caso si tratta di diciassette assiomi:

							$\Delta^-_1.$
	–		–		–		$\neg Block(fl),$
$\Delta^-_2.$	$\neg Floor(a),$	$\Delta^-_3.$	$\neg Floor(b),$	$\Delta^-_4.$	$\neg Floor(c),$		–
$\Delta^-_5.$	$\neg On(a,a),$		–	$\Delta^-_6.$	$\neg On(a,c),$	$\Delta^-_7.$	$\neg On(a,fl),$
$\Delta^-_8.$	$\neg On(b,a),$	$\Delta^-_9.$	$\neg On(b,b),$	$\Delta^-_{10}.$	$\neg On(b,c),$		–
$\Delta^-_{11}.$	$\neg On(c,a),$	$\Delta^-_{12}.$	$\neg On(c,b),$	$\Delta^-_{13}.$	$\neg On(c,c),$		–
$\Delta^-_{14}.$	$\neg On(fl,a),$	$\Delta^-_{15}.$	$\neg On(fl,b),$	$\Delta^-_{16}.$	$\neg On(fl,c),$	$\Delta^-_{17}.$	$\neg On(fl,fl).$

È ovvio che il diagramma positivo e il diagramma negativo sono complementari: ogni fatto esprimibile nel linguaggio deve appartenere al primo o al secondo. Negli schemi precedenti, come è chiaramente visibile, ogni "buco" di uno schema corrisponde a un assioma presente nell'altro schema.

In generale, i fatti che sussistono in un modello sono molto meno numerosi dei fatti che non sussistono; in altre parole, i modelli sono strutture “sparse”. Ciò significa che il diagramma negativo conta in generale molti più assiomi del diagramma positivo.

### Teorie categoriche

Consideriamo ora la teoria assiomatica  $T(\mathbf{M})$  i cui assiomi sono costituiti dall'insieme di enunciati

$$T(\mathbf{M}) = D(\mathbf{M}) \cup \Delta^+(\mathbf{M}) \cup \Delta^-(\mathbf{M}).$$

Possiamo dire che questa teoria caratterizza il modello  $\mathbf{M}$ ? Innanzi tutto dobbiamo stabilire che cosa significhi che una teoria caratterizza un modello. Intuitivamente, vorremmo poter dire che una teoria  $\mathbf{G}$  caratterizza un modello  $\mathbf{M}$  se  $\mathbf{M}$  è l'unico modello di  $\mathbf{G}$ , ovvero l'unico modello che rende veri tutti gli assiomi di  $\mathbf{G}$  (più precisamente, dovremmo dire che tutti i modelli di  $\mathbf{G}$  sono isomorfi a  $\mathbf{M}$ ). Anche se questa proprietà non sussiste, è possibile dimostrare che tutti i *modelli normali* di  $\mathbf{G}$  sono isomorfi al modello descritto nella figura 4.2. In questo senso possiamo dire che  $\mathbf{G}$  caratterizza tale modello (anche se soltanto all'interno della classe dei modelli normali).

Una teoria logica che ammetta un solo modello normale (a meno di isomorfismi) è detta *categorica*. Dunque la teoria  $T(\mathbf{M})$  è una teoria categorica. In generale, se  $\mathbf{G}$  è una teoria categorica di  $\mathbf{M}$  abbiamo che

$$\mathbf{G} \vdash \mathbf{j} \quad \text{se e solo se} \quad \models_{\mathbf{M}} \mathbf{j},$$

ovvero: un enunciato  $\mathbf{j}$  deriva da  $\mathbf{G}$  se e solo se  $\mathbf{j}$  è vero in  $\mathbf{M}$ . Sappiamo infatti che

$$\mathbf{G} \vdash \mathbf{j} \quad \text{se e solo se} \quad \mathbf{G} \models \mathbf{j};$$

ma dato che  $\mathbf{M}$  è l'unico modello di  $\mathbf{G}$ ,

$$\mathbf{G} \models \mathbf{j} \quad \text{se e solo se} \quad \models_{\mathbf{M}} \mathbf{j}.$$

### 4.3 Le assunzioni di chiusura

Consideriamo di nuovo il diagramma positivo di  $\mathbf{M}$ :

- $\Delta^+_1.$  *Block(a)*,
- $\Delta^+_2.$  *Block(b)*,
- $\Delta^+_3.$  *Block(c)*,
- $\Delta^+_4.$  *Floor(fl)*,
- $\Delta^+_5.$  *On(a,b)*,
- $\Delta^+_6.$  *On(b,fl)*,
- $\Delta^+_7.$  *On(c,fl)*.

Non è difficile convincersi che  $\Delta^+$  contiene tutte le informazioni necessarie per ricostruire completamente il modello  $\mathbf{M}$ . Infatti,  $\Delta^+$  contiene quattro costanti che corrispondono ai quattro individui del dominio e descrive i sette fatti che sussistono nel modello. Tuttavia, non è possibile considerare  $\Delta^+$  come una teoria categorica di  $\mathbf{M}$ , perché in assenza degli assiomi di dominio e del diagramma negativo non è vero che ogni enunciato vero in  $\mathbf{M}$  deriva da  $\Delta^+$ . Ne concludiamo che il solo calcolo logico non è in grado di “estrarre” tutta l'informazione contenuta in  $\Delta^+$ . Ciò dipende da due motivi. Il primo è che gli assiomi di  $\Delta^+$  non vietano al dominio di contenere altri individui oltre a quelli denotati dalle costanti presenti; il secondo è che, per così dire, il calcolo “non sa” che  $\Delta^+$  è effettivamente il diagramma positivo del modello.

Una situazione analoga si ha con le basi di dati. In effetti, una base di dati può essere vista come il diagramma positivo di un modello: ogni relazione corrisponde a un predicato, e le tuple che popolano la relazione corrispondono agli enunciati fattuali del diagramma positivo. Nel campo delle basi di dati si adottano le cosiddette *assunzioni di chiusura*:

- *Assunzione di chiusura del dominio*: c'è una corrispondenza biunivoca fra gli individui del dominio e le costanti utilizzate nelle tuple della base di dati.
- *Assunzione del mondo chiuso*: c'è una corrispondenza biunivoca fra i fatti che sussistono nel modello e le tuple della base di dati.

Come è facile notare, l'assunzione di chiusura del dominio sostituisce gli assiomi di dominio, e l'assunzione del mondo chiuso consente di ricavare il diagramma negativo del modello per differenza rispetto al contenuto della base dei dati.

Per ottenere lo stesso risultato mantenendoci all'interno di un sistema logico, definiamo una nuova relazione semantica che consente di estrarre tutte le informazioni contenute in  $\Delta^+$ . Dato un insieme  $F$  di enunciati fattuali (ovvero, enunciati atomici privi di variabili) diciamo che  $j$  è conseguenza logica di  $F$  sotto le *assunzioni di chiusura*, in simboli

$$F \models j,$$

se e solo se  $j$  è vero nel più piccolo modello di cui  $F$  sia il diagramma positivo, dove per “più piccolo modello” intendiamo il modello con il più piccolo dominio possibile (si dimostra che tale modello esiste sempre quando  $F$  è un insieme di enunciati fattuali). È evidente che

$$\text{se } F \models j, \text{ allora } F \models j,$$

mentre in generale non è vero il contrario. Pertanto il calcolo degli alberi semantici (o un altro calcolo equivalente) è corretto anche rispetto a  $\models$ , ma in generale non è più completo. Nel seguito vedremo che certe derivazioni, scorrette in generale, sono invece corrette rispetto a  $\models$ .

#### 4.4 Le teorie generali

Come abbiamo visto, la teoria

$$T(\mathbf{M}) = D(\mathbf{M}) \cup \Delta^+(\mathbf{M}) \cup \Delta^-(\mathbf{M})$$

caratterizza il modello  $\mathbf{M}$  (purché questo sia finito), e adottando le assunzioni di chiusura è addirittura possibile caratterizzare  $\mathbf{M}$  con il solo diagramma positivo  $\Delta^+(\mathbf{M})$ .

Ritorniamo ora all'esempio delle figure 4.1 e 4.2. Possiamo notare che sarebbe possibile ricostruire il modello partendo da un insieme di fatti ancora più piccolo del diagramma positivo. Supponiamo infatti di sapere che:

- gli enunciati  $On(a,b)$ ,  $On(b,fl)$  e  $On(c,fl)$  sono veri;
- nessun altro enunciato fattuale del tipo  $On(-,-)$  è vero.

A questo punto, utilizzando la nostra conoscenza di come è fatto il mondo dei cubetti possiamo dedurre che:

- $a$ ,  $b$  e  $c$  sono cubetti, in quanto solo i cubetti possono trovarsi sopra un altro oggetto;
- $fl$  è un ripiano, in quanto solo un ripiano può non trovarsi sopra un altro oggetto.

Ci chiediamo ora come un agente artificiale potrebbe eseguire le stesse inferenze. La risposta è che la conoscenza sul mondo dei cubetti può essere rappresentata da una teoria basata su assiomi generali, ovvero validi non per una specifica disposizioni di cubetti, ma per tutte le disposizioni possibili.

Una teoria del genere potrebbe basarsi sugli assiomi seguenti:

- A<sub>1</sub>.  $\exists!x \text{Floor}(x)$ ,  
*esiste esattamente un ripiano*<sup>1</sup>.
- A<sub>2</sub>.  $\forall x (\text{Block}(x) \leftrightarrow \neg \text{Floor}(x))$ ,  
*ogni oggetto o è un cubetto o è un ripiano.*<sup>2</sup>
- A<sub>3</sub>.  $\forall x (\text{Block}(x) \rightarrow \exists!y \text{On}(x,y))$ ,  
*ogni cubetto poggia su esattamente un oggetto.*
- A<sub>4</sub>.  $\forall x (\text{Block}(x) \rightarrow \exists'y \text{On}(y,x))$ ,  
*su ogni cubetto poggia al più un oggetto.*
- A<sub>5</sub>.  $\forall x (\text{Floor}(x) \rightarrow \neg \exists y \text{On}(x,y))$ ,  
*un ripiano non poggia su nessun oggetto.*

Questi assiomi, che esprimono proprietà intuitivamente vere nel mondo dei cubetti, non sono sufficienti, perché finora nulla impedisce a una torre di cubetti di essere circolare (il cubetto più in basso poggia sul cubetto più in alto). Per escludere questa eventualità introduciamo un nuovo predicato binario, *Above*(–,–), con il seguente significato intuitivo: *Above*(*x*,*y*) significa che *x* si trova al di sopra di *y* (anche senza contatto). Il predicato *Above* è caratterizzato dal seguente assioma, che lo definisce ricorsivamente a partire da *On*:

- A<sub>6</sub>.  $\forall x \forall y (\text{Above}(x,y) \leftrightarrow \text{On}(x,y) \vee \exists z (\text{On}(x,z) \wedge \text{Above}(z,y)))$ ,  
*un primo oggetto si trova al di sopra di un secondo oggetto se e solo se: il primo oggetto poggia sul secondo oggetto oppure esiste un terzo oggetto tale che il primo oggetto poggia sul terzo oggetto e il terzo oggetto si trova al di sopra del secondo oggetto.*

A questo punto è sufficiente stabilire che ogni cubetto si deve trovare al di sopra di un ripiano:

- A<sub>7</sub>.  $\forall x (\text{Block}(x) \rightarrow \exists y (\text{Floor}(y) \wedge \text{Above}(x,y)))$ ,  
*ogni cubetto si trova al di sopra di un ripiano.*

A questo punto disponiamo di una teoria assiomatica generale dei mondi di cubetti. Supponiamo, come suggerito all'inizio del paragrafo, di sapere che:

- gli enunciati *On*(*a*,*b*), *On*(*b*,*fl*) e *On*(*c*,*fl*) sono veri;
- nessun altro enunciato fattuale del tipo *On*(–,–) è vero.

Utilizzando gli assiomi è possibile dedurre che:

*Block*(*a*),  
*Block*(*b*),  
*Block*(*c*),  
*Floor*(*fl*).

Disporre della teoria generale ci ha quindi consentito di ricostruire tutto il diagramma positivo del modello a partire dai tre enunciati *On*(*a*,*b*), *On*(*b*,*fl*) e *On*(*c*,*fl*). I tre enunciati costituiscono inoltre un

<sup>1</sup> Il quantificatore  $\exists!x$ , “esiste esattamente un *x* tale che” si può definire nella logica predicativa del primo ordine con eguaglianza nel modo seguente. Per prima cosa definiamo il quantificatore  $\exists'x$ , “esiste al più un *x* tale che”:

$$\exists'x \mathbf{j} \stackrel{\text{def}}{=} \forall u \forall v (\mathbf{j}(u/x) \wedge \mathbf{j}(v/x) \rightarrow u = v).$$

A questo punto è possibile definire:

$$\exists!x \mathbf{j} \stackrel{\text{def}}{=} \exists x \mathbf{j} \wedge \exists'x \mathbf{j}.$$

<sup>2</sup> Si noti che una formula del tipo  $(\mathbf{j} \leftrightarrow \neg \mathbf{y})$  equivale a un “or esclusivo” ( $\mathbf{j} \mathbf{xor} \mathbf{y}$ ).

insieme *minimale* al fine di ricostruire il diagramma positivo (ovvero, il diagramma positivo non è ricostruibile a partire da solo due dei tre enunciati).

Un insieme minimale di enunciati fattuali che, unitamente agli assiomi generali, consenta di ricostruire tutto il diagramma positivo del modello è detto *nucleo* del modello. Come si vede dall'esempio, il nucleo di un modello contiene tutta l'informazione necessaria per ricostruire completamente il modello corrispondente.

I concetti di teoria generale, di diagramma positivo e di nucleo sono la base concettuale per le rappresentazioni *basate su fatti* (*fact based*) utilizzate nell'intelligenza artificiale. Nei prossimi capitoli vedremo come utilizzare un *sistema di produzioni* per manipolare una rappresentazione basata su fatti.

## 5 I sistemi di produzioni

I *sistemi di produzioni* (PS, *production systems*) sono sistemi di elaborazione guidati da eventi, spesso utilizzati per implementare sistemi inferenziali.

I PS hanno la loro origine nel concetto di produzione introdotto da Post (1943) come modello di computazione. In seguito sono stati proposti da Newell e Simon (1972) come modello generale per i sistemi intelligenti e come tali sono stati presi in considerazione anche all'interno della psicologia cognitiva. I primi sistemi di questo genere, tuttavia, erano estremamente inefficienti. Il primo PS abbastanza efficiente da poter essere applicato a problemi non banali è stato OPS5, basato sull'algoritmo RETE di Forgy (1979). Attualmente sono diffusi due sistemi che discendono direttamente da OPS5: CLIPS, implementato in ambiente C, e Jess, molto simile a CLIPS ma implementato in ambiente Java. Ambedue i sistemi possono essere scaricati dal web con la relativa documentazione:

- CLIPS: <http://www.ghg.net/clips/CLIPS.html>;
- Jess: <http://herzberg.ca.sandia.gov/jess/index.html>.

Nel seguito farò sempre riferimento a CLIPS. In questo capitolo vedremo l'architettura generale del sistema, gli elementi essenziali di CLIPS e un certo numero di esempi.

### 5.1 L'architettura di un sistema di produzioni

Il sistema CLIPS è costituito da varie componenti, tra cui le principali sono:

- un' *interfaccia* verso l'utente;
- una *base dei fatti*, contenente strutture dati statiche denominate *elementi* o *fatti*;
- una *base delle regole*, contenente strutture eseguibili denominate *produzioni* o *regole*;
- un *valutatore di espressioni funzionali*, che si occupa di *valutare espressioni* del tipo funzione-argomenti;
- un *interprete* o *motore inferenziale* (*inference engine*), che si occupa di *attivare le regole* e di *eseguire le attivazioni* delle regole;

A sua volta, l'interprete utilizza alcune strutture dati essenziali per il suo funzionamento, e in particolare:

- l'*agenda*, che mantiene un insieme ordinato di *attivazioni di regole*;
- il *focus stack*, di cui per ora non parleremo, che mantiene una pila di possibili "fuochi" di attivazione.

L'idea generale è la seguente. Nella base dei fatti vengono inseriti o eliminati dei fatti. L'inserimento o l'eliminazione di un fatto è un *evento* che può attivare una o più regole, e in tal caso un'attivazione viene inserita nell'agenda. L'attivazione in testa all'agenda viene poi eseguita, e in conseguenza di questo altri fatti possono essere inseriti o eliminati dalla base dei fatti. Questo procedimento continua fino a che l'agenda si vuota oppure viene forzata la terminazione.

Vediamo ora il funzionamento delle varie componenti del sistema.

#### L'interfaccia

Nel sistema CLIPS, l'interfaccia utilizza un certo numero di finestre, tra cui:

- la *finestra di dialogo*, in cui avviene il dialogo fra l'utente e l'interprete;

- la *finestra dei fatti*, che mostra lo stato corrente della base dei fatti;
- la *finestra dell'agenda*, che mostra lo stato corrente dell'agenda.

La finestra di dialogo presenta il prompt

CLIPS>

dopo il quale possono essere inserite *espressioni funzionali* da valutare. Le espressioni funzionali di CLIPS, molto simili a quelle del linguaggio Lisp, sono costituite dalla chiamata di una *funzione* applicata a un certo numero di *argomenti*. L'interfaccia trasmette l'espressione al valutatore delle espressioni funzionali; l'eventuale valore restituito viene visualizzato, dopodiché viene presentato un nuovo prompt. Le altre finestre dell'interfaccia verranno descritte in seguito.

### La base dei fatti

La base dei fatti contiene un insieme di strutture dati dette *fatti* o *elementi*. Nel seguito userò sempre il termine "elemento" e riserverò il termine "fatto" a quegli elementi che rappresentino effettivamente dei fatti nel senso visto nei capitoli precedenti. In CLIPS un elemento può essere:

- una *lista*, ovvero una struttura del tipo  $(\text{simbolo costante}_1 \dots \text{costante}_n)$ , dove *simbolo* è un identificatore alfanumerico e *costante* è un simbolo, un numero o una stringa (sequenza di caratteri racchiusa fra doppi apici);
- un *template*, ovvero una struttura del tipo  $(\text{simbolo}_0 (\text{simbolo}_1 \text{costante}_1) \dots (\text{simbolo}_n \text{costante}_n))$ , con le stesse convenzioni di cui sopra.

Per il momento useremo solamente liste.

Un elemento può essere utilizzato liberamente dal programmatore per rappresentare qualcosa. In particolare può essere utilizzato per rappresentare un fatto: il simbolo iniziale rappresenta allora il predicato e le costanti rappresentano gli argomenti. Ad esempio:

(Piove)	predicato a zero argomenti,
(Persona alberto)	predicato a due argomenti,
(Età barbara 22)	predicato a due argomenti.

All'inizio di una sessione di lavoro, la base dei fatti è vuota. Il sistema mette a disposizione una funzione, `assert`, che consente di inserire elementi nella base dei fatti. A ogni elemento inserito viene associato un *indice* intero, che inizia da zero e viene incrementato di uno a ogni elemento inserito. Esempio:

<i>dialogo</i>	<i>base dei fatti</i>
CLIPS>	<i>vuota</i>
CLIPS> (assert (Piove))	0: (Piove)
CLIPS> (assert (Persona alberto)(Persona barbara))	0: (Piove)
	1: (Persona alberto)
	2: (Persona barbara)

Si noti che la valutazione di `(assert elemento)` non ha alcun effetto se *elemento* è già contenuto nella base dei fatti.

Oltre ad `assert` esistono altre funzioni per la manipolazione della base dei fatti, di cui ci occuperemo in seguito.

Dal punto di vista concettuale, la base dei fatti può essere vista come una specie di base di dati relazionale: basta interpretare il simbolo iniziale della lista come il nome di una tabella, e le costanti successive della lista come i valori di una tupla.

### Variabili, ambienti, pattern

Prima di proseguire è necessario introdurre un processo fondamentale per tutti i sistemi di produzioni: l'*accoppiamento di pattern*, o *pattern matching*.

Per prima cosa, definiamo il concetto di pattern. Un *pattern* è una lista che può contenere zero o più variabili; a sua volta, una *variabile* è un simbolo preceduto dal carattere speciale "?". Ecco alcuni esempi di pattern:

```
(Persona ?x)
(Età barbara ?x)
(Età ?x ?y)
```

Un *ambiente* (*environment*) è un insieme di *legami variabile:valore*; il valore di una variabile può essere una qualsiasi *costante* (simbolo, numero o stringa). In un ambiente, una variabile può avere *al più* un valore; in altre parole, in un ambiente una variabile può essere indefinita, ma se è definita ha un valore unico. Ecco alcuni esempi di ambienti:

```
{x:-, y:-}    {x:barbara, y:-}    {x:barbara, y:22}
```

Dati due ambienti  $E_1$  ed  $E_2$ , contenenti *le stesse variabili*, diremo che  $E_2$  *estende*  $E_1$ , e scriveremo

$$E_1 \subseteq E_2,$$

se e solo se ogni variabile che ha un valore definito in  $E_1$  ha lo stesso valore in  $E_2$  (una variabile può invece essere indefinita in  $E_1$  e avere un valore definito in  $E_2$ ). Ad esempio:

$$\{x:barbara, y:-\} \subseteq \{x:barbara, y:22\}.$$

Chiameremo poi *ambiente indefinito* un ambiente in cui tutte le variabili siano indefinite, come:

$$\{x:-, y:-\}.$$

Dati un pattern e un ambiente (che contenga tutte le variabili del pattern, eventualmente indefinite), possiamo operare la *sostituzione* delle variabili del pattern con i loro rispettivi valori nell'ambiente. Il risultato sarà un nuovo pattern, e in casi particolari una lista (quando non rimangono variabili indefinite). Ad esempio:

$$\begin{aligned} \text{sostituzione}(\text{(Età ?x ?y)}, \{x:-, y:-\}) &= \text{(Età ?x ?y)}, \\ \text{sostituzione}(\text{(Età ?x ?y)}, \{x:-, y:22\}) &= \text{(Età ?x 22)}, \\ \text{sostituzione}(\text{(Età ?x ?y)}, \{x:barbara, y:22\}) &= \text{(Età barbara 22)}. \end{aligned}$$

Il *pattern matching* è un'operazione eseguita su una *lista*, un *pattern* (contenente zero o più variabili) e un *ambiente*. Un'operazione di matching può *fallire* o *avere successo*; nel secondo caso viene restituita un'estensione dell'ambiente in ingresso (ovvero, possono essere aggiunti i valori di alcune variabili dell'ambiente precedentemente indefinite).

L'operazione di pattern matching è definita in questo modo: dati una lista  $L$ , un pattern  $P$  e un ambiente  $E$  (che contenga almeno le variabili che compaiono in  $P$ ), il pattern matching fra  $L$  e  $P$  in  $E$ :

- *fallisce* se non esiste nessuna estensione  $E'$  di  $E$  tale che  $L = \text{sostituzione}(P, E')$ ;

- *ha successo* in caso contrario, e allora restituisce la *minima* estensione  $E'$  di  $E$  tale che  $L = \text{sostituzione}(P, E')$ .

Ad esempio:

- $\text{match}(\text{(Età barbara 22)}, \text{(Età ?x ?y)}, \{x:-, y:-\})$  ha successo e restituisce l'ambiente  $\{x:\text{barbara}, y:22\}$ ;
- $\text{match}(\text{(Età barbara 22)}, \text{(Età ?x ?y)}, \{x:-, y:22\})$  ha successo e restituisce l'ambiente  $\{x:\text{barbara}, y:22\}$ ;
- $\text{match}(\text{(Età barbara 22)}, \text{(Età ?x 22)}, \{x:-, y:22\})$  ha successo e restituisce l'ambiente  $\{x:\text{barbara}, y:22\}$ ;
- $\text{match}(\text{(Età barbara 22)}, \text{(Età ?x 22)}, \{x:-, y:-\})$  ha successo e restituisce l'ambiente  $\{x:\text{barbara}, y:-\}$ ;
- $\text{match}(\text{(Età barbara 22)}, \text{(Età ?x ?y)}, \{x:-, y:55\})$  fallisce.

Nel seguito, quando il matching fra  $L$  e  $P$  ha successo dirò che  $L$  *si accoppia con*  $P$ .

#### 4 La base delle regole

La base delle regole è una collezione non ordinata di regole. Ogni regola ha la forma:

```
(defrule nome [commento]
  condizione1
  ...
  condizionen
=>
  azione1
  ...
  azionem)
```

Il *nome* di una regola è un simbolo arbitrario e il *commento*, opzionale, è una stringa arbitraria. Nel caso più semplice, le *condizioni* sono costituite da pattern. Infine, ogni *azione* è un'espressione funzionale composta da una chiamata di funzione applicata a zero o più argomenti, i quali a loro volta possono essere costanti, variabili, liste, pattern o altre espressioni funzionali; ad esempio:

```
(assert (P ?x ?y))
```

L'idea generale è che le condizioni dicono quando la regola va *attivata*, ovvero quando una sua attivazione va inserita nell'agenda. Quando poi l'attivazione di una regola viene prelevata dall'agenda ed eseguita, vengono valutate in sequenza le espressioni che rappresentano le azioni della regola. Tutto ciò risulterà chiaro dopo che avremo descritto il funzionamento dell'interprete.

A ogni regola è associata una struttura composta da due componenti: uno *schema d'attivazione iniziale* e un *ambiente iniziale*. Lo schema d'attivazione iniziale, che rappresenteremo come

*nome*: -, ... , -

è formato dal nome della regola seguito da un campo vuoto per ogni condizione. L'ambiente iniziale è l'ambiente indefinito contenente tutte e sole le variabili che compaiono nelle condizioni della regola. Si noti che tutte le variabili contenute nelle azioni devono comunque appartenere all'ambiente iniziale.

**L'interprete**

L'interprete funziona eseguendo un ciclo, detto *ciclo di attivazione ed esecuzione*. Il funzionamento dell'interprete può essere descritto come segue:

```

esecuzione ← true;
aggiorna(agenda);
while (esecuzione and not empty(agenda)) do
  esegui(top(agenda));
  pop(agenda);
  aggiorna(agenda);
end

```

L'interprete può essere visto come una vera e propria *macchina virtuale* che "esegue" una base di regole. Il ciclo dell'interprete inizia quando viene valutata l'espressione (run).

All'inizio di una sessione di lavoro l'agenda è vuota. L'aggiornamento dell'agenda si basa sulla creazione di nuove attivazioni, sull'inserimento di nuove attivazioni nell'agenda e sulla rimozione di attivazioni precedentemente inserite. Il processo è piuttosto complicato, e sarà quindi descritto con un esempio.

*Esempio 5.1: chiusura transitiva di una relazione binaria*

Questo esempio è l'implementazione in CLIPS del sistema per il calcolo della chiusura transitiva di una relazione binaria. La chiusura transitiva  $R^*$  di una relazione  $R$  è definita dagli assiomi:

- R1.  $R(x,y) \rightarrow R^*(x,y)$ ,  
 R2.  $R^*(x,y) \wedge R(y,z) \rightarrow R^*(x,z)$ ,

rappresentabili in CLIPS nel modo seguente:

```

(defrule R1
  (R ?x ?y)
=>
  (assert (R* ?x ?y))
)
(defrule R2
  (R* ?x ?y)
  (R ?y ?z)
=>
  (assert (R* ?x ?z))
)

```

Gli schemi d'attivazione iniziali e gli ambienti iniziali delle due regole sono rispettivamente:

- R1:- {x:-, y:-}  
 R2:-, - {x:-, y:-, z:-}

Consideriamo ora la relazione binaria descritta nella figura 5.1 e iniziamo a popolare la base dei fatti.

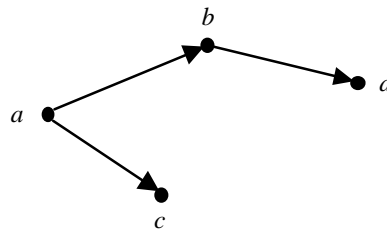


Figura 5.1 Una relazione binaria.

Valutando l'espressione (`assert (R a b)`) causiamo l'inserimento dell'elemento:

0: (R a b)

Questo evento provoca i seguenti effetti:

- L'elemento 0 si accoppia con la prima condizione di R1 a partire dall'ambiente iniziale di R1; ciò provoca la creazione di un'attivazione di R1, che descriveremo come R1:0 (ciò significa che nella regola R1 l'elemento 0 è stato accoppiato con la prima condizione). A questa attivazione è associato l'ambiente  $\{x:a, y:b\}$ . L'attivazione così creata viene inserita nell'agenda (mantenendo l'associazione con l'ambiente); poiché l'agenda è inizialmente vuota, questa attivazione è al momento l'unica contenuta nell'agenda.
- L'elemento 0 si accoppia anche con la seconda condizione di R2 a partire dall'ambiente iniziale di R2; ciò provoca la generazione di un nuovo schema d'attivazione di R2, che descriveremo come R2:-,0 (ciò significa che nella regola R2 l'elemento 0 è stato accoppiato con la seconda condizione). A questo schema è associato l'ambiente  $\{x:-, y:a, z:b\}$ . Poiché non si ha un'attivazione completa, nient'altro viene inserito nell'agenda.

La generazione dei nuovi schemi d'attivazione e dei relativi ambienti si può descrivere come segue:

R1:-  $\{x:-, y:-\}$   
**R1:0**  $\{x:a, y:b\}$   $\longrightarrow$  nell'agenda  
 R2:-, -  $\{x:-, y:-, z:-\}$   
 R2:-, 0  $\{x:-, y:a, z:b\}$

Lo stato dell'agenda è il seguente:

R1:0

Valutiamo ora l'espressione (`assert (R a c)`), che causa l'inserimento nella base dei fatti dell'elemento:

1: (R a c)

Questo evento provoca effetti analoghi al caso precedente. Ragionando allo stesso modo concludiamo che gli schemi d'attivazione sono ora:

R1:-  $\{x:-, y:-\}$   
 R1:0  $\{x:a, x:b\}$   
**R1:1**  $\{x:a, y:c\}$   $\longrightarrow$  nell'agenda  
 R2:-, -  $\{x:-, y:-, z:-\}$   
 R2:-, 0  $\{x:-, y:a, z:b\}$   
 R2:-, 1  $\{x:-, y:a, z:c\}$

Una nuova attivazione viene inserita nell'agenda. Poiché l'agenda non è più vuota, occorre stabilire una *strategia d'inserimento* delle attivazioni nell'agenda. Per il momento adotteremo una strategia LIFO, per cui la nuova attivazione viene posta nell'agenda *al di sopra* della precedente (questa strategia, denominata *depth*, è la strategia di default di CLIPS). Lo stato dell'agenda è ora:

R1:1  
R1:0

Valutiamo ora l'espressione (`assert (R b d)`), che causa l'inserimento nella base dei fatti dell'elemento:

2: (R b d)

Ragionando in modo analogo ai casi precedenti abbiamo che gli schemi d'attivazione sono ora:

R1:- {x:-,y:-}  
 R1:0 {x:a,x:b}  
 R1:1 {x:a,y:c}  
**R1:2 {x:b,y:d} → nell'agenda**  
 R2:-,- {x:-,y:-,z-}  
 R2:-,0 {x:-,y:a,z:b}  
 R2:-,1 {x:-,y:a,z:c}  
 R2:-,2 {x:-,y:b,z:d}

Una nuova attivazione viene inserita nell'agenda:

R1:2  
R1:1  
R1:0

A questo punto possiamo avviare il ciclo dell'interprete valutando l'espressione (`run 1`), che ci permette di eseguire un passo alla volta. Per prima cosa viene estratta dall'agenda ed eseguita l'attivazione R1:2, cui corrisponde l'ambiente {x:b,y:d}. Pertanto, vengono eseguite le azioni della regola R1 nell'ambiente indicato. Ciò provoca l'inserimento nella base dei fatti dell'elemento:

3: (R\* b d)

Si tratta di un nuovo evento, che ha la conseguenza di creare un certo numero di schemi d'attivazione (che per brevità non riporto) ma nessuna attivazione completa. L'agenda è ora:

R1:1  
R1:0

Valutando di nuovo l'espressione (`run 1`), viene estratta dall'agenda ed eseguita l'attivazione R1:1, cui corrisponde l'ambiente {x:a,y:c}. Vengono eseguite le azioni della regola R1 nell'ambiente indicato, il che provoca l'inserimento nella base dei fatti dell'elemento:

4: (R\* a c)

Ancora una volta, ciò ha la conseguenza di creare un certo numero di schemi d'attivazione ma nessuna attivazione completa. L'agenda è ora:

R1:0

Valutando di nuovo l'espressione (`run 1`), viene estratta dall'agenda ed eseguita l'attivazione R1:0, cui corrisponde l'ambiente {x:a,y:b}. Vengono eseguite le azioni della regola R1 nell'ambiente indicato, il che provoca l'inserimento nella base dei fatti dell'elemento:

5: (R\* a b)

Questa volta viene creata una nuova attivazione completa a partire dallo schema preesistente:

R2:-,2 {x:-,y:b,z:d}

L'attivazione creata e inserita nell'agenda è:

R2:5,2 {x:a,y:b,z:d}

L'agenda è ora:

R2:5,2

Valutando di nuovo l'espressione (`run 1`), viene estratta dall'agenda ed eseguita l'attivazione R2:5,2, cui corrisponde l'ambiente {x:a,y:b,z:d}. Vengono eseguite le azioni della regola R2 nell'ambiente indicato, il che provoca l'inserimento nella base dei fatti dell'elemento:

6: (R\* a d)

Ciò ha la conseguenza di creare un certo numero di schemi d'attivazione ma nessuna attivazione completa. Quindi l'agenda è ora vuota, e l'esecuzione termina lasciando la seguente base dei fatti:

0: (R a b)

1: (R a c)

2: (R b d)

3: (R\* b d)

4: (R\* a c)

5: (R\* a b)

6: (R\* a d)

### *Considerazioni sull'esempio*

Come si vede dall'esempio precedente, le attività principali dell'interprete consistono nel creare le attivazioni, nell'inserire nell'agenda le attivazioni create e nell'eseguire le attivazioni estratte dall'agenda.

La creazione delle attivazioni è un po' laboriosa, ma il già menzionato algoritmo RETE di Forgy consente di eseguire le operazioni necessarie in modo abbastanza efficiente. Come si sarà notato, viene creata un'attivazione del tipo  $R:indice_1, \dots, indice_n$  non appena la base dei fatti contiene  $n$  elementi (di indici  $indice_1, \dots, indice_n$ ) che si accoppiano con le  $n$  condizioni della regola  $R$  dando luogo a un unico ambiente di legami.

Quando un'attivazione è stata creata, viene inserita nell'agenda utilizzando un'opportuna strategia; di questo ci occuperemo nel prossimo sottoparagrafo.

Infine, l'attivazione in testa all'agenda viene estratta ed eseguita. Eseguire un'attivazione significa valutare ordinatamente tutte le espressioni funzionali che costituiscono le azioni della regola. Come sempre, la valutazione delle espressioni viene fatta in uno specifico ambiente, ovvero associando specifici valori alle variabili. L'ambiente in cui vengono condotte le valutazioni è quello associato all'attivazione che viene eseguita.

L'inserimento dei fatti iniziali può essere resa più agevole dall'uso della funzione `deffacts`. Ad esempio, possiamo raggruppare i fatti che definiscono la relazione in un unico "pacchetto":

```
(deffacts BinRel1 "A binary relation"
  (R a b)
  (R a c)
  (R b d)
)
```

Valutando nella finestra di dialogo l'espressione (`reset`), la base dei fatti viene così inizializzata:

```
0: (initial-fact)
1: (R a b)
2: (R a c)
3: (R b d)
```

L'uso del fatto iniziale (`initial-fact`), inserito automaticamente dal sistema, verrà spiegato in seguito.

Notiamo inoltre che le regole R1 ed R2 non provocano un ciclo senza fine se la relazione contiene dei cicli. Consideriamo ad esempio la relazione binaria descritta nella figura 5.2.

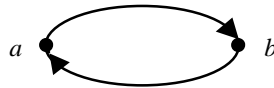


Figura 5.2 Una relazione binaria ciclica.

Ecco una traccia dell'esecuzione, in cui si riportano gli elementi della base dei fatti e il contenuto dell'agenda:

```
0: (R a b)
1: (R b a)
  R1:1
  R1:0
2: (R* b a)
  R2:2,0
  R1:0
3: (R* b b)
  R2:3,1
  R1:0
```

A questo punto, l'esecuzione di `R2:3,1` provoca la valutazione di (`assert (R* ?x ?z)`), con `x:b` e `z:a`. Ciò non ha alcun effetto sulla base dei fatti, dato che l'elemento (`R* b a`) è già presente. Si prosegue:

```
  R1:0
4: (R* a b)
  R2:4,1
5: (R* a a)
  R2:5,0
```

Ancora una volta, l'esecuzione di `R2:5,0` non ha alcun effetto sulla base dei fatti, dato che l'elemento (`R* a b`) è già presente.

L'agenda è ora vuota, e l'esecuzione termina lasciando la seguente base dei fatti:

- 0: (R a b)
- 1: (R b a)
- 2: (R\* b a)
- 3: (R\* b b)
- 4: (R\* a b)
- 5: (R\* a a)

### La gestione dell'agenda

L'inserimento delle nuove attivazioni nell'agenda avviene sulla base di una specifica strategia. In CLIPS, la strategia di default è la strategia *depth*, che segue il criterio LIFO: ogni nuova attivazione viene inserita in testa all'agenda. L'agenda è quindi gestita come una pila.

Il sistema CLIPS consente all'utente di cambiare strategia di inserimento e mette a disposizione un certo numero di strategie predefinite. Ad esempio, la strategia *breadth* segue il criterio FIFO: ogni nuova attivazione viene inserita in fondo all'agenda, che risulta quindi gestita come una coda. Altre strategie verranno presentate in seguito.

Per modificare la strategia è disponibile la funzione *set-strategy*. Ad esempio, valutare l'espressione (*set-strategy breadth*) ha l'effetto seguente:

- le attivazioni attualmente presenti nell'agenda vengono riordinate secondo la strategia *breadth*;
- le attivazioni future verranno inserite secondo la strategia *breadth* (fino a che la strategia non venga ulteriormente modificata).

Che cosa succede nell'esempio della chiusura transitiva se si adotta dall'inizio la strategia *breadth*? La differenza non è molto significativa: vengono creati gli stessi elementi, ma in un ordine differente. In particolare, con la stessa base dei fatti di partenza la base dei fatti finale sarà:

- 0: (R a b)
- 1: (R a c)
- 2: (R b d)
- 3: (R\* a b)
- 4: (R\* a c)
- 5: (R\* b d)
- 6: (R\* a d)

In seguito vedremo casi in cui la scelta della strategia è invece critica.

## 5.2 Produzioni e logica del primo ordine

Non è difficile vedere che esiste uno stretto rapporto fra PS e logica del primo ordine. Per cominciare, gli elementi permettono la rappresentazione immediata di *fatti*: è sufficiente interpretare il simbolo iniziale dell'elemento come un predicato, e le costanti che seguono come i suoi argomenti. Inoltre, un assioma del tipo

$$\forall x_1 \dots \forall x_n (P_1(x_1, \dots, x_n) \wedge \dots \wedge P_m(x_1, \dots, x_n) \rightarrow Q(x_1, \dots, x_k)), \quad \text{con } k \leq n,$$

può essere tradotto in una regola del tipo:

```
(defrule R
  (P1 ?x1 ... ?xn)
  ...
  (Pm ?x1 ... ?xn)
=>
  (assert (Q ?x1 ... ?xk))
)
```

In generale, tuttavia, il “potere deduttivo” della regola è diverso da quello dell’enunciato. Vediamo dapprima in che cosa l’enunciato e la regola si equivalgono. Come abbiamo visto, se i due fatti

```
(R* a b)
(R b d)
```

sono presenti nella base dei fatti, la regola può “dedurre” (cioè inserire nella base dei fatti) il fatto

```
(R* a d)
```

Ciò rispecchia il fatto che

$$R^*(a,b), R(b,d), \forall x \forall y (R^*(x,y) \wedge R(y,z) \rightarrow R^*(x,z)) \vdash R^*(a,d).$$

Vanno però notate due cose. Primo, il linguaggio di CLIPS è più povero di un linguaggio del primo ordine. Ad esempio, una semplice derivazione come

$$R^*(a,b), \neg R^*(a,d), \forall x \forall y (R^*(x,y) \wedge R(y,z) \rightarrow R^*(x,z)) \vdash \neg R(b,d).$$

non ha rispondenza in CLIPS perché i fatti negativi come  $\neg R(b,d)$  non sono esprimibili. Secondo, le regole R1 ed R2 del paragrafo precedente si comportano come enunciati condizionali del primo ordine perché l’azione prevista è un `assert`, ma questa somiglianza potrebbe non sussistere se l’azione fosse di tipo diverso (questo fatto apparirà più chiaro in seguito).

Dalle precedenti considerazioni concludiamo che la corrispondenza fra logica del primo ordine e CLIPS è la seguente: si tratta di due sistemi essenzialmente diversi; tuttavia, un sottoinsieme del CLIPS può essere utilizzato per implementare un frammento significativo della logica del primo ordine. Questa relazione è rappresentata schematicamente nella figura 5.3

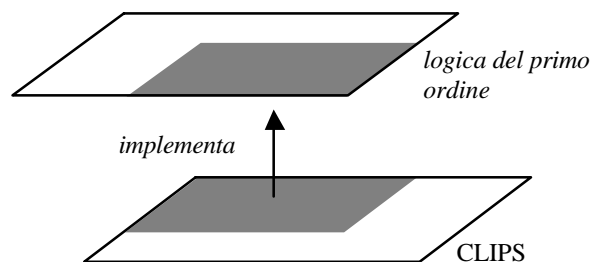


Figura 5.3 La relazione fra CLIPS e logica del primo ordine.

## 6 Interazioni fra regole

Come si è già visto, i PS sono sistemi basati su eventi: un evento può essere l'inserimento o l'eliminazione di un elemento dalla base dei fatti. Ora, programmare un sistema a eventi significa decidere quali azioni vanno eseguite e sotto quali condizioni. A questo scopo il programmatore CLIPS ha a disposizione, come vedremo, un certo numero di strumenti.

### 6.1 Conflitti fra regole

Nel capitolo precedente abbiamo visto un esempio in cui la gestione dell'agenda non era particolarmente critica, perché qualunque ordine di esecuzione delle attivazioni avrebbe portato a un risultato corretto. In generale non è così; anzi, la corretta gestione delle "priorità" fra le regole è essenziale per il buon funzionamento di un programma.

*Esempio 6.1: identificazione di cicli in un grafo orientato*

Consideriamo un esempio sui grafi: dato un grafo orientato arbitrario, vogliamo stabilire se contiene o non contiene un *ciclo* (ovvero, un cammino da un generico nodo a se stesso).

Il primo passo consiste sempre nel decidere come rappresentare i dati di partenza e i risultati. Per quanto riguarda la rappresentazione del grafo, ricordiamo che un grafo può essere definito come una struttura algebrica costituita da due insiemi (l'insieme  $N$  dei nodi e l'insieme  $A$  degli archi) e da due funzioni:

*inizio*:  $A \rightarrow N$ ,

*fine*:  $A \rightarrow N$ ,

che associano ad ogni arco rispettivamente il nodo iniziale e il nodo finale. Dobbiamo ora rappresentare tutto ciò tramite liste CLIPS, naturalmente sotto l'ipotesi che il grafo sia *finito* (il che significa che sia  $N$  sia  $A$  sono insiemi finiti).

Cominciamo con l'insieme dei nodi. Un insieme finito  $A = \{a_1, \dots, a_n\}$  può sempre essere rappresentato associando un predicato  $A(-)$  all'insieme, una costante  $a_k$  ad ogni elemento  $a_k$ , ed elencando gli  $n$  fatti  $A(a_1), \dots, A(a_n)$ , ovvero, in sintassi CLIPS,  $(A\ a_1), \dots, (A\ a_n)$ . Stabiliamo dunque che ogni nodo  $n$  del grafo è rappresentato nella forma  $(Nodo\ n)$ .

Per quanto riguarda gli archi, ogni arco può essere rappresentato nella forma  $(Arco\ a)$ . Restano da rappresentare le funzioni. Anche le funzioni, essendo finite, possono essere rappresentate da fatti. In generale, se  $f(x) = y$  scriviamo  $(f\ x\ y)$ . Quindi, le funzioni *inizio* e *fine* possono essere rappresentate da fatti della forma  $(inizio\ a\ n)$  e  $(fine\ a\ n)$ , dove  $a$  è un arco ed  $n$  è un nodo. In modo più compatto, è possibile rappresentare l'insieme degli archi e le due funzioni con una sola lista di fatti:

- ogni arco  $a$  del grafo è rappresentato nella forma  $(Arco\ a\ n1\ n2)$ , dove  $n1$  è l'inizio di  $a$  e  $n2$  la fine di  $a$ .

Per quanto riguarda la rappresentazione dei risultati, stabiliamo che:

- la presenza di un ciclo sarà indicata dall'elemento  $(Ciclo)$ ;
- l'assenza di cicli sarà indicata dall'elemento  $(Nessun\ ciclo)$ .

Dobbiamo ora decidere come risolvere il problema, e specificare eventuali dati intermedi necessari. Per trovare gli eventuali cicli, costruiamo tutti i possibili cammini del grafo. Poiché i cammini sono

successioni di archi, potremmo rappresentare ogni cammino di  $k$  archi con un elemento (Cammino  $a_1 a_2 \dots a_k$ ). Tuttavia, allo scopo di identificare i cicli è opportuno mettere in evidenza anche il nodo di inizio e il nodo di fine di ogni cammino. Se questi nodi sono rispettivamente  $n_1$  ed  $n_2$  rappresenteremo quindi il cammino con l'elemento (Cammino  $n_1 n_2 a_1 a_2 \dots a_k$ ).

Impostiamo la soluzione nel modo seguente. Il programma sarà costituito da tre *processi* distinti, ognuno costituito da una o più regole:

- P1: il processo P1 ha il compito di costruire tutti i possibili cammini nel grafo, in un ordine qualsiasi ma senza ridondanze inutili.
- P2: il processo P2 ha il compito di stabilire che esiste un ciclo non appena si verificano le condizioni che permettono di affermarlo e di interrompere immediatamente la computazione.
- P3: il processo P3 ha il compito di stabilire che non esistono cicli, non appena si verificano le condizioni che permettono di affermarlo. Si noti che il processo deve attendere la fine della costruzione di *tutti* i cammini prima di poter affermare che non esistono cicli.

È facile stabilire delle *priorità* fra i tre processi; infatti:

- il processo P2 ha priorità su P1, ovvero P2 deve essere in grado di interrompere P1 non appena è stato identificato un ciclo;
- il processo P1 ha però priorità su P3, perché non si può affermare che non esiste un ciclo finché non sono stati creati tutti i cammini.

Quindi:

$$P2 > P1 > P3.$$

In CLIPS è possibile imporre priorità fra processi assegnando un valore intero, detta *rilevanza* (*salience*), alle regole che ne fanno parte. Le attivazioni di regole con rilevanza più alta hanno precedenza sulle attivazioni di regole con rilevanza inferiore; quando non viene dichiarata esplicitamente, la rilevanza di una regola è uguale a zero per default. Per stabilire le priorità corrette possiamo ad esempio attribuire rilevanza 0 alle regole di P1, rilevanza 10 alle regole di P2 e rilevanza -10 alle regole di P3.

#### *Il processo P1*

Definiamo la prima regola del processo P1 (di rilevanza 0).

```
(defrule P1-1 "Crea un cammino di lunghezza uno"
  (Arco ?a ?n1 ?n2)
  =>
  (assert (Cammino ?n1 ?n2 ?a))
)
```

La seconda regola di P1 ha la funzione di concatenare un cammino preesistente con un arco per produrre un cammino più lungo. Dato che i cammini vengono rappresentati da liste con un numero arbitrario di costanti, dobbiamo sfruttare una nuova possibilità di CLIPS: le *multi-variabili*, indicate con il prefisso \$?, che si accoppiano con un numero arbitrario (maggiore o uguale a zero) di costanti.

```
(defrule P1-2 "Aggiungi un arco a un cammino"
  (Cammino ?n1 ?n2 $?c)
  (Arco ?a ?n2 ?n3)
  =>
  (assert (Cammino ?n1 ?n3 $?c ?a))
)
```

### Il processo P2

Il processo P2 consiste di un'unica regola (di rilevanza 10). Nella condizione utilizziamo la *multi-variabile anonima*, che si accoppia con qualunque sequenza di costanti (zero o più) senza modificare l'ambiente. Inoltre, fra le azioni compare l'azione di `halt`; la valutazione di `(halt)` ha l'effetto di assegnare il valore **false** al flag *esecuzione* nel ciclo dell'interprete (par. 4.1).

```
(defrule P2 "Trovato un ciclo"
  (declare (salience 10))
  (Cammino ?n ?n $?)
  =>
  (assert (Ciclo))
  (halt)
)
```

### Il processo P3

Infine, il processo P3 consta anch'esso di un'unica regola (di rilevanza -10). La regola non ha bisogno di condizioni, in quanto l'esecuzione della sua attivazione è determinata unicamente dalla priorità. Tuttavia, in CLIPS non è ammesso definire regole prive di condizioni; inseriamo quindi la *condizione fittizia* (`initial-fact`):

```
(defrule P3 "Non ci sono cicli"
  (declare (salience -10))
  (initial-fact)
  =>
  (assert (Nessun ciclo))
)
```

### Esecuzione

Per brevità consideriamo un grafo semplicissimo (fig. 6.1), definito da:

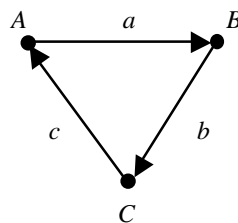
```
(deffacts G0
  (Nodo A)
  (Nodo B)
  (Nodo C)
  (Arco a A B)
  (Arco b B C)
  (Arco c C A)
)
```

La valutazione di `(reset)` ci dà la seguente base dei fatti iniziale:

```

0: (initial-fact)
1: (Nodo A)
2: (Nodo B)
3: (Nodo C)
4: (Arco a A B)
5: (Arco b B C)
6: (Arco c C A)

```

Figura 6.1 Il grafo  $G_0$ .

A questo punto l'agenda (con la testa, al solito, verso l'alto) si presenta come:

```

0    P1-1:4
0    P1-1:5
0    P1-1:6
-10  P3:0

```

dove nella prima colonna ho indicato la rilevanza della regola da cui deriva l'attivazione. L'inserimento di una nuova attivazione nell'agenda avviene nel modo seguente:

- un'attivazione viene inserita *al di sopra* di tutte le attivazioni con rilevanza inferiore, e al di sotto di tutte le attivazioni di rilevanza superiore;
- fra le attivazioni di pari rilevanza vengono applicati i criteri tipici della strategia selezionata (nel nostro caso, la `breadth`).

Di seguito viene descritta la computazione innescata dalla valutazione di `(run)`, alternando la descrizione dei nuovi fatti inseriti alla descrizione dell'agenda.

```

7: (Cammino A B a)
  0  P1-1:5
  0  P1-1:6
  0  P1-2:7,5
-10  P3:0

8: (Cammino B C b)
  0  P1-1:6
  0  P1-2:7,5
  0  P1-2:8,6
-10  P3:0

9: (Cammino C A c)
  0  P1-2:7,5
  0  P1-2:8,6
  0  P1-2:9,4
-10  P3:0

10: (Cammino A C a b)
  0  P1-2:8,6
  0  P1-2:9,4
  0  P1-2:10,6
-10  P3:0

11: (Cammino B A b c)
  0  P1-2:9,4
  0  P1-2:10,6
  0  P1-2:11,4
-10  P3:0

12: (Cammino C B c a)
  0  P1-2:10,6
  0  P1-2:11,4
  0  P1-2:12,5
-10  P3:0

13: (Cammino A A a b c)
 10  P2:13
  0  P1-2:11,4
  0  P1-2:12,5
  0  P1-2:13,4
-10  P3:0

14: (Ciclo)
  0  P1-2:11,4
  0  P1-2:12,5
  0  P1-2:13,4
-10  P3:0

```

A questo punto la computazione termina a causa dell'azione di (`halt`), anche se l'agenda non è vuota.

Vediamo che cosa succede invece se il grafo non ha un ciclo. Ad esempio (fig. 6.2):

```
0: (initial-fact)
1: (Nodo A)
2: (Nodo B)
3: (Nodo C)
4: (Arco a A B)
5: (Arco b B C)
6: (Arco c A C)
```

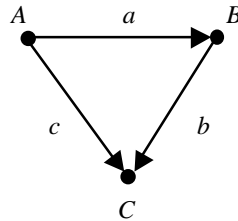


Figura 6.2 Il grafo  $G_1$ .

```

0    P1-1:4
0    P1-1:5
0    P1-1:6
-10  P3:0
7: (Cammino A B a)
0    P1-1:5
0    P1-1:6
0    P1-2:7,5
-10  P3:0
8: (Cammino B C b)
0    P1-1:6
0    P1-2:7,5
-10  P3:0
9: (Cammino A C c)
0    P1-2:7,5
-10  P3:0
10: (Cammino A C a b)
-10  P3:0
11: (NessunCiclo)
```

In questo caso vengono creati tutti i cammini, e solo allora viene eseguita la regola del processo P3. A questo punto la computazione termina perché l'agenda è vuota.

## 6.2 Confronto fra strategie

Mettiamo ora a confronto le due strategie *depth* e *breadth* (il che ci consentirà anche di capire perché si chiamino così).

Come abbiamo visto, ogni evento può provocare l'inserimento di un certo numero di nuove attivazioni nell'agenda. La strategia di gestione dell'agenda stabilisce l'ordine di inserimento e quindi determina

l'ordine di esecuzione delle attivazioni: le stesse attivazioni gestite in modo diverso danno luogo in generale a un processo di computazione differente. È possibile dare una rappresentazione globale dello "spazio" di tutte le computazioni possibili, che dipendono da tutti i possibili modi di ordinare l'agenda (comunque nel rispetto dei valori di rilevanza). Questa rappresentazione può essere data sotto forma di albero. Consideriamo ad esempio le regole P1-1, P1-2, P2 e P3 con i fatti iniziali corrispondenti al grafo della figura 6.3<sup>3</sup> (trascuriamo per semplicità i nodi del grafo, che non sono rilevanti per il problema di cui ci stiamo occupando):

- 0: (initial-fact)
- 1: (Arco a A B)
- 2: (Arco b B A)
- 3: (Arco c C B)

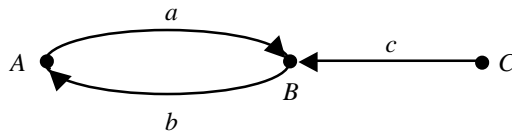
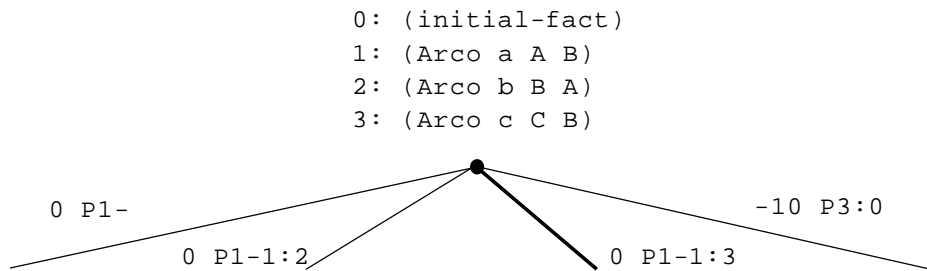
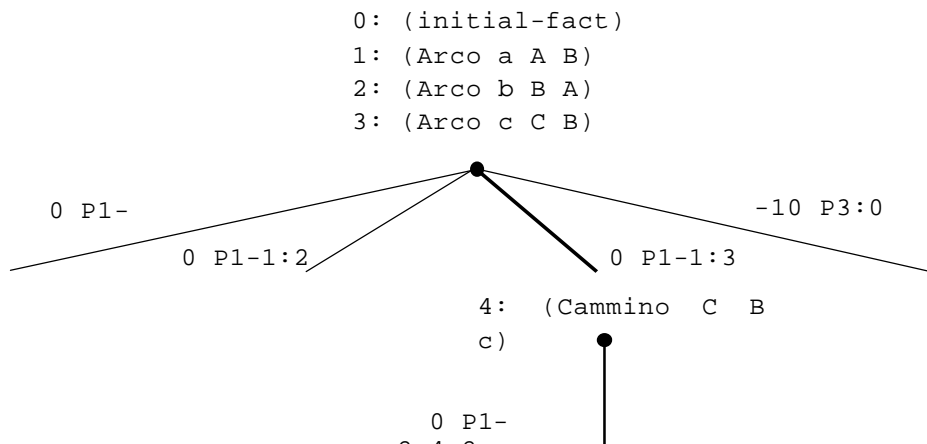


Figura 6.3 Il grafo  $G_2$ .

I fatti iniziali generano quattro attivazioni che, a seconda della strategia di inserimento adottata, potranno essere ordinate in modi diversi. Rappresentiamo questa situazione con un diagramma ad albero:

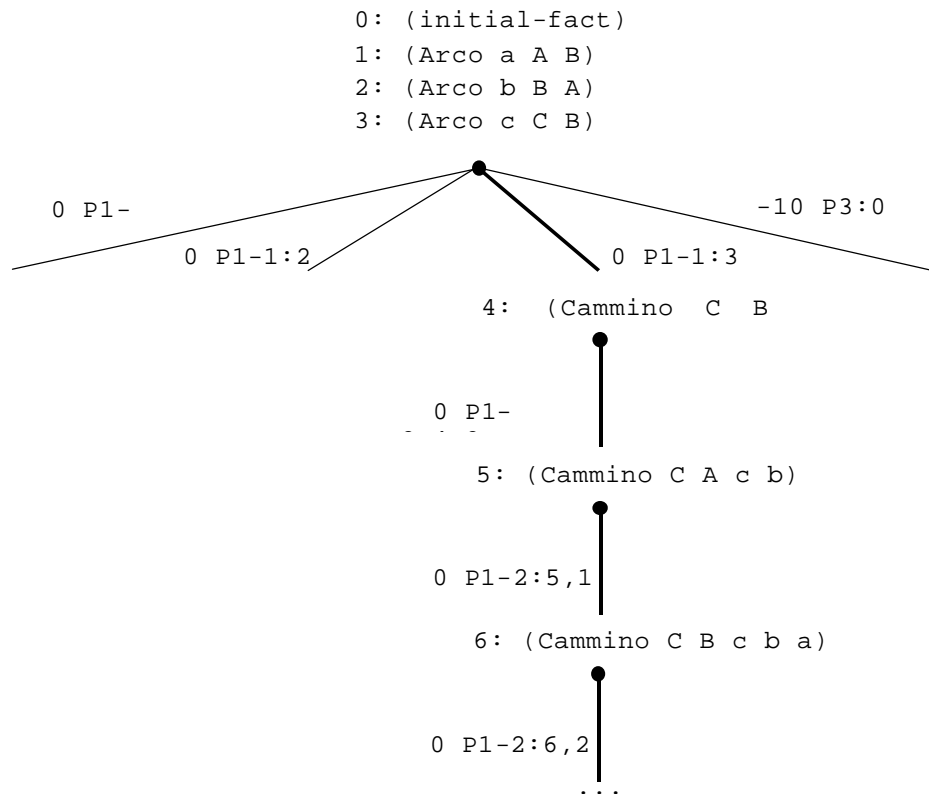


Assumiamo ora di adottare la strategia depth. Ciò significa che verrà eseguita l'attivazione corrispondente al ramo in neretto. La nuova situazione è rappresentata da:

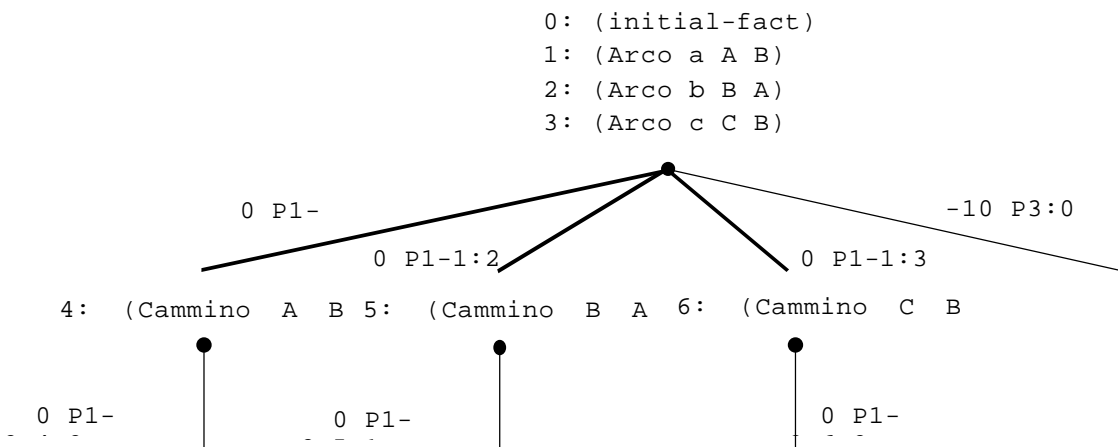


<sup>3</sup> Questo esempio mi è stato cortesemente segnalato da un allievo del corso ICSE 1998/99.

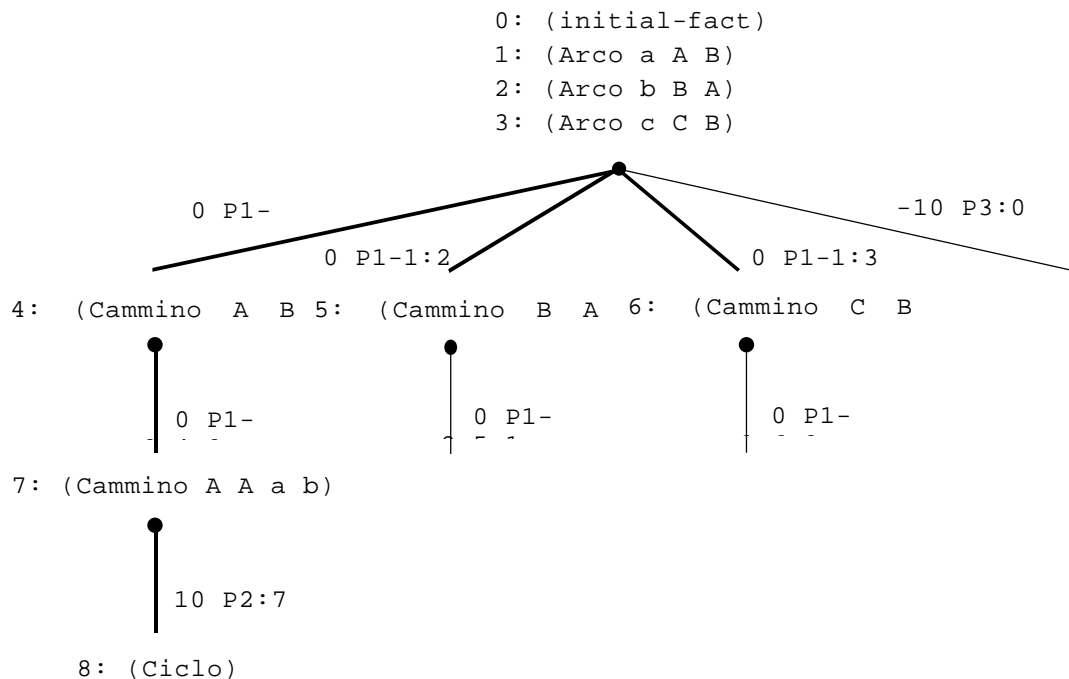
Continuando con la strategia depth:



Ci si convince facilmente che il sistema si è infilato in un ciclo senza fine: proseguendo, verranno costruiti i cammini (Cammino C A c b a b), (Cammino C B c b a b a), e così via, nessuno dei quali sarà identificato come un ciclo dalla regola P2. Questa è una conseguenza dell'adozione della strategia *depth*, che costruisce l'albero nella direzione della *profondità*. Per evitare di finire nel ciclo senza fine, occorre costruire l'albero per livelli, ovvero in *ampiezza*. Ciò può essere fatto adottando la strategia *breadth*, che porta a costruire l'albero seguente:



E proseguendo:



A questo punto, la costruzione dell'albero termina a causa dell'azione di (`halt`).

Concludendo:

- Le attivazioni delle regole possono essere eseguite seguendo diverse strategie, che possono essere viste come diversi modi di costruire un albero di scelte.
- La gestione dell'agenda di CLIPS secondo la strategia *depth*, che tratta l'agenda come una pila, corrisponde alla costruzione dell'albero in profondità; tale strategia può portare a cicli senza fine nel caso l'albero possieda cammini infiniti, anche se nell'albero esiste un cammino finito che porta alla terminazione dell'esecuzione.
- La gestione dell'agenda di CLIPS secondo la strategia *breadth*, che tratta l'agenda come una coda, corrisponde alla costruzione dell'albero in ampiezza; tale strategia non può portare a cicli senza fine nel caso l'albero possieda un cammino finito che porta alla terminazione dell'esecuzione. Molto spesso, però, la strategia *breadth* è inefficiente, perché porta a percorrere troppi rami dell'albero.

### 6.3 I contesti

Supponiamo ora che dopo l'identificazione di un ciclo la computazione non debba terminare, ma piuttosto proseguire con un ulteriore processo, che non ci importa precisare qui e che chiameremo P4. In questo caso non è possibile utilizzare l'istruzione di `halt`, perché blocca la computazione.

Possiamo risolvere il problema menzionato alla fine del paragrafo precedente introducendo una *condizione di contesto*, ad esempio (`CercaCiclo`), nelle regole di P1-P3:

```

(defrule P1-1 "Crea un cammino di lunghezza uno"
  (CercaCiclo)
  (Arco ?a ?n1 ?n2)
  =>
  (assert (Cammino ?n1 ?n2 ?a))
)
(defrule P1-2 "Aggiungi un arco a un cammino"
  (CercaCiclo)
  (Cammino ?n1 ?n2 $?c)
  (Arco ?a ?n2 ?n3)
  =>
  (assert (Cammino ?n2 ?n3 $?c ?a))
)
(defrule P2 "Trovato un ciclo"
  (declare (salience 10))
  ?f <- (CercaCiclo)
  (Cammino ?n ?n $?)
  =>
  (assert (Ciclo))
  (retract ?f)
)
(defrule P3 "Non ci sono cicli"
  (declare (salience -10))
  ?f <- (CercaCiclo)
  =>
  (assert (NessunCiclo))
  (retract ?f)
)

```

L'elemento (CercaCiclo) deve essere inizialmente inserito nella base dei fatti affinché il processo inizi. Tale elemento viene prima o poi rimosso da P2 o P3 (si noti che in P3 non è più necessaria la condizione (initial-fact)), e i tre processi P1-P3 vengono disattivati.

La rimozione di un fatto avviene nel modo seguente. Quando un elemento di indice  $n$  della base dei fatti si accoppia con la condizione

```
?f <- (CercaCiclo),
```

alla variabile viene assegnato come valore il *fact address* dell'elemento di indice  $n$  (il *fact address* di un elemento della base dei fatti è l'indirizzo in memoria dell'elemento, e non va confuso con il *fact index*, che è invece un intero senza segno; all'occorrenza, la funzione CLIPS *fact-index* consente di calcolare l'indice di un fatto a partire dal suo indirizzo). La valutazione di

```
(retract ?f)
```

elimina poi dalla base dei fatti l'elemento di indirizzo ?f (la funzione *retract* accetta in ingresso sia l'indirizzo sia l'indice del fatto da eliminare).

Aggiungendo o togliendo elementi di contesto come (CercaCiclo) è possibile attivare e disattivare a piacimento gruppi di regole: è sufficiente che le regole abbiano un'opportuna condizione di contesto.

## 7 La rappresentazione del cambiamento

### 7.1 Rappresentazione logica del cambiamento

Come si è visto, un sistema deduttivo è in grado di ricavare nuovi enunciati a partire da una KB iniziale. Gli enunciati dedotti sono veri in tutti i modelli della KB (o nel modello minimo se adottiamo le assunzioni di chiusura). Tutti gli enunciati in questione (sia quelli appartenenti alla KB sia quelli dedotti) sono comunque veri (in tutti i modelli o nel modello minimo) indipendentemente da qualsiasi qualificazione temporale. In altre parole, non è che un enunciato diventi vero solo quando viene dedotto; piuttosto, era vero anche prima di essere dedotto, anche se non lo sapevamo ancora.

In certe applicazioni di intelligenza artificiale (e in genere nelle applicazioni della logica alla matematica) ciò è sufficiente, perché non si ha a che fare con enunciati il cui valore di verità cambi nel tempo. In altre applicazioni, invece, lo scorrere del tempo può cambiare ciò che è vero e ciò che è falso. Ad esempio, se un robot afferra un oggetto e lo sposta, la posizione dell'oggetto cambia. Se l'oggetto  $a$  viene spostato dalla posizione 1 alla posizione 2, l'enunciato

$$Pos(a,1)$$

sarà vero prima dello spostamento e falso dopo, mentre l'enunciato

$$Pos(a,2)$$

sarà falso prima dello spostamento e vero dopo.

Anche se originariamente la logica simbolica non è stata sviluppata per trattare problemi di questo genere, è possibile rappresentare in modo rigoroso il cambiamento nella logica del primo ordine. A questo scopo è stato proposto, ad esempio, il *calcolo situazionale* (McCarthy e Hayes, 1969). L'idea è di aggiungere a ogni predicato un ulteriore argomento per fare riferimento alla specifica *situazione* in cui un fatto sussiste. Possiamo così dire che l'oggetto  $a$  ha posizione 1 nella situazione  $s_0$ :

$$Pos(a,1,s_0).$$

Utilizziamo poi un termine funzionale per rappresentare la situazione risultante dall'esecuzione di un'azione. Ad esempio, il termine

$$sposta(a,1,2,s_0)$$

rappresenta la situazione risultante dallo spostamento di  $a$  da 1 a 2 a partire dalla situazione  $s_0$ . Possiamo allora affermare che:

$$Pos(a,2,sposta(a,1,2,s_0)),$$

ovvero: l'oggetto  $a$  ha posizione 2 nella situazione risultante dallo spostamento di  $a$  da 1 a 2 a partire dalla situazione  $s_0$ . Più in generale, varrà l'assioma:

$$Pos(x,y,s) \Rightarrow Pos(x,z,sposta(x,y,z,s)).$$

Questo approccio presenta però un certo numero di problemi tecnici (fra cui il cosiddetto *frame problem*; vedi ad es. Russell e Norvig, 1995), per cui non è molto utilizzato nelle applicazioni.

Un approccio più empirico, ma che può essere sviluppato in modo coerente e rigoroso, consiste nel manipolare i fatti contenuti in una KB con i classici operatori `insert` e `delete` dei DBMS: sarà così possibile inserire nuovi enunciati fattuali nella KB quando diventano veri e cancellare enunciati fattuali dalla KB quando diventano falsi. Questo approccio richiede le assunzioni di chiusura, così che ogni enunciato fattuale assente dalla KB possa essere considerato falso. Si noti che nella terminologia tipica dei sistemi d'intelligenza artificiale l'operatore `insert` è denominato `assert`, e l'operatore `delete` è denominato `retract`.

Utilizzando gli operatori `assert` e `retract` è possibile definire nuovi operatori corrispondenti alle azioni eseguite sul dominio. Ad esempio, lo spostamento di un oggetto da una posizione a un'altra potrebbe essere definito come:

```
sposta(x,y,z) = if Pos(x,y) then
                retract(Pos(x,y));
                assert(Pos(x,z))
            end
```

Nel prossimo paragrafo mostro come un approccio di questo genere possa essere adottato in CLIPS.

## 7.2 Rappresentazione del cambiamento in CLIPS

Vogliamo rappresentare in CLIPS situazioni in cui alcuni cubetti possono essere impilati l'uno sull'altro sopra un ripiano. Vogliamo poi rappresentare gli effetti dello spostamento di un cubetto (figura 8.1)

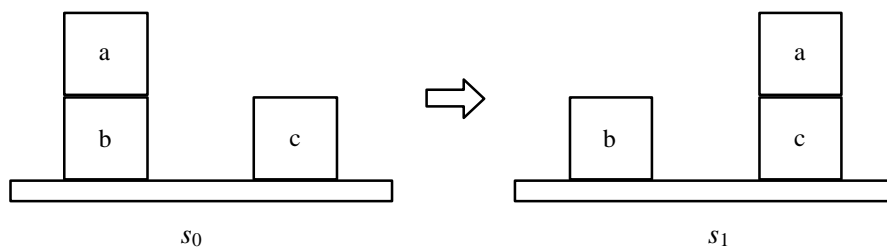


Figura 8.1 Lo spostamento di un cubetto.

Per prima cosa ci poniamo il problema di rappresentare una singola situazione. Sono possibili molte rappresentazioni diverse. Qui stabiliamo di utilizzare:

- una costante distinta per ogni cubetto;
- un predicato *Block(-)* per la proprietà di essere un cubetto;
- un predicato *On(-,-)* per la relazione binaria “stare sopra” fra due cubetti;
- un predicato *Clear(-)* per la proprietà di un cubetto di non avere niente sopra;
- un predicato *OnFloor(-)* per la proprietà di un cubetto di stare sul ripiano.

Stabiliamo inoltre di considerare i predicati *Block* e *On* come primitivi, e di derivare i predicati *Clear* e *OnFloor* tramite i due assiomi seguenti:

$$(A1) \quad Clear(x) \leftrightarrow Block(x) \wedge \neg \exists y On(y,x),$$

$$(A2) \quad OnFloor(x) \leftrightarrow Block(x) \wedge \neg \exists y On(x,y).$$

Oltre a questi due assiomi, possiamo introdurne altri per rappresentare alcune assunzioni restrittive. Ad esempio, assumiamo che un cubetto non possa stare contemporaneamente su due cubetti distinti:

$$(A3) \quad On(x,y) \wedge y \neq z \rightarrow \neg On(x,z).$$

Altri assiomi di questo genere possono essere aggiunti all'occorrenza.

Passiamo ora al CLIPS. La situazione  $s_0$  della figura 8.1 viene allora rappresentata dal seguente nucleo:

```
(deffacts s0
  (Block a)
  (Block b)
  (Block c)
  (On a b)
)
```

Gli assiomi A1 e A2 danno luogo alle seguenti regole di espansione del nucleo:

```
(defrule Exp-Clear
  (Block ?x)
  (not (On ? ?x))
  =>
  (assert (Clear ?x))
)

(defrule Exp-OnFloor
  (Block ?x)
  (not (On ?x ?))
  =>
  (assert (Clear ?x))
)
```

Si noti che queste due regole traducono solo “metà” dei due assiomi originari, in quanto corrispondono a:

$$\text{Clear}(x) \leftarrow \text{Block}(x) \wedge \neg \exists y \text{On}(y,x),$$

$$\text{OnFloor}(x) \leftarrow \text{Block}(x) \wedge \neg \exists y \text{On}(x,y).$$

Vedremo in seguito come utilizzare le due “metà” scartate nonché l'assioma A3. Per ora, abbiamo la seguente esecuzione:

```
0: (initial-fact)
1: (Block a)
2: (Block b)
3: (Block c)
4: (On a b)

  Exp-OnFloor:3,
  Exp-Clear:3,
  Exp-OnFloor:2
  Exp-Clear:1,
...
5: (OnFloor c)
6: (Clear c)
7: (OnFloor b)
8: (Clear a)
```

A questo punto, l'espansione del nucleo è completa e l'esecuzione si ferma.

Vogliamo ora definire un insieme di regole in grado di gestire il passaggio dalla situazione  $s_0$  alla situazione  $s_1$  della figura 8.1. L'idea è la seguente. Rappresentiamo il passaggio alla situazione  $s_1$  aggiungendo il nuovo elemento (On a c) alla base dei fatti. Otteniamo così una base dei fatti *incoerente* (o *inconsistente*), che non corrisponde a nessuna situazione fisicamente possibile.

Predisponiamo allora alcune regole che hanno il compito di ripristinare la coerenza della base dei fatti; ciò viene fatto *contraendo* la base dei fatti, ovvero rimuovendo tutti i fatti incompatibili con il nuovo elemento ( $On\ a\ c$ ). La nuova base così ottenuta viene poi espansa tramite le regole di espansione già viste.

La prima regola di contrazione si occupa dell'incoerenza causata dalla presenza contemporanea nella base dei fatti dei due elementi ( $On\ a\ b$ ) e ( $On\ a\ c$ ). Questi due fatti non possono coesistere sulla base dell'assioma A3. Uno dei due elementi andrà quindi rimosso; ma quale? Se facciamo l'ipotesi che ogni nuovo elemento aggiunto alla base dei fatti rispecchi fedelmente un cambiamento avvenuto nella situazione reale, è ovvio che va rimosso l'elemento "più vecchio", ovvero di indice minore. Ecco allora la regola di contrazione:

```
(defrule Contr-On
  ?f1 <- (On ?x ?)
  ?f2 <- (On ?x ?)
  (test (< (fact-index ?f1)(fact-index ?f2)))
=>
  (retract ?f1)
)
```

Questa regola va spiegata. In CLIPS una condizione

```
(test espressione-booleana)
```

viene gestita in modo diverso dai pattern visti finora. L'espressione booleana viene valutata; se il suo valore è *true* allora la condizione è soddisfatta, altrimenti non è soddisfatta. Come è ovvio, un'espressione booleana della forma

```
(< espressione1 espressione2)
```

vale *true* se il valore (numerico) della prima espressione è minore del valore (numerico) della seconda espressione, e vale *false* altrimenti. Infine, la funzione CLIPS *fact-index* ha lo scopo di trasformare l'*indirizzo* in memoria di un fatto (raccolto dall'assegnamento con  $<-$ ) nel suo *indice* numerico.

Come si vede, la regola *Contr-On* traduce un assioma con *conseguente negativo* (come A3) in un'azione di *retract*. Questo modo di procedere è giustificato dalle assunzioni di chiusura e sarà applicato ancora in seguito.

La regola *Contr-On* si occupa della *contrazione del nucleo*. Dobbiamo ora occuparci della contrazione dell'espansione del nucleo precedente. Infatti certi elementi dell'espansione precedente, come (*Clear c*), non sono più veri dopo l'asserzione del nuovo fatto ( $On\ a\ c$ ). Per procedere in questo senso, ricordiamo che le regole di espansione di *Clear* e *OnFloor* traducevano solo una "metà" degli assiomi A1 e A2. Concentriamoci allora sulle due "metà" trascurate:

$$Clear(x) \rightarrow Block(x) \wedge \neg \exists y On(y,x),$$

$$OnFloor(x) \rightarrow Block(x) \wedge \neg \exists y On(x,y).$$

Dato che sappiamo che l' $x$  di *Clear(x)* e di *OnFloor(x)* è necessariamente un blocco possiamo semplificare questi enunciati come segue:

$$Clear(x) \rightarrow \neg \exists y On(y,x),$$

$$OnFloor(x) \rightarrow \neg \exists y On(x,y).$$

Applicando semplici equivalenze logiche, questi assiomi si trasformano in:

$$\exists y On(y,x) \rightarrow \neg Clear(x),$$

$$\exists y On(x,y) \rightarrow \neg OnFloor(x).$$

Anche in questo caso traduciamo gli assiomi con conseguente negativo in azioni di `retract`. Quindi:

```
(defrule Contr-Clear
  ?f <- (Clear ?x)
        (On ? ?x)
  =>
        (retract ?f)
)
(defrule Contr-OnFloor
  ?f <- (OnFloor ?x)
        (On ?x ?)
  =>
        (retract ?f)
)
```

A questo punto il programma è completo: asserendo il nuovo elemento (`On a c`) si ha una completa ristrutturazione della base dei fatti, che diventa:

```
0: (initial-fact)
1: (Block a)
2: (Block b)
3: (Block c)
4: (On a b)
5: (OnFloor c)
6: (Clear c)
7: (OnFloor b)
8: (Clear a)
9: (On a c)
10: (Clear b)
```

## Riferimenti bibliografici (parte II)

Forgy, C. L. (1979). On the efficient implementation of production systems, Ph.D. dissertation, Carnegie-Mellon University, Pittsburgh, PA.

Newell, A., and H. A. Simon (1972). *Human problem solving*, Prentice Hall, Englewood Cliffs, NJ.

Post, F. L. (1943). Formal reductions of the general combinatorial decision problem. *American Journal of Mathematics*, 65, 197–215.