Politecnico di Milano
Facoltà di ingegneria Milano Leonardo
Corso di Ingegneria della conoscenza e sistemi esperti
Anno accademico 1998/99

Appunti dalle lezioni
Parte III: capitoli 9–11

Marco Colombetti

ATTENZIONE:
IL CAPITOLO 8 DELLE DISPENSE E' STATO SOPPRESSO

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 III)

9. Gli agenti razionali 3

- 9.1 Stati mentali e logica modale
- 9.2 Tipi di stati mentali
- 9.3 Spazi mentali
- 9.4 Azione!
- 9.5 I sotto-obiettivi

10. Architetture funzionali di agenti 11

- 10.1 L'architettura funzionale
- 10.2 Agenti reattivi
- 10.3 Agenti guidati da obiettivi
- 10.4 Agenti deliberativi

11. Altre tecniche di programmazione CLIPS 14

- 11.1 Dipendenze logiche e mantenimento della verità
- 11.2 Rappresentazione esplicita dell'informazione negativa
- 11.3 Ragionamento non monotono

Riferimenti bibliografici (parte III) 23

9 Gli agenti razionali

9.1 Stati mentali e logica modale

Nel capitolo precedente abbiamo visto come un sistema CLIPS possa aggiornare la base dei fatti a fronte di cambiamenti nella situazione di riferimento. Un sistema del genere può essere utilizzato, ad esempio, per mantenere la coerenza della rappresentazione di un ambiente, assumendo che le modifiche vengano operate da un agente esterno e comunicate al sistema. Più interessante è chiedersi come tutto ciò possa far parte di un agente autonomo, in grado di cambiare esso stesso l'ambiente o, in altre parole, di agire.

Perché un sistema decida di modificare l'ambiente occorre che vi sia una discrepanza, una "tensione", fra lo stato effettivo dell'ambiente e uno stato in qualche modo desiderato o considerato preferibile: non sarebbe razionale tentare di cambiare lo stato delle cose se questo va bene com'è. Questa considerazione ci dice qualcosa d'importante sull'architettura di un agente razionale. Non è sufficiente rappresentare una situazione con una KB: occorre invece mantenere la rappresentazione separata di almeno due situazioni, quella effettiva e quella desiderata. In realtà nessun agente, naturale o artificiale, può accedere in modo diretto e infallibile allo stato dell'ambiente. Quindi è più corretto dire che la discrepanza che porta all'azione si forma fra quello che l'agente *crede* e quello che l'agente *desidera* essere lo stato dell'ambiente.

Gli stati mentali sono caratterizzati da una *modalità* e da un *contenuto*. Credenza (*belief*) e desiderio (*desire*) sono due modalità possibili. Quanto al contenuto, si assume che esso sia un qualsiasi stato di cose. Quindi, ad esempio: la credenza che il cubetto a sia sul cubetto b è uno stato mentale la cui modalità è la credenza e il cui contenuto è "il cubetto a è sul cubetto b", e il desiderio che il cubetto a sia sul cubetto c è uno stato mentale la cui modalità è il desiderio e il cui contenuto è "il cubetto a è sul cubetto c". Stati di questo genere possono essere rappresentati da enunciati logici, pur di adottare un linguaggio logico *modale*. Un linguaggio di questo genere possiede, oltre ai simboli logici che già conosciamo, nuovi simboli logici, detti *operatori modali*, qui utilizzati per rappresentare le modalità degli stati mentali. Supponiamo che l'operatore modale BEL rappresenti la credenza e l'operatore modale DES rappresenti il desiderio. I due esempi precedenti si rappresentano allora con gli enunciati:

BEL $On(a,b)$,

DES $On(a,c)$.

I linguaggi modali sono molto espressivi; in particolare permettono di *iterare* gli operatori modali per esprimere stati mentali come "l'agente x desidera che l'agente y creda che ...", e così via. L'approccio agli agenti dotati di stati mentali può quindi essere sviluppato utilizzando gli strumenti della logica modale (vedi ad esempio Chellas, 1980; Hughes and Cresswell, 1996), ma ciò va oltre i limiti di questo corso. Nel seguito adotterò quindi un approccio più empirico.

9.2 Tipi di stati mentali

Come abbiamo già visto, perché un sistema agisca occorre che possieda stati mentali di almeno due tipi diversi:

- stati *epistemici* o *cognitivi*, che rappresentano l'ambiente come l'agente assume che sia; e
- stati *telici* o *volitivi*, che rappresentano l'ambiente come l'agente vorrebbe che fosse.

Chiaramente, le credenze sono stati epistemici e i desideri sono stati telici. Ci sono però altre modalità dei due diversi tipi. Ad esempio, le *percezioni* sono stati epistemici; le *intenzioni* sono stati telici, e così via.

Un agente, per essere tale, deve possedere almeno una modalità epistemica e una modalità telica: è fra i due tipi di stati, infatti, che si crea la "tensione" che porta all'azione. In generale, però, un agente complesso ha diverse modalità di stati mentali dei due tipi. Consideriamo un esempio: mi sento stanco e penso che sia l'ora di prendermi un po' di riposo; decido di partire per il Madagaskar domani mattina e quindi esco subito per andare all'agenzia di viaggi. In questo processo hanno avuto una funzione essenziale i seguenti stati mentali:

- la *percezione* di essere stanco
- il *desiderio* di riposarmi;
- la *credenza* che un lungo viaggio sia un buon modo di riposarsi;
- il *piano* di andare in Madagascar domani;
- la *credenza* che per andare in Madagaskar bisogna prima passare da un'agenzia di viaggi;
- l'*intenzione immediata* di andare all'agenzia di viaggi.

Inoltre, questi stati sono connessi fra loro da *legami causali.* In particolare:

- la percezione di essere stanco causa il desiderio di riposarmi;
- il desiderio di riposarmi, fra tutti i desideri che posso avere nello stesso momento, viene scelto come *obiettivo* da perseguire;
- l'obiettivo di riposarmi, unitamente alla credenza che un lungo viaggio sia un buon modo di riposarsi, causa il piano di andare in Madagaskar domani;
- il piano di andare in Madagaskar domani, unitamente alla credenza che per andare in Madagaskar bisogna prima passare da un'agenzia di viaggi, causa l'intenzione immediata di andare all'agenzia di viaggi.

È interessante notare che gli stati telici, unitamente a degli stati epistemici, possono causare nuovi stati telici. Inoltre, stati epistemici possono causare nuovi stati epistemici. Al contrario, in un agente razionale gli stati volitivi non dovrebbero mai causare stati epistemici. È pur vero che a volte si desidera tanto che le cose stiano in un certo modo da autoconvincersi che le cose stanno proprio così (gli inglesi parlano di *wishful thinking*); ma questo processo non è certo razionale.

Che cosa è, allora, un *agente razionale*? In generale, si considera che un agente sia razionale quando è in grado di *sfruttare le proprie credenze per agire in modo tale da soddisfare i propri desideri*. In pratica, la razionalità lavora in modo tale da ridurre la discrepanza fra stati epistemici e stati telici, tenendo conto il più fedelmente possibile della realtà (non vale cercare di ridurre una discrepanza autoconvincendosi che non c'è!).

9.3 Spazi mentali

Le indicazioni del paragrafo precedente sono molto generiche, ma ci consentono già di fare alcune ipotesi sull'architettura di un agente razionale. Articoliamo le rappresentazioni logiche di un agente in un certo numero di contenitori o *spazi*, uno per ogni modalità.

Per il momento assumeremo che l'agente possieda:

- una modalità epistemica, la credenza (BEL);
- due modalità teliche, gli obiettivi (GOAL) e le azioni intenzionali (DO).

Abbiamo così tre spazi: lo spazio delle credenze, lo spazio degli obiettivi e lo spazio della azioni.

All'interno di ogni spazio possiamo inserire rappresentazioni sotto forma di enunciati logici. Ad esempio (facendo riferimento alla figura 8.1):

spazio delle credenze	***spazio degli obiettivi***	***spazio delle azioni***
Block(*a*)	*On*(*a*,*c*)	
Block(*b*)		
Block(*c*)		
On(*a*,*b*)		

La rappresentazione di questi spazi in CLIPS è molto semplice. Per le credenze e gli obiettivi:

```
(BEL Block a)
(BEL Block b)
(BEL Block c)
(BEL On a b)
(GOAL On a c)
```

Le regole di espansione e contrazione della base dei fatti (cap. 8) vanno ora riformulate tenendo conto della nuova rappresentazione delle credenze. Inoltre, attribuiamo alle regole di espansione e contrazione una priorità abbastanza elevata, perché dovranno avere precedenza sulle regole per la modifica del nucleo che definiremo fra poco. Ad esempio:

```
(defrule Exp-BEL-Clear
  (declare (salience 10))
  (BEL Block ?x)
  (not (BEL On ? ?x))
 =>
  (BEL Clear ?x)
)
```

e così via.

Occorre ora pensare alle regole che determinano le interazioni fra gli obiettivi e le credenze. Mentre per le regole precedenti ci siamo basati su assiomi logici, in questo caso seguiamo un approccio più empirico. Ci poniamo il problema di come l'agente debba gestire le espressioni del tipo `(GOAL …)`. Cominciamo dal caso più banale: l'obiettivo ha lo stesso contenuto di una credenza. Ciò significa, ad esempio, che sono contemporaneamente presenti `(GOAL On a b)` e `(BEL On a b)`. Nei casi di questo genere, stabiliamo che l'agente si limiti a rimuovere l'obiettivo dallo spazio degli obiettivi, dato che è già realizzato:

```
(defrule Contr-GOAL
         (declare (salience 10))
   ?f <- (GOAL $?x)
         (BEL  $?x)
 =>
         (retract ?f)
)
```

Questa regola è adeguata, grazie all'uso della multivariabile `$?x`, a rimuovere gli obiettivi già realizzati di qualunque tipo.

Per quando riguarda le azioni, ne prevediamo di due tipi: lo spostamento di un blocco *x* su un blocco *y*,

```
(DO Move x y)
```

e lo spostamento sul ripiano di un blocco *x*:

```
(DO MoveToFloor x).
```

Anche in questo caso si tratta di stati mentali (la cui modalità è DO), la cui funzione specifica è di provocare l'esecuzione immediata dell'azione descritta. Le regole che portano all'azione sono introdotte nel prossimo paragrafo.

9.4 Azione!

Consideriamo un obiettivo che non è ancora realizzato, ma può essere realizzato con un'unica azione. A questo scopo, assumiamo che il nostro sistema sia interfacciato con un braccio manipolatore in grado di afferrare un blocco (purché sia libero sopra) e di metterlo sul ripiano o sopra un altro blocco (purché anche questo sia libero sopra). Ecco un esempio di obiettivo raggiungibile con una singola azione:

```
(defrule DO-move
  (GOAL On ?x ?y)
  (BEL  Clear ?x)
  (BEL  Clear ?y)
 =>
  (assert (DO Move ?x ?y))
)
```

Analizziamo la regola:

- La prima condizione descrive un obiettivo senza richiedere esplicitamente che esso non sia già realizzato; tale richiesta (che sarebbe comunque esprimibile con la condizione negativa `(not (BEL On ?x ?y)))` non è necessaria, perché sappiamo che gli obiettivi già realizzati vengono immediatamente rimossi da una regola di priorità più alta.
- La seconda e la terza condizione descrivono le condizioni per l'esecuzione dell'azione da parte del braccio manipolatore.
- L'`assert` inserisce nella base dei fatti uno stato mentale che, per definizione, porta all'esecuzione immediata dell'azione.

Ora ci occorre una regola per eseguire l'azione:

```
(defrule Exec-DO-Move
        (declare (saliece 10))
  ?f <- (DO Move ?x ?y)
 =>
        (assert (BEL On ?x ?y))
        (retract ?f)
)
```

Questa regola assume implicitamente che l'azione abbia successo, e quindi asserisce le conseguenze dell'azione (ovvero: BEL *On*(*x*,*y*)). Se il sistema fosse direttamente interfacciato con un braccio manipolatore reale, la regola dovrebbe inviare al programma di controllo del braccio le istruzioni necessarie per eseguire fisicamente l'azione.

L'elemento `(DO ...)` viene rimosso dalla base dei fatti, perché altrimenti la regola potrebbe nuovamente attivarsi in futuro. Qualora si desideri mantenere traccia dell'azione eseguita, si può modificare la regola asserendo anche un fatto del tipo:

```
(Action Move ?x ?y executed)
```

che memorizza l'azione eseguita. Quando viene eseguita un'attivazione di questa regola, viene modificato il nucleo della rappresentazione nello spazio delle credenze. Questo evento scatena l'attivazione delle varie regole di contrazione ed espansione, e gli due spazi mentali delle credenze e degli obiettivi) vengono quindi aggiornati in modo del tutto automatico. Esempio (abbeviato):

```
 0: (initial-fact)
 1  (BEL Block a)
 2: (BEL Block b)
 3: (BEL Block c)
 4: (BEL On a b)
 5: (GOAL On a c)
   10  Exp-BEL-OnFloor:4,
   10  Exp-BEL-Clear:4,
   10  Exp-BEL-OnFloor:3
   10  Exp-Clear:2,
...
 6: (BEL OnFloor c)
 7: (BEL Clear c)
 8: (BEL OnFloor b)
 9: (BEL Clear a)
    0  DO-Move:5,9,7
10: (DO Move a c)
   10  Exec-DO-Move:10
```

~~`10: (DO Move a c)`~~

```
11: (Action Move a c executed)
12: (BEL On a c)
   10  Contr-BEL-Clear:12,7
   10  Contr-GOAL:5,12
   10  Contr-BEL-On:12,4
```

~~`7: (BEL Clear c)`~~
~~`5: (GOAL On a c)`~~
~~`4: (BEL On a b)`~~

```
   10  Exp-BEL-Clear:3,
13: (BEL Clear b)
```

La base dei fatti risultante è quindi:

```
 0: (initial-fact)
 1  (BEL Block a)
 2: (BEL Block b)
 3: (BEL Block c)
 6: (BEL OnFloor c)
 8: (BEL OnFloor b)
 9: (BEL Clear a)
11: (Action Move a c executed)
```

```
12: (BEL On a c)
13: (BEL Clear b)
```

9.5 I sotto-obiettivi

Nel paragrafo precedente abbiamo visto come trattare un obiettivo raggiungibile in una mossa. In generale, però, le cose non sono così semplici. Che cosa succede ad esempio se l'agente (fig. 9.1) desidera che il cubetto *b* si trovi sul cubetto *c*?

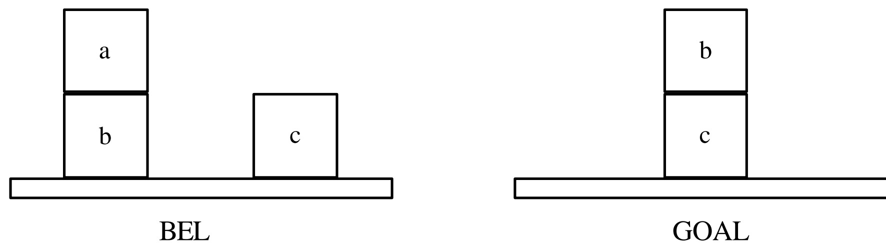

Figura 8.1 *Un compito un po' più impegnativo.*

In questo caso l'obiettivo non può essere realizzato in una sola mossa: il cubetto *b* non può essere spostato finché non sarà *Clear*. Si procede allora generando, a partire dall'obiettivo iniziale

GOAL *On*(*b*,*c*),

un nuovo obiettivo,

GOAL *Clear*(*b*).

Gli obiettivi generati in questo modo sono detti sotto-obiettivi. Gli obiettivi di tipo *On*(–,–) possono generare sotto-obiettivi di tipo *Clear*(–) sia per il primo che per il secondo argomento:

```
(defrule Sub-GOAL-Clear-1
   (GOAL On ?x ?)
   (not (BEL  Clear ?x)
 =>
   (assert (GOAL Clear ?x))
)
(defrule Sub-GOAL-Clear-2
   (GOAL On ? ?y)
   (not (BEL  Clear ?y))
 =>
   (assert (GOAL Clear ?y))
)
```

A questo punto, però, abbiamo introdotto obiettivi di tipo *Clear*(–) ma non abbiamo ancora le regole per trattarli. Anche in questo caso, ci chiediamo in che caso l'obiettivo è raggiungibile in una mossa. Nel caso della figura 9.1, è sufficiente mettere *a* sul ripiano.

Più in generale, e ricordando che per mettere un blocco *x* sul ripiano dobbiamo semplicemente rimuovere il fatto *On*(*x*,–):

```
(defrule DO-MoveToFloor
   (GOAL Clear ?x)
   (BEL  On ?y ?x)
   (BEL  Clear ?y)
 =>
   (assert (DO MoveToFloor ?y))
)
(defrule Exec-DO-MoveToFloor
  ?f1 <- (DO MoveToFloor ?y)
  ?f2 <- (BEL  On ?y ?x)
 =>
         (assert (Action MoveToFloor ?y executed))
         (retract ?f1 ?f2)
)
```

Si noti che i due elementi `?f1` ed `?f2` vengono cancellati per motivi diversi: il primo perché l'azione è stata ormai eseguita, il secondo perché questo è il modo di rappresentare l'effetto dell'azione.

Anche in questo caso le cose potrebbero non essere così semplici; ad esempio, potremmo trovarci nella situazione descritta nella figura 9.2.

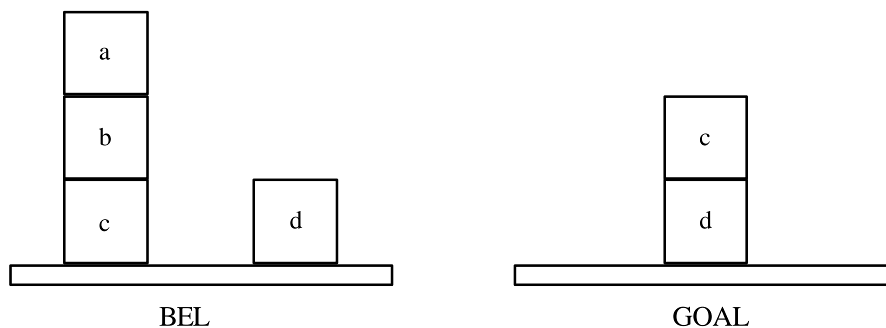

Figura 8.1 *Un compito un po' più impegnativo.*

Nel caso in cui il blocco da spostare sul ripiano non sia esso stesso *Clear*, dobbiamo porci questo fatto come sotto-obiettivo:

```
(defrule Sub-GOAL-Clear-3
   (GOAL Clear ?x)
   (BEL On ?y ?x)
   (not (BEL Clear ?y))
 =>
   (assert (GOAL Clear ?y))
)
```

A questo punto abbiamo finito, perché abbiamo considerato tutti i casi possibili sia per gli obiettivi di tipo *On* che per gli obiettivi di tipo *Clear*.

Come esempio, si consiglia il lettore di eseguire il programma sull'esempio seguente:

```
0: (initial-fact)
1  (BEL Block a)
2: (BEL Block b)
3: (BEL Block c)
4: (BEL Block d)
5: (BEL On a b)
6: (BEL On b c)
5: (GOAL On c d)
```

10 Architetture funzionali di agenti

10.1 L'architettura funzionale

L'*architettura funzionale* di un agente è definita dagli spazi mentali di cui l'agente dispone e dai processi che prendono i loro ingressi da tali spazi e restituiscono i loro valori in tali spazi. Nel seguito descriverò un'architettura funzionale con un multigrafo[1] (etichettato sia ai nodi che agli archi), in cui i nodi rappresentano spazi mentali e i multiarchi rappresentano processi.

10.2 Agenti S-R

Il più semplice tipo di agente ha esattamente uno spazio mentale di tipo epistemico (*percezione*, PERC) e uno di tipo telico (*azione*, DO). Come esempio si può pensare a un robot dotato di certi sensori e attuatori, il cui comportamento è costituito soltanto da reazioni immediate alle percezioni.

In un agente di questo tipo (fig. 10.1), gli stati percettivi sono generati da un processo di *percezione*, che riceve l'input dai sensori diretti verso l'ambiente. Un processo di *reazione* genera azioni sulla base delle percezioni. Le azioni a loro volta controllano gli attuatori attraverso un processo di *esecuzione*.

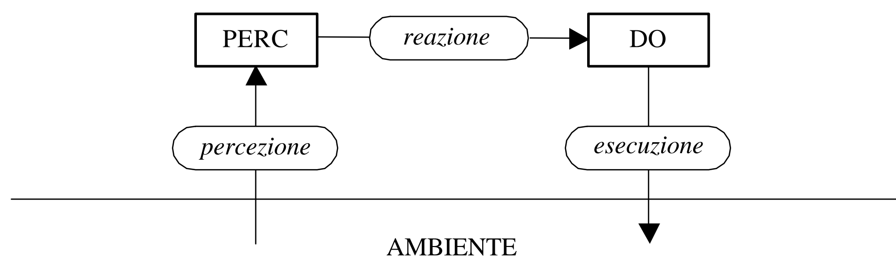

Figura 10.1 *L'architettuira funzionale di un agente SR.*

10.3 Agenti guidati da obiettivi

Un agente guidato da obiettivi ha, oltre agli spazi delle credenze e delle azioni, anche uno spazio degli *obiettivi* (GOAL). Un esempio è costituito dall'agente CLIPS definito nel capitolo 9. La sua architettura tipica è descritta nella figura 10.2, che mette in luce anche il processo di *valutazione*, con cui le percezioni vengono accettate o rifiutate dall'agente (le percezioni accettate diventano credenze). Si noti che l'architettura di un agente SR è comunque contenuta come parte di un agente guidato da obiettivi.

Un agente guidato da obiettivi non è puramente reattivo, in quanto è in grado di "ragionare" sugli obiettivi non immediatamente raggiungibili e di agire allo scopo di renderli tali. Tuttavia, un agente di questo tipo ha un'evidente limitazione: quando un obiettivo è raggiungibile con una sola azione, questa

[1] Un *multigrafo* è una struttura analoga a un grafo, con la differenza che i suoi archi (detti *multiarchi*) hanno una sequenza ordinata di nodi come sorgente (anziché un solo nodo).

viene eseguita immediatamente. Non possono quindi essere costruiti piani d'azione da eseguire in futuro.

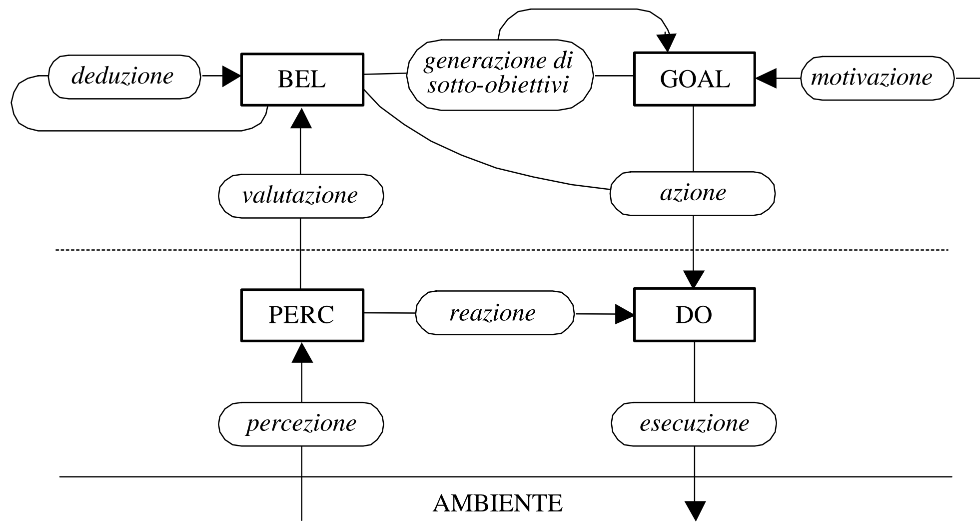

Figura 10.2 *L'architettura funzionale di un agente guidato da obiettivi.*

10.4 Agenti deliberativi

Nella figura 10.3 è descritta un'architettura più complessa, che consente all'agente:

- di memorizzare percezioni interessanti per un eventuale uso futuro (MEM);
- di avere un serbatoio di *desideri* (DES), eventualmente contrastanti;
- di promuovere auno o più desideri a obiettivi da perseguire effettuvamente, in base a un processo di *scelta*;
- di costruire piani d'azione da eseguire in futuro, quando se ne presenterà l'occasione.

Definire in CLIPS un agente di questo tipo è un'operazione non banale, e quindi non approfondiremo qui l'argomento. Nel prossimo capitolo indicherò tuttavia alcune tecniche di programmazione che potrebbero essere utilizzate anche per la realizzazione di un agente deliberativo.

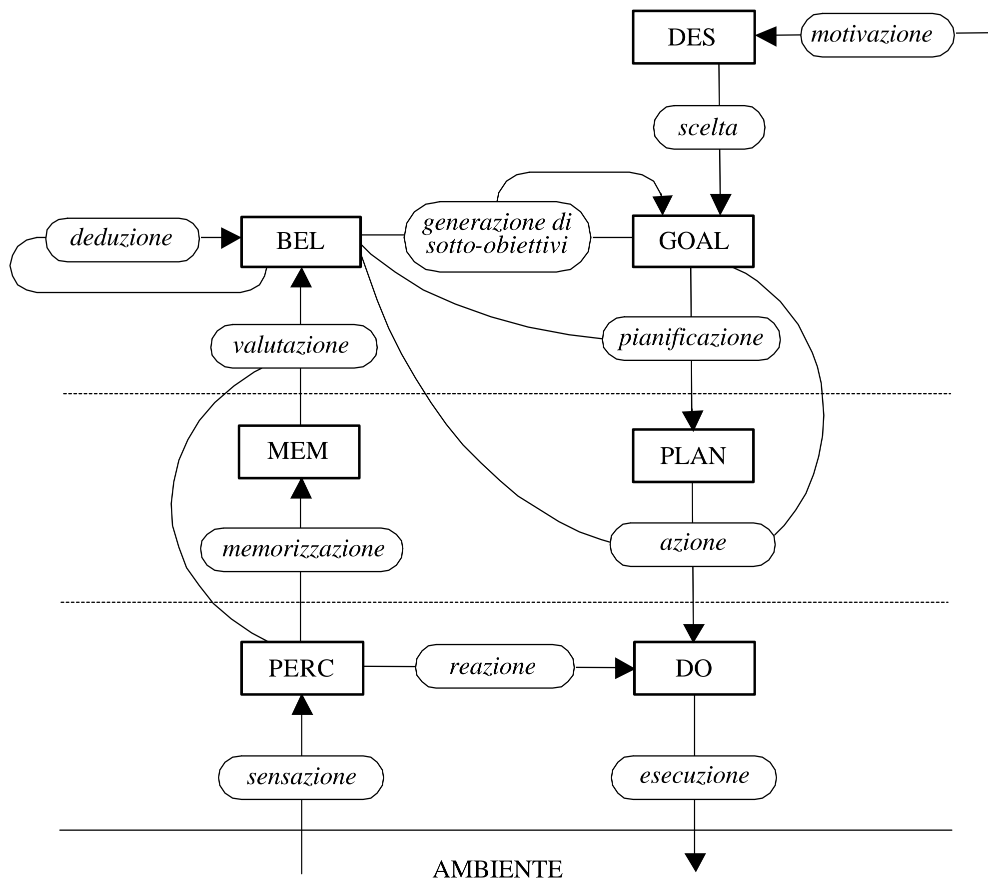

Figura 10.3 *L'architettura funzionale di un agente deliberativo.*

11 Altre tecniche di programmazione CLIPS

11.1 Dipendenze logiche e mantenimento della verità

In molte applicazioni è necessario tenere traccia del processo che porta a dedurre fatti da altri fatti. In questo modo, ad esempio, è possibile rimuovere automaticamente dalla base dei fatti tutte i fatti derivati precedentemente da una premessa, qualora questa venga rimossa. Consideriamo i fatti:

(F1) $P(a)$,

(F2) $Q(a)$,

(F3) $R(a)$,

nonché gli enunciati:

(A1) $P(x) \wedge Q(x) \Rightarrow S(x)$,

(A2) $P(x) \wedge R(x) \Rightarrow S(x)$,

(A3) $S(x) \Rightarrow T(x)$.

Gli enunciati A1–A3 possono essere rappresentati graficamente sotto forma di *albero*[2] *AND-OR* come nella figura 11.1, dove:

- i cerchi contenenti formule atomiche rappresentano *nodi OR*;
- i cerchietti neri rappresentano *nodi AND*;
- i cammini dal basso verso l'alto rappresentano implicazioni logiche;
- le variabili (come x) hanno un campo di visibilità che si estende a tutto l'albero.

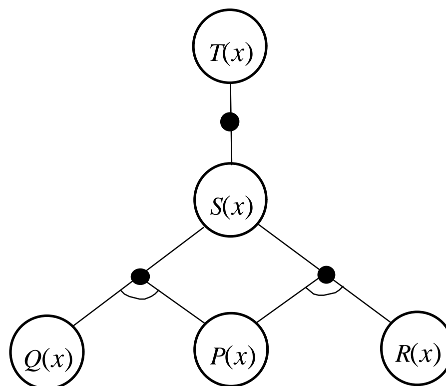

Figura 11.1 *Un albero AND-OR con variabili.*

In questo modo vengono messe in luce le dipendenze logiche fra diverse formule. Ad esempio, si vede facilmente che dai fatti F1–F3 deriva *in due modi distinti* il fatto $S(a)$, e da questo deriva $T(a)$. Se in

[2] Per essere precisi, l'"albero" della figura 11.1 non è un albero, bensì un "albero rientrante" o meglio un grafo orientato aciclico o *DAG* (*directed acyclic graph*).

seguito venisse rimosso il fatto *Q*(*a*) o il fatto *R*(*a*) (ma non ambedue), *S*(*a*) e *T*(*a*) continuerebbero a valere; se invece venisse rimosso *P*(*a*), anche *S*(*a*) e *T*(*a*) andrebbero rimossi.

Il problema che dobbiamo risolvere oro è di tener traccia delle dipendenze fra i fatti dedotti tramite CLIPS. A questo scopo:

- aggiungiamo a ogni fatto un simbolo che funge da *identificatore* univoco del fatto;
- insieme con ogni fatto memorizziamo la sua *giustificazione*, ovvero l'elenco degli identificatori dei fatti da cui dipende.

Per rendere la sintassi più chiara conviene ricorrere a una nuova struttura dati CLIPS: il *template*. Si tratta di una struttura che si presenta come una lista a due livelli (ovvero, una lista di liste), e che corrisponde a un *record* del Modula o a una *struct* del C. Un template si presenta in generale come:

$(simbolo_0\ (simbolo_1\ costante_1)\ \ldots\ (simbolo_n\ costante_n))$

Il primo simbolo del template, $simbolo_0$, è detto *intestazione* (*header*) e le sottoliste sono dette *caselle* (*slot*). Ogni slot contiene un simbolo, detto *nome* dello slot, seguito da una costante, detta *valore* dello slot. Uno slot può contenere anche più valori, e allora è detto *multislot*. Al contrario delle liste semplici, i template vanno dichiarati. Ad esempio,

```
(deftemplate FACT
  (slot id)
  (multislot cont)
)
```

dichiara un template con header `FACT`, uno slot di nome `id` (l'identificatore del fatto) e un multislot di nome `cont` (il contenuto del fatto). Volendo, sarebbe anche possibile dichiarare varie proprietà degli slot, fra cui il *tipo* del valore.

Gli esemplari di un template possono specificare i valori dei loro slot, ma non sono obbligati a farlo. Ad esempio, i seguenti esemplari sono tutti accettabili:

```
(FACT (id 1))
(FACT (cont P a))
(FACT (id 1)(cont P a))
```

Dichiariamo ora un template per le giustificazioni:

```
(deftemplate JUST
  (slot id)
  (multislot just)
)
```

dove `id` è l'identificatore del fatto che viene giustificato, e `just` è l'elenco degli identificatori dei fatti che hanno consentito la deduzione del fatto giustificato. Per motivi che vedremo in seguito, ci occorre anche un template

```
(deftemplate ASSERT
  (multislot cont)
  (multislot just)
)
```

dove `cont` è il contenuto di un fatto nuovo da inserire nella base dei fatti, e `just` è la sua giustificazione. Per convenzione, la giustificazione di una premessa è rappresentata dal simbolo `premise`.

Il nostri fatti iniziali saranno quindi:

```
(deffacts Premises
  (ASSERT (cont P a)(just premise))
  (ASSERT (cont Q a)(just premise))
  (ASSERT (cont R a)(just premise))
)
```

Gli assiomi A1–A3 sono rappresentati dalle seguenti regole:

```
(defrule A1
  (FACT (id ?id1)(cont P ?x))
  (FACT (id ?id2)(cont Q ?x))
 =>
  (assert (ASSERT (cont S ?x)(just ?id1 ?id2)))
)
(defrule A2
  (FACT (id ?id1)(cont P ?x))
  (FACT (id ?id2)(cont R ?x))
 =>
  (assert (ASSERT (cont S ?x)(just ?id1 ?id2)))
)
(defrule A3
  (FACT (id ?id)(cont S ?x))
 =>
  (assert (ASSERT (cont T ?x)(just ?id)))
)
```

Ci occorrono poi due regole per inserire i nuovi fatti, con priorità elevata. La prima regola inserisce il fatto se questo non è già presente nella base dei fatti:

```
(defrule assert-fact
  (declare (salience 10))
  (ASSERT (cont $?c)(just $?j))
  (not (FACT (cont $?c)))
 =>
  (assert (FACT (id (gensym))(cont $?c)))
)
```

Si noti l'uso della funzione CLIPS `gensym`, a zero argomenti, che genera un nuovo simbolo del tipo `gen`*k* ogni volta che viene chiamata. La seconda regola inserisce una nuova giustificazione e rimuove l'`ASSERT`:

```
(defrule justify-fact
         (declare (salience 10))
  ?nf <- (ASSERT (cont $?c)(just $?j))
         (FACT (id ?id)(cont $?c))
 =>
         (assert (JUST (id ?id)(just $?j)))
         (retract ?nf)
)
```

Vediamo che cosa succede con ciò che abbiamo già definito:

```
CLIPS>(set-strategy breadth)
CLIPS>(reset)
CLIPS>(run)
 0: (initial-fact)
 1: (ASSERT (cont P a)(just premise))
 2: (ASSERT (cont Q a)(just premise))
 3: (ASSERT (cont R a)(just premise))
 4: (FACT (id gen1)(cont P a))
 5: (FACT (id gen2)(cont Q a))
 6: (FACT (id gen3)(cont R a))
 7: (JUST (id gen1)(just premise))
 1: (ASSERT (cont P a)(just premise))
 8: (JUST (id gen2)(just premise))
 2: (ASSERT (cont Q a)(just premise))
 9: (JUST (id gen3)(just premise))
 3: (ASSERT (cont R a)(just premise))
10: (FACT (id gen4)(cont S a))
11: (JUST (id gen4)(just gen1 gen2))
12: (JUST (id gen4)(just gen1 gen3))
13: (FACT (id gen5)(cont T a))
14: (JUST (id gen5)(just gen4))
```

Definiamo ora le regole che consentono di cancellare dalla base dei fatti tutte le conseguenze di ogni fatto che venga rimosso. Il processo di cancellazione non deve poter interrompere l'inserimento di nuovi fatti, e avrà quindi priorità inferiore. La prima regola rimuove ogni fatto che sia rimasto privo di giustificazioni:

```
(defrule retract-unjustified-fact"
         (declare (salience 5))
   ?f <- (FACT (id ?id))
         (not (JUST (?id $?)))
 =>
         (retract ?f)
)
```

La seconda regola rimuove ogni giustificazione che faccia riferimento all'identificatore di un fatto rimosso:

```
(defrule retract-unsupported-justification
         (declare (salience 5))
   ?j <- (JUST (just $? ?id&~premise $?))
         (not (FACT (id ?id)))
 =>
         (retract ?j)
)
```

Si noti il pattern:

```
(JUST (just $? ?id&~premise $?))
```

in cui il vincolo `&~premise` aggiunto alla variabile `?id` non consente alla variabile di assumere il valore `premise` durante il matching.

Infine, se vogliamo garantire che i fatti rimossi dall'esterno siano sempre delle premesse, possiamo decidere di rimuovere le premesse inserendo nella base dei fatti un elemento che indica che una premessa va rimossa. Questo elemento può avere la forma:

```
(deftemplate RETRACT
  (multislot cont)
)
```

e la regola di rimozione può essere:

```
(defrule retract-premise
         (declare (salience 10))
   ?r <- (RETRACT (cont $?c))
         (FACT (id ?id)(cont $?c))
   ?j <- (JUST (id ?id)(just premise))
 =>
         (retract ?r ?j)__
)
```

11.2 Rappresentazione esplicita dell'informazione negativa

Fino ad ora abbiamo sempre assunto che l'*informazione positiva*, ovvero i fatti che sussistono, sia rappresentata esplicitamente da elementi nella base dei fatti, mentre l'*informazione negativa*, ovvero i fatti che non sussistono, sia rappresentata *implicitamente* dalla mancanza dei relativi elementi nella base dei fatti. Questa ipotesi ci ha consentito di sfruttare l'assunzione di chiusura nella definizione di regole con pattern negati.

In generale, l'idea di non rappresentare esplicitamente l'informazione negativa è giustificata da ragioni di complessità. Consideriamo ad esempio il problema dei cubetti, con un numero $n>1$ di cubetti. I fatti distinti esprimibili con i predicati *Block*(–), *On*(–,–), *Clear*(–) e *OnFloor*(–) sono n per ogni predicato a un argomento ed n^2 per ogni predicato a due argomenti, quindi in totale:

$$n^2 + 3n.$$

Di questi, i fatti effettivamente sussistenti in una situazione specifica possono variare da un minimo di $2n+1$ (quando tutti i cubetti sono disposti in un'unica pila) a un massimo di $3n$ (quando i cubetti sono tutti sul ripiano). Dunque, i fatti esprimibili sono dell'ordine di n^2, mentre i fatti che sussistono in una data situazione sono dell'ordine di n; ne segue che i fatti che non sussistono in una situazione sono dell'ordine di n^2. È chiaro quindi che non rappresentare esplicitamente i fatti che non sussistono porta a una notevole economia rappresentativa.

Ci sono casi, però in cui l'informazione negativa va rappresentata esplicitamente. Supponiamo, come nel paragrafo precedente, di voler tener traccia delle dipendenze fra fatti nel caso seguente:

(F1) $P(a)$

(A4) $P(x) \land \neg Q(x) \Rightarrow R(x)$

Sotto le assunzioni di chiusura, possiamo dedurre $R(a)$ con la seguente giustificazione: sussiste $P(a)$ e *non* sussiste $Q(a)$. Ma come possiamo registrare nella giustificazione di un fatto l'assenza di un altro fatto? In questo caso, l'unica strada è rappresentare esplicitamente il "fatto negativo" $\neg Q(a)$. Tuttavia, per non rinunciare del tutto all'economia rappresentativa di cui parlavamo poc'anzi, rappresenteremo soltanto i fatti negativi effettivamente utilizzati in almeno una deduzione.

Per rappresentare fatti sia positivi che negativi[3] ci basta aggiungere al template `FACT` uno slot per il valore di verità:

```
(deftemplate FACT
  (slot id)
  (slot val)
  (multislot cont)
)
```

Ad esempio, $P(a)$ e $\neg Q(a)$ sano rappresentabili come:

```
(FACT (id …)(val 1)(cont P a))
(FACT (id …)(val 0)(cont Q a))
```

Ridefiniamo il template `ASSERT` e le regole di asserzione di un fatto:

```
(deftemplate ASSERT
  (slot val)
  (multislot cont)
  (multislot just)
)
(defrule assert-fact
  (declare (salience 10))
  (ASSERT (val ?v)(cont $?c)(just $?j))
  (not (FACT (cont $?c)))
 =>
  (assert (FACT (id (gensym))(val ?v)(cont $?c)))
)
(defrule justify-fact
         (declare (salience 10))
  ?nf <- (ASSERT (val ?v)(cont $?c)(just $?j))
         (FACT (id ?id)(val ?v)(cont $?c))
 =>
         (assert (JUST (id ?id)(just $?j)))
         (retract ?nf)
)
```

Abbiamo poi una nuova regola, che dice che l'asserzione di un fatto positivo cancella automaticamente la giustificazione di un fatto negativo con lo stesso contenuto, se questo è stato introdotto per chiusura:

```
(defrule assert-positive-retract-negative
         (declare (salience 10))
         (ASSERT (val 1)(cont $?c))
         (FACT (id ?id)(val 0)(cont $?c))
   ?j <- (JUST (id ?id)(just closure))
 =>
         (retract ?j)
)
```

Il fatto negativo verrà poi rimosso in quanto ingiustificato, se non possiede altre giustificazioni. Dopo la rimozione del fatto negativo, la regola `assert-fact` potrà asserire il fatto positivo e la regola `justify-fact` introdurrà la giustificazione di quest'ultimo.

Con queste convenzioni, F1 diventa:

---

[3] Nella *semantica situazionale* i fatti positivi e negativi sono considerati le unità elementari di informazione e sono quindi chiamati *infoni* (sul modello di fotoni, elettroni, eccetera); vedi Barwise (1989).

```
(deffacts F1
  (ASSERT (val 1)(P a)(just premise))
)
```

Per quanto riguarda A4, introduciamo due regole: la prima deduce $R(x)$ quando sono già presenti sia $P(x)$ che $\neg Q(x)$; la seconda assume $\neg Q(x)$ per chiusura quando è presente $P(x)$, in modo da rendere poi applicabile la prima regola.

```
(defrule A4
  (FACT (id ?id1)(val 1)(cont P ?x))
  (FACT (id ?id2)(val 0)(cont Q ?x))
 =>
  (assert (ASSERT (val 1)(cont R ?x)(just ?id1 ?id2)))
)
(defrule A4-closure
  (FACT (val 1)(cont P ?x))
  (not (FACT (val 1)(cont Q ?x))
 =>
  (assert (ASSERT (val 0)(cont Q ?x)(just closure)))
)
```

Vediamo che cosa succede:

```
CLIPS> (reset)
 0: (initial-fact)
 1: (ASSERT (val 1)(cont P a)(just premise))
 2: (FACT (id gen1)(val 1)(cont P a))
 3: (JUST (id gen1)(just premise))
-1: (ASSERT (val 1)(cont P a)(just premise))
 4: (ASSERT (val 0)(cont Q a)(just closure))
 5: (FACT (id gen2)(val 0)(cont Q a))
 6: (JUST (id gen2)(just closure))
-4: (ASSERT (val 0)(cont Q a)(just closure))
 7: (ASSERT (val 1)(cont R a)(just gen1 gen2))
 8: (FACT (id gen3)(val 1)(cont R a))
 9: (JUST (id gen3)(just gen1 gen2))
-7: (ASSERT (val 1)(cont R a)(just gen1 gen2))
```

L'esecuzione si ferma quindi lasciando la base dei fatti:

```
 0: (initial-fact)
 2: (FACT (id gen1)(val 1)(cont P a))
 3: (JUST (id gen1)(just premise))
 5: (FACT (id gen2)(val 0)(cont Q a))
 6: (JUST (id gen2)(just closure))
 8: (FACT (id gen3)(val 1)(cont R a))
 9: (JUST (id gen3)(just gen1 gen2))
```

A questo punto vanno riformulate le regole per la rimozione dei fatti in seguito a modifica delle premesse (le regole R1 ed R2 del paragrafo precedente); questo problema è lasciato al lettore come esercizio. Ciò fatto, l'asserzione di $Q(a)$ come premessa porta alla seguente continuazione:

```
10: (ASSERT (val 1)(cont Q a)(just premise))
-6: (JUST (id gen2)(just closure))
-5: (FACT (id gen2)(val 0)(cont Q a))
11: (FACT (id gen3)(val 1)(cont Q a))
```

```
12: (JUST (id gen3)(just premise))
```
~~10: (ASSERT (val 1)(cont Q a)(just premise))~~
~~9: (JUST (id gen3)(just gen1 gen2))~~
~~8: (FACT (id gen3)(val 1)(cont R a))~~

L'esecuzione si ferma quindi lasciando la base dei fatti:

```
 0: (initial-fact)
 2: (FACT (id gen1)(val 1)(cont P a))
 3: (JUST (id gen1)(just premise))
11: (FACT (id gen3)(val 1)(cont Q a))
12: (JUST (id gen3)(just premise))
```

11.3 Ragionamento non monotono

Una caratteristica fondamentale dei sistemi logici standard (come la logica del primo ordine) è che l'aggiunta di nuove premesse non cancella le precedenti conclusioni. In altre parole, se $KB \subseteq KB'$ (ovvero, se KB' si ottiene aggiungendo zero o più enunciati a KB) si ha che:

se $KB \vdash \varphi$, allora $KB' \vdash \varphi$.

Questa proprietà si esprime dicendo che i sistemi logici standard sono *monotoni.* Si tratta di una proprietà molto importante, perché consente il trasferimento di conclusioni precedenti a nuove situazioni. Ad esempio, supponiamo di aver dimostrato una certa proprietà φ di tutti i rettangoli. Dato che un quadrato è qualcosa di più di un rettangolo, sappiamo che φ varrà anche per i quadrati.

Dimostrare che la logica del primo ordine è monotona è molto facile. Possiamo dimostrarlo, ad esempio, facendo riferimento al calcolo di Beth. Dire che $KB \vdash \varphi$ significa dire che una tabella semantica contenente inizialmente tutta la KB nella colonna 1 e φ nella colonna 0 porta a una contraddizione. Ma questa contraddizione non sparisce se aggiungiamo nuovi enunciati alla KB nella colonna 1.

Nella logica di tutti i giorni può sembrare che la monotonia non sussista. Ad esempio, dal fatto che *a* sia un uccello e che gli uccelli volano:

(F4) $\mathit{Uccello}(a)$,

(A5) $\mathit{Uccello}(x) \Rightarrow \mathit{Vola}(x)$,

posso dedurre che *a* vola:

(F5) $\mathit{Vola}(a)$.

Ma basta aggiungere la nuova premessa che *a* è un pinguino per far cadere la conclusione F5. In realtà, questo esempio non dimostra che la logica non sia monotona. La conclusione F5, infatti, poggia su innumerevoli *premesse implicite*: *a* non è un pinguino, *a* non è uno struzzo, *a* non ha le ali tarpate e così via. Aggiungere la premessa che *a* è un pinguino non estende quindi la KB iniziale, ma piuttosto la trasforma in una KB diversa e incompatibile con la precedente. Dal punto di vista pratico, tuttavia, resta il problema che per applicare la logica classica tutte le premesse implicite vanno rese esplicite; e quando sono in ballo problemi di dimensione realistica questo può essere difficile, oneroso o addirittura impossibile.

Nei sistemi di intelligenza artificiale si fa spesso l'ipotesi che le conclusioni vengano derivate dalle premesse sotto l'*assunzione di normalità.* In altre parole, si assume che il mondo si comporti "normalmente", senza anomalie o imprevisti: se *x* è un uccello *normale rispetto al volo*, allora vola. Non solo: questa assunzione viene di solito espressa in termini negativi, così che possa essere assunta per

chiusura in assenza di informazioni contrarie: se *x* è un uccello *e non è anomalo rispetto al volo*, allora vola:

```
(defrule A4
  (FACT (id ?id1)(val 1)(cont Uccello ?x))
  (FACT (id ?id2)(val 0)(cont AnomaloPerIlVolo ?x))
 =>
  (assert (ASSERT (val 1)(cont Vola ?x)(just ?id1 ?id2))
)
(defrule A4-closure
  (FACT (id ?id1)(val 1)(cont Uccello ?x))
  (not (FACT (id ?id2)(val 1)(cont AnomaloPerIlVolo ?x)))
 =>
  (assert (ASSERT (val 0)(cont AnomaloPerIlVolo ?x)(just closure))
)
```

Inoltre, diciamo che i pinguini sono anomali rispetto al volo

```
(defrule Pinguino
  (FACT (id ?id1)(val 1)(cont Pinguino ?x))
 =>
  (assert (ASSERT (val 1)(cont AnomaloPerIlVolo ?x)(just ?id1)))
)
```

Partiamo dalla conoscenza che *a* è un uccello. Eseguiti gli inserimenti e le deduzioni abbiamo

```
 0: (initial-fact)
 2: (FACT (id gen1)(val 1)(cont Uccello a))
 3: (JUST (id gen1)(just premise))
 5: (FACT (id gen2)(val 0)(cont AnomaloPerIlVolo a))
 6: (JUST (id gen2)(just closure))
 8: (FACT (id gen3)(val 1)(cont Vola a))
 9: (JUST (id gen3)(just gen1 gen2))
```

Asseriamo ora che *a* sia un pinguino. Dopo gli inserimenti e le cancellazioni abbiamo:

```
 0: (initial-fact)
 2: (FACT (id gen1)(val 1)(cont Uccello a))
 3: (JUST (id gen1)(just premise))
11: (FACT (id gen4)(val 1)(cont Pinguino a))
12: (JUST (id gen4)(just premise))
14: (FACT (id gen5)(val 1)(cont AnomaloPerIlVolo a))
15: (JUST (id gen5)(just gen4))
```

Riferimenti bibliografici (parte III)

Barwise, J. (1989). *The situation in logic*, CSLI, Stanford, CA.

Chellas, B. F. (1980). *Modal logic: An introduction*, Cambridge University Press, Cambridge, UK.

Hughes, G. E., and M. J. Cresswell (1996). *A new introduction to modal logic*, Rutledge, London, UK.