Introduzione alla Logica Proposizionale

Marta Cialdea Mayer

a.a. 2010/2011

Indice

1 Logica e informatica **2**

2 La logica proposizionale **4**
- 2.1 Il linguaggio della logica proposizionale 4
- 2.2 Semantica: interpretazioni e verità 8
- 2.3 Tavole di verità 10
- 2.4 Connettivi logici e linguaggio naturale 11
- 2.5 Soddisfacibilità, validità, equivalenza logica 12
- 2.6 Conseguenza logica 15
- 2.7 L'isola dei cavalieri e furfanti 18
- 2.8 Forme normali disgiuntive e congiuntive 19
- 2.9 Deduzione automatica 20
- 2.10 Il metodo dei tableaux per la logica proposizionale 20
- 2.11 Esercizi 27
- 2.12 Soluzione di alcuni esercizi 32

3 Implementazione OCaml di alcuni algoritmi **36**
- 3.1 Rappresentazione di formule proposizionali 36
- 3.2 Un parser per le formule proposizionali 37
  - 3.2.1 La specifica della grammatica 39
  - 3.2.2 L'analisi lessicale 40
  - 3.2.3 Compilazione dei file e utilizzazione del parser in modalità interattiva 42
  - 3.2.4 Compilazione separata e codice eseguibile 43
  - 3.2.5 Uso di moduli in modalità interattiva 45
- 3.3 Forme normali 46
- 3.4 Valutazione di una formula in una interpretazione 47
- 3.5 Tavole di verità 47
- 3.6 Conseguenza e equivalenza logica 49
- 3.7 Il metodo dei tableaux 50
- 3.8 Esercizi 53

Riferimenti Bibliografici **54**

1 Logica e informatica

La logica è lo studio delle forme del ragionamento, con l'obiettivo principale di riconoscere e individuare le forme dei ragionamenti corretti, e distinguerle da quelli scorretti. Per questo motivo essa viene a volte chiamata logica "formale", o anche "simbolica", perché le "forme" del ragionamento vengono espresse mediante l'uso di simboli, che consentono di astrarre dal contenuto dei ragionamenti stessi. Astrarre significa semplificare: sbarazzandosi dei dettagli, si guardano oggetti diversi da uno stesso punto di vista, e gli si può dare un trattamento uniforme.

Si considerino ad esempio i due ragionamenti seguenti:

a) Se nevica, la temperatura è di 0^oC.
   Nevica.
   Quindi la temperatura è di 0^oC.

b) Se la matematica è un'opinione, allora $1 + 1 = 0$.
   La matematica è un'opinione.
   Quindi $1 + 1 = 0$.

I due ragionamenti hanno evidentemente la stessa forma, anche se essi riguardano concetti diversi e diverso è il contenuto di verità delle proposizioni in essi coinvolte. Possiamo identificare la loro forma comune utilizzando simboli, A e B, al posto delle proposizioni:

Se A allora B.
A.
Quindi B.

Lo studio della logica, nato nell'antica Grecia con Aristotele e i logici megarici e stoici, ha conosciuto un grande sviluppo a partire dalla fine del secolo scorso, soprattutto come studio del tipo di ragionamento normalmente condotto nel dimostrare teoremi matematici (logica matematica). Per rappresentare proposizioni e ragionamenti, è stato definito un linguaggio formale che, attraverso una serie di evoluzioni, ha raggiunto una forma standard ormai universalmente accettata.

In realtà non è corretto parlare di "logica", al singolare, in quanto sono state studiate diverse "logiche", che trovano applicazione in diversi domini. Ciascuna di esse è caratterizzata da un suo linguaggio formale (con una *sintassi*, che definisce l'insieme delle espressioni corrette del linguaggio, ed una *semantica* che ne definisce il significato) e uno o più metodi per "ragionare" con tali espressioni. La logica più antica e più nota, che costituisce la base di molte altre logiche, è la *logica classica* (proposizionale e dei predicati): un linguaggio formale per la rappresentazione di conoscenza di tipo *dichiarativo.* Altre forme di di conoscenza non possono però essere rappresentate facilmente in logica classica, e sono state studiate altre logiche come, ad esempio la "logica epistemica" (che permette di rappresentare conoscenze espresse in italiano da espressioni del tipo "l'agente sa che ...", "l'agente crede che ...") o le logiche "temporali", che consentono la rappresentazione di un mondo che cambia nel tempo mediante operatori che esprimono concetti quali "sarà sempre vero che ...", "in qualche momento futuro sarà vero che ...".

La logica classica è il formalismo fondamentale del ragionamento matematico: come le proposizioni della matematica, le espressioni della logica classica sono caratterizzate dal fatto di essere o vere o false (indipendentemente dal tempo, dalla nostra conoscenza di esse, ecc.).

Tra i campi di applicazione più importanti della logica rientrano diversi settori dell'informatica. Logica e informatica sono infatti discipline in stretta relazione, nonostante la diversità di "punti di vista": la logica si è principalmente interessata della verità delle proposizioni, cioè di "che cosa" si può dire, e questo viene stabilito mediante dimostrazioni; al contrario, l'informatica è interessata al "come" fare. La logica ha fornito e fornisce importanti contributi all'informatica, sotto diversi aspetti. Viceversa, non soltanto è campo d'applicazione per l'informatica, che fornisce ai logici utili strumenti di lavoro, ma l'informatica ha offerto allo sviluppo della logica nuovi problemi e punti di vista, quali, ad esempio, le problematiche relative allo studio di strutture finite o all'efficienza di procedure logiche.

Per quel che riguarda l'impatto della logica sull'informatica, osserviamo che essa è innanzitutto uno strumento di base per la meta-informatica: ha portato, ad esempio, a definire la nozione di calcolabilità, ai risultati di indecidibilità, alla classificazione dei problemi secondo la loro complessità. Questo è il ruolo esterno e teorico che ha la logica rispetto all'informatica, che riproduce il rapporto che ha e ha sempre avuto con la matematica, relativo alla questione dei loro fondamenti.

La logica ha fornito all'informatica molti concetti fondamentali, come quello di semantica formale, che è alla base della semantica denotazionale dei linguaggi di programmazione, e metodologie generali come il trattamento assiomatico di informazioni, che trova applicazione, ad esempio, nella dimostrazione di proprietà di programmi.

La logica svolge inoltre un ruolo interno e pratico rispetto all'informatica, dove trovano applicazione i suoi metodi e gli strumenti da essa forniti. La logica è infatti il formalismo fondamentale per la rappresentazione della conoscenza in intelligenza artificiale, dove trovano applicazione i metodi automatici per la dimostrazione di teoremi, studiati e realizzati prima ancora della loro applicazione nella costruzione di sistemi di informatica e intelligenza artificiale. Inoltre, i linguaggi di specifica formale e la specifica algebrica di tipi astratti di dati si fondano su metodologie logiche.

Uno dei terreni in cui la logica gioca un ruolo pratico essenziale è relativo ai paradigmi di calcolo. I due fondamentali sono quelli della risoluzione e della riduzione, associati rispettivamente alla *programmazione logica* e alla *programmazione funzionale*.

Il primo approccio è più direttamente basato sulla logica e consiste nel vedere il calcolo come la risoluzione di un problema. Esempio tipico è quello delle equazioni: la soluzione di un'equazione, ad esempio $x^2 - 9 = 0$, è data dai valori delle sue variabili che la rendono vera ($x = 3$ o $x = -3$). La programmazione logica si basa su importanti risultati teorici relativi alla possibilità di definire metodi di dimostrazione automatica e, restringendo opportunamente il linguaggio, metodi *costruttivi*. La costruttività di una dimostrazione consente di estrarre dalla dimostrazione stessa informazioni importanti, che si aggiungono al fatto di aver determinato che una certa formula è un teorema. Un po' più in dettaglio, un *programma logico* P consiste in un isieme di assiomi che descrivono in modo opportuno il dominio di interesse e viene eseguito a fronte di un *obiettivo* $A(x_1, ..., x_n)$; l'esecuzione consiste nella ricerca di una dimostrazione costruttiva

di $\exists x_1 ... \exists x_n A(x_1, ..., x_n)$ a partire da P. Se tale dimostrazione esiste, è possibile da essa estrarre dei valori (termini) $t_1, ..., t_n$ tali che $A(t_1, ..., t_n)$ è dimostrabile da P. Tali termini costituiscono il risultato del calcolo.

Secondo il paradigma della *riduzione*, si calcola riducendo un'espressione a un'altra più semplice o più vicina a un valore, cioè a un'espressione non ulteriormente semplificabile. Ad esempio:

$$(6 + 3) \times (8 - 2) \Rightarrow 9 \times (8 - 2) \Rightarrow 9 \times 6 \Rightarrow 54$$

La *programmazione funzionale* si basa su questo: un programma funzionale è costituito dalla definizione di un insieme di funzioni, che possono richiamarsi l'una con l'altra. Eseguire un programma funzionale consiste nel calcolare il valore di una data espressione, utilizzando tali definizioni, semplificandola fin dove possibile.

Lo scheletro concettuale di molti linguaggi funzionali, come ML, Miranda, Hope e lo stesso LISP è costituito dal *lambda-calcolo* (λ-calcolo). È un linguaggio logico semplice, che consente, esplicitamente, soltanto la definizione di funzioni e la loro applicazione. Tuttavia, dietro questa semplicità sintattica, si cela un linguaggio di grande potenza espressiva: rispetto alla capacità di esprimere algoritmi, il lambda-calcolo è altrettanto potente quanto gli altri linguaggi di programmazione. D'altro canto, il λ-calcolo è una *teoria delle funzioni*. Esso è nato infatti prima dello sviluppo dei linguaggi di programmazione; è stato introdotto da Church negli anni 30, allo scopo di dare una definizione formale di ciò che è calcolabile e studiare i limiti della calcolabilità.

Il ruolo pratico della logica rispetto all'informatica, sopra brevemente illustrato, è di fatto la manifestazione dell'identità profonda esistente tra logica e programmazione. È dimostrato infatti che concepire un programma equivale a dimostrare una proposizione in maniera costruttiva, ed eseguire un programma funzionale equivale a trasformare una dimostrazione in una forma "normale".

Questa dispensa contiene un'introduzione alla logica più semplice, la logica proposizionale classica. Nella prima parte vengono presentate la sintassi e la semantica di tale logica, ed il metodo di dimostrazione automatica basato sui tableaux. La seconda parte propone un modo di rappresentare le formule proposizionali nel linguaggio Objective Caml ed alcuni programmi per operare su di esse.

2 La logica proposizionale

2.1 Il linguaggio della logica proposizionale

Consideriamo di nuovo i semplici ragionamenti del paragrafo precedente. Ciascuno di essi è caratterizzato da un insieme di *premesse* (le affermazioni precedenti la parola "quindi") e una *conclusione* (la proposizione che segue la parola "quindi"). La forma comune dei due ragionamenti può essere schematizzata come segue, separando le premesse dalla conclusione mediante una linea (la *linea di inferenza*), anziché la parola "quindi":

$$\frac{\text{Se } A\text{, allora } B \qquad A}{B}$$

Qui A e B sono simboli, che stanno per proposizioni qualsiasi. Invece "se" e "allora" sono "parole logiche" che connettono proposizioni. Se il ragionamento è corretto, le premesse – se sono vere – giustificano la conclusione. In altri termini, un ragionamento è corretto quando *ogni volta che sono vere* le premesse, è vera anche la conclusione (ovviamente, quando le premesse sono false, non si può dire niente sulla conclusione). È in base al significato delle parole logiche che possiamo dire se un ragionamento è corretto o no. Le parole logiche sono espresse da simboli particolari. Ad esempio, "se–allora" viene chiamato "implicazione" e rappresentato da una freccia. Utilizzando questo simbolismo, la forma di ragionamento sopra riportata si può scrivere così:

$$\frac{A \rightarrow B \qquad A}{B}$$

Ragionamenti di questa forma sono sempre corretti. Questo viene stabilito in base al significato di $\rightarrow$, che assicura, in particolare, che $A \rightarrow B$ è falso solo quando A è vero e B è falso (ciò si vedrà meglio in seguito). Quindi, se sia A che $A \rightarrow B$ sono veri, B deve essere vero.

La logica proposizionale studia il significato di parole logiche che, come l'implicazione, si possono usare per "connettere" proposizioni, e che sono chiamate appunto *connettivi proposizionali* (o *operatori booleani*). I mattoni di base di un linguaggio proposizionale sono costituiti dalle *proposizioni atomiche*, che rappresentano enunciati non ulteriormente analizzati. Un enunciato è un'affermazione di tipo dichiarativo per la quali ha senso chiedersi se sia vera o falsa. Le proposizioni atomiche sono rappresentate da simboli, detti *lettere proposizionali* o *variabili proposizionali*. Gli enunciati semplici possono essere composti per formare enunciati più complessi, mediante i connettivi proposizionali.

Un linguaggio proposizionale è dunque costituito da un insieme di simboli, alcuni dei quali (le lettere proposizionali) possono variare da un linguaggio all'altro, mentre altri (i connettivi) sono un patrimonio comune di tutti i linguaggi proposizionali. I principali connettivi proposizionali sono i seguenti:

$\neg$ è la negazione: si legge "non";

$\wedge$ è la congiunzione: si legge "e";

$\vee$ è la disgiunzione: si legge "oppure";

$\rightarrow$ è l'implicazione: si legge "implica" (o "se ... allora ...");

$\equiv$ è la doppia implicazione: si legge "se e solo se".

Oltre ai connettivi, i linguaggi che considereremo conterranno tutti due *costanti proposizionali*: $\top$, che sta per un enunciato sempre vero, e $\bot$, che sta per un enunciato sempre falso. Le espressioni, semplici o complesse, che si possono formare sulla base di un linguaggio proposizionale sono chiamate *formule proposizionali*.

Ad esempio, l'enunciato "Caino è fratello di Abele" è un atomo, che possiamo rappresentare mediante la lettera proposizionale p; "a ogni paziente piace qualche dottore" è un altro atomo, che rappresentiamo mediante un'altra variabile proposizionale, q. Le frasi a sinistra nella tabella che segue sono allora rappresentate dalle formule a destra:

Caino è fratello di Abele *oppure* a ogni paziente piace qualche dottore	$p \vee q$
Caino *non* è fratello di Abele	$\neg p$
Caino è fratello di Abele *e* a ogni paziente piace qualche dottore	$p \wedge q$
Se Caino è fratello di Abele, *allora* a ogni paziente piace qualche dottore	$p \rightarrow q$
Caino è fratello di Abele *se e solo se* a ogni paziente piace qualche dottore	$p \equiv q$
Se Caino *non* è fratello di Abele, *allora* Caino è fratello di Abele *oppure* a ogni paziente piace qualche dottore	$\neg p \rightarrow (p \vee q)$

La *sintassi* della logica proposizionale determina quali sono le espressioni corrette di tale logica, cioè definisce che cos'è un linguaggio proposizionale. Come abbiamo visto, un linguaggio proposizionale è costruito sulla base di un determinato insieme P di variabili proposizionali. Ad esempio potremmo considerare:

$$P = \{p, q, r\}$$
$$P' = \{p_0, p_1, p_2, ...\}$$
$$P'' = \{caino_fratello_di_abele, pazienti_dottori, ...\}$$

Le espressioni del linguaggio della logica proposizionale sono dette **formule proposizionali** – a volte abbreviato in *fbf* per *"formule ben formate"*. Dato un insieme P di variabili proposizionali, l'insieme delle formule basate su P (il linguaggio proposizionale basato su P), $Prop[P]$, è definito induttivamente come segue.

Definizione 1 *Sia P un insieme, i cui elementi sono chiamati* **lettere (o variabili) proposizionali**. *L'insieme delle formule in $Prop[P]$ è definito induttivamente:*

1. *ogni variabile proposizionale in P è una formula;*
2. *$\top$ e $\bot$ sono formule;*
3. *se A è una formula, allora anche $\neg A$ è una formula;*
4. *se A e B sono formule, allora anche $(A \wedge B)$, $(A \vee B)$, $(A \rightarrow B)$, $(A \equiv B)$ sono formule;*
5. *nient'altro è una formula.*

Una formula della forma $A \wedge B$ si dice congiunzione, e A e B sono i due congiunti. Una formula della forma $A \vee B$ si dice disgiunzione, e A e B sono i due disgiunti. Una formula della forma $A \rightarrow B$ si dice implicazione, A è l'antecedente e B il conseguente dell'implicazione. Una formula della forma $A \equiv B$ si dice doppia implicazione.

Consideriamo ad esempio l'insieme di lettere proposizionali $P = \{p, q, r, s\}$; le espressioni seguenti sono in $Prop[P]$:

$p,\ q,\ r,\ s,\ \top,\ \bot,$
$\neg p,\ \neg q,\ \neg r,\ \neg s,\ \neg\top,\ \neg\bot,$
$(p \wedge q),\ (p \vee q),\ (q \rightarrow r),\ (r \equiv s),\ ...$
$\neg(p \wedge q),\ ((p \vee q) \equiv (q \rightarrow r)),\ ((q \rightarrow r) \vee \neg s),\ ...$
$(\neg(p \wedge q) \rightarrow ((p \vee q) \equiv (q \rightarrow r))),\ ((q \rightarrow r) \vee \neg s) \wedge \neg\top),\ ...$

Convenzioni sull'uso delle parentesi: ometteremo le parentesi dalle formule quando non ci sia rischio di ambiguità, secondo le seguenti convenzioni:

- Si possono omettere le parentesi esterne. Ad esempio, anziché $(p \wedge q)$ si può scrivere $p \wedge q$.
- Associatività: i connettivi associano da sinistra a destra (se non indicato altrimenti dall'uso di parentesi). Ad esempio: $((p \wedge q) \wedge r)$ si può scrivere $p \wedge q \wedge r$, e $((p \rightarrow q) \rightarrow r)$ si può scrivere $p \rightarrow q \rightarrow r$. Ma $(p \rightarrow (q \rightarrow r))$ non si può scrivere $p \rightarrow q \rightarrow r$, che sta invece per $((p \rightarrow q) \rightarrow r)$.
- L'ordine di precedenza dei connettivi è:
$$\neg,\ \wedge,\ \vee,\ \rightarrow,\ \equiv$$
(se non indicato altrimenti dall'uso di parentesi). Ad esempio: $(p \rightarrow (q \wedge r))$ si può scrivere $p \rightarrow q \wedge r$, e $\neg(p \rightarrow (q \wedge r))$ si può scrivere $\neg(p \rightarrow q \wedge r)$. Ma $\neg(p \rightarrow (q \wedge r))$ non si può scrivere $\neg p \rightarrow q \wedge r$, che sta invece per $((\neg p) \rightarrow (q \wedge r))$; e $(p \rightarrow (q \equiv r))$ non si può scrivere $p \rightarrow q \equiv r$, che sta invece per $((p \rightarrow q) \equiv r)$.

Quindi, ad esempio:

$s \rightarrow p \wedge q \vee \neg r$ abbrevia $(s \rightarrow ((p \wedge q) \vee \neg r))$

$p \rightarrow (p \rightarrow r \wedge q)$ abbrevia $(p \rightarrow (p \rightarrow (r \wedge q)))$

$p \equiv \neg q \equiv r \vee s$ abbrevia $((p \equiv \neg q) \equiv (r \vee s))$

Se A è una formula, una sua sottoformula è una qualsiasi sottoespressione (propria o impropria) di A che sia essa stessa una formula, cioè ogni formula che compare nella costruzione di A secondo la definizione 1. Con maggior precisione, l'insieme delle sottoformule di una formula A, $subf(A)$, è definito induttivamente come segue:

1. Se A è atomica allora $subf(A) = \{A\}$.
2. $subf(\neg A) = subf(A) \cup \{\neg A\}$.
3. $subf(A \wedge B) = subf(A) \cup subf(B) \cup \{A \wedge B\}$.
4. $subf(A \vee B) = subf(A) \cup subf(B) \cup \{A \vee B\}$.
5. $subf(A \rightarrow B) = subf(A) \cup subf(B) \cup \{A \rightarrow B\}$.
6. $subf(A \equiv B) = subf(A) \cup subf(B) \cup \{A \equiv B\}$.

Il *connettivo principale* di una formula (non atomica) è l'ultimo connettivo utilizzato nella costruzione della formula:

1. il connettivo principale di $\neg A$ è $\neg$;
2. il connettivo principale di $A \wedge B$ è $\wedge$;
3. il connettivo principale di $A \vee B$ è $\vee$;
4. il connettivo principale di $A \rightarrow B$ è $\rightarrow$;
5. il connettivo principale di $A \equiv B$ è $\equiv$.

Poiché l'insieme delle formule è definito induttivamente, possiamo formulare un corrispondente principio di induzione, il *principio di induzione sulle formule*:

Se P è una proprietà delle formule tale che:

(B) P vale per ogni lettera proposizionale, per $\top$ e $\bot$, e

(PI) per tutte le formule A e B:

- se P vale per A allora P vale per $\neg A$;
- se P vale per A e B, allora P vale per $(A \wedge B)$, per $(A \vee B)$, per $(A \rightarrow B)$, per $(A \equiv B)$;

allora P vale per ogni formula.

2.2 Semantica: interpretazioni e verità

La sintassi di un linguaggio determina la forma delle espressioni corrette nel linguaggio. La semantica assegna un significato alle espressioni sintatticamente corrette di tale linguaggio. Per stabilire la semantica di un linguaggio, occorre specificare innanzituto un **dominio di interpretazione**, cioè l'insieme degli oggetti che costituiscono possibili significati per le espressioni del linguaggio. La semantica di un linguaggio associa allora ad ogni espressione sintattica un oggetto del dominio di interpretazione.

Nel caso della logica proposizionale il dominio di interpretazione è costituito dall'insieme $Bool = \{T, F\}$ dei valori booleani (o valori di verità). Per studiare il significato dei connettivi proposizionali, infatti, si stabilisce che una proposizione atomica può essere vera o falsa. Il significato dei connettivi viene stabilito definendo il valore di verità di proposizioni complesse a partire da quello dei loro componenti immediati.

Il significato (il valore di verità) di una formula in $Prop[P]$ dipende quindi dal significato (il valore di verità) delle lettere proposizionali in P. Dunque, per stabilire il significato di una formula proposizionale occorre *assegnare* un valore di verità a ciascuna variabile proposizionale. Ad esempio, il significato della formula $p \wedge q$ si può stabilire soltanto conoscendo il significato di p e di q; se tali atomi sono entrambi veri, allora $p \wedge q$ è vero, altrimenti è falso.

Definizione 2 *Sia P un insieme di lettere proposizionali. Una* **interpretazione** *o* **assegnazione** $\mathcal{M}$ *per P è una funzione*

$$\mathcal{M} : P \rightarrow Bool$$

Ad esempio, se $P = \{p, q, r\}$, ci sono 8 possibili interpretazioni di P:

	p	q	r
$\mathcal{M}_1$	T	T	T
$\mathcal{M}_2$	T	T	F
$\mathcal{M}_3$	T	F	T
$\mathcal{M}_4$	T	F	F

	p	q	r
$\mathcal{M}_5$	F	T	T
$\mathcal{M}_6$	F	T	F
$\mathcal{M}_7$	F	F	T
$\mathcal{M}_8$	F	F	F

L'interpretazione di una formula dipende dall'interpretazione delle lettere proposizionali che occorrono in essa. Una volta fissata un'interpretazione $\mathcal{M}$ per P, risulta comunque stabilita l'interpretazione di qualsiasi formula in $Prop[P]$, in base al significato delle lettere proposizionali che occorrono in essa e al significato dei connettivi logici. Se A è una formula e $\mathcal{M}$ un'interpretazione di A, la notazione $\mathcal{M} \models A$ indica che A è vera in $\mathcal{M}$, cioè che il significato di A in $\mathcal{M}$ è T (e si dice allora che $\mathcal{M}$ è un *modello* di A). Se A è falsa in $\mathcal{M}$ scriviamo $\mathcal{M} \not\models A$ (e si dice allora che $\mathcal{M}$ è un *contromodello* di A). La verità o falsità di una formula in $\mathcal{M}$ costituisce la sua interpretazione secondo $\mathcal{M}$, ed è definita induttivamente come segue ("sse" sta per "se e solo se"):

Definizione 3 *La verità di una formula $A \in Prop[P]$ in un'interpretazione $\mathcal{M}$ per P è definita induttivamente come segue:*

1. $\mathcal{M} \models p$ *sse* $\mathcal{M}(p) = T$, *se* $p \in P$;
2. $\mathcal{M} \models \top$ *e* $\mathcal{M} \not\models \bot$;
3. $\mathcal{M} \models \neg A$ *sse* $\mathcal{M} \not\models A$
4. $\mathcal{M} \models A \wedge B$ *sse* $\mathcal{M} \models A$ *e* $\mathcal{M} \models B$
5. $\mathcal{M} \models A \vee B$ *sse* $\mathcal{M} \models A$ *oppure* $\mathcal{M} \models B$
6. $\mathcal{M} \models A \rightarrow B$ *sse* $\mathcal{M} \not\models A$ *oppure* $\mathcal{M} \models B$
7. $\mathcal{M} \models A \equiv B$ *sse* $\mathcal{M} \models A$ *e* $\mathcal{M} \models B$, *oppure* $\mathcal{M} \not\models A$ *e* $\mathcal{M} \not\models B$

Se $\mathcal{M} \models A$ diciamo che $\mathcal{M}$ è un modello *di A o che $\mathcal{M}$* soddisfa *A. Se $\mathcal{M} \not\models A$ allora $\mathcal{M}$ è un* contromodello *di A.*

Se S è un insieme di formule, allora $\mathcal{M}$ è un modello di S (e scriviamo $\mathcal{M} \models S$) sse $\mathcal{M} \models A$ per ogni formula $A \in S$.

Osserviamo che dalla definizione precedente segue la definizione di $\mathcal{M} \not\models A$:

1. $\mathcal{M} \not\models p$ sse $\mathcal{M}(p) = F$, se $p \in P$;
2. $\mathcal{M} \not\models \top$ è sempre falso e $\mathcal{M} \not\models \bot$ è sempre vero;
3. $\mathcal{M} \not\models \neg A$ sse $\mathcal{M} \models A$;
4. $\mathcal{M} \not\models A \wedge B$ sse $\mathcal{M} \not\models A$ oppure $\mathcal{M} \not\models B$;
5. $\mathcal{M} \not\models A \vee B$ sse $\mathcal{M} \not\models A$ e $\mathcal{M} \not\models B$;
6. $\mathcal{M} \not\models A \rightarrow B$ sse $\mathcal{M} \models A$ e $\mathcal{M} \not\models B$;
7. $\mathcal{M} \not\models A \equiv B$ sse $\mathcal{M} \models A$ e $\mathcal{M} \not\models B$, oppure $\mathcal{M} \not\models A$ e $\mathcal{M} \models B$;

La definizione 3, di fatto, stabilisce il significato dei connettivi logici. Essa si può riformulare infatti come segue. Se denotiamo con NOT, AND, OR, IMP e IFF le operazioni booleane su $\{T, F\}$, tali che:

	NOT
T	F
F	T

		AND	OR	IMP	IFF
T	T	T	T	T	T
T	F	F	T	F	F
F	T	F	T	T	F
F	F	F	F	T	T

abbiamo stabilito che il significato di:

$\neg$	è	NOT	$\wedge$	è	AND
$\vee$	è	OR	$\rightarrow$	è	IMP
$\equiv$	è	IFF			

Quindi questa definizione stabilisce il significato dei connettivi logici.

Se $A \in Prop[P]$, ad ogni assegnazione di valori di verità T o F alle lettere proposizionali che compaiono in A corrisponde un valore di verità della formula A stessa; questo si può ottenere utilizzando la definizione 3. Ad esempio, possiamo calcolare il valore di verità di $A = (\neg p \vee q) \rightarrow r$ nell'interpretazione $\mathcal{M}$ tale che $\mathcal{M}(p) = \mathcal{M}(q) = T$ e $\mathcal{M}(r) = F$ come segue:

$\mathcal{M} \models (\neg p \vee q) \rightarrow r$
sse $\mathcal{M} \not\models \neg p \vee q$ oppure $\mathcal{M} \models r$
sse $\mathcal{M} \not\models \neg p \vee q$ (perché $\mathcal{M}(r) = F$, quindi $\mathcal{M} \not\models r$)
sse $\mathcal{M} \not\models \neg p$ e $\mathcal{M} \not\models q$
sse $\mathcal{M} \models p$ e $\mathcal{M}(q) = F$
sse $\mathcal{M}(p) = T$ e $\mathcal{M}(q) = F$: FALSO.
Quindi $\mathcal{M} \not\models (\neg p \vee q) \rightarrow r$

2.3 Tavole di verità

Il significato dei connettivi proposizionali si può rappresentare sinteticamente mediante le *tavole di verità* dei connettivi logici, come illustrato nella tabella seguente. Nella prima e nella seconda colonna vengono mostrati i valori di verità di A e B e nelle colonne successive i valori di verità delle formule ottenute mediante applicazione dei connettivi.

A	B	$\neg A$	$A \vee B$	$A \wedge B$	$A \rightarrow B$	$A \equiv B$
T	T	F	T	T	T	T
T	F	F	T	F	F	F
F	T	T	T	F	T	F
F	F	T	F	F	T	T

Quando si vuole calcolare il valore di verità di una formula A in *tutte* le sue interpretazioni, un metodo compatto, alternativo all'applicazione ripetuta della definizione 3, consiste nella costruzione della *tavola di verità* per la formula A che utilizza le tavole di verità dei connettivi logici. Ad esempio, la tavola di verità della formula $A = (\neg p \vee q) \rightarrow r$ si costruisce come segue:

p	q	r	$\neg p$	$\neg p \vee q$	$(\neg p \vee q) \rightarrow r$
T	T	T	F	T	T
F	T	T	T	T	T
T	F	T	F	F	T
F	F	T	T	T	T
T	T	F	F	T	F
F	T	F	T	T	F
T	F	F	F	F	T
F	F	F	T	T	F

Ogni riga rappresenta un'assegnazione di valori di verità alle lettere p, q, r (cioè un'interpretazione per $\{p, q, r\}$) e il corrispondente valore di verità assunto dalle sottoformule di A.

Si noti che, se in una formula ci sono n lettere diverse, si hanno 2^n possibili assegnazioni di valori di verità alle lettere enunciative e quindi 2^n righe nella tavola di verità.

Una tavola di verità può essere abbreviata scrivendo solo la formula completa, mettendo i valori di verità delle lettere proposizionali sotto ciascuna di esse e scrivendo, passo per passo, il valore di verità di ogni sottoformula sotto il suo connettivo principale. Ad esempio, la tavola di verità abbreviata per la formula $(p \vee q) \rightarrow (p \wedge \neg q)$ è la seguente:

$(p$	$\vee$	$q)$	$\rightarrow$	$(p$	$\wedge$	$\neg$	$q)$
T	T	T	F	T	F	F	T
T	T	F	T	T	T	T	F
F	T	T	F	F	F	F	T
F	F	F	T	F	F	T	F

Si noti che, per determinare il valore di verità della formula completa, non sempre è necessario riempire tutti i dettagli di una data riga. Ad esempio, nell'ultima riga della tavola precedente, non appena è stato determinato che il valore di $(p \vee q)$ è F, si può concludere che il valore della formula intera è T, senza bisogno di calcolare il valore di $(p \wedge \neg q)$.

2.4 Connettivi logici e linguaggio naturale

La semantica dei connettivi logici non corrisponde sempre esattamente a quello delle parole corrispondenti utilizzate nel linguaggio naturale. Consideriamo quindi più in dettaglio il loro significato, in relazione al linguaggio naturale.

1. Una congiunzione ($\wedge$) è vera se e solo se entrambi gli argomenti sono veri. Sebbene la congiunzione "e" sia a volte utilizzata in italiano per dire "e poi", la congiunzione logica non ha alcun significato "temporale". Inoltre, $\wedge$ può corrispondere anche ad altre congiunzioni come "ma", "sebbene", ecc.

2. Una doppia implicazione ($\equiv$) è vera se e solo se i due argomenti hanno lo stesso valore di verità: IFF è l'uguaglianza sui booleani. In italiano si può dire "se e solo se" oppure "condizione necessaria e sufficiente". Ad esempio, "sarai promosso (p) *se e solo se* avrai scritto almeno 125 programmi corretti (c)" si può rappresentare con $p \equiv c$. "*Condizione necessaria e sufficiente* per passare l'esame (p) è studiare (s) e avere fortuna (f)" può essere rappresentata con $p \equiv (s \wedge f)$

3. La disgiunzione ($\vee$) è l'OR inclusivo (il *vel* latino): $A \vee B$ è vera anche quando sono veri tutti e due i disgiunti. L'OR esclusivo (XOR), la cui tavola di verità è la seguente (rappresentiamo XOR con il simbolo $\oplus$):

p	q	$p \oplus q$
T	T	F
T	F	T
F	T	T
F	F	F

può essere rappresentato da altre formule. Ad esempio da $(A \vee B) \wedge \neg(A \wedge B)$ (è vera almeno una tra A e B, ma non entrambe), o anche, in modo più compatto, da $\neg(A \equiv B)$ (si faccia la tavola di verità di questa formula e si controlli che si ottiene effettivamente quel che si vuole).

4. L'implicazione ($\rightarrow$) è l'*implicazione materiale*:

   $A \rightarrow B$ è falsa se e solo se A è vera e B è falsa.

   In altri termini, $\rightarrow$ corrisponde alla relazione $\leq$ sui booleani (quando si conviene che $F < T$).

   Anche se si legge come un "se ... allora", l'implicazione ha un significato che non corrisponde completamente al "se ... allora" del linguaggio naturale. Non c'è necessariamente un rapporto di causa-effetto tra l'antecedente A e il conseguente B (che non si può esprimere soltanto in termini di verità e falsità degli enuncitati).

   Il significato logico dell'implicazione è stato oggetto di discussione tra i filosofi e i logici, perché non sempre la riduzione del "se ... allora" in termini di verità e falsità può sembrare adeguata. Essa in ogni caso lo è in logica matematica, dato che tra i fatti matematici non esistono nessi causali o temporali. Per quel che riguarda l'uso della logica per rappresentare conoscenze diverse, si deve aver chiaro che l'unico caso in cui la formula $A \rightarrow B$ è falsa è quello in cui A è vera e B è falsa; dunque, se la formula $A \rightarrow B$ è vera, allora dalla verità di A si può senz'altro derivare che anche B è vera.

   Per convincerci comunque della ragionevolezza della definizione di $\rightarrow$ si consideri il caso seguente: Antonio dice "se piove, vengo a prenderti alla stazione". In quali di questi casi (interpretazioni) pensate che Antonio abbia detto una bugia (la sua affermazione è falsa)?

   (a) Piove, e Antonio va alla stazione

   (b) Piove, e Antonio non va alla stazione

   (c) Non piove, e Antonio va lo stesso

   (d) Non piove, e Antonio non va alla stazione

2.5 Soddisfacibilità, validità, equivalenza logica

La definizione che segue introduce alcune nozioni importanti.

Definizione 4 *Siano A e B formule nel linguaggio $Prop[P]$.*

1. *A è una* **tautologia** *o una formula* **logicamente valida** *sse per ogni interpretazione $\mathcal{M}$ di P, $\mathcal{M} \models A$ (A è vera in ogni interpretazione). Se A è una tautologia, si scrive $\models A$.*

2. *A è* **inconsistente** *(o è una* **contraddizione** *o è* insoddisfacibile*) sse per ogni interpretazione $\mathcal{M}$ di P, $\mathcal{M} \not\models A$ (A è falsa in ogni interpretazione).*

3. *A è* **soddisfacibile** *se esiste un'interpretazione $\mathcal{M}$ tale che $\mathcal{M} \models A$ (A è vera in almeno un'interpretazione).*

4. *Se $A \equiv B$ è una tautologia, allora A e B sono* **logicamente equivalenti**. *In tal caso si scrive $A \leftrightarrow B$.*

Si noti che una formula A è logicamente valida se e solo se $\neg A$ è una contraddizione, e A non è valida se e solo se $\neg A$ è soddisfacibile.

Consideriamo il linguaggio $P = \{p, q\}$ e la formula $A = (p \rightarrow q) \vee q$. A è soddisfacibile, perché esiste un'assegnazione che la soddisfa; ad esempio, $\mathcal{M}$ tale che $\mathcal{M}(p) = F$ e $\mathcal{M}(q) = F$ è un modello di A. Tuttavia A non è valida, infatti, per $\mathcal{M}'$ è tale che $\mathcal{M}'(p) = T$ e $\mathcal{M}'(q) = F$, si ha $\mathcal{M}' \not\models A$ ($\mathcal{M}'$ ̀dunque un contromodello di A).

Per dimostrare che una formula non è valida, è sufficiente trovare un suo contromodello. Ciò si può fare costruendo la tavola di verità della formula e controllando l'esistenza di almeno una riga in cui la formula è falsa, oppure mediante la ricerca diretta di un contromodello. Illustriamo questo secondo metodo mediante un esempio. Sia $A = (p \vee q) \rightarrow (p \rightarrow q)$ e cerchiamo di definire un'assegnazione $\mathcal{M} : \{p, q\} \rightarrow Bool$ tale che $\mathcal{M} \not\models A$. Ora, per avere $\mathcal{M} \not\models A$, si deve avere $\mathcal{M} \models p \vee q$ e $\mathcal{M} \not\models p \rightarrow q$; questo è infatti l'unico caso in cui un'implicazione è falsa. Per avere $\mathcal{M} \not\models p \rightarrow q$, di nuovo, deve essere $\mathcal{M} \models p$ e $\mathcal{M} \not\models q$, quindi $\mathcal{M}(p) = T$ e $\mathcal{M}(q) = F$. Verifichiamo allora che $\mathcal{M} \models p \vee q$; questo segue direttamente da $\mathcal{M} \models p$. Quindi $\mathcal{M}$ è un contromodello di A.

Per dimostrare invece che una formula è una tautologia, si possono controllare tutte le assegnazioni di valori di verità alle sue lettere proposizionali e verificare che la formula è vera in ogni interpretazione. Questo controllo può essere effettuato costruendo la tavola di verità della formula.

Un metodo alternativo per dimostrare la validità di una formula ricorre al ragionamento per assurdo: si dimostra l'impossibilità che la formula abbia un contromodello. Ad esempio, per dimostrare che la formula $\neg(p \vee \neg q) \rightarrow (\neg p \wedge q)$ è valida, assumiamo che esista un'interpretazione $\mathcal{M}$ in cui essa è falsa. Perché tale formula sia falsa, si deve avere $\mathcal{M} \models \neg(p \vee \neg q)$ e $\mathcal{M} \not\models \neg p \wedge q$. Da una parte, perché $\mathcal{M} \models \neg(p \vee \neg q)$, deve essere $\mathcal{M} \not\models p \vee \neg q$, dunque $\mathcal{M} \not\models p$, cioè (1) $\mathcal{M}(p) = F$, e $\mathcal{M} \not\models \neg q$, cioè (2) $\mathcal{M}(q) = T$. D'altro canto, se deve essere $\mathcal{M} \not\models \neg p \wedge q$, si deve avere $\mathcal{M} \not\models \neg p$, cioè $\mathcal{M}(p) = T$, oppure $\mathcal{M}(q) = F$. Ma il primo caso è impossibile per (1), il secondo per (2). Quindi non esiste alcuna assegnazione $\mathcal{M}$ tale che $\mathcal{M} \not\models \neg(p \vee \neg q) \rightarrow (\neg p \wedge q)$.

Lo stesso metodo si può adottare quando non si sa se una formula è valida o no: si cerca di costruirne un contromodello e, se il tentativo ha successo, allora la formula non è valida, altrimenti se si dimostra l'impossibilità di costruirne un contromodello, allora la formula è valida.

Ecco alcune tautologie importanti:

1. Identità: $A \rightarrow A$

2. Affermazione del conseguente: $A \rightarrow (B \rightarrow A)$

3. Negazione dell'antecedente: $\neg A \rightarrow (A \rightarrow B)$

4. *Ex falso quodlibet*: $\bot \rightarrow B$

5. Terzo escluso: $A \vee \neg A$

6. Non contraddizione: $\neg(A \wedge \neg A)$

7. Riduzione all'assurdo: $(A{\rightarrow}B){\rightarrow}((A{\rightarrow}\neg B){\rightarrow}\neg A)$
8. Legge di Pierce: $((A{\rightarrow}B){\rightarrow}A){\rightarrow}A$

Ed alcune equivalenze logiche importanti:

1. Leggi di De Morgan:

$$\begin{array}{lcl} \neg(A \vee B) & \leftrightarrow & \neg A \wedge \neg B \\ \neg(A \wedge B) & \leftrightarrow & \neg A \vee \neg B \end{array}$$

2. Leggi distributive:

$$\begin{array}{lcl} (A \vee (B \wedge C)) & \leftrightarrow & ((A \vee B) \wedge (A \vee C)) \\ (A \wedge (B \vee C)) & \leftrightarrow & ((A \wedge B) \vee (A \wedge C)) \end{array}$$

3. Commutatività e associatività di $\wedge$ e $\vee$:

$$\begin{array}{rcll} A \wedge B & \leftrightarrow & B \wedge A & \text{commutatività di } \wedge \\ A \vee B & \leftrightarrow & B \vee A & \text{commutatività di } \vee \\ A \wedge (B \wedge C) & \leftrightarrow & (A \wedge B) \wedge C & \text{associatività di } \wedge \\ A \vee (B \vee C) & \leftrightarrow & (A \vee B) \vee C & \text{associatività di } \vee \end{array}$$

4. Semplificazioni:

$$\begin{array}{lcl} A \wedge \neg A & \leftrightarrow & \bot \\ A \vee \neg A & \leftrightarrow & \top \\ A \wedge \top & \leftrightarrow & A \\ A \vee \bot & \leftrightarrow & A \\ A \wedge \bot & \leftrightarrow & \bot \\ A \vee \top & \leftrightarrow & \top \end{array}$$

5. Doppia negazione: $A \leftrightarrow \neg\neg A$
6. Leggi di assorbimento:

$$\begin{array}{lcl} A \vee (A \wedge B) & \leftrightarrow & A \\ A \wedge (A \vee B) & \leftrightarrow & A \end{array}$$

7. Definibilità di $\equiv$: $A \equiv B \leftrightarrow (A{\rightarrow}B) \wedge (B{\rightarrow}A)$
8. Interdefinibilità dei connettivi logici $\rightarrow, \wedge, \vee$:

$$\begin{array}{ccccc} A{\rightarrow}B & \leftrightarrow & \neg A \vee B & \leftrightarrow & \neg(A \wedge \neg B) \\ \neg(A{\rightarrow}B) & \leftrightarrow & \neg(\neg A \vee B) & \leftrightarrow & A \wedge \neg B \\ A \wedge B & \leftrightarrow & \neg(\neg A \vee \neg B) & \leftrightarrow & \neg(A{\rightarrow}\neg B) \\ A \vee B & \leftrightarrow & \neg(\neg A \wedge \neg B) & \leftrightarrow & \neg A{\rightarrow}B \end{array}$$

9. Contrapposizione: $(A{\rightarrow}B) \leftrightarrow (\neg B{\rightarrow}\neg A)$

2.6 Conseguenza logica

La definizione che segue introduce l'importante nozione di *conseguenza logica*, che è quella che occorre per formalizzare il concetto di "ragionamento corretto".

Definizione 5 *Sia A una formula e S un insieme di formule nel linguaggio $Prop[P]$. Se A è vera in ogni interpretazione in cui sono vere tutte le formule di S, allora si dice che A è una* **conseguenza logica** *di S (o che S* **implica logicamente** *A), e si scrive $S \models A$. In altri termini, $S \models A$ sse per ogni interpretazione $\mathcal{M}$, se $\mathcal{M} \models S$, allora $\mathcal{M} \models A$.*

D'ora in avanti useremo la notazione $B_1, ..., B_n \models A$, anziché $\{B_1, ..., B_n\} \models A$. Se A non è una conseguenza logica di $\{B_1, ..., B_n\}$, scriviamo $B_1, ..., B_n \not\models A$.

La nozione di conseguenza logica, come abbiamo detto, formalizza quella di "ragionamento corretto". Consideriamo ad esempio il ragionamento già visto nel paragrafo 1:

a) Se nevica, la temperatura è di 0^oC.
   Nevica.
   Quindi la temperatura è di 0^oC.

La forma di tale ragionamento è:

$$\frac{A{\rightarrow}B \qquad A}{B}$$

Il ragionamento è corretto perché la conclusione B è una conseguenza logica di $\{A{\rightarrow}B, A\}$:

$$A{\rightarrow}B, A \models B$$

Per dimostrarlo, possiamo costruire la tavola di verità delle premesse e della conclusione e verificando che il valore della conclusione è T in ogni riga in cui sono vere tutte le premesse:

A	B	$A{\rightarrow}B$	A	B
T	T	T	T	T
T	F	F	T	F
F	T	T	F	T
F	F	T	F	F

L'unica riga in cui sono vere entrambe le premesse è la prima, e in tale riga è vera anche la conclusione. Quindi $A{\rightarrow}B, A \models B$.

Il metodo delle tavole di verità diventa però piuttosto complesso al crescere del numero di variabili proposizionali. In alternativa, possiamo ricorrere ad un ragionamento semantico, dimostrando direttamente che ogni modello delle premesse è necessariamente un modello della conclusione. Nel nostro caso, assumiamo che $\mathcal{M}$ sia una qualsiasi interpretazione tale che $\mathcal{M} \models A{\rightarrow}B$ e $\mathcal{M} \models A$. Per la definizione di $\models$ (Definizione 3), se $\mathcal{M} \models A{\rightarrow}B$, allora $\mathcal{M} \not\models A$ oppure $\mathcal{M} \models B$. Dato che, per ipotesi, $\mathcal{M} \models A$, si deve avere necessariamente che $\mathcal{M} \models B$. Quindi ogni modello di $\{A{\rightarrow}B, A\}$ è anche un modello di B.

In generale, dunque, per dimostrare che un ragionamento è corretto si determina innanzitutto un linguaggio in cui rappresentarlo e si "formalizza" il ragionamento, cioè se ne determina la forma. Infine, si dimostra che la conclusione del ragionamento è una conseguenza logica delle premesse.

Per dimostrare invece che un ragionamento non è corretto, dopo averlo formalizzato, si dimostra che la conclusione non è una conseguenza logica delle premesse, trovando un'interpretazione che è un modello delle premesse e un contromodello della conclusione.

Consideriamo ad esempio il ragionamento seguente:

> Se piove, prendo l'ombrello. Prendo l'ombrello. Dunque piove

Se rappresentiamo piove con la lettera proposizionale A e prendo l'ombrello con B, la forma di tale ragionamento è:

$$\frac{A \to B \qquad B}{A}$$

Il ragionamento non è corretto: infatti se $\mathcal{M}(A) = F$ e $\mathcal{M}(B) = T$, $\mathcal{M} \models \{A \to B, B\}$ e $\mathcal{M} \not\models A$. Dunque $A \to B, B \not\models A$.

Consideriamo ancora un altro esempio, tratto da [5].

> È stato compiuto un furto. Si sa che:
>
> 1. sono implicati tre uomini: Antonio, Biagio, Carlo;
> 2. i ladri sono fuggiti con un furgone;
> 3. Carlo non lavora mai senza la complicità di Antonio;
> 4. Biagio non sa guidare.
>
> Il problema è: Antonio è colpevole?

Iniziamo con la formalizzazione delle informazioni in un linguaggio proposizionale. Utilizziamo le variabili proposizionali A, B, C, con il seguente significato:

A: Antonio è colpevole

B: Biagio è colpevole

C: Carlo è colpevole

L'informazione (1) si può riformulare con "Almeno uno tra Antonio, Biagio e Carlo è colpevole", quindi, nel nostro linguaggio:

F_1: $A \vee B \vee C$

L'informazione (3) è rappresentabile da:

F_2: $C \to A$

(se Carlo è colpevole, allora Antonio è suo complice). Infine, la (2) e la (4) assieme si possono riscrivere in "se Biagio è colpevole, allora anche qualcun altro è colpevole, dato che lui non poteva guidare il furgone"; quindi:

F_3: $B \to (A \vee C)$

Il problema è dunque quello di determinare se

$$F_1, F_2, F_3 \models A$$

Ovviamente, possiamo fare la tavola di verità per le formule F_1, F_2, F_3 e verificare se A è vera in ogni riga in cui esse sono tutte vere. Ma non è questo il modo naturale di ragionare.

Possiamo dedurre la colpevolezza di Antonio ragionando per assurdo: sia $\mathcal{M}$ una qualsiasi interpretazione in cui sono vere F_1, F_2 e F_3, cioè

$$(1)\ \mathcal{M} \models A \vee B \vee C, \quad (2)\ \mathcal{M} \models C \rightarrow A, \quad (3)\ \mathcal{M} \models B \rightarrow (A \vee C)$$

e tuttavia

$$(4)\ \mathcal{M} \not\models A$$

Allora, da (2) e (4) segue

$$(5)\ \mathcal{M} \not\models C$$

per la semantica di $\rightarrow$. Per il significato di $\vee$, da (4) e (5) segue

$$(6)\ \mathcal{M} \not\models A \vee C$$

e, per il significato di $\rightarrow$, da (3) e (6) segue

$$(7)\ \mathcal{M} \not\models B$$

Ma allora, da (4), (5) e (7) segue

$$(8)\ \mathcal{M} \not\models A \vee B \vee C$$

contraddicendo l'ipotesi (1). Poiché dunque assumere l'innocenza di Antonio porta a una contraddizione con le informazioni date, se ne deduce che Antonio è colpevole.

Lo stesso risultato si può seguire ragionando per casi dall'ipotesi che $A \vee B \vee C$ sia vera in $\mathcal{M}$. Secondo questa ipotesi, i casi sono tre: o è colpevole Antonio, oppure è colpevole Biagio oppure è colpevole Carlo. Dimostriamo allora che in tutti e tre i casi Antonio è colpevole. Ne potremo concludere che Antonio è comunque colpevole.

caso 1. Se è vero A, banalmente ne segue che è vero A.

caso 2. Se è vero B, allora, per l'ipotesi (3), è vero anche $A \vee C$; di nuovo, ragioniamo per casi:

sottocaso 2a. se è vero A, allora è vero A;

sottocaso 2b. se è vero C, poiché è vero anche $C \rightarrow A$ (2), allora è vero A.

Quindi possiamo concludere che, nel caso 2, A è comunque vero.

caso 3. Se è vero C, poiché è vero anche $C \rightarrow A$, allora è vero A.

In ogni possibile caso si ha dunque che A è vero. Se ne conclude allora che A è comunque vero: Antonio è colpevole.

2.7 L'isola dei cavalieri e furfanti

In questo paragrafo vediamo come utilizzare la logica dei predicati per risolvere alcuni dei quiz nel libro di R. Smullyan [5], ambientati nell'isola dei cavalieri e furfanti. Quest'isola è abitata da due tipi di persone: i cavalieri, che dicono sempre la verità, e i furfanti, che dicono sempre bugie. Noi siamo visitatori dell'isola e incontriamo due persone, che chiamiamo A e B. A dice: "Io sono un furfante oppure B è un cavaliere". Cosa sono A e B?

Vediamo come rappresentare adeguatamente questo problema in logica proposizionale. Determiniamo innanzitutto un linguaggio appropriato. Utilizziamo a tale scopo due lettere proposizionali: A, che rappresenta l'enunciato "A è un cavaliere", e B, che rappresenta l'enunciato "B è un cavaliere". Poiché nell'isola, eccetto noi, ci sono solo cavalieri e furfanti, l'enunciato "A è un furfante" sarà rappresentato da $\neg A$, e analogamente $\neg B$ rappresenta "B è un furfante".

L'affermazione di A è allora rappresentata dalla formula $\neg A \vee B$. Ma noi non sappiamo se questo è vero o falso, perché A potrebbe essere un furfante. Sappiamo solo che se A è un cavaliere allora $\neg A \vee B$ è vera, e se A è un furfante allora $\neg A \vee B$ è falsa. La nostra conoscenza è allora rappresentata dalle due formule:

1. $A \rightarrow (\neg A \vee B)$
2. $\neg A \rightarrow \neg(\neg A \vee B)$

Mediante le tavole di verità possiamo determinare quali sono le interpretazioni di A e B in cui le due formule 1 e 2 sono entrambe vere, e rispondere dunque alla domanda "cosa sono A e B?":

A	B	$A \rightarrow (\neg A \vee B)$	$\neg A \rightarrow \neg(\neg A \vee B)$
T	T	T	T
T	F	F	T
F	T	T	F
F	F	T	F

L'unica interpetazione in cui 1 e 2 sono entrambe vere è quella in cui A e B sono entrambe vere: A e B sono entrambi cavalieri.

In altri termini, A e B sono conseguenze logiche della nostra conoscenza:

$$A \rightarrow (\neg A \vee B), \neg A \rightarrow \neg(\neg A \vee B) \models A$$

$$A \rightarrow (\neg A \vee B), \neg A \rightarrow \neg(\neg A \vee B) \models B$$

Anziché utilizzare le tavole di verità per risolvere problemi di questo tipo, possiamo utilizzare un ragionamento semantico, come nell'esempio del paragrafo precedente. Consideriamo lo stesso problema visto sopra. Per determinare se A è una conseguenza logica di 1 e 2, dimostriamo che tutti i modelli di 1 e 2 sono modelli di A. Sia $\mathcal{M}$ una qualunque interpretazione che rende vere 1 e 2 e supponiamo (per assurdo) che $\mathcal{M} \not\models A$.

$$\begin{array}{llll} \mathcal{M} \not\models A & \Rightarrow & \mathcal{M} \models \neg A & \text{def. di } \neg \\ & \Rightarrow & \mathcal{M} \models \neg(\neg A \vee B) & \text{2 e def. di } \rightarrow \\ & \Rightarrow & \mathcal{M} \not\models \neg A \vee B & \\ & \Rightarrow & \mathcal{M} \not\models \neg A & \text{def. di } \vee \\ & \Rightarrow & \mathcal{M} \models A : \text{contraddizione} & \end{array}$$

Se fosse $\mathcal{M} \not\models A$, si avrebbe dunque contemporaneamente $\mathcal{M} \models A$ e $\mathcal{M} \not\models A$. Quindi $\mathcal{M} \not\models A$ è assurdo e deve essere $\mathcal{M} \models A$.

Con lo stesso metodo si può determinare che B è una conseguenza logica di 1 e 2.

2.8 Forme normali disgiuntive e congiuntive

Ogni formula è equivalente ad altre formule che hanno una forma particolare, o *normale*.

Definizione 6 *Un* **letterale** *è una lettera proposizionale o la negazione di una lettera proposizionale. Una formula è in* **forma normale disgiuntiva** *(DNF:* Disjunctive Normal Form*) se è una disgiunzione di congiunzioni di letterali. È in* **forma normale congiuntiva** *(CND:* Conjunctive Normal Form*) se è una congiunzione di disgiunzioni di letterali.*

Teorema 1 *Ogni formula è equivalente ad una formula in forma normale disgiuntiva ed è equivalente ad una formula in forma normale congiuntiva.*

La trasformazione di una formula in DNF o CNF può essere eseguita mediante trasformazioni successive, applicando le seguenti regole che trasformano una formula in una formula equivalente:

1. Eliminare le implicazioni e le doppie implicazioni mediante le seguenti trasformazioni:

$$\begin{array}{rcl} A \equiv B & \Longrightarrow & (A \wedge B) \vee (\neg A \wedge \neg B) \\ A \rightarrow B & \Longrightarrow & \neg A \vee B \\ \neg(A \rightarrow B) & \Longrightarrow & A \wedge \neg B \\ \neg(A \equiv B) & \Longrightarrow & (A \vee B) \wedge (\neg A \vee \neg B) \end{array}$$

2. Portare le negazioni sugli atomi, applicando le leggi di De Morgan e la legge di doppia negazione:

$$\begin{array}{rcl} \neg(A \vee B) & \Longrightarrow & (\neg A \wedge \neg B) \\ \neg(A \wedge B) & \Longrightarrow & (\neg A \vee \neg B) \\ \neg\neg A & \Longrightarrow & A \end{array}$$

   Si ottiene così una formula in *forma normale negativa*.

3. Applicare le leggi distributive:

$$\begin{array}{rcl} (A \vee (B \wedge C)) & \Longrightarrow & ((A \vee B) \wedge (A \vee C)) \\ (A \wedge (B \vee C)) & \Longrightarrow & ((A \wedge B) \vee (A \wedge C)) \end{array}$$

Ad esempio, possiamo trasformare in CNF la formula

$$\neg(\neg(p \vee \neg q) \rightarrow \neg(q \vee (r \wedge (s \rightarrow p))))$$

con i seguenti passaggi:

$$
\begin{array}{rl}
\multicolumn{2}{l}{\neg(\neg(p \vee \neg q) \rightarrow \neg(q \vee (r \wedge (s \rightarrow p))))} \\
\Rightarrow & \neg(p \vee \neg q) \wedge \neg\neg(q \vee (r \wedge (\neg s \vee p))) \\
\Rightarrow & \neg(p \vee \neg q) \wedge (q \vee (r \wedge (\neg s \vee p))) \\
\Rightarrow & (\neg p \wedge \neg\neg q) \wedge (q \vee (r \wedge (\neg s \vee p))) \\
\Rightarrow & (\neg p \wedge q) \wedge (q \vee (r \wedge (\neg s \vee p))) \\
\Rightarrow & (\neg p \wedge q) \wedge ((q \vee r) \wedge (q \vee (\neg s \vee p))) \\
= & \neg p \wedge q \wedge (q \vee r) \wedge (q \vee \neg s \vee p)
\end{array}
$$

Si noti che qualsiasi formula ha più di una forma normale congiuntiva o disgiuntiva, sintatticamente distinte anche se logicamente equivalenti.

2.9 Deduzione automatica

Un sistema di deduzione automatica costituisce un apparato per ragionare in modo meccanico. Per automatizzare il ragionamento logico è dunque necessario definire un metodo meccanico per eseguire dimostrazioni. Supponiamo infatti di avere una teoria logica T (un insieme di formule) che rappresenti le conoscenze su un determinato dominio. Come si è visto nel paragrafo 2.6, risolvere un problema nel dominio in oggetto si può spesso ridurre alla questione di determinare se una certa formula A è una conseguenza logica di un insieme dato di formule che descrive il dominio di interesse.

L'obiettivo della deduzione automatica è dunque quello di definire algoritmi che consentano di risolvere problemi della forma: dato un insieme S di formule e una formula A, è vero che $S \models A$?

La maggior parte dei metodi di dimostrazione automatica sono *metodi di refutazione*, che, sfruttano una relazione importante tra la nozione di conseguenza logica e quella di soddisfacibilità, come stabilito dal teorema seguente.

Teorema 2 *Sia S un insieme di formule e A una formula. $S \models A$ sse $S \cup \{\neg A\}$ è insoddisfacibile.*

Un metodo di refutazione dunque riconduce il problema di determinare se $S \models A$ al problema di determinare se $S \cup \{\neg A\}$ è insoddisfacibile. Un metodo di refutazione è dunque un metodo che consente di dimostrare l'insoddisfacibilità di insiemi di formule.

Nel caso della logica proposizionale, il problema dell'insoddisfacibilità di un insieme (finito) di formule è *decidibile*, cioè risolubile in modo meccanico. Il metodo delle tavole di verità è un esempio di procedimento automatizzabile per risolvere tale problema. Tuttavia, dal punto di vista computazionale, le tavole di verità sono un metodo piuttosto costoso: in ogni caso infatti (e non solo nel caso peggiore), la dimensione di una tavola di verità è esponenziale nella dimensione dell'insieme di formule in input. Sebbene il problema della soddisfacibilità di un insieme di formule proposizionale sia comunque inerentemente complesso (è un problema $\mathcal{NP}$-completo), esistono metodi migliori. Il prossimo paragrafo presenta uno di questi.

2.10 Il metodo dei tableaux per la logica proposizionale

In questa sezione presentiamo un semplice metodo di dimostrazione automatica per la logica proposizionale, il metodo dei *tableaux semantici*. Tale metodo

consiste essenzialmente in un sistema di ricerca esaustiva di modelli per una formula o insieme di formule.

Assumiamo qui che $\equiv$ sia un simbolo definito, quindi i simboli logici del linguaggio sono $\neg, \wedge, \vee, \rightarrow$.

Un tableau[1] è un albero, rappresentato con la radice in alto, i cui nodi sono etichettati da formule. Un tableau per una formula F, viene inizializzato con un unico nodo, etichettato proprio dalla formula F. Questo è il "tableau iniziale" per F. La costruzione del tableau procede poi aggiungendo nodi, in accordo con determinate *regole di espansione*, che "analizzano il significato" delle formule. La costruzione di un tableau con radice F essenzialmente consiste nella ricerca esaustiva di un modello per F: se un tale modello viene trovato, la formula F è soddisfacibile, altrimenti si è dimostrato che non esistono modelli di F (e quindi la negazione di F è valida).

Le regole di espansione vengono rappresentate in una delle seguenti forme:

$$\frac{A}{A_0} \qquad \frac{A}{\begin{array}{c}A_0\\A_1\end{array}} \qquad \frac{B}{B_0 \quad B_1}$$

L'applicazione di una regola della prima forma aggiunge la formula A_0 in fondo a ciascun ramo del tableau che contiene A. L'applicazione di una regola della seconda forma aggiunge le formule A_0 e A_1 in fondo a ciascun ramo che contiene la formula A. L'applicazione di regole della terza forma crea invece una ramificazione nell'albero: ciascun ramo che contiene la formula B viene espanso aggiungendo i due nodi B_0 e B_1 come figli dell'ultimo nodo del ramo, provocando la biforcazione del ramo stesso.

Le regole della prima e seconda forma sono chiamate α-regole, quelle della terza forma β-regole: Le regole sono le seguenti:

α-regole:	$\dfrac{A \wedge B}{\begin{array}{c}A\\B\end{array}}$	$\dfrac{\neg(A \vee B)}{\begin{array}{c}\neg A\\\neg B\end{array}}$	$\dfrac{\neg(A \rightarrow B)}{\begin{array}{c}A\\\neg B\end{array}}$	$\dfrac{\neg\neg A}{A}$
β-regole:	$\dfrac{A \vee B}{A \quad B}$	$\dfrac{\neg(A \wedge B)}{\neg A \quad \neg B}$	$\dfrac{A \rightarrow B}{\neg A \quad B}$	

L'intuizione dietro queste regole è la seguente. La prima α-regola (in alto a sinistra) stabilisce che qualsiasi interpretazione è un modello di $A \wedge B$ se e solo se è anche un modello di A e un modello di B. La seconda α-regola che qualsiasi interpretazione è un modello di $\neg(A \vee B)$ se e solo se è anche un modello di $\neg A$ e un modello di $\neg B$. La prima β-regola stabilisce che una qualsiasi interpretazione è un modello di $A \vee B$ se e solo se è un modello di A *oppure* un modello di B.

È conveniente, per compattezza, utilizzare una "notazione uniforme" per le regole di espansione, che utilizza la nozione di α e β-formule, con le rispettive α_0 e α_1, o β_0 e β_1, come descritto nelle seguenti tabelle:

[1]Al singolare: "tableau"; al plurale: "tableaux".

α	α_0	α_1
$A \wedge B$	A	B
$\neg(A \vee B)$	$\neg A$	$\neg B$
$\neg(A \rightarrow B)$	A	$\neg B$
$\neg\neg A$	A	

β	β_0	β_1
$A \vee B$	A	B
$\neg(A \wedge B)$	$\neg A$	$\neg B$
$A \rightarrow B$	$\neg A$	B

Le regole di espansione sono allora semplicemente:

$$\alpha\text{-regola} : \quad \frac{\alpha}{\begin{array}{c}\alpha_0 \\ (\alpha_1)\end{array}} \qquad\qquad \beta\text{-regola} : \frac{\beta}{\beta_0 \qquad \beta_1}$$

Ogni ramo del tableau rappresenta un possibile modello. Se un ramo contiene due formule della forma A e $\neg A$, oppure se contiene $\bot$, esso si dice *chiuso*. Quando, durante la costruzione di un tableau, si genera un ramo chiuso, questo, normalmente, non viene ulteriormente espanso: infatti tutti i rami che sarebbero generati come sua estensione sono rami chiusi, che non possono rappresentare alcuna interpretazione, in quanto A e $\neg A$ dovrebbero essere contemporaneamente vere. Un ramo che non è chiuso si dice *aperto*.

Se è possibile proseguire la costruzione del tableau fino ad ottenere tutti rami chiusi, ottenendo cioè un *tableau chiuso*, allora la formula di partenza non ha modelli, è cioè insoddisfacibile. Altrimenti, se, quando tutte le formule del tableau sono state espanse (ad esse è stata cioè applicata una delle regole applicabili), vi sono ancora rami non chiusi (aperti), allora ciascuno di essi rappresenta un possibile modello (o, in generale, un insieme di modelli) della formula iniziale.

Si noti che, in logica proposizionale, la costruzione di un tableau termina sempre, in quanto le regole di espansione generano formule più semplici della formula espansa.

Come primo esempio, consideriamo l'albero della Figura 1. Esso è un tableau per la formula $(p \wedge q \rightarrow r) \wedge (\neg p \rightarrow s) \wedge q$. I primi nodi, nel ramo unico iniziale, sono ottenuti applicando la regola per formule della forma $A \wedge B$. L'espansione del nodo $p \wedge q \rightarrow r$ provoca la prima ramificazione, con $\neg(p \wedge q)$ a sinistra e r a destra. L'espansione di $\neg p \rightarrow s$ aggiunge un'ulteriore ramificazione alla fine di ciascun ramo, con $\neg\neg p$ a sinistra e s a destra. I due nodi $\neg\neg p$ vengono entrambi espansi aggiungendo p in fondo al rispettivo ramo. Infine, l'espansione di $\neg(p \wedge q)$ aggiunge le ultime ramificazioni sulla sinistra, con $\neg p$ e $\neg q$ in ciascun ramo. I rami marcati con una croce, $\times$, sono chiusi: il primo infatti contiene p e $\neg p$, il secondo e il terzo q e $\neg q$. Vi sono tre rami aperti nel tableau, il primo contenente i letterali $q, s, \neg p$, il secondo q, r, p, il terzo q, r, s. Il tableau non può essere ulteriormente espanso, quindi la formula $(p \wedge q \rightarrow r) \wedge ((\neg p \rightarrow s) \wedge q)$ è soddisfacibile e i tre rami aperti rappresentano modelli per essa. Ad esempio, il primo ramo rappresenta l'insieme delle interpretazioni di p, q, r, s in cui q e s sono vere e p è falsa; ciascuna di tali interpretazioni è un modello di $(p \wedge q \rightarrow r) \wedge ((\neg p \rightarrow s) \wedge q)$.

Se si vuole controllare la validità di una formula A, si inizializza il tableau con la formula $\neg A$: se la ricerca sistematica di un modello di $\neg A$ fallisce, vuol dire che $\neg A$ è insoddisfacibile, dunque A è valida.

Il metodo dei tableaux si può utilizzare anche per dimostrare che una data formula A è una conseguenza logica di un insieme di formule S: il tableau viene

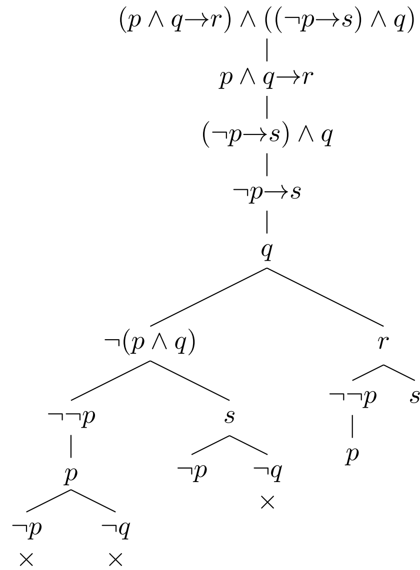


Figura 1: Un tableau per la formula $(p \wedge q \to r) \wedge (\neg p \to s) \wedge q$

allora inizializzato con l'insieme di formule $S \cup \{\neg A\}$. Ciò significa che il tableau iniziale è un albero costituito da un unico ramo, in cui occorrono tutte le formule di $S \cup \{\neg A\}$. Se $S = \{G_1, ..., G_n\}$, tale tableau ha la forma riportata nella figura 2. In generale, se T è un tableau per un insieme di formule S, e T^* si ottiene da T applicando una regola di espansione, allora T^* è ancora un tableaux per S.

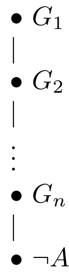


Figura 2: Tableau iniziale per l'insieme di formule $\{G_1, ..., G_n, \neg A\}$.

Ad esempio, allo scopo di dimostrare che $A \wedge B \to C, \neg A \to D \models B \to (C \vee D)$, costruiamo il tableau iniziale riportato in Figura 3, a sinistra. Se in seguito viene espansa la formula $\neg(B \to (C \vee D))$, otteniamo il tableau riportato al centro, dove segnamo con un $\surd$ la formula espansa. Espandendo poi l'ultima formula di tale tableau, si ottiene il tableau riportato a destra, sempre in Figura 3.

In seguito, l'espansione del primo nodo genera il tableau riportato nella Figura 4, a sinistra, e la successiva espansione del nodo $\neg A \to D$ genera il tableau riportato a destra, sempre in Figura 4.

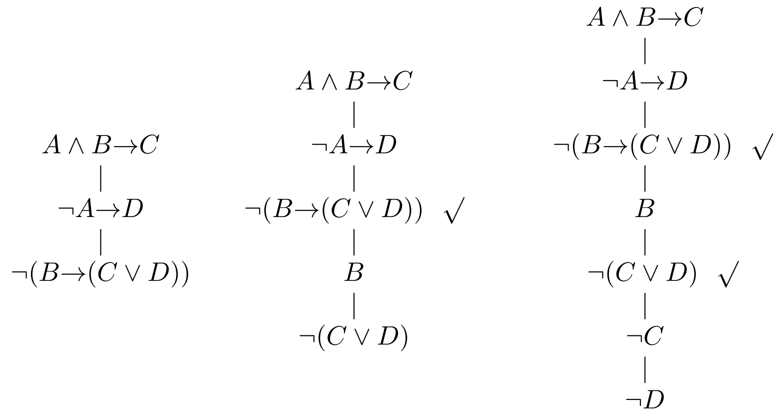


Figura 3: Tableaux per $\{A \wedge B \to C,\ \neg A \to D,\ \neg(B \to (C \vee D))\}$.

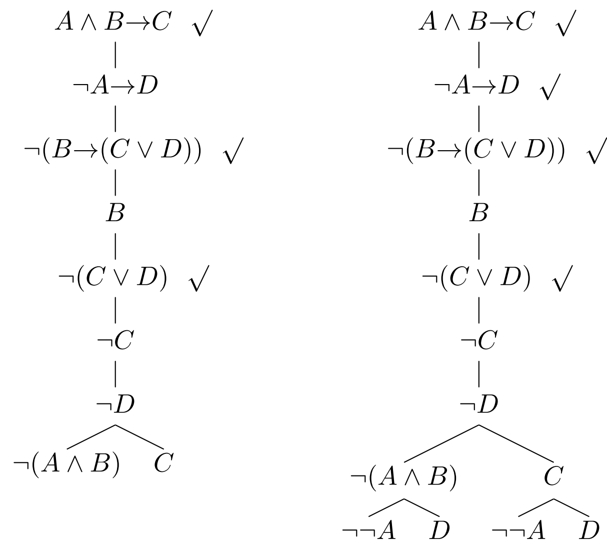


Figura 4: Tableaux per $\{A \wedge B \to C,\ \neg A \to D,\ \neg(B \to (C \vee D))\}$.

Infine, l'espansione dei due nodi etichettati con $\neg\neg A$ e quella di $\neg(A \wedge B)$ genera il tableau chiuso riportato in Figura 5. Il tableau è completamente espanso: i soli nodi non espansi sono etichettati da letterali. Il primo ramo (a partire da sinistra) è chiuso perché contiene A e $\neg A$, il secondo contiene B e $\neg B$, il terzo contiene D e $\neg D$, gli ultimi due contengono C e $\neg C$.

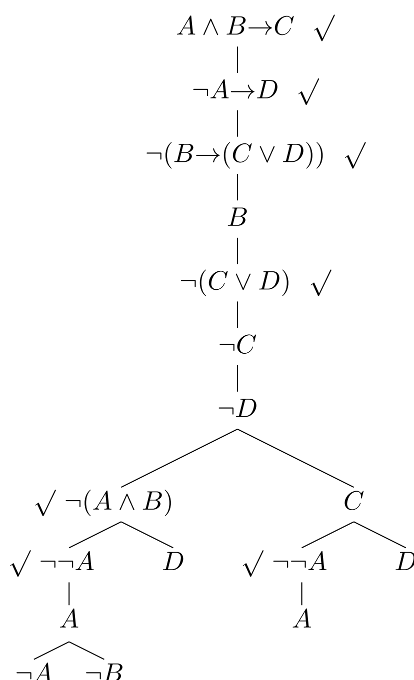


Figura 5: Una dimostrazione mediante tableaux di $A \wedge B \rightarrow C, \neg A \rightarrow D \vdash B \rightarrow (C \vee D)$.

Recapitolando:

- Un ramo di un tableau è **chiuso** se esso contiene sia A che $\neg A$ per qualche formula A, oppure contiene $\bot$.
- Un tableau è chiuso se tutti i suoi rami sono chiusi.
- Una dimostrazione mediante tableau di $S \vdash A$ è un tableau chiuso per $S \cup \{\neg A\}$.
- Un tableau si dice *completo* se ogni nodo che possa essere espanso è stato espanso almeno una volta.

Il metodo dei tableaux per la logica proposizionale è corretto e completo:

$S \models A$ sse esiste una dimostrazione mediante tableaux di $S \vdash A$.

Inoltre, nei tableaux proposizionali:

1. Se S è insoddisfacibile, è sufficiente espandere al massimo una volta ogni formula di un tableau per S per ottenere un tableau chiuso.

2. L'ordine di espansione delle formule è irrilevante per la completezza (*invertibilità delle regole di espansione*).

3. Se A è una formula, T è un tableau completo per A, $\mathcal{B}_1, ..., \mathcal{B}_n$ sono tutti i rami aperti di T, e se $\mathcal{C}(\mathcal{B}_i)$ è la congiunzione dei letterali in $\mathcal{B}_i$, allora:

$$A \leftrightarrow \mathcal{C}(\mathcal{B}_1) \vee .... \vee \mathcal{C}(\mathcal{B}_n)$$

Come esempio relativo al precedente punto 3, si consideri il tableau riportato in Figura 6. Il ramo più a destra è chiuso, quindi i rami aperti sono i primi tre, $\mathcal{B}_1, \mathcal{B}_2$ e $\mathcal{B}_3$, contenenti, rispettivamente i letterali $\{p, \neg q\}$, $\{p, \neg r\}$ e $\{p, q, \neg s\}$. Quindi $\mathcal{C}(\mathcal{B}_1) = p \wedge \neg q$, $\mathcal{C}(\mathcal{B}_2) = p \wedge \neg r$ e $\mathcal{C}(\mathcal{B}_3) = p \wedge q \wedge \neg s$. Quindi $\neg((p{\rightarrow}(q \wedge r)) \wedge (p{\rightarrow}(q{\rightarrow}(s \wedge p))))$ è logicamente equivalente a $(p \wedge \neg q) \vee (p \wedge \neg r) \vee (p \wedge q \wedge \neg s)$.

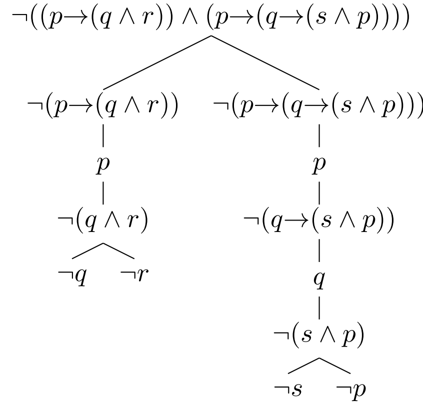


Figura 6: Uso dei tableaux per la trasformazione in forma normale

Come si vede, dunque, il metodo dei tableaux si può utilizzare anche per trasformare le formule proposizionali in forma normale disgiuntiva. Come ulteriore esempio, dal tableau riportato in Figura 1 si ottiene la formula $(q \wedge s \wedge \neg p) \vee (q \wedge r \wedge p) \vee (q \wedge r \wedge s)$, che è logicamente equivalente alla formula iniziale $(p \wedge q{\rightarrow}r) \wedge (\neg p{\rightarrow}s) \wedge q$.

In modo "duale" si può ottenere la forma normale congiuntiva di una formula F: si inizializza il tableau con $\neg F$, si espande il tableau completamente, e si identificano tutti i rami aperti. Per ciascuno di essi si costruisce l'insieme costituito dai *complementi* dei letterali nel ramo (il complemento di un letterale p è $\neg p$ e il complemento di $\neg p$ è p). In tal modo si ottengono insiemi $S_1, ..., S_k$ di letterali. Una forma normale congiuntiva equivalente a F è allora la congiunzione di tutte le disgiunzioni $\bigvee_{\ell \in S_i} \ell$.

In altri termini, per trasformare A in forma normale congiuntiva:

- si costruisce un tableau completo per $\neg A$
- si eliminano i rami chiusi
- si costruisce la congiunzione delle disgiunzioni dei complementi dei letterali nei rami aperti.

Ad esempio, per trasformare in CNF la formula $\neg(p\to(q \wedge r) \vee (q\to s))$, costruiamo innanzitutto un tableau completo per $\neg\neg(p\to(q \wedge r) \vee (q\to s))$, come riportato in Figura 7. Tutti i rami sono aperti e contengono i letterali $\{\neg p\}$ $\{q, r\}$, $\{\neg q\}$, $\{s\}$. I corrispondenti insiemi con i complementi dei letterali sono $\{p\}$ $\{\neg q, \neg r\}$, $\{q\}$, $\{\neg s\}$. Si ottiene quindi la formula in CNF $p\wedge(\neg q\vee\neg r)\wedge q\wedge\neg s$.

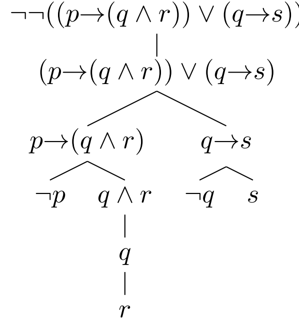


Figura 7: Tableau per $\neg\neg((p\to(q \wedge r)) \vee (q\to s))$

Come ulteriore esempio, dal tableau della figura 6 si ottengono, dai tre rami aperti, i letterali (già complementati) $\{\neg p, q\}$, $\{\neg p, r\}$ e $\{\neg p, \neg q, s\}$. Quindi $(p\to(q\wedge r)) \wedge (p\to(q\to(s \wedge p)))$ è logicamente equivalente a $(\neg p \vee q) \wedge (\neg p \vee r) \wedge (\neg p \vee \neg q \vee s)$.

Il fatto che la CNF di una formula si possa ottenere in questo modo deriva dalla "dualità" tra CNF e DNF: il tableau della figura 7 può servire a costruire, secondo il metodo prima visto, la forma normale disgiuntiva di $\neg\neg((p\to(q\wedge r)) \vee (q\to s))$: otteniamo $\neg p \vee (q \wedge r) \vee \neg q \vee s$, quindi

$$\neg\neg(p\to(q \wedge r)) \vee (q\to s) \;\leftrightarrow\; \neg p \vee (q \wedge r) \vee \neg q \vee s$$

Dunque:

$$\begin{aligned} \neg((p\to(q \wedge r)) \vee (q\to s)) \quad & \leftrightarrow \quad \neg(\neg p \vee (q \wedge r) \vee \neg q \vee s) \\ & \leftrightarrow \quad p \wedge (\neg q \vee \neg r) \wedge q \wedge \neg s \end{aligned}$$

L'ultima equivalenza si ottiene applicando le leggi di De Morgan.

Si noti, infine, che, dato che la costruzione di un tableau proposizionale termina sempre, il metodo dei tableaux costituisce anche un metodo per dimostrare che una formula proposizionale A *non* è valida: si costruisce un tableau completo per $\neg A$; se tale tableau ha un ramo aperto, esso rappresenta un contromodello di A e dunque A non è valida.

2.11 Esercizi

1. Rappresentare le affermazioni seguenti mediante formule proposizionali:
   (a) Se l'umidità è elevata, pioverà questo pomeriggio o questa sera.
   (b) La mancia sarà pagata solo se il servizio è di qualità.
   (c) Il Cagliari vincerà lo scudetto, a meno che oggi non vinca l'Inter [3].

(d) Condizione necessaria e sufficiente affinché uno sceicco sia felice è avere vino, donne e canti [3].

2. Dimostrare che sono tautologie le formule riportate a pagina 13.
3. Dimostrare le equivalenze logiche riportate a pagina 14.
4. Per ciascuna delle formule seguenti determinare se è una tautologia, se è soddisfacibile, se è inconsistente.

   (a) $(p \to q) \vee \neg p$
   (b) $(p \to (q \to r)) \to ((p \to q) \to (p \to r))$
   (c) $\neg\neg p \to p$
   (d) $p \vee (p \to q)$
   (e) $(p \to q) \equiv (\neg p \vee q)$
   (f) $\neg p \to p$
   (g) $p \to \neg p$
   (h) $\neg(p \vee q) \vee (\neg q)$
   (i) $p \vee (q \to \neg p)$
   (j) $p \to \neg(p \vee q)$
   (k) $\neg p \wedge \neg(p \to q)$
   (l) $((p \to q) \to q) \to q$
   (m) $p \equiv (p \vee p)$
   (n) $p \equiv (p \wedge p)$
   (o) $\neg p \to (p \to q)$
   (p) $(p \to q) \equiv ((\neg p) \vee q)$
   (q) $(p \to q) \equiv \neg(p \wedge \neg q)$
   (r) $\neg p \to (p \wedge q)$
   (s) $p \wedge (p \vee q) \equiv p$
   (t) $p \vee (p \wedge q) \equiv p$

5. Un uomo veniva processato per furto. Il pubblico ministero e l'avvocato difensore fecero le seguenti affermazioni.
   Pubblico Ministero: Se l'imputato è colpevole, allora ebbe un complice.
   Avvocato difensore: Non è vero!
   Perché questa fu la cosa peggiore che l'avvocato difensore potesse dire? ([5])

6. Determinare se i seguenti ragionamenti sono corretti.

   (a) Se i controllori di volo scioperano, allora prenderò il treno. O scioperano i controllori, oppure scioperano i piloti. Quindi prenderò il treno.
   (b) Se i controllori di volo scioperano, allora prenderò il treno. Se i piloti scioperano, allora prenderò il treno. O scioperano i controllori, oppure scioperano i piloti. Quindi prenderò il treno.

(c) Se $\overset{\triangle}{ABC}$ è un triangolo equilatero, allora è un poligono regolare. Se $\overset{\triangle}{ABC}$ è un triangolo isoscele, allora è un poligono regolare. $\overset{\triangle}{ABC}$ è un triangolo equilatero oppure isoscele. Quindi $\overset{\triangle}{ABC}$ è un poligono regolare.

(d) Se piove Antonio non va a scuola. Se Antonio non va a scuola, la mamma e la maestra si inquietano. Quindi, se piove la maestra si inquieta.

7. Nell'isola dei cavalieri e furfanti [5, 4]:

   (a) Antonio dice "io sono un furfante". Cos'è Antonio?

   (b) Incontriamo due abitanti, A e B. A dice: "Almeno uno di noi è un furfante". Cosa sono A e B?

   (c) Incontriamo due abitanti, A e B. A dice: "Io sono un furfante ma B non lo è". Cosa sono A e B?

   (d) A dice: "io e mia sorella siamo dello stesso tipo". Scoprite il tipo di almeno uno dei due.

   (e) A dice: "B e C sono entrambi cavalieri". Poi gli chiedete: "È vero che B è un cavaliere?" e A risponde: "No". Cos'è A e cosa sono B e C?

   (f) Chiedete ad A: "B e C sono entrambi cavalieri?" Risponde di no. Poi gli chiedete: "B è un cavaliere?" Risponde: "Si". Cosa sono A, B e C?

   (g) Ci sono tre abitanti, A, B e C. A dice: "Siamo tutti furfanti". B dice: "Esattamente uno di noi è un cavaliere". Cosa sono A, B e C?

   (h) Ci sono tre persone: A, B e C. A dice: "B è un furfante". B dice: "A e C sono dello stesso tipo". Cos'è C?

   (i) Ci sono tre persone: A, B e C. A dice: "B e C sono dello stesso tipo". Poi chiedete a C: "A e B sono dello stesso tipo?". Cosa risponde C?

   (j) A dice: "Io sono un furfante oppure 2+2=5". Cosa si può concludere?

   (k) D vi ruba la chiave della macchina e la mette in uno di 3 cassetti: A, B, C. Gli chiedete: "È vero che la chiave è in A o in B?" Risponde di no. Chiedete allora: "È nel cassetto A?" Risponde di sì. Dov'è la chiave?

   (l) Tornate in albergo e non vi ricordate con precisione il numero della vostra stanza, che comunque deve essere il 33 o il 35. Chiedete al portiere: "È vero che il numero della mia stanza è il 33 o il 35?" Risponde di si. Poi gli chiedete: "è vero che è il 33?" e dice di no. Qual è il numero della vostra stanza?

8. Trovare forme normali disgiuntive e congiuntive equivalenti a:

   (a) $(\neg p \wedge q) \rightarrow r$

   (b) $p \rightarrow ((q \wedge r) \rightarrow s)$

   (c) $p \vee (\neg p \wedge q \wedge (r \vee q))$

(d) $(p \rightarrow q) \rightarrow r$

9. È possibile che una formula sia contemporaneamente in CNF e DNF?

10. Sono corretti i seguenti ragionamenti? Per rispondere seguire la procedura seguente:

Formalizzare innanzitutto il ragionamento. In seguito:

- Per dimostrare che un ragionamento è corretto, si possono seguire due metodi:

  (1) dare una dimostrazione semantica. La dimostrazione può essere diretta (supponiamo che esista un'interpretazione $\mathcal{M}$ in cui sono vere tutte le premesse, allora ... in $\mathcal{M}$ è vera anche la conclusione) oppure indiretta (supponiamo che esista un'interpretazione in cui sono vere tutte le premesse e falsa la conclusione, allora .... ASSURDO). Un terzo metodo semantico è quello che fa uso delle tavole di verità, ma cercate di evitarlo.

  (2) dare una dimostrazione sintattica, utilizzando il metodo dei tableaux.

- Per dimostrare che un ragionamento non è corretto può essere utilizzato un unico metodo: definire un'interpretazione in cui sono vere tutte le premesse e falsa la conclusione. Per far questo, è comunque possibile utilizzare il metodo dei tableaux.

(a) Se piove, o prendo l'ombrello o resto a casa. Se resto a casa non mi bagno. Quindi non mi bagno.

(b) Se piove, o prendo l'ombrello o resto a casa. Se resto a casa, sono al coperto. Se prendo l'ombrello, sono al coperto. Non mi bagno solo se sono al coperto. Quindi non mi bagno.

(c) Se piove, l'erba è bagnata. Non piove. Quindi l'erba è asciutta.

(d) Se piove, l'erba è bagnata. L'erba è bagnata. Quindi piove.

(e) Se Napoleone fosse tedesco, sarebbe asiatico. Napoleone non è asiatico. Quindi non è tedesco.

(f) Se l'investimento di capitali rimane costante, allora cresceranno le spese del governo oppure ci sarà disoccupazione. Se non aumenteranno le spese del governo, allora potranno essere ridotte le tasse. Se le tasse potranno essere ridotte e l'investimento di capitali rimane costante, allora non si avranno fenomeni di disoccupazione. Quindi aumenteranno le spese del governo [3].

(g) Se Bianchi non ha incontrato Rossi la notte scorsa, allora o Rossi era l'assassino, oppure Bianchi mente. Se Rossi non era l'assassino, allora Bianchi non ha incontrato Rossi la notte scorsa e il delitto è avvenuto dopo la mezzanotte. Se il delitto è avvenuto dopo la mezzanotte, allora o Rossi era l'assassino o Bianchi mente. Quindi Rossi era l'assassino [3].

(h) Aggiungere alle ipotesi di (10g): Se Bianchi mente, allora Bianchi ha incontrato Rossi la notte scorsa

(i) Aggiungere alle ipotesi di (10g): Bianchi non mente

11. Formulare regole di espansione per $A \equiv B$ e $\neg(A \equiv B)$, che siano "regole derivate" nel sistema dei tableaux.

12. Dimostrare, utilizzando i tableaux per la ricerca di modelli, che:

    (a) $p \to \neg q, \neg p \not\models q$

    (b) $\neg p \to \neg q, \neg q \not\models \neg p$

    (c) $p \to q \not\models q \to p$

    (d) $p \vee q \not\models p$

    (e) $p \to q \not\models p \wedge q$

    (f) $p \to q \not\models \neg p \to \neg q$

    (g) $p \vee q \not\models p$

    (h) $p \wedge (q \vee r) \not\models \neg(q \to r)$

13. Dimostrare mediante tableaux che:

    (a) $p \to (q \to r), p \vee s \vdash (q \to r) \vee s$

    (b) $p, q \to \neg p \vdash \neg q$

    (c) $\neg q \to q, p \to q \vdash q$

    (d) $(p \to q) \to p \vdash p$

    (e) $p \to q, q \to p, p \vee q \vdash p \wedge q$

    (f) $p \vee q, p \vee (q \wedge r) \vdash p \vee r$

    (g) $p \to (q \to r), p \to q \vdash p \to r$

    (h) $p \to \neg q, q \vdash \neg p$

    (i) $p \to (q \to r) \vdash q \to (p \to r)$

    (j) $q \to r \vdash (\neg q \to \neg p) \to (p \to r)$

    (k) $q \to (p \to r), \neg r, q \vdash \neg p$

    (l) $p \to \neg q \vdash q \to \neg p$

    (m) $\neg p \to \neg q \vdash q \to p$

    (n) $p \to q \vdash (q \to r) \to (p \to r)$

    (o) $p \to (q \to r) \vdash (p \wedge q) \to r$

    (p) $p \to q \vdash (r \wedge p) \to (r \wedge q)$

    (q) $p \vee q \vdash q \vee p$

    (r) $q \to r \vdash (p \vee q) \to (p \vee r)$

    (s) $p \vee (q \vee r) \vdash q \vee (p \vee r)$

    (t) $p \to q, p \to \neg q \vdash \neg p$

    (u) $p \to \neg p \vdash \neg p$

    (v) $p \vdash q \to (p \wedge q)$

(w) $(p \to q) \land (p \to r) \vdash p \to (q \land r)$

(x) $(p \to r) \land (q \to r) \vdash (p \lor q) \to r$

(y) $(p \to q) \land (r \to s) \vdash (p \land r) \to (q \land s)$

(z) $\neg p \to p \vdash p$

14. È vero o falso che $\models (p \to q) \lor (q \to p)$? Risolvere il problema utilizzando i tableaux.

15. Dimostrare la validità delle leggi di De Morgan, utilizzando il metodo dei tableaux con la regola per $\equiv$ formulata al punto 11.

16. Dimostrare mediante il metodo dei tableaux che le formule a pagina 13 sono tautologie e che le coppie di formule introdotte a pagina 14 sono logicamente equivalenti.

17. Dare dimostrazioni sintattiche (che utilizzano il metodo dei tableaux) per risolvere gli esercizi 7a, 7b, 7c e 7j relativi a cavalieri e furfanti. Si consideri anche un esempio di problema con tre personaggi e si risolva sempre utilizzando i tableaux.

2.12 Soluzione di alcuni esercizi

(1b) L'affermazione "la mancia sarà pagata solo se il servizio è di qualità" si può riformulare con "condizione necessaria perché la mancia venga pagata è che il servizio sia di qualità". In altri termini, se viene pagata la mancia, ciò significa che il servizio è stato di qualità (altrimenti niente mancia). Quindi, se rappresentiamo "la mancia sarà pagata" con l'atomo p e "il servizio è di qualità" con l'atomo q, la formula che rappresenta correttamente l'enunciato è: $p \to q$. La formula $q \to p$ rappresenta invece l'enunciato "se il servizio è di qualità, allora la mancia sarà pagata", o, in altri termini "condizione sufficiente perché la mancia venga pagata è che il servizio sia di qualità". Cioè l'implicazione è nel senso inverso rispetto a quello voluto: affermando che la mancia sarà pagata solo se il servizio è di qualità non si afferma che il servizio di qualità garantisce il pagamento della mancia.

(1c) Rappresentiamo "il Cagliari vincerà lo scudetto" con l'atomo p e "oggi vince l'Inter" con l'atomo q. L'enunciato "il Cagliari vincerà lo scudetto, a meno che oggi non vinca l'Inter" afferma che se oggi non vince l'Inter allora certamente il Cagliari vincerà lo scudetto. Dunque: $\neg q \to p$. Con questa interpretazione di "a meno che" non si esclude il caso in cui il Cagliari vinca comunque lo scudetto nonostante il fatto che oggi vinca l'Inter. Ma le frasi in linguaggio naturale sono spesso ambigue. Con l'enunciato considerato, potremmo intendere anche che la vittoria dell'Inter esclude la possibilità che il Cagliari vinca lo scudetto, allora possiamo riformulare l'enunciato in "il Cagliari vincerà lo scudetto se e solo se oggi non vince l'Inter", dunque la rappresentazione adeguata è in questo caso più forte: $p \equiv \neg q$.

(2) Negazione dell'antecedente (3): per dimostrare che $\models \neg A \to (A \to B)$ assumiamo, per assurdo, che esista un'interpretazione $\mathcal{M}$ tale che $\mathcal{M} \not\models \neg A \to (A \to B)$. L'unico caso in cui un'implicazione è falsa è quando è vero

l'anteceente e falso il conseguente; quindi deve essere (1) $\mathcal{M} \models \neg A$ e (2) $\mathcal{M} \not\models A \rightarrow B$. Da (1) segue (3) $\mathcal{M} \not\models A$ e da (2) seguono (4) $\mathcal{M} \models A$ e $\mathcal{M} \not\models B$. Ma (3) e (4) costituiscono una contraddizione, quindi non esiste alcuna interpretazione $\mathcal{M}$ tale che $\mathcal{M} \not\models \neg A \rightarrow (A \rightarrow B)$.

Legge di Pierce (8): per mostrare che $\models ((A \rightarrow B) \rightarrow A) \rightarrow A$, ragioniamo di nuovo per assurdo: assumiamo che, per qualche interpretazione $\mathcal{M}$, si abbia $\mathcal{M} \not\models ((A \rightarrow B) \rightarrow A) \rightarrow A$. Dunque (1) $\mathcal{M} \models (A \rightarrow B) \rightarrow A$ e (2) $\mathcal{M} \not\models A$. Ma se è falso il conseguente (2) di un'implicazione, deve essere falso anche l'antecedente, quindi (3) $\mathcal{M} \not\models A \rightarrow B$. Se un'implicazione è falsa, il suo antecedente è vero, quindi deve essere $\mathcal{M} \models A$, contraddicendo (2). Di conseguenza non esiste alcuna interpretazione $\mathcal{M}$ tale che $\mathcal{M} \not\models ((A \rightarrow B) \rightarrow A) \rightarrow A$.

(3) Dimostriamo la prima Legge di De Morgan: $\neg(A \vee B) \leftrightarrow \neg A \wedge \neg B$. Sia $\mathcal{M}$ un'interpretazione qualsiasi. Per il significato della negazione, $\mathcal{M} \models \neg(A \vee B)$ se e solo se $\mathcal{M} \not\models A \vee B$. L'unico caso in cui una disgiunzione è falsa è quando sono falsi entrambi i disgiunti. Quindi $\mathcal{M} \not\models A \vee B$ se e solo se $\mathcal{M} \not\models A$ e $\mathcal{M} \not\models B$. Per la semantica della negazione, ciò vale se e solo se $\mathcal{M} \models \neg A$ e $\mathcal{M} \models \neg B$, che equivale, per il significato della congiunzione, a $\mathcal{M} \models \neg A \wedge \neg B$.

(4a) Cerchiamo di costruire un contromodello di $(p \rightarrow q) \vee \neg p$: se $\mathcal{M} \not\models (p \rightarrow q) \vee \neg p$, deve essere (1) $\mathcal{M} \not\models p \rightarrow q$ e (2) $\mathcal{M} \not\models \neg p$. (1) equivale a dire che $\mathcal{M} \models p$ e $\mathcal{M} \not\models q$ e (2) equivale a $\mathcal{M} \models p$. Dunque, se $\mathcal{M}(p) = T$ e $\mathcal{M}(q) = F$, allora $\mathcal{M} \not\models (p \rightarrow q) \vee \neg p$. Di conseguenza la formula non è valida.

Per verificare se è soddisfacibile occorre determinare un'interpretazione in cui è vera. Perché una disgiunzione sia vera è sufficiente che sia vero uno dei due disgiunti: ad esempio $\mathcal{M} \models \neg p$. Dunque la formula è soddisfatta in qualsiasi interpretazione $\mathcal{M}$ tale che $\mathcal{M}(p) = F$, ed è allora soddisfacibile. Dato che è soddisfacibile, non è una contraddizione.

(4i) Cerchiamo di costruire un'interpretazione $\mathcal{M}$ tale che $\mathcal{M} \not\models p \vee (q \rightarrow \neg p)$. $\mathcal{M}$ deve essere tale che (1) $\mathcal{M} \not\models p$ e (2) $\mathcal{M} \not\models q \rightarrow \neg p$. Da (2) segue che $\mathcal{M} \models q$ e $\mathcal{M} \not\models \neg p$, quindi $\mathcal{M} \models p$, contraddicendo (1). Di conseguenza la formula è valida.

Se è valida, è ovviamente soddisfacibile e non è inconsistente.

(4r) La formula non è certamente valida: se p è falso, $\neg p$ è vero e $p \wedge q$ è falso, quindi $\neg p \rightarrow (p \wedge q)$ è falso.

Verifichiamo se la formula è soddisfacibile. Per far ciò è sufficiente verificare se esiste un'interpretazione in cui l'antecedente è falso. Se $\mathcal{M}(p) = T$, allora $\mathcal{M} \not\models \neg p$, quindi $\mathcal{M} \models \neg p \rightarrow (p \wedge q)$.

(5) Un'implicazione è falsa (affermazione dell'avvocato difensore) solo se l'antecedente è vero (e il conseguente falso): dunque l'imputato è colpevole.

(7) Nella soluzione di questi esercizi rappresentiamo con A, B e C gli enunciati "A è un cavaliere", "B è un cavaliere" e "C è un cavaliere", rispettivamente. Per ciascun esercizio risolto, indichiamo le formule che rappresentano la

nostra conoscenza e le rispettive tavole di verità, oppure un ragionamento semantico che consente di risolvere il problema.

(7b) Le formule $A \to \neg A \vee \neg B$ e $\neg A \to \neg(\neg A \vee \neg B)$ implicano logicamente A e $\neg B$:

A	B	$A \to \neg A \vee \neg B$	$\neg A \to \neg(\neg A \vee \neg B)$
T	T	F	T
T	F	T	T
F	T	T	F
F	F	T	F

(7c) Le formule $A \to \neg A \wedge B$ e $\neg A \to \neg(\neg A \wedge B)$ implicano logicamente $\neg A$ e $\neg B$:

A	B	$A \to \neg A \wedge B$	$\neg A \to \neg(\neg A \wedge B)$
T	T	F	T
T	F	F	T
F	T	T	F
F	F	T	T

(7g) Le formule che rappresentano la nostra conoscenza sono:

F_1. $A \to \neg A \wedge \neg B \wedge \neg C$
F_2. $\neg A \to A \vee B \vee C$
F_3. $B \to (A \wedge \neg B \wedge \neg C) \vee (\neg A \wedge B \wedge \neg C) \vee (\neg A \wedge \neg B \wedge C)$
F_4. $\neg B \to (\neg A \wedge \neg B \wedge \neg C) \vee (A \wedge B) \vee (A \wedge C) \vee (B \wedge C)$

Esse implicano logicamente $\neg A$, B e $\neg C$:

A	B	C	F_1	F_2	F_3	F_4
T	T	T	F			
T	T	F	F			
T	F	T	F			
T	F	F	F			
F	T	T	T	T	F	
F	T	F	T	T	T	T
F	F	T	T	T	T	F
F	F	F	T	F		

(7h) La conoscenza può essere rappresentata dalle quattro formule seguenti:

F_1. $A \to \neg B$
F_2. $\neg A \to B$
F_3. $B \to (A \wedge C) \vee (\neg A \wedge \neg C)$
F_4. $\neg B \to (A \wedge \neg C) \vee (\neg A \wedge C)$

Questa volta risolviamo il problema mediante un ragionamento semantico. Supponiamo che $\mathcal{M}$ sia un'interpretazione in cui sono vere le quattro formule date sopra. Ora, i casi sono due: $\mathcal{M} \models A$ oppure $\mathcal{M} \models \neg A$. Mostriamo che in entrambi i casi $\mathcal{M} \models \neg C$.

* Caso 1: $\mathcal{M} \models A$. Allora, dato che anche $\mathcal{M} \models A \to \neg B$ (F_1), $\mathcal{M} \models \neg B$, e poichè $\mathcal{M} \models F_4$, $\mathcal{M} \models (A \wedge \neg C) \vee (\neg A \wedge C)$. Quindi i casi sono due: (a) $\mathcal{M} \models A \wedge \neg C$, oppure (b) $\mathcal{M} \models \neg A \wedge C$. Mostriamo che il secondo caso è assurdo: se fosse $\mathcal{M} \models \neg A \wedge C$, allora, per la semantica della congiunzione, $\mathcal{M} \models \neg A$, contraddicendo l'ipotesi del caso 1. Quindi deve essere vero (a) e, sempre per la semantica della congiunzione, $\mathcal{M} \models \neg C$.
* Caso 2: $\mathcal{M} \models \neg A$. Poiché anche $\mathcal{M} \models F_2$, $\mathcal{M} \models B$ e, poiché $\mathcal{M} \models F_3$, si ha $\mathcal{M} \models (A \wedge C) \vee (\neg A \wedge \neg C)$. Quindi abbiamo due casi: (c) $\mathcal{M} \models A \wedge C$, oppure (d) $\mathcal{M} \models \neg A \wedge \neg C$. Mostriamo che il caso (c) è assurdo: in tal caso infatti si avrebbe $\mathcal{M} \models A$, contraddicendo l'ipotesi del caso 2. Quindi deve essere vero (d), dunque $\mathcal{M} \models \neg C$ per il significato della congiunzione.

Poiché in entrambi i casi possibili si ha $\mathcal{M} \models \neg C$, ne segue che $F_1, F_2, F_3, F_4 \models \neg C$.

(7i) Indichiamo in questo caso soltanto come impostare il problema. Le formule che rappresentano la nostra conoscenza sono:

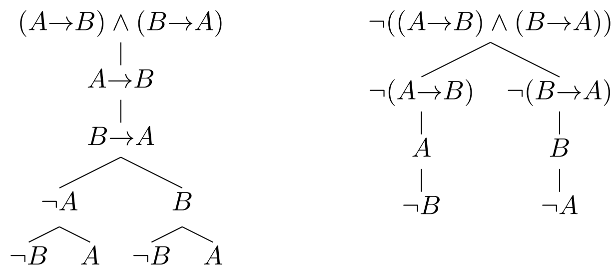
$$\begin{array}{ll} F_1. & A \to (B \wedge C) \vee (\neg B \wedge \neg C) \\ F_2. & \neg A \to (B \wedge \neg C) \vee (\neg B \wedge C) \end{array}$$

Il problema può essere risolto determinando se una delle due formule seguenti è una conseguenza logica di $\{F_1, F_2\}$:

$$\begin{array}{ll} (SI) & (C \to (A \wedge B) \vee (\neg A \wedge \neg B)) \wedge (\neg C \to (A \wedge \neg B) \vee (\neg A \wedge B)) \\ (NO) & (C \to (A \wedge \neg B) \vee (\neg A \wedge B)) \wedge (\neg C \to (A \wedge B) \vee (\neg A \wedge \neg B)) \end{array}$$

(9) Un letterale, una congiunzione di letterali, una disgiunzione di letterali sono tutte formule in CNF e anche in DFN.

(11) Consideriamo la definizione di $A \equiv B$ come $(A \to B) \wedge (B \to A)$ ed i seguenti tableaux:



Poiché il secondo e terzo ramo del tableau a sinistra sono chiusi, essi dimostrano che le espansioni:

$$\begin{array}{cc} \underline{\quad A \equiv B \quad} \\ \neg A \quad\quad B \\ \neg B \quad\quad A \end{array} \qquad \begin{array}{cc} \underline{\quad \neg(A \equiv B) \quad} \\ A \quad\quad B \\ \neg B \quad\quad \neg A \end{array}$$

sono regole derivate.

3 Implementazione OCaml di alcuni algoritmi

In questo capitolo mostriamo come rappresentare in Objective Caml le formule della logica proposizionale e come implementare alcune operazioni su di esse.

3.1 Rappresentazione di formule proposizionali

Le formule della logica proposizionale, costruite a partire da un insieme di variabili proposizionali e dalle costanti $\top$ e $\bot$ applicando i connettivi $\neg, \wedge, \vee, \rightarrow$, possono essere rappresentate mediante il tipo di dati così definito:

```
type form =
    True
  | False
  | Prop of string
  | Not of form
  | And of form * form
  | Or of form * form
  | Imp of form * form
```

Si noti che la doppia implicazione non è considerata un costruttore. Formule della forma $A \equiv B$ saranno rappresentate come $(A \rightarrow B) \wedge (B \rightarrow A)$.

Come esempi, le formule seguenti:

$$\begin{array}{rclrcl} f_1 & = & \neg p \vee q & f_2 & = & \neg(p \rightarrow q) \\ f_3 & = & f_1 \wedge f_2 & f_4 & = & f_3 \vee (r \wedge s) \\ f_5 & = & \neg(\neg p \rightarrow q) & f_6 & = & \neg(p \vee \neg q) \\ f_7 & = & \neg(f_5 \wedge f_6) & & & \end{array}$$

sono rappresentate, rispettivamente, dai valori così definiti:

```
let f1 = Or(Not(Prop "p"),Prop "q");;
let f2 = Not(Imp(Prop "p",Prop "q"));;
let f3 = And(f1,f2);;
let f4 = Or(f3,And(Prop "r", Prop "s"));;
let f5 = Not(Imp(Not(Prop "p"),Prop "q"));;
let f6 = Not(Or(Prop "p",Not(Prop "q")));;
let f7 = Not(And(f5,f6));;
```

Il tipo di dati "formula" è un tipo ricorsivo. Sulle formule si può quindi lavorare ricorsivamente, come ad esempio sugli alberi binari (in effetti le `form` sono un tipo particolare di alberi binari). Ad esempio, si può definire l'insieme degli atomi che occorrono in una formula come segue (la funzione utilizza `union`):

```
(* setadd : 'a -> 'a list -> 'a list *)
(* aggiunge un elemento a una lista se non vi occorre gia` *)
let setadd x xs = if List.mem x xs then xs else x::xs

(* union : 'a list -> 'a list -> 'a list *)
(* riporta l'unione degli elementi di due liste, assumendo che
   i due argomenti siano anch'essi liste senza ripetizioni *)
let rec union lst ys =
```

```
  match lst with
    [] -> ys
  | x::xs -> setadd x (union xs ys);;

(* atomlist : form -> string list *)
let rec atomlist = function
    True | False -> []
  | Prop s -> [s]
  | Not f -> atomlist f
  | Or(f1,f2) -> union (atomlist f1)
                       (atomlist f2)
  | And(f1,f2) -> union (atomlist f1)
                        (atomlist f2)
  | Imp(f1,f2) -> union (atomlist f1)
                        (atomlist f2)
```

Sarà utile nel seguito disporre di una funzione per trasformare formule in stringhe, definita anch'essa in modo ricorsivo:

```
(* f2s : form -> string *)
let rec f2s = function
    True -> "T"
  | False -> "F"
  | Prop p -> p
  | Not f -> "-"^(f2s f)
  | And(f,g) -> String.concat "" ["(";f2s f;"&"; f2s g;")"]
  | Or(f,g) -> String.concat "" ["(";f2s f;"|"; f2s g;")"]
  | Imp(f,g) -> String.concat "" ["(";f2s f;"->"; f2s g;")"]

# f2s f3;;
- : string = "((-p|q)&-(p->q))"
```

3.2 Un parser per le formule proposizionali

In questo paragrafo mostriamo come costruire un semplice parser per formule della logica proposizionale, utilizzando i "costruttori di parser" `ocamllex` e `ocamlyacc`.

A questo scopo è necessario avere alcune nozioni sul sistema dei moduli di OCaml e la corrispondenza tra moduli e files. Ne diamo qui soltanto alcuni cenni, per una trattazione più approfondita si veda il capitolo 5 del libro [1].

Un modulo (chiamato *struttura* in OCaml) è un ambiente separato dall'ambiente principale, ai cui elementi si accede precedendoli con il nome del modulo (che inizia sempre con una lettera maiuscola), seguito da un punto e dal nome della variabile cui si vuole accedere. Così ad esempio, si accede al contenuto dei moduli della libreria standard di OCaml: `List.mem` è la funzione `mem` definita nel modulo `List`.

Un modulo può essere definito dal programmatore mediante una *dichiarazione di struttura*, oppure implicitamente, suddividendo il programma in file separati: ogni file, allora, corrisponde a un modulo, il cui nome è uguale a quello del file stesso (senza l'estensione `.ml`), ma con la prima lettera maiuscola.

Ad esempio, assumiamo che il file `language.ml` contienga la dichiarazione del tipo `form`, cioè che il suo contenuto sia il seguente:

```
type form =
    True
  | False
  | Prop of string
  | Not of form
  | And of form * form
  | Or of form * form
  | Imp of form * form
```

Allora gli altri moduli (file) del programma "vedranno" un tipo `Language.form`, con i costruttori `Language.True, Language.False, Language.Prop`, ...

Perché gli altri moduli possano vedere il contenuto di `Language`, occorre prima compilare il file, mediante il comando (da una shell di comandi):

```
ocamlc -c language.ml
```

Tale comando genera i file `language.cmi` (l'interfaccia compilata del modulo) e `language.cmo` (*bytecode* del modulo `Language`).

Per utilizzare ora il modulo `Language` in modalità interattiva, occorre caricarlo in memoria, utilizzando la *direttiva* `#load`:[2]

```
# #load "language.cmo";;
```

Ma i costruttori sono ora relativi al modulo `Language`:

```
# Prop "p";;
Characters 0-8:
  Prop "p";;
  ^^^^^^^^
Unbound constructor Prop
# Language.Prop "p";;
- : Language.form = Language.Prop "p"
```

Se si vuole evitare di scrivere `Language.Prop, Language.And`, ..., occorre "aprire" il modulo:

```
# open Language;;
# Prop "p";;
- : Language.form = Prop "p"
```

Abbiamo scritto il file `language.ml` perché questo sarà utilizzato dal parser, il quale sarà costruito, come già detto, dai costruttori di parser `ocamllex` e `ocamlyacc`. Un generatore di parser (chiamato anche *compiler-compiler*) è uno strumento per la generazione del codice sorgente di un parser, a partire dalla descrizione data nella forma di grammatica (in genere BNF). I nomi `ocamllex` e `ocamlyacc` derivano dai due più noti generatori di parser, `Lex` e `Yacc`. `Lex` genera codice C per un *lex*ical analyzer. `Yacc` (acronimo per "Yet Another

[2]L'uso di `#load` è simile a quello di `#use`, ma il primo deve essere richiamato su file con estensione `.cmo` (bytecode), mentre il secondo su file contenenti codice sorgente (con estensione `.ml`).

Compiler Compiler!"), a partire dalla descrizione di una grammatica, genera un *parser* o analizzatore sintattico (in C): la sequenza di *token* prodotta dall'analisi lessicale è organizzata in un albero sintattico.

Analogamente, in OCaml, `ocamllex` produce il codice OCaml dell'analizzatore lessicale corrispondente ai pattern specificati in un file con estensione `mll`. E `ocamlyacc` produce il codice OCaml di un parser, corrispondente alla grammatica specificata.

3.2.1 La specifica della grammatica

La descrizione della grammatica viene dichiarata in un file con estensione `mly`. Nel nostro caso, definiamo la grammatica delle formule nel file `parser.mly`.

Quando un file con estensione `.mly` viene dato in input a `ocamlyacc`:

```
ocamlyacc parser.mly
```

vengono prodotti due file: `parser.ml`, con il codice OCaml del parser, e la sua interfaccia (o *segnatura*, nella terminologia di OCaml), `parser.mli`.

Un file `mly` inizia con un *header* (codice OCaml che verrà copiato all'inizio del file che conterrà il codice OCaml del parser) e la dichiarazione dei simboli terminali (*token*) della grammatica.

Nel nostro caso, il file `parser.mly` inizia con:

```
%{ open Language %}

%token <string> ATOM
%token TRUE FALSE NOT AND OR IMP IFF RPAREN LPAREN EOF END
%left IFF
%left IMP
%left OR
%left AND
%nonassoc NOT
```

L'*header*, come si vede, "apre" la struttura `Language`.

Le dichiarazioni `%token` definiscono il tipo dei simboli terminali (token), e le dichiarazioni `%left`, `%right` e `%nonassoc` associano la precedenza e l'associatività ai simboli corrispondenti: ogni simbolo ha precedenza più alta di quelli che lo precedono. Il token speciale `EOF` (End Of File) è il token che segnala la fine del dell'input.

Le dichiarazioni dei token produrranno, nel file `parser.ml`, la corrispondente dichiarazione di tipo:

```
type token =
    ATOM of string
  | TRUE
  | FALSE
  | NOT
  | AND
  | OR
  | IMP
  | IFF
  | RPAREN
```

```
  | LPAREN
  | EOF
```

Il file `parser.mly` prosegue con la dichiarazione degli *entry point* della grammatica:

```
%start formula
%type <Language.form> formula
```

Per ogni entry point, il modulo prodotto da `ocamlyacc` avrà una funzione con lo stesso nome. Nel nostro caso, abbiamo un unico *entry point*, chiamato `formula`, il cui tipo è `Language.form`: il parser avrà dunque una funzione `formula` che, applicata agli argomenti appropriati, riporta una `Language.form`. In particolare, il tipo di `Parser.formula` è:

```
formula :
  (Lexing.lexbuf  -> token) -> Lexing.lexbuf -> Language.form
```

(il tipo degli argomenti di `formula` sarà chiarito in seguito).

Ogni simbolo non terminale deve poi essere definito mediante regole sintattiche, che associano un *attributo* (un'espressione OCaml) a ogni caso di espansione.

```
formula :
    ATOM                        { Prop($1) }
  | TRUE                        { True }
  | FALSE                       { False }
  | NOT formula                 { Not $2}
  | formula IFF formula         { And(Imp($1,$3),Imp($3,$1)) }
  | formula IMP formula         { Imp($1,$3) }
  | formula OR formula          { Or($1,$3) }
  | formula AND formula         { And($1,$3) }
  | LPAREN formula RPAREN       { $2 }
;
```

La sintassi da utilizzare è simile a quella utilizzata per specificare una grammatica in BNF. Ogni caso è costituito da una sequenza di simboli (terminali o no), nella parte sinsitra, a cui è associata un'espressione OCaml (o "attributo semantico", nella parte destra) che sarà valutata a riportata come valore nel caso corrispondente. In tale espressione si può fare riferimento agli attributi semantici corrispondenti ai simboli che occorrono nella parte sinistra mediante la notazione `$`: `$1` corrisponde all'attributo del primo simbolo, `$2` quello del secondo simbolo, ecc.

3.2.2 L'analisi lessicale

Il comando `ocamllex` produce un analizzatore lessicale da un insieme di espressioni regolari con associate "azioni semantiche".

Ad esempio, il contenuto del file per l'analisi lessicale di formule proposizionali, `lexer.mll`, che viene dato in input a `ocamllex` è:

```
{ open Parser }

rule token = parse
  [' ' '\t' '\n' ] {token lexbuf }
| "T" { TRUE }
| "F" { FALSE }
| '&' { AND }
| '|'  { OR }
| "=>" | "->" { IMP }
| "<=>" | "<->" {IFF }
| '-'  { NOT }
| '(' {  LPAREN }
| ')' {  RPAREN }
| ['a'-'z' 'A'-'Z'] ['a'-'z' 'A'-'Z' '0'-'9' '_']*
    { ATOM(Lexing.lexeme lexbuf) }
| eof  { EOF }
| _  { print_string "Token sconosciuto ignorato\n "; token lexbuf }
```

Anche qui abbiamo un *header*, che apre la struttura `Parser`. Il lexer infatti presuppone la definizione del tipo `token` contenuta nel file `parser.ml`.

Nel file `lexer.mll`, `token` è un "entry point". (nel nostro caso l'unico, ma possono essercene diversi).

L'esecuzione del comando

```
ocamllex lexer.mll
```

produce codice OCaml per l'analizzatore lessicale nel file `lexer.ml`. Tale file conterrà la definizione di una *funzione di lexing* per ogni *entry point* del file `lexer.mll`. Le funzioni del file `lexer.ml` avranno lo stesso nome del corrispondente entry point. Nel nostro caso, dunque, `lexer.ml` conterrà la definizione della funzione `token`.

L'argomento delle funzioni di lexing è un *lexer buffer*: il tipo dei *lexer buffer* è un tipo astratto implementato nel modulo `Lexing` della libreria standard. Tale modulo contiene anche le funzioni `Lexing.from_channel` e `Lexing.from_string`, che creano un *lexer buffer* a partire da un canale di input, o una stringa, rispettivamente (si veda il manuale di OCaml).

Le funzioni di lexing sono definite per casi, ciascuno dei quali è costituito da una parte sinistra in cui è specificata un'espressione regolare (si veda il manuale di OCaml per la sintassi delle espressioni regolari), con associato "attributo semantico", nella parte destra (espressione OCaml). Il valore riportato dalle funzioni di lexing è l'attributo semantico del caso corrispondente.

Nella definizione di `token` l'unico caso complesso è quello della definizione degli `ATOM`: in questo caso, se il lexer buffer inizia con un'espressione che corrisponde all'espressione regolare specificata (carattere alfabetico, seguito da un numero qualsiasi di caratteri alfanumerici o `'_'`), viene riportato il valore `ATOM str`), dove `str` è la stringa letta dal buffer che corrisponde all'espressione regolare specificata (`Lexing.lexeme lexbuf`).

Il file `lexer.ml` prodotto da `ocamllex` conterrà la definizione della funzione `token`:

```
token : Lexing.lexbuf -> Parser.token
```

Si ricordi che il tipo del primo argomento della funzione `Parser.formula` è proprio quello di `Lexer.token`. Quindi la funzione `Lexer.token` può essere passata come primo argomento a `Parser.formula`.

Per ottenere una formula a partire, ad esempio, da una stringa `str`, si applicherà `Parser.formula` nel modo seguente:

```
Parser.formula Lexer.token (Lexing.from_string str)
```

Nella valutazione di questa espressione, la stringa `str` viene convertita in un *lexer buffer*, e tale buffer è analizzato dal parser in base all'analisi lessicale eseguita su di esso da `Lexer.token`.

3.2.3 Compilazione dei file e utilizzazione del parser in modalità interattiva

Assumiamo ora di aver compilato il file `language.ml`, generando il file `language.cmo` (il modulo `Language`):

```
ocamlc -c language.ml
```

Ora è possibile generare il parser, mediante i comandi:

```
ocamlyacc parser.mly
ocamllex lexer.mll
```

`ocamlyacc` genera i file `parser.ml` e `parser.mli` (l'interfaccia); ocamllex genera `lexer.ml`.

Questi file devono essere a loro volta compilati:

```
ocamlc -c parser.mli parser.ml lexer.ml
```

Il comando genera i corrispondenti file `cmi` e `cmo`.

Per utilizzarli nella modalità interattiva di OCaml, occorre caricare in memoria i moduli definiti in tali file:

```
# #load "language.cmo";;
# #load "parser.cmo";;
# #load "lexer.cmo";;
```

Ora abbiamo a disposizione i moduli `Language`, `Parser` e `Lexer`.

Dopo aver aperto la struttura `Language`, possiamo definire una funzione `parse` che esegue l'analisi sintattica di una stringa, convertendola in una `Language.form`, e scrivere una funzione `read` che legge una stringa da tastiera e riporta la `Language.form` corrispondente:

```
open Language;;

(* parse: string -> Language.form *)
(* solleva Parsing.parse_error in caso di fallimento *)
(* vedi modulo Parsing della libreria standard *)
let parse stringa =
  let lexbuf = Lexing.from_string stringa (* vedi modulo Lexing della
                                             libreria standard *)
  in Parser.formula Lexer.token lexbuf
```

```
(* lettura di una formula da tastiera: *)
let read() =
  try parse (read_line())
  with Parsing.Parse_error ->
    print_string "Syntax error\n";
    False
```

La funzione `read` può essere utilizzata ad esempio nel modo seguente:

```
# let f=read();;
-p=>(pippo|-(pluto=>paperino))
val f : Language.form =
  Imp (Not (Prop "p"),
   Or (Prop "pippo", Not (Imp (Prop "pluto", Prop "paperino"))))

# let g=read();;
A<=>B
val g : Language.form =
  And (Imp (Prop "A", Prop "B"), Imp (Prop "B", Prop "A"))

# f2s g;;
- : string = "((A->B)&(B->A))"
```

3.2.4 Compilazione separata e codice eseguibile

Le estensioni dei file OCaml segnalano, come abbiamo visto, se il file contiene il codice di una struttura (modulo) o di una segnatura (interfaccia):

- `<nome-file>.mli` contiene il codice sorgente per un'interfaccia. La compilazione produce un'interfaccia compilata nel file `<nome-file>.cmi`
- `<nome-file>.ml` contiene codice sorgente per una unità di compilazione (modulo, o struttura). La compilazione produce bytecode nel file `<nome-file>.cmo`

Per compilare separatamente un'unità o un'interfaccia, si utilizza il comando `ocamlc -c`:

```
ocamlc -c <nome-file>.mli
ocamlc -c <nome-file>.ml
```

Mediante *linking* del bytecode (file `.cmo`) si ottiene un programma eseguibile indipendente.

```
ocamlc -o <nome-programma> <file_1.cmo> ... <file_n.cmo>
```

Il comando `ocamlc -o` può anche essere invocato direttamente sui file sorgente:

```
ocamlc -o <nome-programma> <f_1.mli> <f_1.ml> ... <f_n.ml>
```

Questi comandi creano il bytecode del programma nel file `<nome-programma>`. Per eseguirlo, si invoca `ocamlrun`, l'interprete di bytecode:

```
ocamlrun <nome-programma>
```

È anche possibile creare programmi *stand-alone*, che possono essere eseguiti anche su macchine su cui non sia istallato OCaml, mediante il comando `ocamlc` invocato con l'opzione `-custom`:

```
  ocamlc -custom -o <nome-programma> ........
```

(per la creazione di programma stand-alone su sistemi Windows, vedere il sito di OCaml)

Se vogliamo eseguire un programma creato con

```
  ocamlc -o <nome-programma> ....
```

il file "principale" (quello dal quale non dipende nessun altro), deve contenere una richiesta di valutazione che dia l'avvio all'esecuzione.

Ad esempio:

```
let _ = <Espressione-da-valutare>
```

È anche possibile scrivere programmi che accettano argomenti: vedere il modulo `Arg` della libreria standard.

Nel nostro caso, consideriamo come semplice esempio un programma che esegue un ciclo, in cui, ad ogni iterazione legge da tastiera una stringa, la converte in `Language.form` e la stampa su video. Il ciclo termina quando viene immessa la stringa vuota.

Supponiamo allora di aver scritto i file `language.ml`, `lexer.mll` e `parser.mly`, e di aver creato il parser (i file `lexer.ml, parser.mli` e `parser.mly`), mediante i comandi:

```
ocamlc -c language.ml
ocamlyacc parser.mly
ocamllex lexer.mll
```

Ora scriviamo un file `prova.ml` con il seguente contenuto:

```
open Language (* "apriamo" il modulo Language *)

(* string -> Language.form *)
let parse stringa =
  let lexbuf = Lexing.from_string stringa
  in Parser.formula Lexer.token lexbuf

(* Language.form -> string *)
let rec f2s = function
    True -> "T"
  | False -> "F"
  | Prop p -> p
  | Not f -> "-"^(f2s f)
  | And(f,g) -> String.concat "" ["(";f2s f;"&"; f2s g;")"]
  | Or(f,g) -> String.concat "" ["(";f2s f;"|"; f2s g;")"]
  | Imp(f,g) -> String.concat "" ["(";f2s f;"->"; f2s g;")"]

(* loop: unit -> unit *)
```

```
let rec loop() =
  print_string "\nScrivi una formula: ";
  let stringa=read_line()
  in if stringa="" then ()
  else ((try (print_string(f2s (parse stringa)))
         with Parsing.Parse_error ->
            print_string "Syntax error\n");
        loop())

(* invocazione principale *)
let _=loop()
```

Compiliamo, creando il file `prova`:

```
ocamlc -o prova language.ml parser.mli parser.ml lexer.ml prova.ml
```

che ora possiamo eseguire (da una command shell):

```
ocamlrun prova

Scrivi una formula: p&q
(p&q)

Scrivi una formula: -r->(s<->q)
(-r->((s->q)&(q->s)))

Scrivi una formula: p+q
Token sconosciuto ignorato
 Syntax error

Scrivi una formula:
```

3.2.5 Uso di moduli in modalità interattiva

Come abbiamo visto, se abbiamo compilato `language.ml`, `parser.ml` (con la sua interfaccia `parser.mli`) e `lexer.ml`:

```
ocamlc -c language.ml parser.mli parser.ml lexer.ml
```

(creando i rispettivi file `cmi` e `cmo`), possiamo caricarli nella modalità interattiva di Ocaml, e possiamo, ad esempio, definire una funzione che converte una stringa nella rappresentazione interna della formula corrispondente utilizzando il parser.

Negli esempi di esecuzioni che vedremo nel seguito, assumiamo che tutte le definizioni siano contenute in un file `logic.ml` che inizia come segue:

```
# #load "language.cmo";;
# #load "parser.cmo";;
# #load "lexer.cmo";;

(* si apre il modulo Language *)
open Language

(* string -> Language.form *)
```

```
(* solleva Parsing.Parse_error se il parsing fallisce *)
let parse stringa =
  let lexbuf = Lexing.from_string stringa
  in Parser.formula Lexer.token lexbuf
```

Il file sarà caricato in memoria con il comando `#use`:

```
# #use "logic.ml";;
```

3.3 Forme normali

La funzione `nnf` trasforma una formula in "forma normale negativa" (NNF: *Negation Normal Form*), cioè in una formula equivalente in cui non occorrono implicazioni e dove nessun connettivo occorre nello scopo di una negazione (le uniche sottoformule della forma $\neg A$ sono tali che A è un atomo). La funzione utilizza le equivalenze $A{\rightarrow}B \equiv \neg A \vee B$, $\neg(A{\rightarrow}B) \equiv A \wedge \neg B$ e le leggi di De Morgan.

```
(* nnf: form -> form *)
let rec nnf = function
  | And(f,g) -> And(nnf f,nnf g)
  | Or(f,g) -> Or(nnf f,nnf g)
  | Imp(f,g) -> nnf(Or(Not f,g))
  | Not(And(f,g)) -> Or(nnf(Not f),nnf(Not g))
  | Not(Or(f,g))  -> And(nnf(Not f),nnf(Not g))
  | Not(Imp(f,g)) -> And(nnf f,nnf(Not g))
  | Not(Not f) -> nnf f
  | f -> f

# f2s (nnf (parse  "-(-(p|q)|-(p|-q))"));;
- : string = "((p|q)&(p|-q))"
```

Un algoritmo per la trasformazione in CFN secondo il metodo indicato a pagina 19 è il seguente:

- Eliminare le implicazioni e le doppie implicazioni e portare le negazioni sugli atomi (trasformare A in NNF)
- Ricorsivamente:
  - se A ha la forma $B \vee C$, distribuire l'OR più esterno su tutte le congiunzioni nella formula $CNF(B) \vee CNF(C)$
  - se A ha la forma $B \wedge C$, il risultato è $CNF(B) \wedge CNF(C)$
  - altrimenti (A è un letterale) il risultato è A

La funzione che segue è una implementazione OCaml di tale algoritmo, che sfrutta la definizione della funzione `nnf` precedentemente definita.

```
(* cnf : form -> form *)
let rec cnf f =
  let rec distrib  = function
      (* applicazione delle distributive *)
```

```
      Or(f,g) ->
        (match (distrib f,distrib g) with
          (And(f1,g1),h) | (h,And(f1,g1)) ->
            distrib (And(Or(h,f1), Or(h,g1)))
        | (f1,h) -> Or(f1,h))
       (* ricorsione generale *)
    | And(f,g) -> And(distrib f,distrib g)
    | f -> f
  in distrib(fnn f)
```

La trasformazione di una formula in forma normale disgiuntiva è lasciata per esercizio.

3.4 Valutazione di una formula in una interpretazione

Rappresentiamo qui un'interpretazione $\mathcal{M}$ per un linguaggio della logica proposizionale mediante una lista di stringhe, contenente i nomi di tutte e solo le lettere proposizionali vere in $\mathcal{M}$. Definiamo il tipo corrispondente:

```
type interpretation = string list
```

La definizione della funzione `models`, che riporta il valore di verità di una formula in un'interpretazione, ricalca esattamente la definizione ricorsiva di verità di una formula in una interpretazione.

```
(* models: form -> interpretation -> bool *)
let rec models f emme =
  match f with
    True -> true
  | False -> false
  | Prop name -> List.mem name emme
  | Not f1 -> not(models f1 emme)
  | And(f1,f2) -> models f1 emme & models f2 emme
  | Or(f1,f2) -> models f1 emme or models f2 emme
  | Imp(f1,f2) -> not(models f1 emme) or models f2 emme

# let emme = ["p";"q"];;
val emme : string list = ["p"; "q"]
# models (parse "-p|q") emme;;
- : bool = true
# models (parse "-(p->q)") emme;;
- : bool = false
```

3.5 Tavole di verità

Descriviamo qui un programma per controllare la validità di una formula con il metodo delle tavole di verità.

Per costruire la tavola di verità di una formula, occorre generare tutte le interpretazioni dell'insieme delle lettere proposizionali che occorrono nella formula. La funzione `atomlist: form -> string list`, che abbiamo già visto, estrae da una formula le variabili proposizionali che in essa occorrono.

```
(* setadd : 'a -> 'a list -> 'a list *)
let setadd x xs
    = if List.mem x xs then xs else x::xs

(* union : 'a list -> 'a list -> 'a list *)
let rec union lst ys =
  match lst with
    [] -> ys
  | x::xs -> setadd x (union xs ys);;

(* atomlist : form -> string list *)
let rec atomlist = function
    True | False -> []
  | Prop s -> [s]
  | Not f -> atomlist f
  | Or(f1,f2) -> union (atomlist f1) (atomlist f2)
  | And(f1,f2) -> union (atomlist f1) (atomlist f2)
  | Imp(f1,f2) -> union (atomlist f1) (atomlist f2)

# atomlist (parse "((-p|q)&-(p->q))|(r&s)");;
- : string list = ["p"; "q"; "r"; "s"]
```

La funzione `powset: 'a list -> 'a list list`, applicata a una lista di lettere proposizionali, genera una lista con tutte le interpretazioni di tali lettere proposizionali, cioè tutti i sottoinsiemi dell'insieme delle lettere proposizionali. La funzione è definita ricorsivamente. Chiaramente, l'unica interpretazione dell'insieme vuoto di lettere proposizionali è rappresentata dalla lista vuota. Per il caso ricorsivo, assumiamo di saper calcolare l'insieme $[\mathtt{int}_1, \ldots, \mathtt{int}_n]$ di tutte le interpretazioni di una lista `props` di lettere proposizionali. Allora ciascuna interpretazione di `props` è ancora un'interpretazione di `p::props` (in cui `p` è falso); inoltre, si devono aggiungere tutte le interpretazioni che si ottengono aggiungendo `p` a ciascuna $\mathtt{int}_i$ (l'estensione di $\mathtt{int}_i$ in cui `p` è vero).

```
(* cons : 'a -> 'a list -> 'a list *)
let cons x xs = x::xs;;

(* powset : 'a list -> 'a list list *)
let rec powset = function
    [] -> [[]]
  | p::props ->
      let all_ints = powset props
      in all_ints @ List.map (cons p) all_ints

# powset ["p";"q"];;
- : string list list = [[]; ["q"]; ["p"]; ["p"; "q"]]
# powset ["p";"q";"r"];;
- : string list list =
[[]; ["r"]; ["q"]; ["q"; "r"]; ["p"]; ["p"; "r"]; ["p"; "q"];
 ["p"; "q"; "r"]]
```

Infine, la funzione `valid` controlla la validità di una formula, cioè se ogni interpretazione `emme` delle lettere proposizionali in `f` è tale che `models f emme`.

```
(* valid : form -> bool *)
let valid f =
  List.for_all (models f) (powset (atomlist f));;

# valid (parse "(-p|q)");;
- : bool = false
# valid (parse "-(-(-p->q)&-(p|-q))");;
- : bool = true
```

Utilizzando le funzioni precedenti, possiamo definire una funzione che genera la tavola di verità di una formula. Una tavola di verità è rappresentata mediante una lista di coppie; il primo elemento di ogni coppia è la rappresentazione di una assegnazione $\mathcal{M}$, il secondo è il valore della formula secondo $\mathcal{M}$.

```
(* truthtable : form -> (interpretation * bool) list *)
let truthtable f =
  let mkrow f emme = (emme, models f emme)
  in List.map (mkrow f)(powset (atomlist f));;

# truthtable  (parse "-p|q");;
- : (string list * bool) list =
[([], true); (["q"], true); (["p"], false); (["p"; "q"], true)]
# truthtable (parse "-(p|-q)");;
- : (string list * bool) list =
[([], false); (["q"], true); (["p"], false); (["p"; "q"], false)]
# truthtable (parse "p|-p");;
- : (string list * bool) list = [([], true); (["p"], true)]
# let pierce = parse "((A->B)->A)->A";;
val pierce : Language.form =
  Imp (Imp (Imp (Prop "A", Prop "B"), Prop "A"), Prop "A")
# truthtable pierce;;
- : (string list * bool) list =
[([], true); (["A"], true); (["B"], true); (["B"; "A"], true)]
```

3.6 Conseguenza e equivalenza logica

Per controllare se una formula è una conseguenza logica di un insieme di formule, possiamo costruire la corrispondente implicazione e controllarne la validità:

```
(* mkand: form list -> form *)
let rec mkand = function
    [] -> True
  | [f] -> f
  | f::rest -> And(f,mkand rest)

(* consequence: form list -> form -> bool *)
let consequence set f =
```

```
  valid (Imp(mkand set,f))

# consequence  [parse "p"; parse "q->-p"] (parse "-q");;
- : bool = true
```

Alternativamente, si può verificare se la formula data come secondo argomento è vera in tutti i modelli della lista di formule data come primo artomento. Questo secondo metodo è lasciato come esercizio (ci si può ispirare al modo in cui viene controllata l'equivalenza logica nel seguito).

Oppure, ricordando che $S \models A$ se e solo se $S \cup \{\neg A\}$ è insoddisfacibile:

```
let consequence set f =
  not(sat (And(Not f,mkand set)))
```

La funzione `logequiv` verifica se due formule sono logicamente equivalenti applicando la definizione di equivalenza logica:

```
(* logequiv: form -> form -> bool *)
let logequiv f g =
  List.for_all
    (function emme -> models f emme = models g emme)
    (powset (union (atomlist f)(atomlist g)))

# logequiv (parse "A | A&B") (parse "A");;
- : bool = true
```

3.7 Il metodo dei tableaux

In questo paragrafo mostriamo come implementare in OCaml il metodo dei tableaux per controllare la validità di una formula, per ricercarne un modello e per trasformarla in forma normale disgiuntiva.

Per rendere più compatta la costruzione dei tableaux è utile definire alcune funzioni ausiliarie. La funzione `alpha`, applicata a formule a cui si possa applicare una α-regola (regola che non fa ramificare) e che produrrebbe l'aggiunta di due formule al ramo, riporta la coppia costituita da queste due formule. Negli altri casi solleva un'eccezione. Analogamente, `beta` applicata a una β-formula (una formula a cui si può applicare una β-regola, cioè una regola che produce una ramificazione), riporta la coppia delle due formule che andrebbero aggiunte, rispettivamante, al ramo di sinistra e quello di destra.

Infine, la funzione `complement`, applicata a un letterale, ne riporta il complemento.

```
(* alpha : form -> form * form *)
let alpha = function
    And(f,g) -> (f,g)
  | Not(Or(f,g)) -> (Not f,Not g)
  | Not(Imp(f,g)) -> (f,Not g)
  | _ -> failwith "alpha"

(* val beta : form -> form * form *)
let beta = function
    Or(f,g) -> (f,g)
```

```
  | Not(And(f,g)) -> (Not f,Not g)
  | Imp(f,g) -> (Not f,g)
  | _ -> failwith "beta"

(* complement : form -> form *)
let complement = function
    Prop p -> Not(Prop p)
  | Not(Prop p) -> Prop p
  | _ -> failwith "complement"
```

La funzione `valid_tab: form -> bool`, applicata a una formula F, riporta `true` se F è valida, `false` altrimenti. Il controllo è eseguito mediante il metodo dei tableaux, implementato dalla funzione ausiliaria `closed` che, applicata a una lista di formule (le formule ancora "attive" del ramo) e una lista di letterali (i letterali presenti nel ramo), controlla se l'espansione del ramo genera un tableau chiuso (cioè con tutti i rami chiusi).

```
(* valid_tab : form -> bool *)
let valid_tab f =
  let rec closed pending lits =
    match pending with
      [] -> false  (* il ramo e' aperto *)
    |  f::rest ->  match f with
        | True | Not False -> closed rest lits
        | False | Not True -> true (* il ramo e' chiuso *)
        | Prop _ | Not(Prop _) ->
            if List.mem f lits then closed rest lits
            else List.mem (complement f) lits
                 or closed rest (f::lits)
        | Not(Not f) -> closed (f::rest) lits
        | And(_,_) | Not(Or(_,_)) | Not(Imp(_,_)) ->
            let (f1,f2)= alpha f in
            closed (f1::f2::rest) lits
        | Or(_,_) | Not(And(_,_)) | Imp(_,_) ->
            let (f1,f2) = beta f in
            closed (f1::rest) lits & closed (f2::rest) lits
  in closed [Not f] [];;

# valid_tab (parse "(p->-p)|(-p->p)");;
- : bool = true
```

La funzione `valid` può essere modificata in modo da controllare la soddisfacibilità di un insieme di formule e riportare la rappresentazione di un'interpretazione in cui tutte le formule dell'insieme sono vere, se esiste. La funzione solleva un'eccezione se l'insieme è insoddisfacibile. La ricerca dell'interpretazione avviene esaminando in profondità un tableau generato dall'insieme di formule e utilizzando la tecnica del *backtracking* (se un ramo chiude, quindi la ricerca di un modello lungo quel ramo fallisce, si esplora un'altra possibilità, cioè un altro ramo). La funzione riporta una lista di letterali, che rappresenta l'insieme di tutte le interpretazioni che assegnano *true* agli atomi presenti nella lista e $false$

agli atomi p tali che $\neg p$ è presente nella lista. Il valore di verità degli atomi p tali che né p né $\neg p$ è presente nella lista può essere qualsiasi.

```
exception NotSat;;

(* form list -> form list *)
let sat formlist =
  let rec aux pending lits =
    match pending with
      [] -> lits
    | f::rest ->
        match f with
          True | Not False -> aux rest lits
        | False | Not True -> raise NotSat
        | Prop _ | Not(Prop _) ->
            if List.mem f lits then aux rest lits
            else if List.mem (complement f) lits  then raise NotSat
            else aux rest (f::lits)
        | Not(Not f) -> aux (f::rest) lits
        | And(_,_) | Not(Or(_,_)) | Not(Imp(_,_)) ->
            let (f1,f2)= alpha f in
            aux (f1::f2::rest) lits
        | Or(_,_) | Not(And(_,_)) | Imp(_,_) ->
            let (f1,f2) = beta f in
            try aux (f1::rest) lits
            with NotSat -> aux (f2::rest) lits
   in aux formlist []
```

Possiamo allora, ad esempio, risolvere il problema di determinare cosa sono A e B se, nell'isola dei cavalieri e furfanti, A dice: “io sono un furfante oppure B e‘ un cavaliere”:

```
# sat [parse "A=> -A|B"; parse "-A=> -(-A|B)"];;
- : Language.form list = [Prop "A"; Prop "B"]
```

Nell'isola dei cavalieri e furfanti non è possibile che un abitante dica “io sono un furfante”:

```
# sat [parse "A=> -A"; parse "-A=> A"];;
Exception: NotSat.
```

La ricerca di modelli si può ancora modificare, in modo che la funzione riporti una lista che rappresenta tutti i modelli di un insieme di formule.

```
(* all_models: form list -> form list list *)
let all_models formlist =
  let rec aux pending lits =
    match pending with
      [] -> [lits]
    | f::rest ->
        match f with
          True | Not False -> aux rest lits
```

```
      | False | Not True -> []
      | Prop _ | Not(Prop _) ->
          if List.mem f lits then aux rest lits
          else if List.mem (complement f) lits  then []
          else aux rest (f::lits)
      | Not(Not f) -> aux (f::rest) lits
      | And(_,_) | Not(Or(_,_)) | Not(Imp(_,_)) ->
          let (f1,f2)= alpha f in
          aux (f1::f2::rest) lits
      | Or(_,_) | Not(And(_,_)) | Imp(_,_) ->
          let (f1,f2) = beta f in
          (aux (f1::rest) lits) @ (aux (f2::rest) lits)
   in aux formlist []
```

Infine, la funzione `all_models` può essere utilizzata per trasformare una formula in DNF: vengono collezionati tutti i rami aperti del tableau, per ciascuno di essi si costruisce la congiunzione dei letterali presenti nel ramo, ed infine si mettono in disgiunzione tutte le congiunzioni ottenute.

```
(* mkdnf : form form list -> form *)
let rec mkdnf = function
    [] -> True
  | [branch] -> mkand branch
  | branch::rest ->
      Or(mkand branch,mkdnf rest)

(* dnf_tab: form -> form *)
let dnf_tab f =
  mkdnf(all_models [f])
```

3.8 Esercizi

1. Definire una funzione `dnf`, analoga alla funzione `cnf` definita nel paragrafo 3.3, che trasformi una formula in una forma normale disgiuntiva equivalente.

2. Definire una funzione `mkor: form list -> form` che, applicata a una lista di formule, riporta la disgiunzione di tutte le formule nella lista. Se la lista è vuota, la funzione riporta `False`.

3. Una forma normale congiuntiva di una formula F si può ottenere mediante il metodo dei tableaux in modo analogo a quello con cui si ottiene una DFN: si raccolgono i complementi dei letterali dei rami aperti in un tableau completo per $\neg F$, ottenendo una lista di liste di letterali $[\texttt{flist}_1;\ldots;\texttt{flist}_n]$, e si costruisce la congiunzione di tutte le formule che si ottengono disgiungendo le formule in ciascuna delle $\texttt{flist}_i$ in `L`, cioè $\bigwedge_{i=1}^{n}(\bigvee_{A\in\texttt{flist}_i} A)$. Scrivere una funzione che riporti una forma normale disgiuntiva di una formula seguendo tale metodo.

4. Raffinare le funzioni che implementano il metodo dei tableaux, in modo tale che esse applichino la strategia di dilazionare finché possibile le ramificazioni (l'applicazione di β-regole).

Riferimenti bibliografici

[1] M. Cialdea Mayer and C. Limongelli. *Introduzione alla Programmazione Funzionale.* Esculapio, 2002.

[2] X. Leroy. The Objective Caml System, release 3.06. Documentation and user's manual. Disponibile su `http://caml.inria.fr/`, 2001.

[3] E. Mendelson. *Introduction to Mathematical Logic.* Van Nostrand, 1964. Trad. it. Boringhieri.

[4] M. Mondadori and M. D'Agostino. *Logica.* Bruno Mondadori, 1997.

[5] R. Smullyan. *What is the name of this book?* Prentice Hall, 1978. trad. it. Zanichelli.