dispense dell'insegnamento di

Programmazione Procedurale e Logica

Marco Bernardo Edoardo Bontà Alessandro Aldini

Università degli Studi di Urbino "Carlo Bo"
Corso di Laurea in Informatica Applicata

versione del 19/04/2018

Queste dispense sono state preparate con LaTeX e sono reperibili in formato PDF su `https://blended.uniurb.it/`.
Esse costituiscono soltanto un ausilio per il docente, quindi non sono in nessun modo sostitutive dei testi consigliati.

Si richiede di portare le dispense alle esercitazioni di laboratorio o in aula, ma non alle lezioni teoriche.
Quando non si capisce un argomento, fare domande a lezione o sul forum di Moodle e usufruire del ricevimento.

In aula e in laboratorio, seguire le attività in silenzio per non disturbare il docente e gli altri studenti.
Se si arriva tardi a lezione o si prevede di andare via in anticipo, sedersi nei posti vicini all'uscita.

Si consiglia di studiare durante tutto il periodo didattico, evitando di ridursi agli ultimi giorni prima dell'esame.
In ogni caso, è opportuno studiare i testi consigliati prima di svolgere il progetto d'esame, non dopo.

Indice

Capitolo 1

Introduzione alla programmazione degli elaboratori

1.1 Definizioni di base dell'informatica

- Informatica: disciplina che studia il trattamento automatico delle informazioni, comprendendo aspetti *scientifici* (Computer Science – teoria della calcolabilità, teoria della complessità computazionale, teoria degli automi e linguaggi formali), aspetti *metodologici* (Software Architecture & Engineering) e aspetti *tecnologici* (Information & Communication Technology).
- Computer o elaboratore elettronico: insieme di dispositivi elettromeccanici *programmabili* per l'immissione, la memorizzazione, l'elaborazione e l'emissione di informazioni sotto forma di numeri, testi, immagini, suoni e video.
- Hardware: insieme dei dispositivi elettromeccanici che costituiscono un computer.
- Software: insieme dei programmi eseguibili da un computer.
- L'hardware rappresenta l'insieme delle risorse di calcolo che abbiamo a disposizione, mentre il software rappresenta l'insieme delle istruzioni che impartiamo alle risorse di calcolo per svolgere certi compiti.
- Esempi di hardware: processore, memoria principale, memoria secondaria (dischi, nastri, ecc.), tastiera, mouse, schermo, stampante, modem.
- Esempi di software: sistema operativo, compilatore e debugger per un linguaggio di programmazione, strumento per la scrittura di testi, strumento per fare disegni, visualizzatore di documenti, visualizzatore di immagini, riproduttore di video e audio, foglio elettronico, sistema per la gestione di basi di dati, programma di posta elettronica, navigatore Internet.
- Impatto dell'informatica dal punto di vista socio-economico:
  - Trasferimento di attività ripetitive dalle persone alle macchine.
  - Capacità di effettuare calcoli complessi in tempi brevi.
  - Capacità di trattare grandi quantità di informazioni in tempi brevi.
  - Capacità di trasmettere notevoli quantità di informazioni in tempi brevi.

1.2 Cenni di storia dell'informatica

- Nel 1642 Pascal costruì una macchina meccanica capace di fare addizioni e sottrazioni.
- Nel 1672 Leibniz costruì una macchina meccanica capace di fare anche moltiplicazioni e divisioni (precursore delle calcolatrici tascabili a quattro funzioni).

- Nel 1834 Babbage progettò una macchina meccanica general purpose e programmabile, chiamata Analytical Engine, dotata delle quattro componenti – la sezione di input (lettore di schede perforate), il mulino (unità di computazione), il magazzino (memoria), la sezione di output (output perforato e stampato) – e delle istruzioni di base – addizione, sottrazione, moltiplicazione, divisione, trasferimento da/verso memoria e salto (in)condizionato – che troviamo in tutti i moderni computer. Il primo programmatore della storia fu Ada Byron, assunta da Babbage per programmare l'Analytical Engine.

- Nel 1935 Turing formalizzò il concetto intuitivo di algoritmo mediante un modello operazionale che oggi chiamiamo macchina di Turing. Inoltre introdusse l'idea di non costruire più macchine specializzate per scopi specifici, ma di costruire un'unica macchina in grado di svolgere tutti i compiti, che formalizzò attraverso un modello operazionale che oggi chiamiamo macchina di Turing universale. I modelli matematici di Turing sono tuttora alla base della teoria della calcolabilità e, a quel tempo, diedero luogo per la prima volta ad una visione del software – fino ad allora limitato alle schede perforate introdotte nel 1804 da Jacquard per controllare i telai – inteso come schema di computazione non più cablato nell'hardware, ma indipendente da esso in quanto rappresentato attraverso una delle enumerabili combinazioni di un insieme finito di simboli (i futuri linguaggi di programmazione).

- Nel 1943 il governo britannico fece costruire Colossus, il primo elaboratore elettronico, il quale aveva l'obiettivo specifico di decifrare i messaggi trasmessi in codice Enigma dalla marina tedesca. Turing partecipò alla sua progettazione contribuendo pertanto alla fine della seconda guerra mondiale.

- Nel 1946 Mauchley ed Eckert costruirono ENIAC, il primo grande elaboratore elettronico general purpose, grazie ad un finanziamento dell'esercito americano. Pesava 30 tonnellate ed occupava una stanza di 9 per 15 metri. Era composto da 18000 tubi a vuoto e 1500 relè. Aveva 20 registri, ciascuno dei quali poteva contenere un numero decimale a 10 cifre. Veniva programmato impostando 6000 interruttori e collegando una moltitudine di cavi.

- Nel 1952 Von Neumann costruì la macchina IAS. Influenzato dal modello della macchina di Turing universale, introdusse l'idea di *computer a programma memorizzato* (dati e programmi caricati insieme in memoria) al fine di evitare la programmazione attraverso l'impostazione di interruttori e cavi. Riconobbe inoltre che la pesante aritmetica decimale seriale usata in ENIAC poteva essere sostituita dall'aritmetica binaria parallela. Come nell'Analytical Engine, il cuore della macchina IAS era composto da un'unità di controllo e un'unità aritmetico-logica che interagivano con la memoria e i dispositivi di input/output.

- Nel periodo 1955-1965 vennero costruiti computer basati su transistor, che resero obsoleti i precedenti computer basati su tubi a vuoto. Elea 9003, prodotto in Italia dalla Olivetti nel 1957, fu il primo computer interamente basato su transistor, seguito da IBM 7090, DEC PDP-1 e PDP-8, CDC 6600. Inoltre P101, prodotto in Italia dalla Olivetti nel 1964, fu il primo personal computer della storia.

- Nel periodo 1965-1980 vennero costruiti computer basati su circuiti integrati, cioè circuiti nei quali era possibile includere decine di transistor. Questi computer, tra i quali si annoverano IBM 360 e DEC PDP-11, erano più piccoli, veloci ed economici dei loro predecessori. Nel 1969 nacque Internet.

- Dal 1980 i computer sono basati sulla tecnologia VLSI, che permette di includere milioni di transistor in un singolo circuito. Iniziò l'era dei personal computer, perché i costi di acquisto diventarono sostenibili anche dai singoli individui (citiamo IBM, Apple, Commodore, Amiga, Atari). Nello stesso tempo le reti di computer – su diverse scale geografiche – diventarono sempre più diffuse. Il continuo aumento del numero di transistor integrabili su un singolo circuito ha determinato il continuo aumento della velocità dei processori e della capacità delle memorie. Tali aumenti sono stati e sono tuttora necessari per far fronte alla crescente complessità delle applicazioni software e dei dati da trattare.

- A partire dal 1950 analoghe evoluzioni hanno avuto luogo nell'ambito dello sviluppo del software, con particolare riferimento a linguaggi e paradigmi di programmazione. Tra questi ultimi citiamo il paradigma imperativo di natura procedurale (Fortran, Cobol, Algol, Basic, Ada, C, Pascal) o di natura orientata agli oggetti (Simula, Smalltalk, C++, Java, C#) e il paradigma dichiarativo di natura funzionale (Lisp, ML, Caml, Scheme, Haskell) o di natura logica (Prolog).

1.3 Elementi di architettura degli elaboratori

- L'architettura – ispirata da Babbage, Turing e Von Neumann – di un moderno computer è riportata in Fig. 1.1. La conoscenza della sua struttura e del funzionamento dei suoi componenti è necessaria per comprendere come i programmi vengono eseguiti.
- Computer a programma memorizzato: sia i programmi che i dati devono essere caricati in memoria principale per poter essere elaborati, entrambi rappresentati come sequenze di cifre binarie. Cambiando il programma caricato in memoria principale, cambiano le operazioni effettuate dal computer.
- Passi per l'esecuzione di un programma in un computer a programma memorizzato:
  1. Il programma risiede in memoria secondaria, perché questa è una memoria non volatile.
  2. Il programma viene trasferito dalla memoria secondaria alla memoria principale per poter essere eseguito.
  3. Il programma viene eseguito dall'unità centrale di elaborazione. Prima il programma acquisisce i dati di ingresso dai dispositivi periferici di ingresso e/o dalla memoria, poi produce i dati di uscita sui dispositivi periferici di uscita e/o sulla memoria.

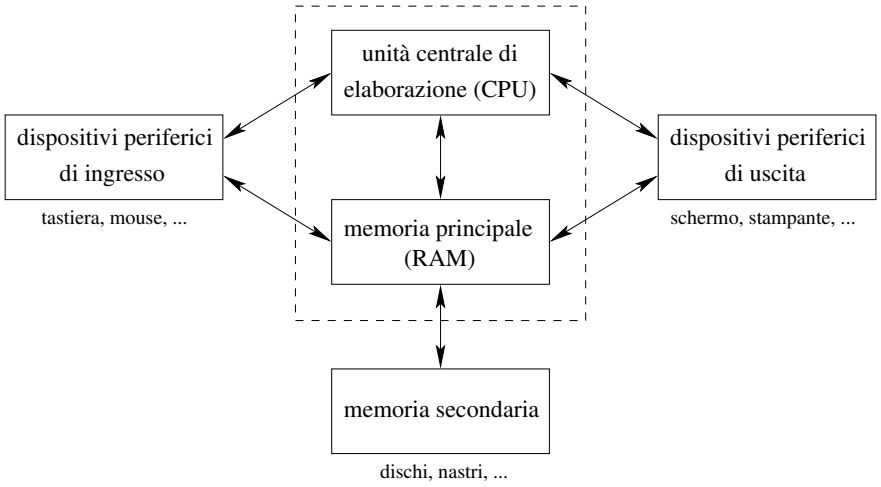


Figura 1.1: Architettura di un computer

- Dal punto di vista di un programma:
  - L'unità centrale di elaborazione è la risorsa che lo esegue.
  - La memoria è la risorsa che lo contiene.
  - I dispositivi periferici di ingresso/uscita sono le risorse che acquisiscono i suoi dati di ingresso e comunicano i suoi dati di uscita.
- L'unità centrale di elaborazione (CPU – central processing unit) svolge i seguenti due compiti:
  - Coordinare tutte le operazioni del computer a livello hardware.
  - Eseguire le operazioni aritmetico-logiche sui dati.
- La CPU è composta dai seguenti dispositivi:
  - Un'unità di controllo (CU – control unit) che, sulla base delle istruzioni di un programma caricato in memoria principale, determina quali operazioni eseguire in quale ordine, trasmettendo gli opportuni segnali di controllo alle altre componenti hardware del computer.
  - Un'unità aritmetico-logica (ALU – arithmetic-logic unit) che esegue le operazioni aritmetico-logiche segnalate dall'unità di controllo sui dati stabiliti dall'istruzione corrente.

  - Un insieme di registri che contengono l'istruzione corrente, i dati correnti e altre informazioni. Questi registri sono piccole unità di memoria che, diversamente dalla memoria principale, hanno una velocità di accesso confrontabile con la velocità dell'unità di controllo.

- Dal punto di vista di un programma caricato in memoria principale, la CPU compie il seguente ciclo di esecuzione dell'istruzione (ciclo fetch-decode-execute):

  1. Trasferire dalla memoria principale all'instruction register (IR) la prossima istruzione da eseguire, la quale è indicata dal registro program counter (PC).
  2. Aggiornare il PC per farlo puntare all'istruzione successiva a quella caricata nell'IR.
  3. Interpretare l'istruzione contenuta nell'IR per determinare quali operazioni eseguire.
  4. Trasferire dalla memoria principale ad appositi registri i dati sui quali opera l'istruzione contenuta nell'IR.
  5. Eseguire le operazioni determinate dall'istruzione contenuta nell'IR (si tratta di una modifica del PC in caso di salto).
  6. Ritornare all'inizio del ciclo.

- La memoria è una sequenza di locazioni chiamate celle, ognuna delle quali ha un indirizzo univoco. Il numero di celle è solitamente 2^n, con indirizzi compresi tra 0 e $2^n - 1$ che possono essere rappresentati tramite n cifre binarie.

- Il contenuto di una cella di memoria è una sequenza di cifre binarie chiamate bit (bit = binary digit). Il numero di bit contenuti in una cella è detto dimensione della cella. Una cella di dimensione d può contenere uno fra 2^d valori diversi, compresi fra 0 e $2^d - 1$.

- L'unità di misura della capacità di memoria è il byte (byte = binary octette), dove 1 byte = 8 bit. I suoi multipli più comunemente usati sono Kbyte, Mbyte e Gbyte, dove: 1 Kbyte = 2^{10} byte, 1 Mbyte = 2^{20} byte, 1 Gbyte = 2^{30} byte.

- Il sistema di numerazione in base 2 è particolarmente adeguato per la rappresentazione delle informazioni nella memoria di un computer. I due valori 0 ed 1 sono infatti facilmente associabili a due diverse gamme di valori di tensione come pure a due polarizzazioni opposte in un campo magnetico.

- La memoria principale è un dispositivo di memoria con le seguenti caratteristiche:

  - Accesso casuale (RAM – random access memory): il tempo di accesso ad una cella non dipende dalla sua posizione fisica.
  - Volatile: il contenuto viene perso quando cessa l'erogazione di energia elettrica.

- La memoria secondaria è un dispositivo di memoria con le seguenti caratteristiche:

  - Accesso sequenziale: il tempo di accesso ad una cella dipende dalla sua posizione fisica.
  - Permanente: il contenuto viene mantenuto anche quando cessa l'erogazione di energia elettrica.

  Tipici supporti secondari sono dischi magnetici (rigidi e floppy), nastri magnetici, compact disk e chiavette USB.

- La gerarchia di memoria "registri-principale-secondaria" è caratterizzata da:

  - Velocità decrescenti.
  - Capacità crescenti.
  - Costi decrescenti.

- Dal punto di vista dei programmi:

  - La memoria secondaria li contiene tutti.
  - La memoria principale contiene il programma da eseguire ora.
  - I registri contengono l'istruzione di programma da eseguire ora. ■fltpp_1

1.4 Elementi di sistemi operativi

- Il primo strato di software che viene installato su un computer è il sistema operativo. Esso è un insieme di programmi che gestiscono l'interazione degli utenti del computer con le risorse hardware del computer stesso. Anche la conoscenza della struttura del sistema operativo e del funzionamento dei suoi componenti è necessaria per comprendere come i programmi vengono eseguiti.
- Negli anni 1960 esistevano pochissimi centri di calcolo, ognuno dotato di un computer molto costoso e di un operatore a cui gli utenti lasciavano i propri programmi e dati per l'esecuzione (modalità batch). Il ruolo dei sistemi operativi divenne fondamentale con l'avvento dei sistemi di elaborazione:
  - multiprogrammati, in cui più programmi (anziché uno solo) possono risiedere contemporaneamente in memoria principale pronti per l'esecuzione, così da incrementare l'utilizzo del processore in caso di frequenti richieste di input/output da parte del programma in esecuzione;
  - time sharing, in cui più utenti condividono le risorse hardware del centro di calcolo interagendo con il sistema di elaborazione tramite apposite postazioni di lavoro, così da stabilire un dialogo più diretto e rapido tra utente e macchina (modalità interattiva).
- In un ambiente multiprogrammato o time sharing, è necessaria la presenza di un sistema operativo per coordinare le risorse del computer al fine di eseguire correttamente i comandi impartiti dagli utenti direttamente o da programma.
- Obiettivi di un sistema operativo:
  - Dal punto di vista prestazionale, il sistema operativo deve garantire tempi di esecuzione accettabili per i vari programmi caricati in memoria che devono essere eseguiti, nonché percentuali di utilizzo adeguate per le varie risorse del computer.
  - Dal punto di vista dell'usabilità, il sistema operativo deve creare un ambiente di lavoro amichevole per gli utenti, al fine di incrementare la loro produttività e il loro grado di soddisfazione.
- Compiti specifici di un sistema operativo:
  - Schedulare i programmi: in che ordine eseguire i programmi pronti per l'esecuzione?
  - Gestire la memoria principale: in che modo caricare i programmi pronti per l'esecuzione?
  - Gestire la memoria secondaria: in che ordine evadere le richieste di accesso?
  - Gestire i dispositivi periferici: in che ordine evadere le richieste di servizio?
  - Trattare i file: come organizzarli sui supporti di memoria secondaria?
  - Preservare l'integrità (errori interni) e la sicurezza (attacchi esterni) dei dati.
  - Acquisire comandi dagli utenti ed eseguirli:
    - Interfaccia a linea di comando: Unix, Microsoft DOS, Linux.
    - Interfaccia grafica dotata di finestre, icone, menù e puntatore (approccio WIMP): Mac OS, Microsoft Windows, Unix con ambiente XWindow, Linux con ambienti KDE o Gnome.

1.5 Elementi di linguaggi di programmazione e compilatori

- La programmazione di un computer richiede un linguaggio in cui esprimere i programmi.
- Ogni computer mette a disposizione degli utenti due linguaggi di programmazione:
  - Linguaggio macchina: ogni istruzione è costituita dal codice operativo espresso in binario (addizione, sottrazione, moltiplicazione, divisione, trasferimento dati o salto istruzioni) e dai valori o dagli indirizzi degli operandi espressi in binario. È direttamente interpretabile dal computer.
  - Linguaggio assemblativo: ogni istruzione è costituita dal codice operativo espresso in formato mnemonico e dai valori o dagli indirizzi degli operandi espressi in formato mnemonico. Mantiene dunque una corrispondenza uno-a-uno con il linguaggio macchina, ma necessita di un traduttore detto assemblatore per rendere i suoi programmi eseguibili.

- Inconvenienti dei linguaggi macchina e assemblativo:
  - Basso livello di astrazione: è difficile scrivere programmi in questi linguaggi.
  - Dipendenza dall'architettura di un particolare computer: i programmi espressi in questi linguaggi non sono portabili su computer diversi, cioè non possono essere eseguiti su altri computer aventi un'architettura diversa.
- Caratteristiche dei linguaggi di programmazione di alto livello (sviluppati a partire dal 1955 con Fortran, per le applicazioni scientifiche, e Cobol, per le applicazioni gestionali):
  - Alto livello di astrazione: è più facile scrivere programmi in questi linguaggi perché sono più vicini al linguaggio naturale delle persone e alla notazione matematica di uso comune.
  - Indipendenza dall'architettura di un particolare computer: i programmi espressi in questi linguaggi sono portabili su computer diversi.
  - Necessità di un traduttore nel linguaggio macchina o assemblativo dei vari computer: una singola istruzione di alto livello corrisponde ad una sequenza di più istruzioni di basso livello. Il traduttore è chiamato compilatore o interprete a seconda che la traduzione venga effettuata separatamente dall'esecuzione del programma da tradurre o contestualmente a quest'ultima.
- Il compilatore di un linguaggio di programmazione di alto livello per un particolare computer traduce i programmi (comprensibili dagli umani) espressi nel linguaggio di alto livello in programmi equivalenti (eseguibili dal computer) espressi nel linguaggio macchina o assemblativo del computer.
- Componenti di un compilatore:
  - Analizzatore lessicale: organizza la sequenza di simboli presenti nel programma di alto livello in lessemi (come ad esempio parole riservate, identificatori, numeri e simboli specifici) e segnala le sottosequenze di caratteri che non formano lessemi legali del linguaggio.
  - Analizzatore sintattico: organizza la sequenza precedentemente riconosciuta di lessemi in frasi sulla base delle regole grammaticali del linguaggio (relative per esempio a istruzioni, espressioni, dichiarazioni e direttive) e segnala le sottosequenze di lessemi che violano tali regole.
  - Analizzatore semantico: mediante un sistema di tipi verifica se le varie parti del programma di alto livello hanno un significato compiuto (ad esempio non ha senso scrivere in un programma la somma tra un numero e un vettore di numeri).
  - Generatore di codice: traduce il programma di alto livello in un programma equivalente espresso in linguaggio macchina o assemblativo.
  - Ottimizzatore di codice: modifica le istruzioni macchina o assemblative del programma precedentemente generato al fine di ridurne il tempo di esecuzione e/o la dimensione senza però alterarne la semantica.
- Procedimento di scrittura, compilazione, linking, caricamento ed esecuzione di un programma:
  1. Il programma viene scritto in un linguaggio di programmazione di alto livello e poi memorizzato in un file detto file sorgente.
  2. Il file sorgente viene compilato. Se non ci sono errori lessicali, sintattici e semantici, si prosegue con il passo successivo, altrimenti si torna al passo precedente. In caso di successo viene prodotto un file detto file oggetto.
  3. Il file oggetto viene collegato con altri eventuali file oggetto contenenti parti di codice richiamate nel programma originario ma non ivi definite. Il risultato è un file detto file eseguibile.
  4. Il file eseguibile viene caricato in memoria principale.
  5. Il file eseguibile viene eseguito sulla CPU – la quale ripete il ciclo fetch-decode-execute per ogni istruzione – sotto la supervisione del sistema operativo – il quale coordina l'esecuzione di tutti i file eseguibili caricati in memoria principale.

- Il primo passo è svolto tramite uno strumento per la scrittura di testi. Il secondo e il terzo passo sono svolti dal compilatore. Il quarto e il quinto passo hanno luogo all'atto del lancio in esecuzione del programma. Esiste un ulteriore passo, detto debugging, che si rende necessario qualora si manifestino errori durante l'esecuzione del programma (sono errori di malfunzionamento, da non confondere con quelli linguistici rilevati dal compilatore). Tale passo consiste nell'individuazione delle cause degli errori nel file sorgente, nella loro rimozione dal file sorgente e nella ripetizione dei passi 2, 3, 4 e 5 per il file sorgente modificato.

1.6 Una metodologia di sviluppo software "in the small"

- I passi precedentemente indicati devono essere preceduti da una adeguata attività di progettazione. Questo perché la programmazione non può essere improvvisata – soprattutto quando si ha a che fare con lo sviluppo di applicazioni software complesse in gruppi di lavoro – ma deve essere guidata da una metodologia ben precisa.

- Noi utilizzeremo una metodologia di sviluppo software "in the small" (cioè per programmi di piccole dimensioni) che è composta dalle seguenti fasi:

  1. **Specifica del problema**. Enunciare il problema in maniera chiara e precisa. Questa fase richiede solitamente diverse interazioni con chi ha posto il problema, al fine di evidenziare gli aspetti rilevanti del problema stesso.
  2. **Analisi del problema**. Individuare i dati di ingresso (che verranno forniti) e i dati di uscita (che dovranno essere prodotti) risultanti dalla specifica del problema, assieme alle principali relazioni intercorrenti tra di essi da sfruttare ai fini della soluzione del problema, astraendo dagli aspetti algoritmici che verranno successivamente progettati così come dal linguaggio implementativo.
  3. **Progettazione dell'algoritmo**. Nel contesto del problema specificato e della sua analisi, illustrare e motivare le principali scelte di progetto (rappresentazione degli input e degli output, idea alla base della soluzione, strutture dati utilizzate, ecc.) e riportare i principali passi con eventuali raffinamenti dell'algoritmo creato per risolvere il problema, astraendo dallo specifico linguaggio di programmazione di alto livello che verrà impiegato per l'implementazione.
  4. **Implementazione dell'algoritmo**. Tradurre l'algoritmo nel prescelto linguaggio di programmazione di alto livello (passi di scrittura, compilazione e linking del programma).
  5. **Testing del programma**. Effettuare diversi test significativi di esecuzione completa del programma, riportando fedelmente per ciascun test sia i dati di ingresso introdotti che i corrispondenti risultati ottenuti (passi di caricamento ed esecuzione del programma). Se alcuni test danno risultati diversi da quelli attesi, ritornare alle fasi precedenti per correggere gli errori di malfunzionamento rilevati (passo di debugging del programma).
  6. **Manutenzione del programma**. Modificare il programma dopo la sua distribuzione qualora errori d'esecuzione, prestazioni scadenti, falle di sicurezza o limiti di usabilità vengano riscontrati dagli utenti del programma (manutenzione correttiva), come pure nel caso si rendano necessari degli aggiornamenti a seguito di nuove esigenze degli utenti o sviluppi tecnologici (manutenzione evolutiva). Questa fase richiede l'adozione nelle fasi precedenti di un appropriato stile di programmazione e la produzione di un'adeguata documentazione interna (commenti) ed esterna (relazione tecnica e manuale d'uso) per il programma.

- Le peggiori cose che un programmatore possa fare sono scrivere un programma senza averlo prima progettato e non produrre alcuna documentazione oppure produrla tutta solo alla fine. ■fltpp_2

Capitolo 2

Programmazione procedurale: il linguaggio ANSI C

2.1 Cenni di storia del C

- La nascita e l'evoluzione del linguaggio C sono legate a quelle del sistema operativo Unix:
  - Nel 1969 Thompson e Ritchie implementarono la prima versione di Unix per il DEC PDP-11 presso i Bell Lab della AT&T.
  - Nel 1972 Ritchie mise a punto il linguaggio di programmazione C, il cui scopo era di consentire agli sviluppatori di Unix di implementare e sperimentare le loro idee.
  - Nel 1973 Thompson e Ritchie riscrissero il nucleo di Unix in C. Da allora tutte le chiamate di sistema di Unix vennero definite come funzioni C. Ciò rese di fatto il C il linguaggio da utilizzare per scrivere programmi applicativi in ambiente Unix.
  - Nel 1974 i Bell Lab cominciarono a distribuire Unix alle università, quindi il C iniziò a diffondersi all'interno di una comunità più ampia.
  - Nel 1976 venne implementata la prima versione portabile di Unix, incrementando di conseguenza la diffusione del C.
  - Nel 1983 l'ANSI (American National Standard Institute) nominò un comitato per stabilire una definizione del linguaggio C che fosse non ambigua e indipendente dal computer.
  - Nel 1988 il comitato concluse i suoi lavori con la produzione del linguaggio ANSI C.
- Il C è un linguaggio di programmazione:
  - di alto livello di astrazione, quindi i programmi C necessitano di essere compilati prima della loro esecuzione e sono portabili su computer diversi;
  - general purpose, cioè non legato a nessun ambito applicativo in particolare, anche se esso mantiene una certa vicinanza al basso livello di astrazione (data la sua storia) che lo rende particolarmente adeguato per lo sviluppo di sistemi software di base come sistemi operativi e compilatori;
  - *imperativo* di natura *procedurale*, in quanto i programmi C sono sequenze di istruzioni:
    - che prescrivono come modificare il contenuto di locazioni di memoria;
    - raggruppate in blocchi detti procedure dotati di parametri impostabili di volta in volta.
- I programmi C si compongono di espressioni matematiche e di istruzioni imperative raggruppate in procedure parametrizzate che manipolano dati di vari tipi.
- Sebbene il C sia nato con Unix, esso è ormai supportato da tutti i sistemi operativi di largo uso.

2.2 Formato di un programma con una singola funzione

- Questo è il formato più semplice che un programma C può avere:

```
◁direttive al preprocessore▷
◁intestazione della funzione main▷
{
   ◁dichiarazioni▷
   ◁istruzioni▷
}
```

- L'intestazione della funzione contiene nome, parametri e tipo del risultato della funzione stessa, mentre ciò che è racchiuso tra parentesi graffe costituisce il corpo della funzione.
- Esempio di programma: conversione di miglia in chilometri.
  1. **Specifica del problema.** Convertire una distanza espressa in miglia nell'equivalente distanza espressa in chilometri.
  2. **Analisi del problema.** Per questo semplice problema, l'input è rappresentato dalla distanza in miglia, mentre l'output è rappresentato dalla distanza equivalente in chilometri. La relazione fondamentale da sfruttare è 1 mi = 1.609 km.
  3. **Progettazione dell'algoritmo.** Data la semplicità del problema, non ci sono particolari scelte di progetto da compiere. I passi dell'algoritmo sono i seguenti:
     - Acquisire la distanza in miglia.
     - Convertire la distanza in chilometri.
     - Comunicare la distanza in chilometri.
  4. **Implementazione dell'algoritmo**. Questa è la traduzione dei passi in C:

```
/*****************************************************************************/
/* programma per la conversione di miglia in chilometri (versione interattiva) */
/*****************************************************************************/

/*****************************/
/* inclusione delle librerie */
/*****************************/

#include <stdio.h>

/*****************************************/
/* definizione delle costanti simboliche */
/*****************************************/

#define KM_PER_MI 1.609   /* fattore di conversione */

/***********************************/
/* definizione della funzione main */
/***********************************/

int main(void)
{
  /* dichiarazione delle variabili locali alla funzione */
  double miglia,        /* input: distanza in miglia */
         chilometri;    /* output: distanza in chilometri */

  /* acquisire la distanza in miglia */
  printf("Digita la distanza in miglia: ");
  scanf("%lf",
        &miglia);

  /* convertire la distanza in chilometri */
  chilometri = KM_PER_MI * miglia;
```

```
  /* comunicare la distanza in chilometri */
  printf("La stessa distanza in chilometri e': %f\n",
         chilometri);
  return(0);
}
```

- Notare l'indentazione (cioè la sequenza di spazi iniziali che determinano un rientro verso destra) rispetto alle parentesi graffe, introdotta per favorire la leggibilità del corpo della funzione.
- Ciò che è racchiuso tra `/*` e `*/` costituisce dei commenti interni al programma (che verranno ignorati dal compilatore) usati per esplicitare il legame tra identificatori di costanti e variabili e input e output del problema e loro relazioni, oppure tra passi dell'algoritmo e sequenze di istruzioni che li implementano.

2.3 Inclusione di libreria

- La direttiva di inclusione di libreria:
  `#include <◁`*file di intestazione di libreria standard*`▷>`
  oppure:
  `#include "◁`*file di intestazione di libreria del programmatore*`▷"`
  imparte al preprocessore C (parte iniziale del compilatore) il comando di sostituire nel testo del programma la direttiva stessa con il contenuto del file di intestazione, così da permettere l'utilizzo di identificatori di costanti simboliche, tipi, variabili e funzioni definiti/dichiarati nella libreria.

2.4 Funzione `main`

- Ogni programma C deve contenere almeno la funzione `main`, il cui corpo (racchiuso tra parentesi graffe) si compone di una sezione di dichiarazioni di variabili locali e di una sezione di istruzioni che manipolano tali variabili.
- La prima istruzione che viene eseguita in un programma C è la prima istruzione della funzione `main`. Le rimanenti istruzioni vengono poi eseguite una alla volta nell'ordine in cui sono state scritte.

2.5 Identificatori

- Gli identificatori del linguaggio C sono sequenze di lettere, cifre decimali e sottotratti che non iniziano con una cifra decimale. Le lettere minuscole sono considerate diverse dalle corrispondenti lettere maiuscole (case sensitivity).
- Gli identificatori denotano nomi di costanti simboliche (p.e. `KM_PER_MI`), tipi (p.e. `int`, `void`, `double`), variabili (p.e. `miglia`, `chilometri`), funzioni (p.e. `main`, `printf`, `scanf`) e istruzioni.
- Gli identificatori si dividono in riservati (p.e. `int`, `void`, `double`, `return`), standard (p.e. `printf`, `scanf`) e introdotti dal programmatore (p.e. `KM_PER_MI`, `miglia`, `chilometri`).
- I seguenti identificatori riservati sono predefiniti dal linguaggio C e hanno un significato fissato, quindi all'interno dei programmi essi non sono utilizzabili per scopi diversi da quelli stabiliti dal linguaggio:

`auto`	`double`	`int`	`struct`
`break`	`else`	`long`	`switch`
`case`	`enum`	`register`	`typedef`
`char`	`extern`	`return`	`union`
`const`	`float`	`short`	`unsigned`
`continue`	`for`	`signed`	`void`
`default`	`goto`	`sizeof`	`volatile`
`do`	`if`	`static`	`while`

- Gli identificatori standard sono definiti all'interno delle librerie standard del linguaggio C, ma possono essere ridefiniti e usati per altri scopi, anche se ciò è fortemente sconsigliato.

- Gli identificatori introdotti dal programmatore sono tutti gli altri identificatori che compaiono all'interno di un programma. Ecco alcune regole prescrittive e stilistiche per scegliere buoni identificatori:
  - Devono essere diversi dagli identificatori riservati.
  - È consigliabile che siano diversi anche dagli identificatori standard.
  - È consigliabile dare nomi significativi che ricordino ciò che gli identificatori rappresentano.
  - È consigliabile usare sottotratti quando un identificatore è composto da più (abbreviazioni di) parole.
  - È consigliabile evitare identificatori troppo lunghi o molto simili, in quanto questi aumentano la probabilità di commettere errori di scrittura del programma non rilevabili dal compilatore.
  - È una convenzione comunemente seguita in C quella di usare lettere tutte maiuscole negli identificatori di costanti simboliche e lettere tutte minuscole in tutti gli altri identificatori.

2.6 Tipi di dati predefiniti: `int`, `double`, `char`

- Ogni identificatore di costante simbolica, variabile o funzione ha un tipo ad esso associato, il quale stabilisce – come se fosse una struttura algebrica – l'insieme di valori che l'identificatore può assumere e l'insieme di operazioni nelle quali l'identificatore può essere coinvolto. Il linguaggio C mette a disposizione tre tipi predefiniti: `int`, `double`, `char`.
- Il tipo `int` denota l'insieme dei numeri interi rappresentabili con un certo numero di bit (sottoinsieme finito di $\mathbb{Z}$). Essi vengono espressi tramite la notazione consueta composta da segno e valore assoluto.
- Il tipo `double` denota l'insieme dei numeri reali rappresentabili con un certo numero di bit (sottoinsieme finito di $\mathbb{R}$). Vengono espressi in virgola fissa (p.e. `13.72`) o in virgola mobile (p.e. `0.1372e2`). Ogni numero in virgola fissa è considerato di tipo `double` anche se la parte frazionaria è nulla (p.e. `-13.0`).
- Il tipo `char` denota l'insieme dei caratteri, i quali vengono espressi racchiusi tra apici (p.e. `'a'`). Tale insieme comprende le 26 lettere minuscole, le 26 lettere maiuscole, le 10 cifre decimali, i simboli di punteggiatura, le varie parentesi, gli operatori aritmetici e relazionali e i caratteri di spaziatura (spazio, tabulazione, andata a capo).
- Un'estensione del tipo `char` è il tipo stringa. Esso denota l'insieme delle sequenze finite di caratteri, ciascuna espressa racchiusa tra virgolette (p.e. `"ciao"`).
- Esiste inoltre il tipo `void`, che viene usato per rappresentare in una funzione l'assenza di parametri o l'assenza di risultati da restituire. ■fltpp_3

2.7 Funzioni di libreria per l'input/output interattivo

- In modalità interattiva, un programma C dialoga con l'utente durante la sua esecuzione acquisendo dati tramite tastiera e comunicando risultati tramite schermo. Il file di intestazione di libreria standard che mette a disposizione le relative funzioni è `stdio.h`.
- La funzione di libreria standard per acquisire dati tramite tastiera ha la seguente sintassi:
  `scanf(`◁*stringa formato*$_s$▷`,`
  ◁*sequenza indirizzi variabili*▷`)`
  dove:
  - *stringa formato*$_s$ è una sequenza non vuota racchiusa tra virgolette dei seguenti segnaposto:
    - `%d` per un valore di tipo `int`;
    - `%lf` per un valore di tipo `double` in virgola fissa;
    - `%le` per un valore di tipo `double` in virgola mobile;
    - `%lg` per un valore di tipo `double` in virgola fissa o mobile;
    - `%c` per un valore di tipo `char` (usare la funzione `getchar` per acquisire un solo carattere);
    - `%s` per un valore di tipo stringa (scrivere `%`◁*numero*▷`s` per limitare il numero di caratteri).

  - *sequenza indirizzi variabili* è una sequenza non vuota di indirizzi di variabili separati da virgola (tali indirizzi sono solitamente ottenuti applicando l'operatore "`&`" agli identificatori delle variabili, vedi Sez. 5.7).
  - L'ordine, il tipo e il numero dei segnaposto nella *stringa formato*$_s$ devono corrispondere all'ordine, al tipo e al numero delle variabili nella *sequenza indirizzi variabili*.

- La semantica della funzione `scanf`, cioè il suo effetto a tempo d'esecuzione, è di acquisire i valori da tastiera e memorizzarli da sinistra a destra nelle variabili della *sequenza indirizzi variabili*:

  - L'introduzione di un valore della sequenza tramite tastiera avviene premendo i tasti corrispondenti e viene terminata (anche nel caso di un valore di tipo carattere o stringa) premendo la barra spaziatrice, il tabulatore o il tasto invio. L'introduzione dell'ultimo (o unico) valore della sequenza deve necessariamente essere terminata premendo il tasto invio.
  - L'acquisizione di un valore di tipo numerico o stringa avviene saltando eventuali spazi, tabulazioni e andate a capo precedentemente digitati. Ciò accade anche per l'acquisizione di un valore di tipo carattere solo se il corrispondente segnaposto `%c` è preceduto da uno spazio nella *stringa formato*$_s$ (nessun salto è possibile quando si usa la funzione `getchar`).

- La funzione `scanf` restituisce il numero di valori acquisiti correttamente rispetto ai segnaposto specificati nella *stringa formato*$_s$ (mentre la funzione `getchar` restituisce il carattere acquisito), il che è molto utile per la validazione stretta degli input (vedi Sez. 4.4). Ulteriori valori eventualmente introdotti prima dell'andata a capo finale sono considerati come non ancora acquisiti.

- La funzione di libreria standard per stampare su schermo ha la seguente sintassi:
  `printf(`◁*stringa formato*$_p$▷`,`
  `        `◁*sequenza espressioni*▷`)`
  dove:

  - *stringa formato*$_p$ è una sequenza non vuota racchiusa tra virgolette di caratteri tra i quali possono comparire i seguenti segnaposto:
    * `%d` per un valore di tipo `int`;
    * `%f` per un valore di tipo `double` in virgola fissa;
    * `%e` per un valore di tipo `double` in virgola mobile;
    * `%g` per un valore di tipo `double` in virgola fissa o mobile;
    * `%c` per un valore di tipo `char` (usare la funzione `putchar` per stampare un solo carattere);
    * `%s` per un valore di tipo stringa;

    assieme ai seguenti caratteri speciali:
    * `\n` per un'andata a capo;
    * `\t` per una tabulazione.
  - *sequenza espressioni* è una sequenza eventualmente vuota di espressioni separate da virgola.
  - L'ordine, il tipo e il numero dei segnaposto nella *stringa formato*$_p$ devono corrispondere all'ordine, al tipo e al numero delle espressioni nella *sequenza espressioni*.

- All'atto dell'esecuzione, la funzione `printf` stampa su schermo *stringa formato*$_p$ dopo aver sostituito da sinistra a destra gli eventuali segnaposto con i valori delle espressioni nella *sequenza espressioni* e dopo aver tradotto gli eventuali caratteri speciali (la funzione `putchar` stampa il carattere specificato).

- Formattazione per la stampa allineata di valori di tipo `int`:

  - `%`◁*numero*▷`d` specifica che il valore deve essere stampato su *numero* colonne, includendo l'eventuale segno negativo.
  - Se il valore ha meno cifre delle colonne, esso viene allineato a destra con spazi a precedere se positivo, allineato a sinistra con spazi a seguire se negativo.
  - Se il valore ha più cifre delle colonne, la formattazione viene ignorata.

- Formattazione per la stampa allineata di valori di tipo `double` in virgola fissa:
  - `%`◁*numero*$_1$▷`.`◁*numero*$_2$▷`f` specifica che il valore deve essere stampato su *numero*$_1$ colonne, includendo il punto decimale e l'eventuale segno negativo, delle quali *numero*$_2$ sono riservate alla parte frazionaria (*numero*$_1$ può essere omesso).
  - Se il valore ha meno cifre nella parte intera, esso viene stampato allineato a destra con spazi a precedere.
  - Se il valore ha più cifre nella parte intera, *numero*$_1$ viene ignorato.
  - Se il valore ha meno cifre nella parte frazionaria, vengono aggiunti degli zeri a seguire.
  - Se il valore ha più cifre nella parte frazionaria, questa viene arrotondata.
- Formattazione per la stampa allineata di valori di tipo stringa:
  - `%`◁*numero*▷`s` con *numero* positivo specifica che il valore deve essere stampato su almeno *numero* colonne, con allineamento a sinistra se la lunghezza del valore è minore di *numero*.
  - `%`◁*numero*▷`s` con *numero* negativo specifica che il valore deve essere stampato su almeno -*numero* colonne, con allineamento a destra se la lunghezza del valore è minore di -*numero*.

2.8 Funzioni di libreria per l'input/output tramite file

- In modalità batch, un programma C acquisisce dati tramite file preparati dall'utente prima dell'esecuzione e comunica risultati tramite altri file che l'utente consulterà dopo l'esecuzione. Il file di intestazione di libreria standard che mette a disposizione le relative funzioni è `stdio.h`.
- I file possono essere gestiti direttamente all'interno del programma come segue:
  - Dichiarare una variabile di tipo standard puntatore a file per ogni file da gestire:
    `FILE *`◁*variabile file*▷`;`
    Tale variabile conterrà l'indirizzo di un'area di memoria detta buffer del file, in cui verranno temporaneamente memorizzate le informazioni lette dal file o da scrivere sul file.
  - Aprire ogni file per creare una corrispondenza tra la variabile precedentemente dichiarata – che ne rappresenta il nome logico – e il suo nome fisico all'interno del file system del sistema operativo. L'operazione di apertura specifica se il file deve essere letto (nel qual caso il file deve esistere):
    ◁*variabile file*▷ `= fopen("`◁*nome file*▷`",`
    `"r");`
    oppure scritto (nel qual caso il precedente contenuto del file, se esistente, viene perso):
    ◁*variabile file*▷ `= fopen("`◁*nome file*▷`",`
    `"w");`
    Se il tentativo di apertura del file fallisce, il risultato della funzione `fopen` che viene assegnato alla variabile è il valore della costante simbolica standard `NULL`.
  - La funzione di libreria standard per leggere da un file aperto in lettura ha la seguente sintassi:
    `fscanf(`◁*variabile file*▷`,`
    ◁*stringa formato*$_s$▷`,`
    ◁*sequenza indirizzi variabili*▷`)`
    Se `fscanf` incontra il carattere di terminazione file, il risultato che essa restituisce è il valore della costante simbolica standard `EOF`.
  - La funzione di libreria standard per verificare se è stata raggiunta la fine di un file aperto in lettura ha la seguente sintassi:
    `feof(`◁*variabile file*▷`)`

  - La funzione di libreria standard per scrivere su un file aperto in scrittura ha la seguente sintassi:
    `fprintf(`◁*variabile file*▷`,`
    ◁*stringa formato*$_p$▷`,`
    ◁*sequenza espressioni*▷`)`
  - Chiudere ogni file dopo l'ultima operazione di lettura/scrittura su di esso al fine di rilasciare il relativo buffer e rendere definitive eventuali modifiche del contenuto del file:
    `fclose(`◁*variabile file*▷`)`
  - Il numero di file che possono rimanere aperti contemporaneamente durante l'esecuzione del programma è limitato e coincide con il valore della costante simbolica standard `FOPEN_MAX`. Se all'atto della terminazione del programma ci sono ancora dei file aperti, questi vengono automaticamente chiusi dal sistema operativo.

- In alternativa, se il programma opera in modalità batch su un solo file di input e un solo file di output, i due file possono essere specificati mediante il meccanismo di ridirezione nel comando con il quale il programma viene lanciato in esecuzione:
  ◁*file eseguibile*▷ < ◁*file input*▷ > ◁*file output*▷
  In questo caso, bisogna usare le stesse funzioni di libreria standard illustrate in Sez. 2.7.

- Se nessuno dei due precedenti meccanismi viene usato, si intende che i dati vengano letti dal file `stdin` associato alla tastiera (come fa `scanf`) e che i risultati vengano scritti sul file `stdout` associato allo schermo (come fa `printf`). Questi due file, assieme al file `stderr` sul quale vengono riportati eventuali messaggi di errore a tempo di esecuzione, sono messi a disposizione dal file di intestazione di libreria standard `stdio.h`.

- Esempio di programma: conversione di miglia in chilometri usando file.

```
/*************************************************************************/
/* programma per la conversione di miglia in chilometri (versione batch) */
/*************************************************************************/

/*****************************/
/* inclusione delle librerie */
/*****************************/

#include <stdio.h>

/***************************************/
/* definizione delle costanti simboliche */
/***************************************/

#define KM_PER_MI 1.609   /* fattore di conversione */

/***********************************/
/* definizione della funzione main */
/***********************************/

int main(void)
{
  /* dichiarazione delle variabili locali alla funzione */
  double miglia,             /* input: distanza in miglia */
         chilometri;         /* output: distanza in chilometri */
  FILE   *file_miglia,       /* lavoro: puntatore al file di input */
         *file_chilometri;   /* lavoro: puntatore al file di output */
```

```
    /* aprire i file */
    file_miglia = fopen("miglia.txt",
                        "r");
    file_chilometri = fopen("chilometri.txt",
                            "w");

    /* acquisire la distanza in miglia */
    fscanf(file_miglia,
           "%lf",
           &miglia);

    /* convertire la distanza in chilometri */
    chilometri = KM_PER_MI * miglia;

    /* comunicare la distanza in chilometri */
    fprintf(file_chilometri,
            "La stessa distanza in chilometri e': %f\n",
            chilometri);

    /* chiudere i file */
    fclose(file_miglia);
    fclose(file_chilometri);
    return(0);
}
```

- Esempio di lancio in esecuzione del programma di Sez. 2.2 con ridirezione dei file standard per `scanf` (da `stdin` a `miglia.txt`) e `printf` (da `stdout` a `chilometri.txt`):

  `conversione_mi_km < miglia.txt > chilometri.txt` ■fltpp_4

Capitolo 3

Espressioni

3.1 Definizione di costante simbolica

- Una costante può comparire all'interno di un programma C in forma letterale – nel qual caso è direttamente espressa tramite il suo valore (p.e. `-13`, `17.5`, `'A'`, `"ciao"`) – oppure in forma simbolica, cioè attraverso un identificatore ad essa associato – che eredita il tipo del suo valore.

- La direttiva di definizione di costante simbolica:
  `#define` ◁*identificatore della costante*▷ ◁*valore della costante*▷
  imparte al preprocessore C il comando di sostituire nel testo del programma ogni occorrenza dell'identificatore della costante con il valore della costante prima di compilare il programma (quindi nessuno spazio di memoria viene riservato per l'identificatore di costante).

- Questo meccanismo consente di raccogliere all'inizio di un programma tutti i valori costanti usati nel programma, dando loro dei nomi da usare all'interno del programma che dovrebbero richiamare ciò che i valori costanti rappresentano. Ciò incrementa la leggibilità del programma e agevola le eventuali successive modifiche di tali valori. Infatti non serve andare a cercare tutte le loro occorrenze all'interno del programma, in quanto basta cambiare le loro definizioni all'inizio del programma.

3.2 Dichiarazione di variabile

- Una variabile in un programma C funge da contenitore per un valore. Diversamente dal valore di una costante, il valore di una variabile può cambiare, anche più volte, durante l'esecuzione del programma.

- La dichiarazione di variabile:
  ◁*tipo*▷ ◁*identificatore della variabile*▷`;`
  che in caso di contestuale inizializzazione diventa:
  ◁*tipo*▷ ◁*identificatore della variabile*▷ `=` ◁*valore iniziale*▷`;`
  associa un nome simbolico alla porzione di memoria necessaria per contenere un valore di un certo tipo, dove sia la dimensione della porzione di memoria che la gamma di valori sono determinati dal tipo.

- Più variabili dello stesso tipo possono essere raccolte in un'unica dichiarazione separando i loro identificatori con delle virgole:
  ◁*tipo*▷ ◁*identificatore della variabile*$_1$▷`,`
  ◁*identificatore della variabile*$_2$▷`,`
  ⋮
  ◁*identificatore della variabile*$_n$▷`;`
  dove gli identificatori delle variabili compaiono allineati su linee separate per aumentare la leggibilità della dichiarazione.

3.3 Operatori aritmetici

- Operatori aritmetici unari (prefissi):
  - `+`◁*espressione*▷: valore dell'espressione (questo operatore non viene mai usato).
  - `-`◁*espressione*▷: valore dell'espressione cambiato di segno.
- Operatori aritmetici binari additivi (infissi):
  - ◁*espressione*$_1$▷ `+` ◁*espressione*$_2$▷: somma dei valori delle due espressioni.
  - ◁*espressione*$_1$▷ `-` ◁*espressione*$_2$▷: differenza dei valori delle due espressioni.
- Operatori aritmetici binari moltiplicativi (infissi):
  - ◁*espressione*$_1$▷ `*` ◁*espressione*$_2$▷: prodotto dei valori delle due espressioni.
  - ◁*espressione*$_1$▷ `/` ◁*espressione*$_2$▷: quoziente dei valori delle due espressioni (se il valore di *espressione*$_2$ è zero, il risultato è `NaN` – not a number).
  - ◁*espressione*$_1$▷ `%` ◁*espressione*$_2$▷: resto della divisione dei valori delle due espressioni intere (se il valore di *espressione*$_2$ è zero, il risultato è `NaN` – not a number).
- Per ogni operatore abbiamo indicato la sintassi prima dei due punti e la semantica dopo i due punti. Notare lo spazio prima e dopo ogni operatore binario per incrementare la leggibilità delle espressioni.

3.4 Operatori relazionali

- Operatori relazionali d'uguaglianza (binari infissi):
  - ◁*espressione*$_1$▷ `==` ◁*espressione*$_2$▷: vero se i valori delle due espressioni sono uguali.
  - ◁*espressione*$_1$▷ `!=` ◁*espressione*$_2$▷: vero se i valori delle due espressioni sono diversi.
- Operatori relazionali d'ordine (binari infissi):
  - ◁*espressione*$_1$▷ `<` ◁*espressione*$_2$▷: vero se il valore di *espressione*$_1$ è minore del valore di *espressione*$_2$.
  - ◁*espressione*$_1$▷ `<=` ◁*espressione*$_2$▷: vero se il valore di *espressione*$_1$ è minore del o uguale al valore di *espressione*$_2$.
  - ◁*espressione*$_1$▷ `>` ◁*espressione*$_2$▷: vero se il valore di *espressione*$_1$ è maggiore del valore di *espressione*$_2$.
  - ◁*espressione*$_1$▷ `>=` ◁*espressione*$_2$▷: vero se il valore di *espressione*$_1$ è maggiore del o uguale al valore di *espressione*$_2$.
- Se vera, l'espressione complessiva ha valore uno, altrimenti ha valore zero.

3.5 Operatori logici

- In C i valori di verità falso e vero vengono rappresentati numericamente come segue: zero è interpretato come falso, ogni altro numero è interpretato come vero.
- Se un operatore logico è soddisfatto, allora il valore che esso restituisce è uno, altrimenti è zero.
- Operatori logici unari (prefissi):
  - `!`◁*espressione*▷: negazione logica del valore dell'espressione.
- Operatori logici binari (infissi):
  - ◁*espressione*$_1$▷ `&&` ◁*espressione*$_2$▷: congiunzione logica dei valori delle due espressioni.
  - ◁*espressione*$_1$▷ `||` ◁*espressione*$_2$▷: disgiunzione logica dei valori delle due espressioni.
- In C avviene la cortocircuitazione dell'applicazione degli operatori logici binari:
  - In ◁*espressione*$_1$▷ `&&` ◁*espressione*$_2$▷, se il valore di *espressione*$_1$ è falso, *espressione*$_2$ non viene valutata in quanto si può già stabilire che il valore dell'espressione complessiva è falso.
  - In ◁*espressione*$_1$▷ `||` ◁*espressione*$_2$▷, se il valore di *espressione*$_1$ è vero, *espressione*$_2$ non viene valutata in quanto si può già stabilire che il valore dell'espressione complessiva è vero.

3.6 Operatore condizionale

- L'operatore condizionale (ternario infisso):
  `(`◁*espressione*$_1$▷`)? ◁`*espressione*$_2$▷`: ◁`*espressione*$_3$▷
  dà come valore quello di *espressione*$_2$ oppure quello di *espressione*$_3$ a seconda che il valore di *espressione*$_1$ sia vero o falso.

- L'operatore condizionale permette di scrivere i programmi in maniera più concisa, però non bisogna abusarne al fine di non compromettere la leggibilità dei programmi stessi.

- Esempi:

  - Determinazione del massimo tra i valori delle variabili `x` ed `y`:
    ```
    (x > y)? x: y
    ```
  - Calcolo della biimplicazione logica applicato alle variabili `x` ed `y`:
    ```
    (x == y || (x != 0 && y != 0))? 1: 0
    ```
  - Gestione di singolare e plurale:
    ```
    printf("Hai vinto %d centesim%c.\n",
           importo,
           (importo == 1)? 'o': 'i');
    ```

3.7 Operatori di assegnamento

- Operatori di assegnamento (binari infissi):

  - ◁*variabile*▷ `=` ◁*espressione*▷: il valore dell'espressione diventa il nuovo valore della variabile (attraverso la sua memorizzazione nella porzione di memoria riservata alla variabile).
  - ◁*variabile*▷ `+=` ◁*espressione*▷: la somma del valore corrente della variabile e del valore dell'espressione diventa il nuovo valore della variabile.
  - ◁*variabile*▷ `-=` ◁*espressione*▷: la differenza tra il valore corrente della variabile e il valore dell'espressione diventa il nuovo valore della variabile.
  - ◁*variabile*▷ `*=` ◁*espressione*▷: il prodotto del valore corrente della variabile e del valore dell'espressione diventa il nuovo valore della variabile.
  - ◁*variabile*▷ `/=` ◁*espressione*▷: il quoziente del valore corrente della variabile e del valore dell'espressione diventa il nuovo valore della variabile.
  - ◁*variabile*▷ `%=` ◁*espressione*▷: il resto della divisione del valore corrente della variabile intera per il valore dell'espressione intera diventa il nuovo valore della variabile intera.

- In generale ◁*variabile*▷ ◁*op*▷`=` ◁*espressione*▷ sta per:
  ◁*variabile*▷ `=` ◁*variabile*▷ ◁*op*▷ `(`◁*espressione*▷`)`
  dove le parentesi attorno all'espressione sono necessarie per applicare gli operatori nell'ordine corretto.

- Il valore di un'espressione di assegnamento è il valore assegnato, il che permette l'assegnamento simultaneo del valore di una singola espressione a molteplici variabili:
  ◁*variabile*$_1$▷ `=` ◁*variabile*$_2$▷ `= ... =` ◁*variabile*$_n$▷ `=` ◁*espressione*▷

- Il simbolo "=", usato in C per denotare l'operatore di assegnamento, non va confuso con il simbolo "=" tradizionalmente usato in matematica per denotare l'operatore relazionale di uguaglianza, che è rappresentato col simbolo "==" in C.

3.8 Operatori di incremento/decremento

- Operatori di incremento/decremento postfissi (unari):
  - ◁*variabile*▷`++`: il valore della variabile, la quale viene poi incrementata di una unità.
  - ◁*variabile*▷`-`: il valore della variabile, la quale viene poi decrementata di una unità.
- Operatori di incremento/decremento prefissi (unari):
  - `++`◁*variabile*▷: il valore della variabile incrementato di una unità.
  - `-`◁*variabile*▷: il valore della variabile decrementato di una unità.
- Gli operatori di incremento/decremento sono applicabili solo a variabili e comportano sempre la variazione di un'unità del valore delle variabili cui sono applicati. Da questo punto di vista, l'operatore di incremento equivale a ◁*variabile*▷ `+= 1`, mentre l'operatore di decremento equivale a ◁*variabile*▷ `-= 1`.
- Il valore dell'intera espressione di incremento/decremento cambia a seconda che l'operatore di incremento/decremento sia postfisso o prefisso (importante all'interno di un'espressione più grande).
- Esempi:
  - Consideriamo l'espressione `x += y++` e assumiamo che, prima della sua valutazione, la variabile `x` valga `15` e la variabile `y` valga `3`. Al termine della valutazione, la variabile `y` vale `4` e la variabile `x` vale `18` perché il valore di `y++` è quello di `y` prima dell'incremento.
  - Consideriamo l'espressione `x += ++y` e assumiamo le stesse condizioni iniziali dell'esempio precedente. Al termine della valutazione, la variabile `y` vale `4` (come nell'esempio precedente) ma la variabile `x` vale `19` (diversamente dall'esempio precedente) perché il valore di `++y` è quello di `y` dopo l'incremento.

3.9 Operatore virgola

- L'operatore virgola (binario infisso):
  ◁$espressione_1$▷`,` ◁$espressione_2$▷
  dà come valore quello di $espressione_2$.
- L'operatore virgola viene usato come separatore in una sequenza di espressioni che debbono essere valutate una dopo l'altra all'interno della medesima istruzione (vedi Sez. 4.4). ■fltpp_5

3.10 Tipo delle espressioni

- Le espressioni aritmetico-logiche del linguaggio C sono formate da occorrenze dei precedenti operatori (aritmetici, relazionali, logici, condizionale, di assegnamento, di incremento/decremento, virgola) applicati a costanti letterali o simboliche e variabili (e risultati di invocazioni di funzioni – vedi Sez. 5.5) di tipo `int` o `double` (o `char` – vedi Sez. 6.5 – o enumerato – vedi Sez. 6.6).
- Esempi:
  - Stabilire se `x` ed `y` sono entrambe maggiori di `z`: `x > z && y > z`
  - Stabilire se `x` vale `1` oppure `3`: `x == 1 || x == 3`
  - Stabilire se `z` è compresa tra `x` ed `y` (assumendo $x \leq y$): `x <= z && z <= y`
  - Stabilire se `z` non è compresa tra `x` ed `y`: `!(x <= z && z <= y)`
  - Stabilire se `z` non è compresa tra `x` ed `y` in modo diverso: `z < x || y < z`
  - Stabilire se `n` è pari: `n % 2 == 0`
  - Stabilire se `n` è dispari: `n % 2 == 1`
  - Stabilire se `anno` è bisestile: `(anno % 4 == 0 && anno % 100 != 0) || (anno % 400 == 0)`

- Il tipo di un'espressione aritmetico-logica dipende dagli operatori presenti in essa e dal tipo dei relativi operandi. La regola generale è la seguente:
  - Se tutti i suoi operandi sono di tipo `int`, l'espressione è di tipo `int`.
  - Se almeno uno dei suoi operandi è di tipo `double`, l'espressione è di tipo `double`.
- Nel caso dell'operatore "`%`", i suoi due operandi devono essere di tipo `int`.
- Nel caso degli operatori di assegnamento:
  - Se l'espressione è di tipo `int` e la variabile è di tipo `double`, il tipo dell'espressione viene automaticamente convertito in `double` e il relativo valore viene modificato aggiungendogli una parte frazionaria nulla (`.0`) prima che avvenga l'assegnamento.
  - Se l'espressione è di tipo `double` e la variabile è di tipo `int`, il tipo dell'espressione viene automaticamente convertito in `int` e il relativo valore viene modificato troncandone la parte frazionaria prima che avvenga l'assegnamento.
- Esempi:
  - Il valore di `5 / 2` è `2`, mentre il valore di `5.0 / 2` oppure `5 / 2.0` oppure `5.0 / 2.0` è `2.5`.
  - Dato l'assegnamento `x = 3.75`, se `x` è di tipo `int` allora il suo nuovo valore è `3`.

3.11 Precedenza e associatività degli operatori

- Al fine di determinare il valore di un'espressione aritmetico-logica, occorre stabilire l'ordine in cui gli operatori (diversi o uguali) presenti nell'espressione debbono essere applicati.
- L'ordine in cui occorrenze di operatori diversi debbono essere applicate è stabilito dalla precedenza degli operatori, di seguito riportata in ordine decrescente:
  - Operatori unari aritmetici, logici, di incremento/decremento: "`+`", "`-`", "`!`", "`++`", "`--`".
  - Operatori aritmetici moltiplicativi: "`*`", "`/`", "`%`".
  - Operatori aritmetici additivi: "`+`", "`-`".
  - Operatori relazionali d'ordine: "`<`", "`>`", "`<=`", "`>=`".
  - Operatori relazionali d'uguaglianza: "`==`", "`!=`".
  - Operatori logici moltiplicativi: "`&&`".
  - Operatori logici additivi: "`||`".
  - Operatore condizionale: "`?:`".
  - Operatori di assegnamento: "`=`", "`+=`", "`-=`", "`*=`", "`/=`", "`%=`".
  - Operatore virgola: "`,`".
- L'ordine in cui occorrenze dello stesso operatore debbono essere applicate è stabilito dall'associatività dell'operatore. Tutti gli operatori riportati sopra sono associativi da sinistra, ad eccezione di quelli di incremento/decremento – che non sono associativi, cioè sono applicabili una sola volta ad una variabile – e degli altri unari e di quelli di assegnamento – che sono associativi da destra.
- Le regole di precedenza e associatività possono essere alterate inserendo delle parentesi tonde.
- Un ausilio grafico per determinare l'ordine in cui applicare gli operatori di un'espressione aritmetico-logica è costituito dall'albero di valutazione dell'espressione. Come illustrato in Fig. 3.1, in questo albero le foglie corrispondono a costanti e variabili (e risultati di invocazioni di funzioni) presenti nell'espressione, mentre i nodi interni corrispondono agli operatori presenti nell'espressione e vengono collocati in base alle regole di precedenza e associatività.

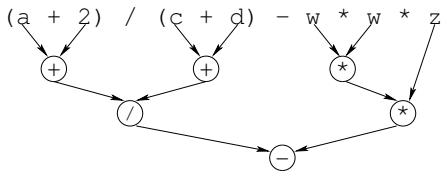


Figura 3.1: Albero di valutazione di `(a + 2) / (c + d) - w * w * z`

- Esempio di programma: determinazione del valore di un insieme di monete.
  1. **Specifica del problema**. Calcolare il valore complessivo di un insieme di monete che vengono depositate da un cliente presso una banca, espresso come numero di euro e frazione di euro.
  2. **Analisi del problema**. L'input è rappresentato dal numero di monete di ogni tipo (1, 2, 5, 10, 20, 50 centesimi e 1, 2 euro) che vengono depositate. L'output è rappresentato dal valore complessivo delle monete depositate, espresso in euro e frazione di euro. Le relazioni da sfruttare sono le seguenti: 1 centesimo = 0.01 euro, 2 centesimi = 0.02 euro, 5 centesimi = 0.05 euro, 10 centesimi = 0.10 euro, 20 centesimi = 0.20 euro, 50 centesimi = 0.50 euro, 1 euro = 100 centesimi, 2 euro = 200 centesimi.
  3. **Progettazione dell'algoritmo**. Come scelta di progetto, decidiamo di calcolare prima il valore totale in centesimi perché questo agevola poi la determinazione del valore totale in euro e frazione di euro. I passi sono i seguenti:
     - Acquisire il numero di monete depositate di ogni tipo.
     - Calcolare il valore totale delle monete in centesimi.
     - Convertire il valore totale delle monete in euro e frazione di euro.
     - Comunicare il valore totale delle monete in euro e frazione di euro.
  4. **Implementazione dell'algoritmo**. Questa è la traduzione dei passi in C:

```
/*****************************************************************/
/* programma per determinare il valore di un insieme di monete */
/*****************************************************************/

/*****************************/
/* inclusione delle librerie */
/*****************************/

#include <stdio.h>

/***********************************/
/* definizione della funzione main */
/***********************************/

int main(void)
{
  /* dichiarazione delle variabili locali alla funzione */
  int num_monete_01c,     /* input: numero di monete da 1 centesimo */
      num_monete_02c,     /* input: numero di monete da 2 centesimi */
      num_monete_05c,     /* input: numero di monete da 5 centesimi */
      num_monete_10c,     /* input: numero di monete da 10 centesimi */
      num_monete_20c,     /* input: numero di monete da 20 centesimi */
      num_monete_50c,     /* input: numero di monete da 50 centesimi */
      num_monete_01e,     /* input: numero di monete da 1 euro */
      num_monete_02e;     /* input: numero di monete da 2 euro */
  int valore_euro,        /* output: valore espresso in numero di euro */
      frazione_euro;      /* output: numero di centesimi della frazione di euro */
  int valore_centesimi;   /* lavoro: valore espresso in numero di centesimi */
```

```
  /* acquisire il numero di monete depositate di ogni tipo */
  printf("Digita il numero di monete da 1 centesimo: ");
  scanf("%d",
        &num_monete_01c);
  printf("Digita il numero di monete da 2 centesimi: ");
  scanf("%d",
        &num_monete_02c);
  printf("Digita il numero di monete da 5 centesimi: ");
  scanf("%d",
        &num_monete_05c);
  printf("Digita il numero di monete da 10 centesimi: ");
  scanf("%d",
        &num_monete_10c);
  printf("Digita il numero di monete da 20 centesimi: ");
  scanf("%d",
        &num_monete_20c);
  printf("Digita il numero di monete da 50 centesimi: ");
  scanf("%d",
        &num_monete_50c);
  printf("Digita il numero di monete da 1 euro: ");
  scanf("%d",
        &num_monete_01e);
  printf("Digita il numero di monete da 2 euro: ");
  scanf("%d",
        &num_monete_02e);

  /* calcolare il valore totale delle monete in centesimi */
  valore_centesimi =   1 * num_monete_01c +
                       2 * num_monete_02c +
                       5 * num_monete_05c +
                      10 * num_monete_10c +
                      20 * num_monete_20c +
                      50 * num_monete_50c +
                     100 * num_monete_01e +
                     200 * num_monete_02e;

  /* convertire il valore totale delle monete in euro e frazione di euro */
  valore_euro = valore_centesimi / 100;
  frazione_euro = valore_centesimi % 100;

  /* comunicare il valore totale delle monete in euro e frazione di euro */
  printf("Il valore delle monete e' di %d euro e %d centesim%c.\n",
         valore_euro,
         frazione_euro,
         (frazione_euro == 1)? 'o': 'i');
  return(0);
}
```

Il programma è ridondante in termini di numero di variabili di input dichiarate, numero di `printf` e `scanf` utilizzate per acquisire i valori delle variabili di input e numero di addendi presenti nel calcolo del valore totale in centesimi. Per evitare tale ridondanza, bisogna ricorrere ad istruzioni di ripetizione (vedi Sez. 4.4) e a strutture dati di tipo array (vedi Sez. 6.8) e stringa (vedi Sez. 6.9).

■fltpp_6

Capitolo 4

Istruzioni

4.1 Istruzione di assegnamento

- L'istruzione di assegnamento è l'istruzione più semplice del linguaggio C. Sintatticamente, essa è un'espressione di assegnamento terminata da "`;`".
- L'istruzione di assegnamento consente di modificare il contenuto di una porzione di memoria. Questa istruzione è quella fondamentale nel paradigma di programmazione imperativo di natura procedurale, in quanto tale paradigma si basa su modifiche ripetute del contenuto della memoria. Quest'ultimo viene assunto essere lo stato della computazione.

4.2 Istruzione composta

- Un'istruzione composta del linguaggio C è una sequenza di una o più istruzioni, eventualmente racchiuse tra parentesi graffe. Nel seguito, con *istruzione* intenderemo sempre un'istruzione composta.
- Le istruzioni che formano un'istruzione composta vengono eseguite una alla volta nell'ordine in cui sono state scritte. Per cambiare questo, occorre utilizzare istruzioni di controllo del flusso quali le istruzioni di selezione e le istruzioni di ripetizione.

4.3 Istruzioni di selezione: `if`, `switch`

- Le istruzioni di selezione `if` e `switch` messe a disposizione dal linguaggio C esprimono una scelta tra diverse istruzioni composte, dove la scelta viene operata sulla base del valore di un'espressione aritmetico-logica (scelta deterministica).
- L'istruzione `if` ha diversi formati (nei quali si usa l'indentazione per motivi di leggibilità):
  - Formato con singola alternativa:
    `if (`◁*espressione*▷`)`
    ◁*istruzione*▷
    La sua semantica, cioè il suo effetto a tempo d'esecuzione, è che l'istruzione composta viene eseguita solo se l'espressione è vera.
  - Formato con due alternative:
    `if (`◁*espressione*▷`)`
    ◁$istruzione_1$▷
    `else`
    ◁$istruzione_2$▷
    Se l'espressione è vera, viene eseguita $istruzione_1$, altrimenti viene eseguita $istruzione_2$.

  - Formato con più alternative correlate annidate:

    ```
    if (◁espressione1▷)
       ◁istruzione1▷
    else if (◁espressione2▷)
       ◁istruzione2▷
      ⋮
    else if (◁espressione_{n−1}▷)
       ◁istruzione_{n−1}▷
    else
       ◁istruzione_n▷
    ```

    Le espressioni vengono valutate una dopo l'altra nell'ordine in cui sono scritte, fino ad individuarne una che è vera; se questa è *espressione*$_i$, viene eseguita *istruzione*$_i$. Se nessuna delle espressioni è vera, viene eseguita *istruzione*$_n$. Le alternative sono correlate nel senso che tutte le espressioni hanno una parte comune.

- Esempi:

  - Scambio dei valori delle variabili `x` ed `y` se il valore della prima è maggiore del valore della seconda:

    ```
    if (x > y)
    {
      tmp = x;
      x = y;
      y = tmp;
    }
    ```

  - Controllo del divisore:

    ```
    if (y != 0)
      risultato = x / y;
    else
      printf("Impossibile calcolare il risultato: divisione illegale.\n");
    ```

  - Classificazione dei livelli di rumore:

    ```
    if (rumore <= 50)
      printf("quiete\n");
    else if (rumore <= 70)
      printf("leggero disturbo\n");
    else if (rumore <= 90)
      printf("disturbo\n");
    else if (rumore <= 110)
      printf("forte disturbo\n");
    else
      printf("rumore insopportabile\n");
    ```

- Nel formato generale, un'istruzione `if` può contenere altre istruzioni `if` arbitrariamente annidate. In tal caso, ogni `else` è associato all'`if` pendente più vicino che lo precede. Questa regola di associazione può essere alterata inserendo delle parentesi graffe nell'istruzione `if` complessiva.

- Esempi:

  - Uso errato delle parentesi graffe:

    ```
    if (y == 0)
    {
      printf("E' impossibile calcolare il risultato ");
      printf("perche' e' stata incontrata una divisione ");
    }
    printf("nella quale il divisore e' nullo.\n");
    ```

– Associazione degli `else` agli `if`:

```
if (x == 0)                if (x == 0)                if (x == 0)
  if (y >= 0)              {                          {
    x += y;                  if (y >= 0)                if (y >= 0)
  else                         x += y;                    x += y;
    x -= y;                  else                     }
                               x -= y;                else
                           }                            x -= y;
```

La prima istruzione è equivalente alla seconda, ma non alla terza.

- L'istruzione `switch` ha il seguente formato (notare l'allineamento delle clausole `case` e l'indentazione all'interno di ciascun insieme di clausole `case` per motivi di leggibilità):

```
switch (◁espressione▷)
{
  case ◁valore_{1,1}▷:
  case ◁valore_{1,2}▷:
   ...
  case ◁valore_{1,m_1}▷:
    ◁istruzione_1▷
    break;
  case ◁valore_{2,1}▷:
  case ◁valore_{2,2}▷:
   ...
  case ◁valore_{2,m_2}▷:
    ◁istruzione_2▷
    break;
   ⋮
  case ◁valore_{n,1}▷:
  case ◁valore_{n,2}▷:
   ...
  case ◁valore_{n,m_n}▷:
    ◁istruzione_n▷
    break;
  default:
    ◁istruzione_{n+1}▷
    break;
}
```

- L'espressione su cui si basa la selezione deve essere di tipo `int` o `char` (o enumerato – vedi Sez. 6.6). Se il suo valore è uguale ad uno dei valori indicati in una delle clausole `case`, tutte le istruzioni composte che seguono quella clausola `case` vengono eseguite fino ad incontrare un'istruzione `break` (o la fine dell'istruzione `switch`). Se invece il valore dell'espressione è diverso da tutti i valori indicati nelle clausole `case`, viene eseguita l'istruzione composta specificata nella clausola opzionale `default`.

- La presenza di un'istruzione `break` dopo ogni istruzione composta dell'istruzione `switch` garantisce la corretta strutturazione dell'istruzione `switch` stessa, in quanto permette ad ogni istruzione composta di essere eseguita solo se il valore dell'espressione è uguale al valore di una delle clausole `case` associate all'istruzione composta medesima. Ogni istruzione composta ha quindi un unico punto di ingresso – l'insieme delle clausole `case` che immediatamente la precedono – e un unico punto di uscita – l'istruzione `break` che immediatamente la segue (vedi Sez. 4.6).

- La precedente istruzione `switch` è equivalente alla seguente istruzione `if` con alternative correlate:

```
if (◁espressione▷ == ◁valore_{1,1}▷ ||
    ◁espressione▷ == ◁valore_{1,2}▷ ||
     ...
    ◁espressione▷ == ◁valore_{1,m_1}▷)
  ◁istruzione_1▷
else if (◁espressione▷ == ◁valore_{2,1}▷ ||
         ◁espressione▷ == ◁valore_{2,2}▷ ||
          ...
         ◁espressione▷ == ◁valore_{2,m_2}▷)
  ◁istruzione_2▷
  ⋮
else if (◁espressione▷ == ◁valore_{n,1}▷ ||
         ◁espressione▷ == ◁valore_{n,2}▷ ||
          ...
         ◁espressione▷ == ◁valore_{n,m_n}▷)
  ◁istruzione_n▷
else
  ◁istruzione_{n+1}▷
```

- L'istruzione `switch` è quindi più leggibile ma meno espressiva dell'istruzione `if`. Infatti, l'espressione di selezione dell'istruzione `switch` può essere solo di tipo `int` o `char` (o enumerato – vedi Sez. 6.6), può essere confrontata solo attraverso l'operatore relazionale di uguaglianza e gli elementi con cui effettuare i confronti possono essere solo delle costanti (letterali o simboliche).

- Esempio di riconoscimento dei colori della bandiera italiana:

```
char colore;

switch (colore)
{
  case 'V':
  case 'v':
    printf("colore presente nella bandiera italiana: verde\n");
    break;
  case 'B':
  case 'b':
    printf("colore presente nella bandiera italiana: bianco\n");
    break;
  case 'R':
  case 'r':
    printf("colore presente nella bandiera italiana: rosso\n");
    break;
  default:
    printf("colore non presente nella bandiera italiana\n");
    break;
}
```

■fltpp_7

- Esempio di programma: calcolo della bolletta dell'acqua.
  1. **Specifica del problema**. Calcolare la bolletta dell'acqua per un utente sulla base di una quota fissa di 15 euro e una quota variabile di 2.50 euro per ogni metro cubo d'acqua consumato nell'ultimo periodo, più una mora di 10 euro per eventuali bollette non pagate relative a periodi precedenti. Evidenziare l'eventuale applicazione della mora.
  2. **Analisi del problema**. L'input è costituito dalla lettura del contatore alla fine del periodo precedente, dalla lettura del contatore alla fine del periodo corrente (cui la bolletta si riferisce) e dall'importo di eventuali bollette precedenti ancora da pagare. L'output è costituito dall'importo della bolletta del periodo corrente, evidenziando anche l'eventuale applicazione della mora. Le relazioni da sfruttare sono che il consumo di acqua nel periodo corrente è dato dalla differenza tra le ultime due letture del contatore e che l'importo della bolletta è dato dalla somma della quota fissa, del costo al metro cubo moltiplicato per il consumo di acqua nel periodo corrente, dell'importo di eventuali bollette arretrate e dell'eventuale mora.
  3. **Progettazione dell'algoritmo**. Non ci sono particolari scelte di progetto da compiere. Osservato che l'importo della bolletta è diverso a seconda che vi siano bollette precedenti ancora da pagare o meno, i passi sono i seguenti:
     - Acquisire le ultime due letture del contatore.
     - Acquisire l'importo di eventuali bollette precedenti ancora da pagare.
     - Calcolare l'importo della bolletta:
       * Calcolare l'importo derivante dal consumo di acqua nel periodo corrente.
       * Determinare l'applicabilità della mora.
       * Sommare le varie voci.
     - Comunicare l'importo della bolletta evidenziando l'eventuale mora.
  4. **Implementazione dell'algoritmo**. Questa è la traduzione dei passi in C:

```
/*************************************************/
/* programma per calcolare la bolletta dell'acqua */
/*************************************************/

/*****************************/
/* inclusione delle librerie */
/*****************************/

#include <stdio.h>

/*****************************************/
/* definizione delle costanti simboliche */
/*****************************************/

#define QUOTA_FISSA    15.00   /* quota fissa */
#define COSTO_PER_M3    2.50   /* costo per metro cubo */
#define MORA           10.00   /* mora */

/***********************************/
/* definizione della funzione main */
/***********************************/

int main(void)
{
  /* dichiarazione delle variabili locali alla funzione */
  int    lettura_prec,        /* input: lettura alla fine periodo precedente */
         lettura_corr;        /* input: lettura alla fine periodo corrente */
  double importo_arretrato;   /* input: importo delle bollette arretrate */
  double importo_bolletta;    /* output: importo della bolletta */
  double importo_consumo,     /* lavoro: importo del consumo */
         importo_mora;        /* lavoro: importo della mora se dovuta */
```

```
  /* acquisire le ultime due letture del contatore */
  printf("Digita il consumo risultante dalla lettura precedente: ");
  scanf("%d",
        &lettura_prec);
  printf("Digita il consumo risultante dalla lettura corrente: ");
  scanf("%d",
        &lettura_corr);

  /* acquisire l'importo di eventuali bollette precedenti ancora da pagare */
  printf("Digita l'importo di eventuali bollette ancora da pagare: ");
  scanf("%lf",
        &importo_arretrato);

  /* calcolare l'importo derivante dal consumo di acqua nel periodo corrente */
  importo_consumo = (lettura_corr - lettura_prec) * COSTO_PER_M3;

  /* determinare l'applicabilita' della mora */
  importo_mora = (importo_arretrato > 0.0)?
                  MORA:
                  0.0;

  /* sommare le varie voci */
  importo_bolletta = QUOTA_FISSA +
                     importo_consumo +
                     importo_arretrato +
                     importo_mora;

  /* comunicare l'importo della bolletta evidenziando l'eventuale mora */
  printf("\nTotale bolletta: %.2f euro.\n",
         importo_bolletta);
  if (importo_mora > 0.0)
  {
    printf("\nLa bolletta comprende una mora di %.2f euro",
           importo_mora);
    printf(" per un arretrato di %.2f euro.\n",
           importo_arretrato);
  }
  return(0);
}
```

4.4 Istruzioni di ripetizione: `while`, `for`, `do-while`

- Le istruzioni di ripetizione `while`, `for` e `do-while` messe a disposizione dal linguaggio C esprimono l'esecuzione reiterata di un'istruzione composta, dove la terminazione dell'iterazione dipende dal valore di un'espressione aritmetico-logica detta condizione di continuazione.

- Formato dell'istruzione `while` (in cui si usa l'indentazione per motivi di leggibilità):

  `while (`◁*espressione*▷`)`
  ◁*istruzione*▷

  La sua semantica, cioè il suo effetto a tempo d'esecuzione, è che l'istruzione composta che si trova all'interno viene eseguita finché l'espressione è vera. Se all'inizio l'espressione è falsa, l'istruzione composta non viene eseguita affatto.

- Formato dell'istruzione `for` (in cui si usa l'indentazione per motivi di leggibilità):

  ```
  for (◁espressione1▷;
       ◁espressione2▷;
       ◁espressione3▷)
     ◁istruzione▷
  ```

  L'istruzione `for` è una variante articolata dell'istruzione `while`, dove *espressione*$_1$ è l'espressione di inizializzazione delle variabili (di controllo del ciclo) presenti nella condizione di continuazione, *espressione*$_2$ è la condizione di continuazione ed *espressione*$_3$ è l'espressione di aggiornamento delle variabili (di controllo del ciclo) presenti nella condizione di continuazione.
  L'istruzione `for` equivale alla seguente istruzione composta contenente un'istruzione `while`:

  ```
  ◁espressione1▷;
  while (◁espressione2▷)
  {
     ◁istruzione▷
     ◁espressione3▷;
  }
  ```

- Formato dell'istruzione `do-while` (in cui si usa l'indentazione per motivi di leggibilità):

  ```
  do
     ◁istruzione▷
  while (◁espressione▷);
  ```

  L'istruzione `do-while` è una variante dell'istruzione `while` in cui l'istruzione composta che si trova all'interno viene eseguita almeno una volta, quindi equivale alla seguente istruzione composta contenente un'istruzione `while`:

  ```
  ◁istruzione▷
  while (◁espressione▷)
     ◁istruzione▷
  ```

- Quando si usano istruzioni di ripetizione, è fondamentale definire le loro condizioni di continuazione in maniera tale da evitare iterazioni senza termine, come pure racchiudere le loro istruzioni composte tra parentesi graffe se queste istruzioni sono formate da più di un'istruzione.

- Esistono diversi tipi di controllo della ripetizione:

  - Ripetizione controllata tramite contatore (quando si conosce a priori il numero di iterazioni).
  - Ripetizione controllata tramite sentinella (quando non si conosce il numero di iterazioni).
  - Ripetizione controllata tramite fine file (caso particolare del precedente).
  - Ripetizione relativa all'acquisizione e alla validazione di un valore.

- Esempi:

  - Uso di un contatore nel calcolo della media di un insieme di numeri naturali:

    ```
    for (contatore_valori = 1,
           somma_valori = 0;
         (contatore_valori <= numero_valori);
         contatore_valori++,
           somma_valori += valore)
    {
      printf("Digita il prossimo valore: ");
      scanf("%d",
            &valore);
    }
    printf("La media e': %d.\n",
           somma_valori / numero_valori);
    ```

– Uso di un valore sentinella nel calcolo della media di un insieme di numeri naturali:

```
numero_valori = somma_valori = valore = 0;
while (valore >= 0)
{
  printf("Digita il prossimo valore (negativo per terminare): ");
  scanf("%d",
        &valore);
  if (valore >= 0)
  {
    numero_valori++;
    somma_valori += valore;
  }
}
printf("La media e': %d.\n",
       somma_valori / numero_valori);
```

– Uso di fine file nel calcolo della media di un insieme di numeri naturali memorizzati su file:

```
for (numero_valori = somma_valori = 0;
     (fscanf(file_valori,
             "%d",
             &valore) != EOF);
     numero_valori++,
       somma_valori += valore);
printf("La media e': %d.\n",
       somma_valori / numero_valori);
```

– Validazione lasca di un valore acquisito in ingresso:

```
do
{
  printf("Digita il numero di valori di cui calcolare la media (> 0): ");
  scanf("%d",
        &numero_valori);
}
while (numero_valori <= 0);
```

– Validazione stretta dello stesso valore sfruttando il risultato della funzione `scanf` per eliminare dal buffer eventuali valori non conformi al segnaposto nonché eventuali ulteriori valori (questi valori sono considerati come non acquisiti e quindi potrebbero determinare la non terminazione del ciclo di validazione lasca):

```
/* dichiarare esito_lettura di tipo int */

do
{
  printf("Digita il numero di valori di cui calcolare la media (> 0): ");
  esito_lettura = scanf("%d",
                        &numero_valori);
  if (esito_lettura != 1 || numero_valori <= 0)
    printf("Input non accettabile!\n");
  while (getchar() != '\n');
}
while (esito_lettura != 1 || numero_valori <= 0);
```

■fltpp_8

- Esempio di programma: calcolo dei livelli di radiazione.

  1. **Specifica del problema**. In un laboratorio può verificarsi una perdita di un materiale pericoloso, il quale produce un certo livello iniziale di radiazione che poi si dimezza ogni tre giorni. Calcolare il livello delle radiazioni ogni tre giorni, fino a raggiungere il giorno in cui il livello delle radiazioni scende al di sotto di un decimo del livello di sicurezza quantificato in 0.466 mrem.
  2. **Analisi del problema**. L'input è costituito dal livello iniziale delle radiazioni. L'output è rappresentato dal giorno in cui viene ripristinata una situazione di sicurezza, con il valore del livello delle radiazioni calcolato ogni tre giorni sino a quel giorno. La relazione da sfruttare è il dimezzamento del livello delle radiazioni ogni tre giorni.
  3. **Progettazione dell'algoritmo**. Non ci sono particolari scelte di progetto da compiere. Osservato che il problema richiede il calcolo ripetuto del livello delle radiazioni sino a quando tale livello non scende sotto una certa soglia, i passi sono i seguenti:
     - Acquisire il livello iniziale delle radiazioni.
     - Calcolare e comunicare il livello delle radiazioni ogni tre giorni finché il livello non scende al di sotto di un decimo del livello di sicurezza.
     - Comunicare il giorno in cui il livello delle radiazioni scende al di sotto di un decimo del livello di sicurezza.
  4. **Implementazione dell'algoritmo**. Questa è la traduzione dei passi in C:

```
/*****************************************************/
/* programma per calcolare i livelli di radiazione */
/*****************************************************/

/*****************************/
/* inclusione delle librerie */
/*****************************/

#include <stdio.h>

/*****************************************/
/* definizione delle costanti simboliche */
/*****************************************/

#define SOGLIA_SICUREZZA    0.0466   /* soglia di sicurezza */
#define FATTORE_RIDUZIONE   2.0      /* fattore di riduzione delle radiazioni */
#define NUMERO_GIORNI       3        /* numero di giorni di riferimento */

/***********************************/
/* definizione della funzione main */
/***********************************/

int main(void)
{
  /* dichiarazione delle variabili locali alla funzione */
  double livello_iniziale;   /* input: livello iniziale delle radiazioni */
  int    giorno_sicurezza;   /* output: giorno di ripristino della sicurezza */
  double livello_corrente;   /* output: livello corrente delle radiazioni */
```

```
  /* acquisire il livello iniziale delle radiazioni */
  do
  {
    printf("Digita il livello iniziale delle radiazioni (> 0): ");
    scanf("%lf",
          &livello_iniziale);
  }
  while (livello_iniziale <= 0.0);

  /* calcolare e comunicare il livello delle radiazioni ogni tre giorni finche'
     il livello non scende al di sotto di un decimo del livello di sicurezza */
  for (livello_corrente = livello_iniziale,
         giorno_sicurezza = 0;
       (livello_corrente >= SOGLIA_SICUREZZA);
       livello_corrente /= FATTORE_RIDUZIONE,
         giorno_sicurezza += NUMERO_GIORNI)
    printf("Il livello delle radiazioni al giorno %3d e' %9.4f.\n",
           giorno_sicurezza,
           livello_corrente);

  /* comunicare il giorno in cui il livello delle radiazioni scende al di sotto
     di un decimo del livello di sicurezza */
  printf("Giorno in cui si puo' tornare in laboratorio: %d.\n",
         giorno_sicurezza);
  return(0);
}
```

4.5 Istruzione `goto`

- Il flusso di esecuzione delle istruzioni di un programma C può essere modificato in maniera arbitraria tramite la seguente istruzione:

  `goto` ◁*etichetta*▷`;`

  la quale fa sì che la prossima istruzione da eseguire non sia quella ad essa immediatamente successiva nel testo del programma, ma quella prefissata da:

  ◁*etichetta*▷`:`

  dove *etichetta* è un identificatore.

- Il C eredita l'istruzione `goto` dai linguaggi assemblativi, nei quali non sono disponibili istruzioni di controllo del flusso di esecuzione di alto livello di astrazione – come quelle di selezione e ripetizione – ma solo istruzioni di salto incondizionato e condizionato.

- Esempio di uso di istruzioni di salto incondizionato e condizionato per comunicare se il valore di una variabile è pari o dispari:

```
(1)                if (n % 2 == 0)
(2)                  goto scrivi_pari;
(3)                printf("Il numero e' dispari.\n");
(4)                goto continua;
(5)  scrivi_pari:  printf("Il numero e' pari.\n");
(6)  continua:     ...
```

  Se `n` è pari, allora il flusso di esecuzione è `(1)-(2)-(5)-(6)`, altrimenti è `(1)-(3)-(4)-(6)`.

- L'uso dell'istruzione `goto` rende i programmi più difficili da leggere e da mantenere, quindi è bene evitarlo.

4.6 Teorema fondamentale della programmazione strutturata

- I programmi sono rappresentabili graficamente attraverso schemi di flusso, i quali sono costituiti dai blocchi e dai nodi mostrati in Fig. 4.1.

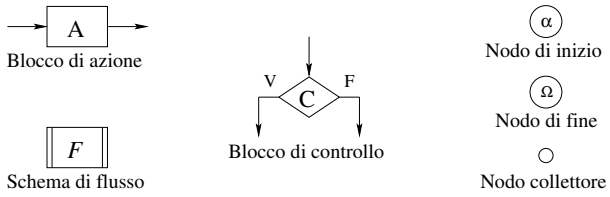


Figura 4.1: Blocchi e nodi degli schemi di flusso

- L'insieme degli schemi di flusso strutturati è definito per induzione sulla struttura grafica degli schemi di flusso come il più piccolo insieme di schemi di flusso tale che:
  - Il primo schema di flusso in Fig. 4.2 è strutturato.
  - Se F_1, F_2 ed F sono schemi di flusso strutturati dai quali sono stati tolti il nodo di inizio e il nodo di fine, allora ciascuno degli altri tre schemi di flusso in Fig. 4.2 è strutturato.

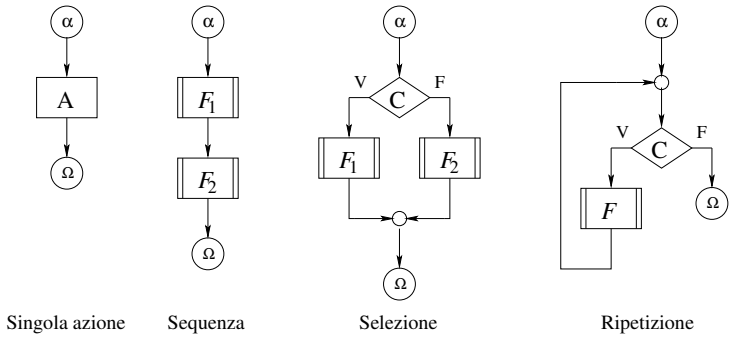
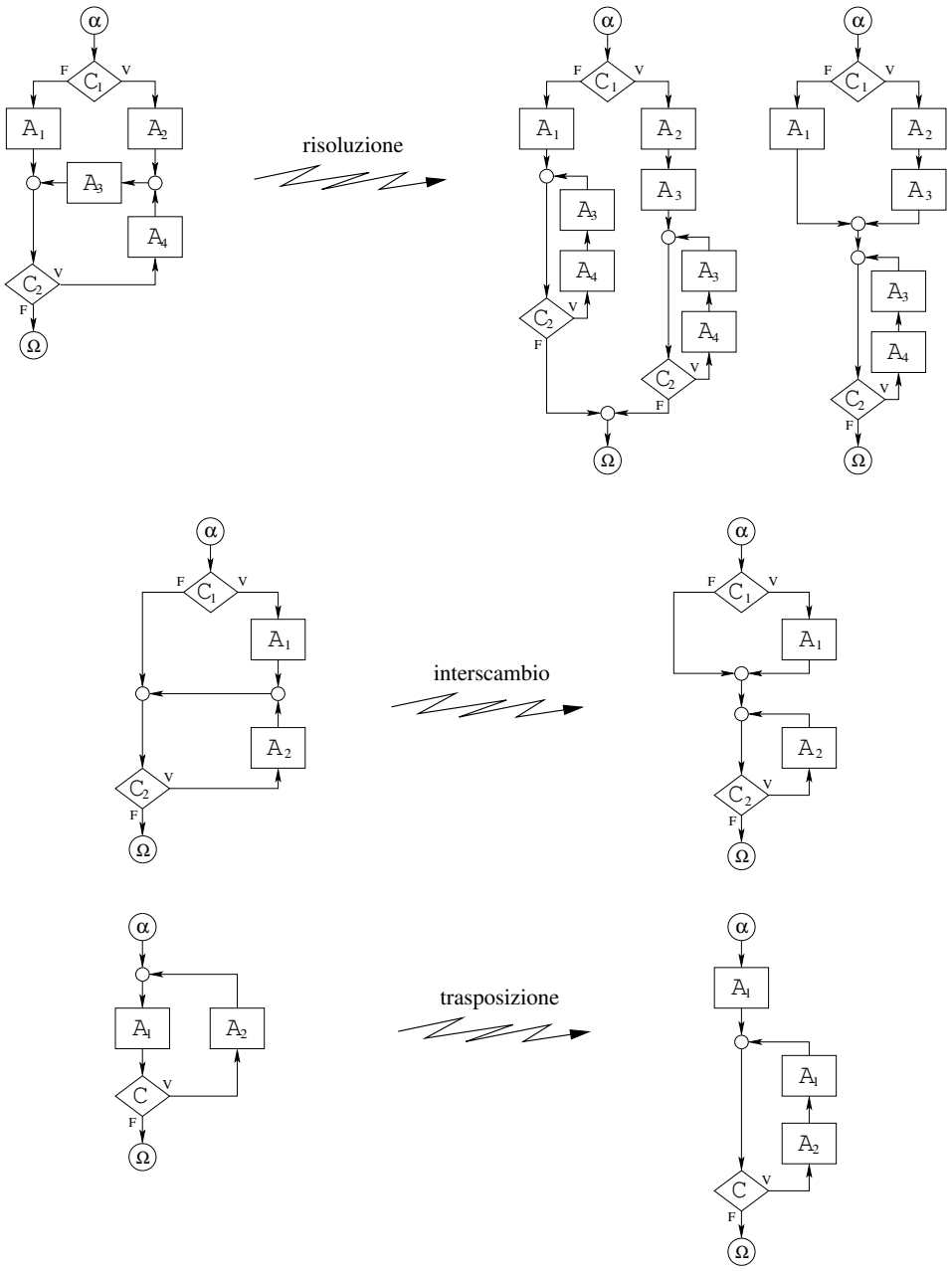


Figura 4.2: Schemi di flusso strutturati

- Proprietà: ogni schema di flusso strutturato ha un unico nodo di inizio e un unico nodo di fine.
- Teorema fondamentale della programmazione strutturata (Böhm e Jacopini): Dato un programma P ed uno schema di flusso F che lo descrive, è sempre possibile determinare un programma P' *equivalente* a P che è descrivibile con uno schema di flusso F' *strutturato*.
- In virtù del teorema precedente, è sempre possibile evitare l'uso dell'istruzione `goto` qualora il linguaggio impiegato metta a disposizione dei meccanismi di sequenza, selezione e ripetizione.
- Se uno schema di flusso non è strutturato, è possibile renderlo tale applicando le seguenti regole:
  - Risoluzione: duplicare e separare blocchi condivisi aggiungendo ulteriori nodi collettori.
  - Interscambio: scambiare le linee di ingresso/uscita di nodi collettori adiacenti.
  - Trasposizione: scambiare un blocco di azione con un nodo collettore o un blocco di controllo, a patto di preservare la semantica.

- Esempi di applicazione delle regole di strutturazione:



Nel primo esempio, ci sono due alternative a seconda che si separino le due vie dell'istruzione di selezione iniziale (duplicando l'istruzione di ripetizione finale) oppure l'istruzione di selezione iniziale dall'istruzione di ripetizione finale (duplicando soltanto il blocco di azione `A3`). La seconda alternativa è dunque preferibile perché comporta meno ridondanza della prima alternativa. ■fltpp_9

Capitolo 5

Procedure

5.1 Formato di un programma con più funzioni su un singolo file

- Un problema complesso viene di solito affrontato suddividendolo in sottoproblemi più semplici. Ciò è supportato dal paradigma di programmazione imperativo di natura procedurale attraverso la possibilità di articolare il programma che dovrà risolvere il problema in più sottoprogrammi o procedure – detti funzioni nel linguaggio C – che dovranno risolvere i sottoproblemi. Tale meccanismo prende il nome di progettazione top-down e consente lo sviluppo di programmi modulari per raffinamenti successivi.
- Una funzione C è una sequenza di istruzioni logicamente correlate che lavorano su un insieme di dati parametrizzati. È utile esprimere tale sequenza di istruzioni come funzione quando la loro esecuzione è ripetuta in punti diversi del programma (se la loro esecuzione fosse ripetuta in un solo punto, basterebbe inserirle all'interno di un'istruzione di ripetizione).
- Vantaggi derivanti dalla suddivisione di un programma C in funzioni:
  - Migliore articolazione del programma con conseguente snellimento della funzione `main`.
  - Riuso del software: una funzione viene definita una sola volta, ma può essere usata più volte sia all'interno del programma in cui è definita che in altri programmi (minore lunghezza dei programmi, minore tempo necessario per scrivere i programmi, maggiore affidabilità dei programmi).
  - Possibilità di suddividere il lavoro in modo coordinato all'interno di un gruppo di programmatori.
- Una funzione C è assimilabile ad una funzione matematica $f : A \to B$, $f(a) = b$, dove diciamo che f è l'identificatore della funzione, A è il tipo dei parametri, B è il tipo del risultato, $f : A \to B$ è la dichiarazione della funzione ed $f(a) = b$ è la definizione della funzione.
- Formato di un programma C con più funzioni su un singolo file:
  ◁*direttive al preprocessore*▷
  ◁*definizione dei tipi*▷
  ◁*dichiarazione delle variabili globali*▷
  ◁*dichiarazione delle funzioni (esclusa* `main`*)*▷
  ◁*definizione delle funzioni (a partire da* `main`*)*▷
- L'ordine in cui le funzioni vengono dichiarate/definite è inessenziale dal punto di vista della loro esecuzione. È invece importante che le funzioni vengano tutte dichiarate prima di essere definite affinché il compilatore possa risolvere eventuali invocazioni incrociate tra più funzioni. L'esecuzione del programma inizia sempre dalla prima istruzione della funzione `main` e poi continua seguendo l'ordine testuale delle istruzioni presenti nella funzione `main` e nelle funzioni che vengono via via invocate.
- Le variabili globali sono utilizzabili all'interno di ciascuna funzione che segue la loro dichiarazione, ad eccezione di quelle funzioni in cui i loro identificatori vengono ridichiarati come parametri formali o variabili locali. Per motivi legati alla correttezza dei programmi, l'uso delle variabili globali è sconsigliato, perché il fatto che più funzioni possano modificare il valore di tali variabili rende difficile tenere sotto controllo l'evoluzione dei valori che le variabili stesse assumono a tempo di esecuzione.

5.2 Dichiarazione di funzione

- Una funzione C viene dichiarata nel seguente modo:
  ◁*tipo risultato*▷ ◁*identificatore funzione*▷(◁*tipi parametri formali*▷);
- Il tipo del risultato rappresenta il tipo del valore che viene restituito dalla funzione quando l'esecuzione della funzione termina. Tale tipo è `void` se la funzione non restituisce alcun risultato.
- I tipi dei parametri formali sono costituiti dalla sequenza dei tipi degli argomenti della funzione separati da virgole. Tale sequenza è `void` se la funzione non ha argomenti.

5.3 Definizione di funzione e parametri formali

- Una funzione C viene definita nel seguente modo:
  ◁*tipo risultato*▷ ◁*identificatore funzione*▷(◁*dichiarazione parametri formali*▷)
  {
  ◁*dichiarazione variabili locali*▷
  ◁*istruzioni*▷
  }
  dove la prima linea costituisce l'intestazione della funzione, mentre ciò che è racchiuso tra parentesi graffe costituisce il corpo della funzione ed è indentato per motivi di leggibilità.
- La dichiarazione dei parametri formali è costituita da una sequenza di dichiarazioni di variabili separate da virgole che rappresentano gli argomenti della funzione. Tale sequenza è `void` se la funzione non ha argomenti.
- L'intestazione della funzione deve coincidere con la dichiarazione della funzione a meno dei nomi dei parametri formali e del punto e virgola finale.
- Gli identificatori dei parametri formali e delle variabili locali sono utilizzabili solo all'interno del corpo della funzione.

5.4 Invocazione di funzione e parametri effettivi

- Una funzione C viene invocata nel seguente modo:
  ◁*identificatore funzione*▷(◁*parametri effettivi*▷)
- I parametri effettivi sono costituiti da una sequenza di espressioni separate da virgole, i cui valori sono usati ordinatamente da sinistra a destra per inizializzare i parametri formali della funzione invocata. Se la funzione invocata non ha argomenti, la sequenza è vuota.
- Parametri effettivi e parametri formali devono corrispondere per numero, ordine e tipo. Se un parametro formale e il corrispondente parametro effettivo hanno tipi diversi compresi nell'insieme {`int`, `double`}, valgono le considerazioni fatte in Sez. 3.10 per gli operatori di assegnamento.
- Dal punto di vista dell'esecuzione delle istruzioni, l'effetto dell'invocazione di una funzione è quello di far diventare la prima istruzione di quella funzione la prossima istruzione da eseguire.

5.5 Istruzione `return`

- Se il tipo del risultato di una funzione è diverso da `void`, nel corpo della funzione sarà presente la seguente istruzione:
  `return`(◁*espressione*▷);
  la quale restituisce come risultato della funzione il valore dell'espressione, che deve essere del tipo dichiarato per il risultato della funzione.
- Dal punto di vista dell'esecuzione delle istruzioni, l'effetto dell'istruzione `return` è quello di far diventare l'istruzione successiva a quella contenente l'invocazione originale della funzione la prossima istruzione da eseguire.
- Per coerenza con i principi della programmazione strutturata, una funzione che restituisce un risultato deve contenere un'unica istruzione `return`. Inoltre, nel corpo della funzione non ci dovrebbero essere ulteriori istruzioni dopo l'istruzione `return`, in quanto queste non potrebbero mai essere eseguite.

5.6 Parametri e risultato della funzione `main`

- La funzione `main` è dotata di due parametri formali inizializzati dal sistema operativo in base alle stringhe (opzioni e nomi di file) presenti nel comando con cui il programma viene lanciato in esecuzione.
- Se tali parametri debbono essere utilizzabili all'interno del programma, l'intestazione della funzione `main` deve essere estesa come segue (notare l'allineamento dei parametri per motivi di leggibilità):
```
int main(int   argc,
         char *argv[])
```
- Il parametro `argc` contiene il numero di stringhe presenti nel comando, incluso il nome del file eseguibile del programma.
- Il parametro `argv` è un vettore contenente le stringhe presenti nel comando, incluso il nome del file eseguibile del programma.
- Esempio di lancio in esecuzione di un programma il cui file eseguibile si chiama `pippo`:
```
pippo -r dati.txt
```
dove l'opzione specificata stabilisce se il file che la segue deve essere letto o scritto. In questo caso, `argc` vale `3` e `argv` contiene le stringhe `"pippo"`, `"-r"` e `"dati.txt"`.
- Il risultato restituito dalla funzione `main` attraverso l'istruzione `return` è un valore di controllo che viene passato al sistema operativo per verificare se l'esecuzione del programma è andata a buon fine. Il valore che viene normalmente restituito è `0`. ■fltpp_10

5.7 Passaggio di parametri per valore e per indirizzo

- Nella dichiarazione di un parametro di una funzione occorre stabilire se il corrispondente parametro effettivo deve essere passato per valore o per indirizzo:
  - Se il parametro effettivo viene passato per valore, il valore della relativa espressione viene copiato nell'area di memoria riservata al corrispondente parametro formale.
  - Se il parametro effettivo viene passato per indirizzo, la relativa espressione viene interpretata come un indirizzo di memoria e questo viene copiato nell'area di memoria riservata al corrispondente parametro formale.
- Qualora il parametro effettivo sia una variabile, nel primo caso il valore della variabile non può essere modificato dalla funzione invocata durante la sua esecuzione, in quanto ciò che viene passato è una copia del valore di quella variabile. Per contro, nel secondo caso il corrispondente parametro formale contiene l'indirizzo della variabile, quindi attraverso questo parametro la funzione invocata può modificare il valore della variabile durante la sua esecuzione.
- Diversamente dal passaggio per valore, il passaggio per indirizzo deve essere esplicitamente dichiarato:
  - Se un parametro effettivo passato per indirizzo è di tipo *`tipo`*, il corrispondente parametro formale deve essere dichiarato di tipo *`tipo *`*.
  - Se `p` è un parametro formale di tipo *`tipo *`*, all'interno delle istruzioni della funzione in cui `p` è dichiarato si denota con `p` l'indirizzo contenuto in `p`, mentre si denota con `*p` il valore contenuto nell'area di memoria il cui indirizzo è contenuto in `p` (vedi operatore valore-di in Sez. 6.11).
  - Se `v` è una variabile passata per indirizzo, essa viene denotata con `&v` all'interno dell'invocazione di funzione (vedi operatore indirizzo-di in Sez. 6.11).
- Normalmente i parametri di una funzione sono visti come dati di input, nel qual caso il passaggio per valore è sufficiente. Se però in una funzione alcuni parametri rappresentano dati di input/output, oppure la funzione – come nel caso della `scanf` – deve restituire più risultati (l'istruzione `return` permette di restituirne uno solo), allora è necessario ricorrere al passaggio per indirizzo.

- Esempio di passaggio per valore e passaggio per indirizzo di una variabile:

```
...
  pippo1(v);
  w = v + 3;
...
void pippo1(int n)
{
  n += 10;
  printf("valore incrementato: %d",
         n);
}
...
```

```
...
  pippo2(&v);
  w = v + 3;
...
void pippo2(int *n)
{
  *n += 10;
  printf("valore incrementato: %d",
         *n);
}
...
```

  Se `v` ha valore `5`, in entrambi i casi il valore che viene stampato è `15`. La differenza è che nel primo caso il valore che viene assegnato a `w` è `8`, mentre nel secondo caso è `18`.

- Esempio di programma: aritmetica con le frazioni.

  1. **Specifica del problema**. Calcolare il risultato della addizione, sottrazione, moltiplicazione o divisione di due frazioni, mostrandolo ancora in forma di frazione.
  2. **Analisi del problema**. L'input è costituito dalle due frazioni e dall'operatore aritmetico da applicare ad esse. L'output è costituito dal risultato dell'applicazione dell'operatore aritmetico alle due frazioni, con il risultato da esprimere ancora sotto forma di frazione. Le relazioni da sfruttare sono le leggi dell'aritmetica con le frazioni: date $\frac{n_1}{d_1}$ ed $\frac{n_2}{d_2}$ dove $n_1, n_2 \in \mathbb{Z}$ e $d_1, d_2 \in \mathbb{Z} \setminus \{0\}$, vale che $\frac{n_1}{d_1} \pm \frac{n_2}{d_2} = \frac{n_1 \cdot d_2 \pm n_2 \cdot d_1}{d_1 \cdot d_2}$, $\frac{n_1}{d_1} \cdot \frac{n_2}{d_2} = \frac{n_1 \cdot n_2}{d_1 \cdot d_2}$, $\frac{n_1}{d_1} : \frac{n_2}{d_2} = \frac{n_1}{d_1} \cdot \frac{d_2}{n_2}$ (se $n_2 \neq 0$).
  3. **Progettazione dell'algoritmo**. Poiché una frazione è costituita da due numeri interi detti rispettivamente numeratore e denominatore, dove il denominatore deve essere diverso da zero, conveniamo di rappresentare il segno della frazione nel numeratore, cosicché il denominatore deve essere un numero intero strettamente positivo. Stabiliamo inoltre di rappresentare l'espressione in notazione infissa, cosicché l'operatore deve essere acquisito tra le due frazioni.
     I passi dell'algoritmo sono i seguenti:
     - Acquisire la prima frazione.
     - Acquisire l'operatore aritmetico.
     - Acquisire la seconda frazione.
     - Applicare l'operatore aritmetico.
     - Comunicare il risultato sotto forma di frazione.

     I passi riportati sopra possono essere svolti attraverso altrettante chiamate di funzioni. Si può pensare di sviluppare una funzione per la lettura di una frazione (che verrà richiamata due volte), una funzione per la lettura di un operatore aritmetico e una funzione per la stampa di una frazione. Si può inoltre pensare di sviluppare una funzione per ciascuna delle quattro operazioni aritmetiche. In realtà, abbiamo soltanto bisogno di una funzione per l'addizione – utilizzabile anche in una sottrazione a patto di cambiare preventivamente il segno del numeratore della seconda frazione – e di una funzione per la moltiplicazione – utilizzabile anche in una divisione a patto di scambiare preventivamente tra loro numeratore e denominatore della seconda frazione.
  4. **Implementazione dell'algoritmo**. Questa è la traduzione dei passi in C:

```
/*********************************************/
/* programma per l'aritmetica con le frazioni */
/*********************************************/
```

```
/*****************************/
/* inclusione delle librerie */
/*****************************/

#include <stdio.h>

/*********************************/
/* dichiarazione delle funzioni */
/*********************************/

void leggi_frazione(int *,
                    int *);
char leggi_operatore(void);
void somma_frazioni(int,
                    int,
                    int,
                    int,
                    int *,
                    int *);
void moltiplica_frazioni(int,
                         int,
                         int,
                         int,
                         int *,
                         int *);
void stampa_frazione(int,
                     int);

/*****************************/
/* definizione delle funzioni */
/*****************************/

/* definizione della funzione main */
int main(void)
{
  /* dichiarazione delle variabili locali alla funzione */
  int  n1,   /* input: numeratore della prima frazione */
       d1,   /* input: denominatore della prima frazione */
       n2,   /* input: numeratore della seconda frazione */
       d2;   /* input: denominatore della seconda frazione */
  char op;   /* input: operatore aritmetico da applicare */
  int  n,    /* output: numeratore della frazione risultato */
       d;    /* output: denominatore della frazione risultato */

  /* acquisire la prima frazione */
  leggi_frazione(&n1,
                 &d1);

  /* acquisire l'operatore aritmetico */
  op = leggi_operatore();
```

```
  /* acquisire la seconda frazione */
  leggi_frazione(&n2,
                 &d2);

  /* applicare l'operatore aritmetico */
  switch (op)
  {
    case '+':
      somma_frazioni(n1,
                     d1,
                     n2,
                     d2,
                     &n,
                     &d);
      break;
    case '-':
      somma_frazioni(n1,
                     d1,
                     -n2,
                     d2,
                     &n,
                     &d);
      break;
    case '*':
      moltiplica_frazioni(n1,
                          d1,
                          n2,
                          d2,
                          &n,
                          &d);
      break;
    case ':':
      if (n2 != 0)
        moltiplica_frazioni(n1,
                            d1,
                            d2,
                            n2,
                            &n,
                            &d);
      break;
  }

  /* comunicare il risultato sotto forma di frazione */
  if ((op != ':') || (n2 != 0))
    stampa_frazione(n,
                    d);
  else
    printf("Divisione illegale!");
  return(0);
}
```

```
/* definizione della funzione per leggere una frazione */
void leggi_frazione(int *num,   /* output: numeratore della frazione */
                    int *den)   /* output: denominatore della frazione */
{
  /* dichiarazione delle variabili locali alla funzione */
  char sep;   /* lavoro: carattere che separa numeratore e denominatore */

  /* leggere e validare la frazione */
  do
  {
    printf("Digita una frazione come coppia di interi separati da \"/\" ");
    printf("con il secondo intero strettamente positivo: ");
    scanf("%d %c%d",
          num,
          &sep,
          den);
  }
  while ((sep != '/') ||
         (*den <= 0));
}

/* definizione della funzione per leggere un operatore aritmetico */
char leggi_operatore(void)
{
  /* dichiarazione delle variabili locali alla funzione */
  char op;   /* output: operatore aritmetico */

  /* leggere e validare l'operatore aritmetico */
  do
  {
    printf("Digita un operatore aritmetico (+, -, *, :): ");
    scanf(" %c",
          &op);
  }
  while ((op != '+') &&
         (op != '-') &&
         (op != '*') &&
         (op != ':'));
  return(op);
}

/* definizione della funzione per sommare due frazioni */
void somma_frazioni(int n1,   /* input: numeratore della prima frazione */
                    int d1,   /* input: denominatore della prima frazione */
                    int n2,   /* input: numeratore della seconda frazione */
                    int d2,   /* input: denominatore della seconda frazione */
                    int *n,   /* output: numeratore della frazione risultato */
                    int *d)   /* output: denominatore della frazione risultato */
{
  /* sommare le due frazioni */
  *n = n1 * d2 + n2 * d1;
  *d = d1 * d2;
}
```

```
/* definizione della funzione per moltiplicare due frazioni */
void moltiplica_frazioni(int n1,   /* input: numeratore della prima frazione */
                         int d1,   /* input: denominatore della prima frazione */
                         int n2,   /* input: numeratore della seconda frazione */
                         int d2,   /* input: denominatore della seconda frazione */
                         int *n,   /* output: numeratore della frazione risultato */
                         int *d)   /* output: denominatore della frazione risultato */
{
  /* moltiplicare le due frazioni */
  *n = n1 * n2;
  *d = d1 * d2;
}

/* definizione della funzione per stampare una frazione */
void stampa_frazione(int num,   /* input: numeratore della frazione */
                     int den)   /* input: denominatore della frazione */
{
  /* stampare la frazione */
  printf("La frazione risultato e' %d/%d\n",
         num,
         den);
}
```

■fltpp_11

5.8 Funzioni ricorsive

- Una funzione ricorsiva è una funzione che invoca direttamente o indirettamente se stessa. La ricorsione è un meccanismo fondamentale per risolvere in modo elegante o efficace determinati problemi.
- Essendo basata sul principio di induzione, la ricorsione è adeguata per risolvere qualsiasi problema che sia suddivisibile in sottoproblemi più semplici *della stessa natura di quello originario*, cioè:
  - Bisogna individuare dei casi base per i quali si può ricavare direttamente la soluzione del problema.
  - Il caso generale deve essere definito attraverso un insieme di sottoproblemi della stessa natura di quello originario che sono più vicini *di un passo* ai casi base. La soluzione del caso generale è data dalla combinazione delle soluzioni dei sottoproblemi tramite i quali il caso stesso è stato definito.
- Esempi:
  - In virtù dell'assiomatizzazione di Peano, le quattro operazioni aritmetiche su $\mathbb{N}$ possono essere espresse in modo ricorsivo utilizzando solo gli operatori di incremento e decremento di un'unità e gli operatori relazionali:

```
int addizione(int m,   /* m >= 0 */
              int n)   /* n >= 0 */
{
  int somma;

  if (n == 0)
    somma = m;
  else
    somma = addizione(m + 1,
                      n - 1);
  return(somma);
}
```

```
int sottrazione(int m,   /* m >= n */
                int n)   /* n >= 0 */
{
  int differenza;

  if (n == 0)
    differenza = m;
  else
    differenza = sottrazione(m - 1,
                             n - 1);
  return(differenza);
}

int moltiplicazione(int m,   /* m >= 0 */
                    int n)   /* n >= 0 */
{
  int prodotto;

  if (n == 0)
    prodotto = 0;
  else
    prodotto = addizione(m,
                         moltiplicazione(m,
                                         n - 1));
  return(prodotto);
}

void divisione(int m,            /* m >= 0 */
               int n,            /* n > 0 */
               int *quoziente,
               int *resto)
{
  if (m < n)
  {
    *quoziente = 0;
    *resto = m;
  }
  else
  {
    divisione(sottrazione(m,
                          n),
              n,
              quoziente,
              resto);
    *quoziente += 1;
  }
}
```

– Altre operazioni matematiche su $\mathbb{N}$ possono essere espresse in modo ricorsivo utilizzando solo le quattro operazioni aritmetiche e gli operatori relazionali:

```
int elevamento(int m,   /* m >= 0 */
               int n)   /* n >= 0, n != 0 se m == 0 */
{
  int potenza;

  if (n == 0)
    potenza = 1;
  else
    potenza = m * elevamento(m,
                             n - 1);
  return(potenza);
}

int massimo_comun_divisore(int m,   /* m >= n */
                           int n)   /* n > 0 */
{
  int mcd;

  if (m % n == 0)
    mcd = n;
  else
    mcd = massimo_comun_divisore(n,
                                 m % n);
  return(mcd);
}

int fattoriale(int n)   /* n >= 0 */
{
  int fatt;

  if (n == 0)
    fatt = 1;
  else
    fatt = n * fattoriale(n - 1);
  return(fatt);
}
```

– L'n-esimo numero di Fibonacci fib_n è il numero di coppie di conigli esistenti nel periodo n sotto le seguenti ipotesi: nel periodo 1 viene ad esistere la prima coppia di conigli, nessuna coppia è fertile nel primo periodo successivo al periodo in cui è avvenuta la sua nascita, ogni coppia produce un'ulteriore coppia in ciascuno degli altri periodi successivi. Nel periodo 1 abbiamo dunque una sola coppia di conigli, che non è ancora fertile nel periodo 2 e comincia a produrre nuove coppie a partire dal periodo 3. Il numero di coppie esistenti nel generico periodo n è il numero di coppie esistenti nel periodo $n-1$ più il numero di nuove coppie nate nel periodo n, le quali sono tante quante le coppie fertili nel periodo n, che coincidono a loro volta con le coppie esistenti nel periodo $n-2$:

```
int fibonacci(int n)   /* n >= 1 */
{
  int fib;

  if ((n == 1) || (n == 2))
    fib = 1;
  else
    fib = fibonacci(n - 1) + fibonacci(n - 2);
  return(fib);
}
```

– Il problema delle torri di Hanoi è il seguente. Date tre aste di altezza sufficiente con n dischi diversi accatastati sulla prima asta in ordine di diametro decrescente dal basso verso l'alto, portare i dischi sulla terza asta rispettando le due seguenti regole: (i) è possibile spostare un solo disco alla volta e (ii) un disco non può mai essere appoggiato sopra un disco di diametro inferiore. Osservato che per $n = 1$ la soluzione è banale, quando $n \geq 2$ si può adottare un meccanismo ricorsivo del seguente tipo in cui i ruoli delle aste si scambiano. Spostare gli $n - 1$ dischi di diametro più piccolo dalla prima alla seconda asta usando questo meccanismo ricorsivo, poi spostare il disco di diametro più grande direttamente dalla prima alla terza asta, e infine spostare gli $n - 1$ dischi di diametro più piccolo dalla seconda alla terza asta usando questo meccanismo ricorsivo:

```
void hanoi(int n,            /* n >= 1 */
           int partenza,
           int arrivo,
           int intermedia)
{
  if (n == 1)
    printf("Sposta da %d a %d.\n",
           partenza,
           arrivo);
  else
  {
    hanoi(n - 1,
          partenza,
          intermedia,
          arrivo);
    printf("Sposta da %d a %d.\n",
           partenza,
           arrivo);
    hanoi(n - 1,
          intermedia,
          arrivo,
          partenza);
  }
}
```

■fltpp_12

5.9 Modello di esecuzione sequenziale basato su pila

- Le istruzioni di un programma C vengono eseguite una alla volta nell'ordine in cui sono state scritte e la loro esecuzione è sequenziale a meno di invocazioni di funzioni. Quando il programma viene lanciato in esecuzione, il sistema operativo riserva tre aree distinte di memoria principale per il programma:
  – Un'area per contenere la versione eseguibile delle istruzioni del programma.
  – Un'area destinata come stack (o pila) per contenere un record di attivazione per ogni invocazione di funzione la cui esecuzione non è ancora terminata.
  – Un'area destinata come heap per l'allocazione/disallocazione delle strutture dati dinamiche.
- A seguito dell'invocazione di una funzione, il sistema operativo compie i seguenti passi (in maniera del tutto trasparente all'utente del programma):
  – Un record di attivazione per la funzione viene allocato in cima allo stack, la cui dimensione è tale da poter contenere i valori di parametri formali e variabili locali della funzione, più alcune informazioni di controllo. Questo record di attivazione viene a trovarsi subito sopra a quello della funzione che ha invocato la funzione in esame.
  – Lo spazio riservato ai parametri formali viene inizializzato con i valori dei corrispondenti parametri effettivi contenuti nell'invocazione.

  - Tra le informazioni di controllo viene memorizzato l'indirizzo dell'istruzione eseguibile successiva a quella contenente l'invocazione (indirizzo di ritorno), il quale viene preso dal program counter.
  - Il registro program counter viene impostato con l'indirizzo della prima istruzione eseguibile della funzione invocata, così da continuare l'esecuzione del programma da quella istruzione.

- A seguito della terminazione dell'esecuzione di una funzione, il sistema operativo compie i seguenti passi (in maniera del tutto trasparente all'utente del programma):

  - L'eventuale risultato restituito dalla funzione viene memorizzato nel record di attivazione della funzione che ha invocato la funzione in esame.
  - Il registro program counter viene impostato con l'indirizzo di ritorno precedentemente memorizzato nel record di attivazione, così da riprendere l'esecuzione del programma da quel punto.
  - Il record di attivazione viene disallocato dalla cima dello stack, cosicché il record di attivazione della funzione che ha invocato la funzione in esame viene di nuovo a trovarsi in cima allo stack.

- Invece di allocare staticamente un record di attivazione per ogni funzione all'inizio dell'esecuzione del programma, si utilizza un modello di esecuzione a pila (implementato attraverso lo stack dei record di attivazione) in cui i record di attivazione sono associati alle invocazioni delle funzioni – non alle funzioni – e vengono dinamicamente allocati/disallocati in ordine last-in-first-out (LIFO).

- Il motivo per cui si usa il modello di esecuzione a pila anziché il più semplice modello statico è che quest'ultimo non supporta la corretta esecuzione delle funzioni ricorsive. Prevedendo un unico record di attivazione per ciascuna funzione ricorsiva (invece di un record distinto per ogni invocazione di una funzione ricorsiva), il modello statico provoca interferenza tra le diverse invocazioni ricorsive di una funzione. Infatti in tale modello ogni invocazione ricorsiva finisce per sovrascrivere dentro al record di attivazione della funzione i valori dei parametri formali e delle variabili locali della precedente invocazione ricorsiva, determinando così il calcolo di un risultato errato.

5.10 Formato di un programma con più funzioni su più file

- Un programma C articolato in funzioni può essere distribuito su più file. Questa organizzazione, ormai prassi consolidata nel caso di grossi sistemi software, si basa sullo sviluppo e sull'utilizzo di librerie, ciascuna delle quali contiene funzioni e strutture dati logicamente correlate tra loro.
- L'organizzazione di un programma C complesso su più file enfatizza i vantaggi dell'articolazione del programma in funzioni:
  - Le funzionalità offerte dalle funzioni e dalle strutture dati di una libreria ("cosa") possono essere separate dai relativi dettagli implementativi ("come"), che rimangono di conseguenza nascosti a chi utilizza la libreria e possono essere modificati in ogni momento senza alterare le funzionalità offerte dalla libreria stessa.
  - Il grado di riuso del software aumenta, in quanto una funzione o una struttura dati di una libreria può essere usata più volte non solo all'interno di un unico programma, ma all'interno di tutti i programmi che includeranno la libreria (in generale, sviluppare una libreria ha senso solo se è ragionevole prevederne l'uso in più programmi).

- Una libreria C consiste in un file di implementazione e un file di intestazione aventi nomi coerenti tra loro:
  - Il file di implementazione (`.c`) ha il seguente formato:

```
◁direttive al preprocessore (costanti simboliche da esportare e interne)▷
◁definizione dei tipi (da esportare e interni)▷
◁dichiarazione delle variabili globali (da esportare e interne)▷
◁dichiarazione delle funzioni (da esportare e interne)▷
◁definizione delle funzioni (da esportare e interne) (no main)▷
```

In altri termini, il formato è lo stesso di un programma con più funzioni su un singolo file, ad eccezione della funzione `main` che non può essere definita all'interno di una libreria. Viene inoltre fatta distinzione tra gli identificatori da esportare – cioè utilizzabili nei programmi che includeranno il file di intestazione della libreria – e gli identificatori interni alla libreria – cioè la cui definizione è di supporto alla definizione degli identificatori da esportare.

– Il file di intestazione (`.h`) ha il seguente formato:
  ◁*ridefinizione delle costanti simboliche esportate*▷
  ◁*ridefinizione dei tipi esportati*▷
  ◁*ridichiarazione delle variabili globali esportate (precedute da* `extern`*)*▷
  ◁*ridichiarazione delle funzioni esportate (precedute da* `extern`*)*▷
  Questo file di intestazione rende disponibili per l'uso tutti gli identificatori in esso contenuti – i quali sono definiti nel corrispondente file di implementazione – ai programmi che includeranno il file di intestazione stesso. La ridichiarazione delle variabili globali e delle funzioni esportate deve essere preceduta da `extern`.

- In un programma organizzato su più file esiste solitamente un modulo principale – che è un file `.c` – il quale contiene la definizione della funzione `main` – che deve essere unica in tutto il programma – e include i file di intestazione di tutte le librerie necessarie. Il modulo principale e i file di implementazione delle librerie incluse vengono compilati separatamente in modo parziale, producendo così altrettanti file oggetto che vengono poi collegati assieme per ottenere un unico file eseguibile.

- Durante la compilazione parziale del modulo principale, il fatto che le ridichiarazioni delle funzioni importate dal modulo stesso siano precedute da `extern` consente al compilatore di sapere che i relativi identificatori sono definiti altrove, così da rimandare la ricerca delle loro definizioni al passo di linking. Se la ridichiarazione di una funzione importata dal modulo principale non fosse preceduta da `extern`, il compilatore cercherebbe la definizione di quella funzione nel modulo principale e, non trovandola, segnalerebbe errore.

- Esempio di libreria: aritmetica con le frazioni.

  – Il file di implementazione `frazioni.c` è uguale a quello riportato nella Sez. 5.7 dopo aver tolto la definizione della funzione `main`, quindi contiene tutte le dichiarazioni e definizioni di funzioni.

  – Il file di intestazione `frazioni.h`, da non includere nel file di implementazione di cui sopra, è il seguente:

```
/*****************************************************************/
/* intestazione della libreria per l'aritmetica con le frazioni */
/*****************************************************************/

/*******************************************/
/* ridichiarazione delle funzioni esportate */
/*******************************************/

extern void leggi_frazione(int *,
                           int *);
extern char leggi_operatore(void);
extern void somma_frazioni(int,
                           int,
                           int,
                           int,
                           int *,
                           int *);
```

```
extern void moltiplica_frazioni(int,
                                int,
                                int,
                                int,
                                int *,
                                int *);
extern void stampa_frazione(int,
                            int);
```

– Il modulo principale (`.c`) contiene `#include "frazioni.h"`, così da poter invocare le funzioni esportate dalla libreria, e la definizione della funzione main della Sez. 5.7.

5.11 Visibilità degli identificatori locali e non locali

- Ad ogni identificatore presente in un programma C è associato un campo di visibilità. Questo definisce la regione del programma in cui l'identificatore è utilizzabile.
- Un identificatore locale denota un parametro formale o una variabile locale di una funzione e ha come campo di visibilità soltanto la funzione stessa. All'inizio dell'esecuzione di un'invocazione della funzione, ogni parametro formale è inizializzato col valore del corrispondente parametro effettivo contenuto nell'invocazione, mentre il valore di ciascuna variabile locale è indefinito a meno che la variabile non sia esplicitamente inizializzata nella sua dichiarazione. Gli identificatori dei parametri formali e delle variabili locali di una funzione devono essere tutti distinti.
- Un identificatore non locale denota una costante simbolica, un tipo, una variabile globale o una funzione e ha come campo di visibilità la parte del file di implementazione in cui l'identificatore è definito/dichiarato compresa tra la sua definizione/dichiarazione e il termine del file, escluse quelle funzioni in cui viene dichiarato un parametro formale o una variabile locale con lo stesso nome di quell'identificatore. Gli identificatori non locali di un programma devono essere tutti distinti.
- Esistono inoltre i seguenti qualificatori per modificare il campo di visibilità di identificatori non locali:
  – Se `static` precede la dichiarazione di una variabile globale o di una funzione in un file `.c`, il relativo identificatore non può essere esportato al di fuori di quel file di implementazione (utile nei file di implementazione delle librerie per impedire di esportare identificatori la cui definizione è solo di supporto agli identificatori da esportare). In altri termini, `static` congela il campo di visibilità dell'identificatore di una variabile globale o di una funzione limitandolo al file di implementazione nel quale l'identificatore è definito.
  – Se `extern` precede la ridichiarazione di una variabile globale o di una funzione in un file `.h`, il relativo identificatore è definito in un file di implementazione diverso da quello che include quel file di intestazione. In altri termini, `extern` permette di ampliare il campo di visibilità dell'identificatore di una variabile globale o di una funzione, rendendo l'identificatore visibile al di fuori del file di implementazione nel quale è definito (fondamentale per poter attuare l'esportazione di identificatori attraverso i file di intestazione delle librerie).

- Esistono infine i seguenti qualificatori per modificare la memorizzazione di identificatori locali:
  – Se `static` precede la dichiarazione di una variabile locale, la variabile locale viene allocata una volta per tutte all'inizio dell'esecuzione del programma, invece di essere allocata in cima allo stack dei record di attivazione ad ogni invocazione della relativa funzione. Ciò consente alla variabile di mantenere il valore che essa aveva al termine dell'esecuzione dell'invocazione precedente quando inizia l'esecuzione dell'invocazione successiva della relativa funzione (utile per alcune applicazioni, come i generatori di numeri pseudo-casuali, per evitare il ricorso a variabili globali).
  – Se `register` precede la dichiarazione di un parametro formale o di una variabile locale, il parametro formale o la variabile locale viene allocato, se possibile, in un registro della CPU anziché in una cella di memoria (utile per fare accesso più rapidamente ai parametri formali e alle variabili locali più frequentemente utilizzate). ■fltpp_13

Capitolo 6

Tipi di dati

6.1 Classificazione dei tipi di dati e operatore `sizeof`

- Un tipo di dato denota – come una struttura algebrica – un insieme di valori ai quali sono applicabili solo determinate operazioni. La dichiarazione del tipo degli identificatori presenti in un programma consente quindi al compilatore di rilevare errori staticamente (cioè senza eseguire il programma).
- In generale i tipi di dati si suddividono in scalari e strutturati:
  - I tipi scalari denotano insiemi di valori scalari, cioè non ulteriormente strutturati al loro interno (come i numeri e i caratteri).
  - I tipi strutturati denotano invece insiemi di valori aggregati i cui elementi possono essere omogenei (come nel caso di vettori e stringhe) oppure eterogenei (come nel caso di record e strutture lineari, gerarchiche e reticolari di dimensione dinamicamente variabile).
- In relazione ai tipi di dati del linguaggio C, si fa distinzione tra tipi scalari predefiniti, tipi standard, costruttori di tipo e tipi definiti dal programmatore:
  - I tipi scalari predefiniti sono `int` (e le sue varianti), `double` (e le sue varianti) e `char`.
  - I tipi standard sono definiti nelle librerie standard (p.e. `FILE`).
  - I costruttori di tipo sono `enum`, array ("`[]`"), `struct`, `union` e puntatore ("`*`").
  - Nuovi tipi di dati possono essere definiti dal programmatore nel seguente modo:
    `typedef` ◁*definizione del tipo*▷ ◁*identificatore del tipo*▷`;`
    usando in *definizione del tipo* i tipi scalari predefiniti, i tipi standard e i costruttori di tipo.
- L'informazione sulla quantità di memoria in byte necessaria per rappresentare un valore di un certo tipo, che dipende dalla specifica implementazione del linguaggio C, è reperibile nel seguente modo:
  `sizeof(`◁*tipo*▷`)`

6.2 Tipo `int`: rappresentazione e varianti

- Il tipo `int` denota l'insieme dei numeri interi rappresentabili con un certo numero di bit (sottoinsieme finito di $\mathbb{Z}$).
- Il minimo (risp. massimo) numero intero rappresentabile è indicato dalla costante simbolica `INT_MIN` (risp. `INT_MAX`) definita nel file di intestazione di libreria standard `limits.h`. Se si esce dalla gamma di valori `INT_MIN` .. `INT_MAX`, si ha un errore di overflow con generazione del valore `NaN` (not a number).
- Il numero di bit usati per la rappresentazione di un numero intero è dato da $\log_2$ `INT_MAX`, più un bit per la rappresentazione del segno.

- Varianti del tipo `int` e relative gamme minime di valori stabilite dallo standard ANSI, con indicazione del numero di bit usati per la rappresentazione:

`int`	-32767 .. 32767	1 + 15 bit
`unsigned`	0 .. 65535	16 bit
`short`	-32767 .. 32767	1 + 15 bit
`unsigned short`	0 .. 65535	16 bit
`long`	-2147483647 .. 2147483647	1 + 31 bit
`unsigned long`	0 .. 4294967295	32 bit

6.3 Tipo `double`: rappresentazione e varianti

- Il tipo `double` denota l'insieme dei numeri reali rappresentabili con un certo numero di bit (sottoinsieme finito di $\mathbb{R}$).
- Ogni numero reale (r) è rappresentato in memoria nel formato in virgola mobile attraverso due numeri interi espressi in formato binario detti mantissa (m) ed esponente (e), rispettivamente, tali che:
$$r = m \cdot 2^e$$
Il numero di bit riservati alla mantissa determina la precisione della rappresentazione, mentre il numero di bit riservati all'esponente determina l'ordine di grandezza della rappresentazione.
- Il minimo (risp. massimo) numero reale rappresentabile è indicato dalla costante simbolica `DBL_MIN` (risp. `DBL_MAX`) definita nel file di intestazione di libreria standard `float.h`. Se si esce dalla gamma di valori `DBL_MIN` .. `DBL_MAX`, si ha un errore di overflow con generazione del valore `NaN` (not a number).
- Il numero limitato di bit riservati alla mantissa, ovvero concettualmente il numero limitato di cifre rappresentabili dopo la virgola, può provocare anche errori di arrotondamento. In particolare, qualora un numero reale il cui valore assoluto è compreso tra 0 ed 1 venga rappresentato come 0, si ha un errore di underflow.
- Il numero di bit usati per la rappresentazione di un numero reale è dato da $\log_2$ *mantissa*(`DBL_MAX`) + $\log_2$ *esponente*(`DBL_MAX`), più un bit per la rappresentazione del segno della mantissa e un bit per la rappresentazione del segno dell'esponente.
- Varianti del tipo `double` e relative gamme minime di valori positivi stabilite dallo standard ANSI, con indicazione approssimativa del numero di bit usati per la rappresentazione dell'esponente:

`double`	10^{-307} .. 10^{308}	(2^{1024})	10 bit per l'esponente
`float`	10^{-37} .. 10^{38}	(2^{128})	7 bit per l'esponente
`long double`	10^{-4931} .. 10^{4932}	(2^{16384})	14 bit per l'esponente

6.4 Funzioni di libreria matematica

- Principali funzioni matematiche messe a disposizione dal linguaggio C, con indicazione dei relativi file di intestazione di libreria standard (spesso richiedono l'uso dell'opzione `-lm` nel comando di compilazione):

`int abs(int x)`	`stdlib.h`	$\lvert x \rvert$	
`double fabs(double x)`	`math.h`	$\lvert x \rvert$	
`double ceil(double x)`	`math.h`	$\lceil x \rceil$	
`double floor(double x)`	`math.h`	$\lfloor x \rfloor$	
`double sqrt(double x)`	`math.h`	$\sqrt{x}$	$x \geq 0$
`double exp(double x)`	`math.h`	e^x	
`double pow(double x, double y)`	`math.h`	x^y	$x < 0 \Rightarrow y \in \mathbb{Z}, x = 0 \Rightarrow y \neq 0$
`double log(double x)`	`math.h`	$\log_\mathrm{e} x$	$x > 0$
`double log10(double x)`	`math.h`	$\log_{10} x$	$x > 0$
`double sin(double x)`	`math.h`	$\sin x$	x espresso in radianti
`double cos(double x)`	`math.h`	$\cos x$	x espresso in radianti
`double tan(double x)`	`math.h`	$\tan x$	x espresso in radianti

6.5 Tipo char: rappresentazione e funzioni di libreria

- Il tipo `char` denota l'insieme dei caratteri comprendente le 26 lettere minuscole, le 26 lettere maiuscole, le 10 cifre decimali, i simboli di punteggiatura, le parentesi, gli operatori aritmetici e relazionali e i caratteri di spaziatura (spazio, tabulazione, andata a capo).
- Ogni carattere è rappresentato attraverso una sequenza lunga solitamente 8 bit in conformità ad un certo sistema di codifica, quale ASCII (American Standard Code for Information Interchange), EBCDIC (Extended Binary Coded Decimal Interchange Code), CDC (Control Data Corporation), ecc.
- Per non limitare la portabilità di un programma C, ogni valore di tipo `char` usato nel programma deve essere espresso attraverso la relativa costante (p.e. `'A'`) anziché il rispettivo codice (p.e. `65` nel caso di ASCII) perché quest'ultimo potrebbe cambiare a seconda del sistema di codifica adottato nei computer su cui il programma verrà eseguito.
- Lo standard ANSI richiede che, qualunque sia il sistema di codifica adottato, esso garantisca che:
  - Le 26 lettere minuscole siano ordinatamente rappresentate attraverso 26 codici consecutivi.
  - Le 26 lettere maiuscole siano ordinatamente rappresentate attraverso 26 codici consecutivi.
  - Le 10 cifre decimali siano ordinatamente rappresentate attraverso 10 codici consecutivi.
- Poiché i caratteri sono codificati attraverso numeri interi, c'è piena compatibilità tra il tipo `char` e il tipo `int`. Ciò significa che variabili e valori di tipo `char` possono far parte di espressioni aritmetico-logiche.
- Esempi resi possibili dalla consecutività dei codici delle lettere minuscole, delle lettere maiuscole e delle cifre decimali:
  - Verifica del fatto che il carattere contenuto in una variabile di tipo `char` sia una lettera maiuscola:

```
char c;

if (c >= 'A' && c <= 'Z')
  ...
```

  - Trasformazione del carattere che denota una cifra decimale nel valore numerico corrispondente alla cifra stessa:

```
char c;
int  n;

n = c - '0';
```

- Principali funzioni per il tipo `char` messe a disposizione dal linguaggio C, con indicazione dei relativi file di intestazione di libreria standard:

`int getchar(void)`	`stdio.h`	acquisisce un carattere da tastiera
`int putchar(int c)`	`stdio.h`	stampa un carattere su schermo
`int isalnum(int c)`	`ctype.h`	è un carattere alfanumerico?
`int isalpha(int c)`	`ctype.h`	è una lettera?
`int islower(int c)`	`ctype.h`	è una lettera minuscola?
`int isupper(int c)`	`ctype.h`	è una lettera maiuscola?
`int isdigit(int c)`	`ctype.h`	è una cifra decimale?
`int ispunct(int c)`	`ctype.h`	è un carattere diverso da lettera, cifra, spazio?
`int isspace(int c)`	`ctype.h`	è un carattere di spaziatura?
`int iscntrl(int c)`	`ctype.h`	è un carattere di controllo?
`int tolower(int c)`	`ctype.h`	trasforma una lettera in minuscolo
`int toupper(int c)`	`ctype.h`	trasforma una lettera in maiuscolo

6.6 Tipi enumerati

- Nel linguaggio C è possibile costruire ulteriori tipi scalari nel seguente modo:
  `enum {◁`*identificatori dei valori*`▷}`
  il quale prevede l'esplicita enumerazione degli identificatori dei valori assumibili dalle espressioni di questo tipo, con gli identificatori separati da virgole.

- Gli identificatori dei valori che compaiono nella definizione di un tipo enumerato non possono comparire nella definizione di un altro tipo enumerato.

- Se gli identificatori dei valori sono n, essi sono rappresentati da sinistra a destra mediante i numeri interi compresi tra 0 ed $n-1$. Ciò implica la piena compatibilità tra i tipi enumerati e il tipo `int`, quindi variabili e valori di un tipo enumerato possono far parte di espressioni aritmetico-logiche.

- Gli identificatori dei valori di un tipo enumerato sono assimilabili a constanti simboliche e perciò sono più comprensibili dell'uso diretto dei numeri.

- Esempi:

  – Definizione di un tipo per i valori di verità compatibile col fatto che 0 rappresenta falso:

```
typedef enum {falso,
              vero}  booleano_t;
```

  – Definizione di un tipo per i giorni della settimana:

```
typedef enum {lunedi,
              martedi,
              mercoledi,
              giovedi,
              venerdi,
              sabato,
              domenica}  giorno_t;
```

  – Definizione di un tipo per i mesi dell'anno:

```
typedef enum {gennaio,
              febbraio,
              marzo,
              aprile,
              maggio,
              giugno,
              luglio,
              agosto,
              settembre,
              ottobre,
              novembre,
              dicembre}  mese_t;
```

  – Calcolo del giorno successivo:

```
giorno_t oggi,
         domani;

domani = (oggi + 1) % 7;
```

6.7 Conversioni di tipo e operatore di cast

- Durante la valutazione delle espressioni aritmetico-logiche vengono effettuate le seguenti conversioni automatiche tra i tipi scalari visti sinora:
  - Se un operatore binario è applicato ad un operando di tipo `int` e un operando di tipo `double`, il valore dell'operando di tipo `int` viene convertito nel tipo `double` aggiungendogli una parte frazionaria nulla (`.0`) prima di applicare l'operatore.
  - Se un'espressione di tipo `int` deve essere assegnata ad una variabile di tipo `double`, il valore dell'espressione viene convertito nel tipo `double` aggiungendogli una parte frazionaria nulla (`.0`) prima di essere assegnato alla variabile.
  - Se un'espressione di tipo `double` deve essere assegnata ad una variabile di tipo `int`, il valore dell'espressione viene convertito nel tipo `int` tramite troncamento della parte frazionaria prima di essere assegnato alla variabile.
  - L'assegnamento dei valori dei parametri effettivi contenuti nell'invocazione di una funzione ai corrispondenti parametri formali della funzione invocata segue le regole precedenti.
- È inoltre possibile imporre delle conversioni esplicite di tipo alle espressioni attraverso l'operatore di cast:
  (◁*tipo*▷)◁*espressione*▷
  Esso ha l'effetto di convertire nel tipo specificato il valore dell'espressione cui è applicato prima che questo valore venga successivamente utilizzato. Se applicato ad una variabile, l'operatore di cast non ne altera il contenuto e produce il suo effetto solo nel contesto dell'espressione in cui è applicato (cioè il tipo originariamente dichiarato per la variabile viene preservato).
- L'operatore di cast "`()`", che è unario e prefisso, ha la stessa precedenza degli operatori unari aritmetico-logici (vedi Sez. 3.11).
- Esempi:
  - Dato:

    ```
    double x;
    int    y,
           z;

    x = y / z;
    ```

    se `y` vale `3` e `z` vale `2`, il risultato della loro divisione è `1`, il quale viene automaticamente convertito in `1.0` prima di essere assegnato ad `x`.
  - Dato:

    ```
    double x;
    int    y,
           z;

    x = (double)y / (double)z;
    ```

    se `y` vale `3` e `z` vale `2`, questi valori vengono esplicitamente convertiti in `3.0` e `2.0` rispettivamente prima di effettuare la divisione, cosicché il valore assegnato ad `x` è `1.5`.

6.8 Array: rappresentazione e operatore di indicizzazione

- Il costruttore di tipo array del linguaggio C dà luogo ad un valore aggregato formato da un numero finito di elementi dello stesso tipo, i quali sono memorizzati in celle consecutive di memoria.
- Una variabile di tipo array viene dichiarata come segue:
  ◁*tipo elementi*▷ ◁*identificatore variabile*▷[◁*espr_dich*▷];
  oppure con contestuale inizializzazione:
  ◁*tipo elementi*▷ ◁*identificatore variabile*▷[◁*espr_dich*▷] = {◁*sequenza valori*▷};
  dove:
  - Il numero di elementi (o lunghezza) della variabile di tipo array è dato da un'espressione di tipo `int` il cui valore deve essere positivo e i cui operandi devono essere delle costanti (no variabili). Ciò implica che il numero di elementi della variabile di tipo array è fissato staticamente.
  - Il numero di elementi può essere omesso se è specificata una sequenza di valori di inizializzazione separati da virgole aventi tutti tipo compatibile con quello dichiarato per gli elementi.
  - L'identificatore della variabile di tipo array rappresenta in forma simbolica l'indirizzo della locazione di memoria che contiene il valore del primo elemento dell'array.
- Essendo assimilabile ad una costante simbolica, l'identificatore di una variabile di tipo array non può comparire in un'istruzione a sinistra di un operatore di assegnamento. Ciò implica in particolare che il risultato di una funzione non può essere di tipo array.
- Ogni elemento di una variabile di tipo array è selezionato all'interno di un'istruzione tramite il suo indice:
  ◁*identificatore variabile*▷[◁*espr_indice*▷]
  dove l'espressione deve essere di tipo `int` (variabili ammesse). In virtù della memorizzazione consecutiva degli elementi, l'indirizzo dell'elemento considerato è dato dalla somma tra l'indirizzo denotato dalla variabile di tipo array (cioè l'indirizzo del primo elemento dell'array) e il valore dell'espressione: ◁*identificatore variabile*▷ + ◁*espr_indice*▷.
- Se il numero di elementi di una variabile di tipo array è n, gli elementi sono indicizzati da 0 a $n-1$. Il valore dell'espressione utilizzato all'interno di un operatore di indicizzazione deve rientrare nei limiti stabiliti, altrimenti viene selezionato un elemento che sta al di fuori dello spazio di memoria riservato alla variabile di tipo array (salvo casi specifici, nessun messaggio d'errore viene emesso dal sistema operativo per segnalare tale situazione).
- L'operatore di indicizzazione "`[]`", che è unario e postfisso, ha precedenza sugli operatori unari aritmetico-logici (vedi Sez. 3.11).
- Un parametro formale di tipo array può essere dichiarato come segue:
  ◁*tipo elementi*▷ ◁*identificatore parametro*▷[]
  oppure nel seguente modo:
  `const` ◁*tipo elementi*▷ ◁*identificatore parametro*▷[]
  dove:
  - Ogni parametro effettivo di tipo array è passato per indirizzo, in quanto l'identificatore di una variabile di tipo array rappresenta l'indirizzo del primo elemento dell'array in forma simbolica. Ciò significa che le modifiche apportate ai valori contenuti in un parametro formale di tipo array durante l'esecuzione di una funzione vengono effettuate direttamente sui valori contenuti nel corrispondente parametro effettivo di tipo array.
  - Poiché non viene effettuata una copia di un parametro effettivo di tipo array, non è richiesta la specifica del numero di elementi del corrispondente parametro formale di tipo array (se serve, tale numero viene passato come ulteriore parametro).
  - Il qualificatore `const` stabilisce che i valori contenuti nel parametro formale di tipo array non possono essere modificati durante l'esecuzione della funzione. Ciò garantisce che i valori contenuti nel corrispondente parametro effettivo passato per indirizzo non vengano modificati durante l'esecuzione della funzione.

- Un array può avere più dimensioni:
  - La dichiarazione di una variabile di tipo array multidimensionale deve specificare il numero di elementi racchiuso tra parentesi quadre per ciascuna dimensione (p.e. `int tabella[10][15]`).
  - Nel caso di dichiarazione con contestuale inizializzazione, i valori debbono essere racchiusi entro parentesi graffe rispetto a tutte le dimensioni.
  - La selezione di un elemento di una variabile di tipo array multidimensionale all'interno di un'istruzione deve specificare l'indice racchiuso tra parentesi quadre per ciascuna dimensione (p.e. `tabella[i][j]`).
  - La dichiarazione di un parametro formale di tipo array multidimensionale deve specificare il numero di elementi per ciascuna dimensione tranne la prima e non può contenere `const`. ■fltpp_15

- Esempio di programma: statistica delle vendite.

  1. **Specifica del problema**. Calcolare il totale delle vendite effettuate da ciascun venditore in ciascuna stagione sulla base delle registrazioni delle singole vendite (venditore, stagione, importo) contenute in un apposito file, riportando anche i totali per venditore e per stagione.
  2. **Analisi del problema**. L'input è costituito dalle registrazioni delle singole vendite contenute in un apposito file. L'output è costituito dal totale delle vendite effettuate da ciascun venditore in ciascuna stagione, più i totali per venditore e per stagione. L'operatore aritmetico di addizione stabilisce le relazioni tra input e output.
  3. **Progettazione dell'algoritmo**. Le registrazioni delle singole vendite contenute sul file – quindi accessibili solo in modo sequenziale – debbono essere preventivamente trasferite su una struttura dati che agevoli il calcolo dei totali per venditore e per stagione. A tale scopo, risulta particolarmente adeguata una struttura dati di tipo array bidimensionale – i cui elementi siano indicizzati dai venditori e dalle stagioni – che viene riempita man mano che si procede con la lettura delle registrazioni delle singole vendite dal file. Chiameremo questa struttura la tabella delle vendite.
     I passi dell'algoritmo – realizzabili attraverso altrettante funzioni – sono i seguenti:
     - Azzerare la tabella delle vendite e i totali per venditore e per stagione.
     - Trasferire le registrazioni del file delle vendite nella tabella delle vendite.
     - Calcolare i totali delle vendite per venditore.
     - Calcolare i totali delle vendite per stagione.
     - Stampare la tabella delle vendite e i totali per venditore e per stagione.
  4. **Implementazione dell'algoritmo**. Questa è la traduzione dei passi in C:

```
/*********************************************/
/* programma per la statistica delle vendite */
/*********************************************/

/*****************************/
/* inclusione delle librerie */
/*****************************/

#include <stdio.h>

/*****************************************/
/* definizione delle costanti simboliche */
/*****************************************/

#define VENDITORI    9              /* numero di venditori dell'azienda */
#define STAGIONI     4              /* numero di stagioni */
#define FILE_VENDITE "vendite.txt"  /* nome fisico del file delle vendite */
```

```
/************************/
/* definizione dei tipi */
/************************/

typedef enum {autunno,
              inverno,
              primavera,
              estate}    stagione_t;   /* tipo stagione */

/*********************************/
/* dichiarazione delle funzioni */
/*********************************/

void azzera_strutture(double tabella_vendite[][STAGIONI],
                      double totali_venditore[],
                      double totali_stagione[]);
void trasf_reg_vendite(double tabella_vendite[][STAGIONI]);
void calc_tot_venditore(double tabella_vendite[][STAGIONI],
                        double totali_venditore[]);
void calc_tot_stagione(double tabella_vendite[][STAGIONI],
                       double totali_stagione[]);
void stampa_strutture(      double tabella_vendite[][STAGIONI],
                      const double totali_venditore[],
                      const double totali_stagione[]);

/*****************************/
/* definizione delle funzioni */
/*****************************/

/* definizione della funzione main */
int main(void)
{
  /* dichiarazione delle variabili locali alla funzione */
  double tabella_vendite[VENDITORI][STAGIONI],   /* output: tabella delle vendite */
         totali_venditore[VENDITORI],            /* output: totali per venditore */
         totali_stagione[STAGIONI];              /* output: totali per stagione */

  /* azzerare la tabella delle vendite e i totali per venditore e per stagione */
  azzera_strutture(tabella_vendite,
                   totali_venditore,
                   totali_stagione);

  /* trasferire le registrazioni del file delle vendite nella tabella delle vendite */
  trasf_reg_vendite(tabella_vendite);

  /* calcolare i totali delle vendite per venditore */
  calc_tot_venditore(tabella_vendite,
                     totali_venditore);

  /* calcolare i totali delle vendite per stagione */
  calc_tot_stagione(tabella_vendite,
                    totali_stagione);
```

```
  /* stampare la tabella delle vendite e i totali per venditore e per stagione */
  stampa_strutture(tabella_vendite,
                   totali_venditore,
                   totali_stagione);
  return(0);
}

/* definizione della funzione per azzerare la tabella delle vendite
   e i totali per venditore e per stagione */
void azzera_strutture(double tabella_vendite[][STAGIONI],   /* output: tab. vend. */
                      double totali_venditore[],            /* output: tot. vend. */
                      double totali_stagione[])             /* output: tot. stag. */
{
  /* dichiarazione delle variabili locali alla funzione */
  int        i;   /* lavoro: indice per i venditori */
  stagione_t j;   /* lavoro: indice per le stagioni */

  /* azzerare la tabella delle vendite */
  for (i = 0;
       (i < VENDITORI);
       i++)
    for (j = autunno;
         (j <= estate);
         j++)
      tabella_vendite[i][j] = 0.0;

  /* azzerare i totali per venditore */
  for (i = 0;
       (i < VENDITORI);
       i++)
    totali_venditore[i] = 0.0;

  /* azzerare i totali per stagione */
  for (j = autunno;
       (j <= estate);
       j++)
    totali_stagione[j] = 0.0;
}

/* definizione della funzione per trasferire le registrazioni del file delle vendite
   nella tabella delle vendite */
void trasf_reg_vendite(double tabella_vendite[][STAGIONI])   /* output: tab. vend. */
{
  /* dichiarazione delle variabili locali alla funzione */
  FILE       *file_vendite;   /* input: file delle vendite */
  int        venditore;       /* input: venditore letto nella registrazione */
  stagione_t stagione;        /* input: stagione letta nella registrazione */
  double     importo;         /* input: importo letto nella registrazione */

  /* aprire il file delle vendite */
  file_vendite = fopen(FILE_VENDITE,
                       "r");
```

```
  /* trasferire le registrazioni del file delle vendite nella tabella delle vendite */
  while (fscanf(file_vendite,
                "%d%d%lf",
                &venditore,
                (int *)&stagione,
                &importo) != EOF)
    tabella_vendite[venditore][stagione] += importo;

  /* chiudere il file delle vendite */
  fclose(file_vendite);
}

/* definizione della funzione per calcolare i totali delle vendite per venditore */
void calc_tot_venditore(double tabella_vendite[][STAGIONI],   /* i.: tab. vend. */
                        double totali_venditore[])            /* o.: tot. vend. */
{
  /* dichiarazione delle variabili locali alla funzione */
  int        i;   /* lavoro: indice per i venditori */
  stagione_t j;   /* lavoro: indice per le stagioni */

  /* calcolare i totali delle vendite per venditore */
  for (i = 0;
       (i < VENDITORI);
       i++)
    for (j = autunno;
         (j <= estate);
         j++)
      totali_venditore[i] += tabella_vendite[i][j];
}

/* definizione della funzione per calcolare i totali delle vendite per stagione */
void calc_tot_stagione(double tabella_vendite[][STAGIONI],   /* i.: tab. vend. */
                       double totali_stagione[])             /* o.: tot. stag. */
{
  /* dichiarazione delle variabili locali alla funzione */
  int        i;   /* lavoro: indice per i venditori */
  stagione_t j;   /* lavoro: indice per le stagioni */

  /* calcolare i totali delle vendite per stagione */
  for (j = autunno;
       (j <= estate);
       j++)
    for (i = 0;
         (i < VENDITORI);
         i++)
      totali_stagione[j] += tabella_vendite[i][j];
}
```

```
/* definizione della funzione per stampare la tabella delle vendite
   e i totali per venditore e per stagione */
void stampa_strutture(      double tabella_vendite[][STAGIONI],   /* i.: tab. vend. */
                      const double totali_venditore[],            /* i.: tot. vend. */
                      const double totali_stagione[])             /* i.: tot. stag. */
{
  /* dichiarazione delle variabili locali alla funzione */
  int        i;   /* lavoro: indice per i venditori */
  stagione_t j;   /* lavoro: indice per le stagioni */

  /* stampare l'intestazione di tutte le colonne */
  printf("Venditore   Autunno   Inverno  Primavera  Estate    Totale\n");

  /* stampare la tabella delle vendite e i totali per venditore */
  for (i = 0;
       (i < VENDITORI);
       i++)
  {
    printf("%5d     ",
           i);
    for (j = autunno;
         (j <= estate);
         j++)
      printf("%8.2f  ",
             tabella_vendite[i][j]);
    printf("%8.2f  \n",
           totali_venditore[i]);
  }

  /* stampare l'intestazione dell'ultima riga */
  printf("\nTotale    ");

  /* stampare i totali per stagione */
  for (j = autunno;
       (j <= estate);
       j++)
    printf("%8.2f  ",
           totali_stagione[j]);
  printf("\n");
}
```

6.9 Stringhe: rappresentazione e funzioni di libreria

- Un valore di tipo stringa, denotato come sequenza di caratteri racchiusa tra virgolette, viene rappresentato nel linguaggio C attraverso un array di elementi di tipo `char`. Per le stringhe valgono quindi tutte le considerazioni fatte per gli array ad una singola dimensione.
- Diversamente dai valori dei tipi visti finora, i quali occupano tutti la stessa quantità di memoria, i valori di tipo stringa occupano quantità di memoria diverse a seconda del numero di caratteri che li compongono.
- Una variabile array di tipo stringa stabilisce il numero massimo di caratteri che possono essere contenuti. Poiché il valore di tipo stringa contenuto nella variabile in un certo momento può avere un numero di caratteri inferiore al massimo prestabilito, la fine di questo valore deve essere esplicitamente marcata con un carattere speciale, che è '`\0`'.

- Se la dichiarazione di una variabile di tipo stringa contiene anche l'inizializzazione della variabile, invece di esprimere i singoli caratteri del valore iniziale tra parentesi graffe è possibile indicare l'intero valore iniziale tra virgolette. Il carattere '\0' non va indicato nel valore iniziale in quanto viene aggiunto automaticamente all'atto della memorizzazione del valore nella variabile.

- Quando il valore di una variabile di tipo stringa viene acquisito tramite `scanf` o sue varianti, il carattere '\0' viene automaticamente aggiunto all'atto della memorizzazione del valore nella variabile. L'identificatore di tale variabile non necessita di essere preceduto dall'operatore "`&`" quando compare nella `scanf` in quanto, essendo di tipo array, rappresenta già un indirizzo.

- Quando il valore di una variabile di tipo stringa viene comunicato tramite `printf` o sue varianti, vengono considerati tutti e soli i caratteri che precedono il carattere '\0' nel valore della variabile.

- Questioni da tenere presente quando si utilizza una variabile di tipo stringa:

  - Lo spazio di memoria riservato alla variabile deve essere sufficiente per contenere il valore di tipo stringa più il carattere '\0' al termine di ciascun utilizzo della variabile. In particolare, quando si acquisisce un valore di tipo stringa, è meglio usare `scanf` con segnaposto `%`◁*numero*▷`s` oppure acquisire un solo carattere alla volta tramite `getchar`, così da essere sicuri che lo spazio di memoria riservato alla variabile in cui il valore sarà memorizzato sia abbastanza grande.
  - Poiché tutte le funzioni di libreria standard per le stringhe fanno affidamento sulla presenza del carattere '\0' alla fine del valore di tipo stringa contenuto nella variabile, è fondamentale che tale carattere sia presente all'interno dello spazio di memoria riservato alla variabile al termine di ciascun utilizzo della variabile.

- Esempi:

  - Definizione di una costante simbolica di tipo stringa:

    ```
    #define FILE_VENDITE "vendite.txt"
    ```

  - Dichiarazione con contestuale inizializzazione di una variabile di tipo stringa:

    ```
    char messaggio[20] = "benvenuto";
    ```

  - Dichiarazione con contestuale inizializzazione di una variabile di tipo array di stringhe:

    ```
    char mese[12][10] = {"gennaio",
                         "febbraio",
                         "marzo",
                         "aprile",
                         "maggio",
                         "giugno",
                         "luglio",
                         "agosto",
                         "settembre",
                         "ottobre",
                         "novembre",
                         "dicembre"};
    ```

- Principali funzioni messe a disposizione dal linguaggio C per il tipo stringa con indicazione dei relativi file di intestazione di libreria standard (il tipo standard `size_t` è assimilabile al tipo predefinito `unsigned int`, il tipo `char *` è assimilabile al tipo stringa per ciò che vedremo in Sez. 6.11):

  - `size_t strlen(const char *s)` `<string.h>`
    Restituisce la lunghezza di `s`, cioè il numero di caratteri attualmente in `s` escluso '\0'.

- `char *strcpy(char *s1, const char *s2)` `<string.h>`
  Copia il contenuto di `s2` in `s1` (che deve avere spazio sufficiente).
- `char *strncpy(char *s1, const char *s2, size_t n)` `<string.h>`
  Copia i primi `n` caratteri di `s2` in `s1` (che deve avere spazio sufficiente).
- `char *strcat(char *s1, const char *s2)` `<string.h>`
  Concatena il contenuto di `s2` a quello di `s1` (che deve avere spazio sufficiente).
- `char *strncat(char *s1, const char *s2, size_t n)` `<string.h>`
  Concatena i primi `n` caratteri di `s2` a quelli di `s1` (che deve avere spazio sufficiente).
- `int strcmp(const char *s1, const char *s2)` `<string.h>`
  Confronta i contenuti di `s1` ed `s2` sulla base dell'ordinamento lessicografico restituendo:
  * `-1` se `s1 < s2`,
  * `0` se `s1 = s2`,
  * `1` se `s1 > s2`,

  dove `s1 < s2` se:
  * `s1` è più corta di `s2` e tutti i caratteri di `s1` coincidono con i corrispondenti caratteri di `s2`, oppure
  * i primi `n` caratteri di `s1` ed `s2` coincidono a due a due e `s1[n] < s2[n]` rispetto alla codifica usata per i caratteri.
- `int strncmp(const char *s1, const char *s2, size_t n)` `<string.h>`
  Come la precedente, considerando solo i primi `n` caratteri di `s1` ed `s2`.
- `int sprintf(char *s, const char *formato`$_p$`, ◁`*espressioni*`▷)` `<stdio.h>`
  Scrive su `s` (in particolare, permette di convertire numeri in stringhe).
- `int sscanf(const char *s, const char *formato`$_s$`, ◁`*indirizzi variabili*`▷)` `<stdio.h>`
  Legge da `s` (in particolare, permette di estrarre numeri da stringhe).
- `int atoi(const char *s)` `<stdlib.h>`
  Converte `s` in un numero intero.
- `double atof(const char *s)` `<stdlib.h>`
  Converte `s` in un numero reale. ■fltpp_16

6.10 Strutture e unioni: rappresentazione e operatore punto

- Il costruttore di tipo struttura del linguaggio C – noto più in generale come record – dà luogo ad un valore aggregato formato da un numero finito di elementi non necessariamente dello stesso tipo. Per questo motivo gli elementi non saranno selezionabili mediante indici come negli array, ma dovranno essere singolarmente dichiarati e identificati.

- Una variabile di tipo struttura viene dichiarata come segue:
  `struct {◁`*dichiarazione elementi*`▷} ◁`*identificatore variabile*`▷;`
  oppure con contestuale inizializzazione:
  `struct {◁`*dichiarazione elementi*`▷} ◁`*identificatore variabile*`▷ = {◁`*sequenza valori*`▷};`
  dove:
  - Ogni elemento (o campo) è dichiarato come segue:
    `◁`*tipo elemento*`▷ ◁`*identificatore elemento*`▷;`
  - Se presenti, i valori di inizializzazione sono separati da virgole e vengono ordinatamente assegnati da sinistra a destra ai corrispondenti elementi a patto che i rispettivi tipi siano compatibili.

- Diversamente dal tipo array, una variabile di tipo struttura può comparire in entrambi i lati di un assegnamento – quindi il risultato di una funzione può essere di tipo struttura – e può essere passata sia per valore che per indirizzo ad una funzione.

- Ogni elemento di una variabile di tipo struttura è selezionato all'interno di un'istruzione tramite il suo identificatore:
  ◁*identificatore variabile*▷.◁*identificatore elemento*▷
- L'operatore punto, che è unario e postfisso, ha la stessa precedenza dell'operatore di indicizzazione.
- Esempi:
  - Definizione di un tipo per i pianeti del sistema solare:

```
typedef struct
{
  char   nome[9];           /* nome del pianeta */
  double diametro;          /* diametro equatoriale in km */
  int    lune;              /* numero di lune */
  double tempo_orbita,      /* durata dell'orbita attorno al sole in anni */
         tempo_rotazione;   /* durata della rotazione attorno all'asse in ore */
} pianeta_t;
```

  - Dichiarazione con contestuale inizializzazione di una variabile per un pianeta:

```
pianeta_t pianeta = {"Giove",
                     142800.0,
                     16,
                     11.9,
                     9.925};
```

  - Acquisizione da tastiera dei dati relativi ad un pianeta:

```
pianeta_t pianeta;

scanf("%s%lf%d%lf%lf",
      pianeta.nome,
      &pianeta.diametro,
      &pianeta.lune,
      &pianeta.tempo_orbita,
      &pianeta.tempo_rotazione);
```

  - Funzione per verificare l'uguaglianza del contenuto di due variabili di tipo `pianeta_t`:

```
int pianeti_uguali(pianeta_t pianeta1,
                   pianeta_t pianeta2)
{
  return((strcmp(pianeta1.nome,
                 pianeta2.nome) == 0) &&
         (pianeta1.diametro == pianeta2.diametro) &&
         (pianeta1.lune == pianeta2.lune) &&
         (pianeta1.tempo_orbita == pianeta2.tempo_orbita) &&
         (pianeta1.tempo_rotazione == pianeta2.tempo_rotazione));
}
```

  - Definizione di un tipo per i numeri complessi in forma algebrica (sebbene i due elementi siano entrambi numeri reali, essi hanno ruoli ben diversi e ciò giustifica il ricorso ad una struttura piuttosto che un array):

```
typedef struct
{
  double parte_reale,   /* parte reale del numero complesso */
         parte_immag;   /* parte immaginaria del numero complesso */
} num_compl_t;
```

  - Funzione per calcolare la somma di due numeri complessi in forma algebrica:

```
num_compl_t somma_num_compl(num_compl_t n1,
                            num_compl_t n2)
{
  num_compl_t n;

  n.parte_reale = n1.parte_reale + n2.parte_reale;
  n.parte_immag = n1.parte_immag + n2.parte_immag;
  return(n);
}
```

- Il costruttore di tipo unione del linguaggio C – noto più in generale come record variant – dà luogo ad un valore aggregato formato da un numero finito di elementi non necessariamente dello stesso tipo, i quali sono in alternativa tra di loro. Ciò consente di rappresentare dati che possono avere diverse interpretazioni a seconda dell'input del programma.
- Mentre lo spazio di memoria da riservare ad una variabile di tipo struttura è la somma delle `sizeof` dei tipi degli elementi della struttura, lo spazio di memoria da riservare ad una variabile di tipo unione è il massimo delle `sizeof` dei tipi degli elementi dell'unione, perché in questo caso i vari elementi sono in alternativa tra loro e quindi basta una quantità di spazio tale da contenere l'elemento più grande.
- La dichiarazione di una variabile di tipo unione ha la stessa forma della dichiarazione di una variabile di tipo struttura, con `struct` sostituito da `union`.
- Ogni elemento di una variabile di tipo unione è selezionato all'interno di un'istruzione tramite il suo identificatore utilizzando l'operatore punto. Per garantire che tale elemento sia coerente con l'interpretazione corrente della variabile, occorre testare preventivamente un'ulteriore variabile contenente l'interpretazione corrente.
- Esempi:
  - Definizione di un tipo per le figure geometriche (per chiarezza iniziamo dalla `typedef` che deve essere introdotta per ultima in quanto fa uso di tutte le altre):

```
typedef struct
{
  forma_t forma;         /* interpretazione corrente */
  union
  {
      triangolo_t  dati_triangolo;    /* dati del triangolo */
      rettangolo_t dati_rettangolo;   /* dati del rettangolo */
      double       lato_quadrato;     /* lunghezza del lato del quadrato */
      double       raggio_cerchio;    /* lunghezza del raggio del cerchio */
  }       dati_figura;   /* dati della figura */
  double  perimetro,     /* perimetro della figura */
          area;          /* area della figura */
} figura_t;

typedef enum {triangolo,
              rettangolo,
              quadrato,
              cerchio}    forma_t;
```

```
typedef struct
{
  double lato1,     /* lunghezza del primo lato */
         lato2,     /* lunghezza del secondo lato */
         lato3,     /* lunghezza del terzo lato */
         altezza;   /* altezza riferita al primo lato come base */
} triangolo_t;

typedef struct
{
  double lato1,   /* lunghezza del primo lato */
         lato2;   /* lunghezza del secondo lato */
} rettangolo_t;
```

– Funzione per calcolare il perimetro e l'area di una figura (che deve essere passata per indirizzo perché i risultati vengono memorizzati nei suoi elementi `perimetro` e `area`):

```
void calcola_perimetro_area(figura_t *figura)
{
  switch ((*figura).forma)
  {
    case triangolo:
      (*figura).perimetro =
        (*figura).dati_figura.dati_triangolo.lato1 +
        (*figura).dati_figura.dati_triangolo.lato2 +
        (*figura).dati_figura.dati_triangolo.lato3;
      (*figura).area =
        (*figura).dati_figura.dati_triangolo.lato1 *
        (*figura).dati_figura.dati_triangolo.altezza / 2;
      break;
    case rettangolo:
      (*figura).perimetro =
        2 * ((*figura).dati_figura.dati_rettangolo.lato1 +
             (*figura).dati_figura.dati_rettangolo.lato2);
      (*figura).area =
        (*figura).dati_figura.dati_rettangolo.lato1 *
        (*figura).dati_figura.dati_rettangolo.lato2;
      break;
    case quadrato:
      (*figura).perimetro =
        4 * (*figura).dati_figura.lato_quadrato;
      (*figura).area =
        (*figura).dati_figura.lato_quadrato *
        (*figura).dati_figura.lato_quadrato;
      break;
    case cerchio:
      (*figura).perimetro =
        2 * PI_GRECO * (*figura).dati_figura.raggio_cerchio;
      (*figura).area =
        PI_GRECO * (*figura).dati_figura.raggio_cerchio *
                   (*figura).dati_figura.raggio_cerchio;
      break;
  }
}
```

■fltpp_17

6.11 Puntatori: operatori e funzioni di libreria

- Il costruttore di tipo puntatore del linguaggio C viene utilizzato per denotare indirizzi di memoria, che sono valori scalari. Un valore di tipo puntatore non rappresenta quindi un dato, ma l'indirizzo di memoria al quale un dato può essere reperito.

- L'insieme dei valori di tipo puntatore include un valore speciale, denotato con la costante simbolica `NULL` definita nel file di intestazione di libreria standard `stdio.h`, il quale rappresenta l'assenza di un indirizzo specifico.

- Una variabile di tipo puntatore ad un dato di un certo tipo viene dichiarata come segue:
  ◁*tipo dato*▷ *◁*identificatore variabile*▷;
  oppure con contestuale inizializzazione:
  ◁*tipo dato*▷ *◁*identificatore variabile*▷ = ◁*indirizzo*▷;
  Lo spazio di memoria da riservare ad una variabile di tipo puntatore è indipendente dal tipo del dato cui il puntatore fa riferimento.

- Poiché gli indirizzi di memoria vengono rappresentati attraverso numeri interi, i valori di tipo puntatore sono assimilabili ai valori di tipo `int`. Tuttavia, oltre all'operatore di assegnamento, ai valori di tipo puntatore è ragionevole applicare solo alcuni degli operatori aritmetico-logici:

  – Addizione/sottrazione di un valore di tipo `int` ad/da un valore di tipo puntatore: serve per far avanzare/indietreggiare il puntatore di un certo numero di porzioni di memoria, ciascuna della dimensione tale da poter contenere un valore del tipo di riferimento del puntatore.
  – Sottrazione di un valore di tipo puntatore da un altro valore di tipo puntatore: serve per calcolare il numero di porzioni di memoria – ciascuna della dimensione tale da poter contenere un valore del tipo di riferimento dei due puntatori – comprese tra i due puntatori.
  – Confronto di uguaglianza/diversità di due valori di tipo puntatore.

- Esistono inoltre degli operatori specifici per i valori di tipo puntatore:

  – L'operatore valore-di "`*`", applicato ad una variabile di tipo puntatore il cui valore è diverso da `NULL`, restituisce il valore contenuto nella locazione di memoria il cui indirizzo è contenuto nella variabile. Se viene applicato a una variabile di tipo puntatore il cui valore è `NULL`, il sistema operativo emette un messaggio di errore e l'esecuzione del programma viene interrotta.
  – L'operatore indirizzo-di "`&`", applicato ad una variabile, restituisce l'indirizzo della locazione di memoria in cui è contenuto il valore della variabile.
  – L'operatore freccia "`->`", il quale riguarda le variabili di tipo puntatore a struttura o unione, consente di abbreviare:
    (*◁*identificatore variabile puntatore*▷).◁*identificatore elemento*▷
    in:
    ◁*identificatore variabile puntatore*▷->◁*identificatore elemento*▷

  I primi due operatori sono unari e prefissi e hanno la stessa precedenza degli operatori unari aritmetico-logici (vedi Sez. 3.11), mentre l'operatore freccia è unario e postfisso e ha la stessa precedenza degli operatori di indicizzazione e punto.

- Poiché l'identificatore di una variabile di tipo array rappresenta l'indirizzo della locazione di memoria che contiene il valore del primo elemento dell'array, vale quanto segue:

  – Un parametro formale di tipo array può essere indifferentemente dichiarato come segue:
    ◁*tipo elementi*▷ ◁*identificatore parametro array*▷[]
    oppure nel seguente modo:
    ◁*tipo elementi*▷ *◁*identificatore parametro array*▷

  - Un elemento di una variabile di tipo array può essere indifferentemente selezionato come segue:
    ◁*identificatore variabile array*▷[◁*espr_indice*▷]
    oppure nel seguente modo:
    ◁*identificatore variabile array*▷ + ◁*espr_indice*▷
    Nel secondo caso, il valore dell'elemento selezionato viene ottenuto nel seguente modo:
    *(◁*identificatore variabile array*▷ + ◁*espr_indice*▷)

- Il costruttore di tipo struttura e il costruttore di tipo puntatore usati congiuntamente consentono la definizione di tipi di dati ricorsivi. Il costruttore di tipo struttura permette infatti di associare un identificatore alla struttura stessa nel seguente modo:
  `struct` ◁*identificatore struttura*▷ {◁*dichiarazione elementi*▷}
  In via di principio, ciò rende possibile la presenza di uno o più elementi della seguente forma all'interno di {◁*dichiarazione elementi*▷}:
  `struct` ◁*identificatore struttura*▷ ◁*identificatore elemento*▷;
  come pure di elementi della seguente forma:
  `struct` ◁*identificatore struttura*▷ *◁*identificatore elemento*▷;
  Tuttavia, solo gli elementi della seconda forma sono ammissibili in quanto rendono la definizione ricorsiva ben posta. Il motivo è che per gli elementi di questa forma è noto lo spazio di memoria da riservare, mentre ciò non vale per gli elementi della prima forma.

- Esempio di tipo di dato ricorsivo costituito dalla lista ordinata di numeri interi (o è vuota, o è composta da un numero intero collegato a una lista ordinata di numeri interi di valore maggiore del numero che li precede), i cui valori sono accessibili solo se si conosce l'indirizzo della sua prima componente:

```
typedef struct comp_lista
{
  int               valore;    /* numero intero memorizzato nella componente */
  struct comp_lista *succ_p;   /* puntatore alla componente successiva */
} comp_lista_t;
```

- Oltre a consentire il passaggio di parametri per indirizzo, i puntatori permettono il riferimento a strutture dati dinamiche, cioè strutture dati – tipicamente implementate attraverso la definizione di tipi ricorsivi – che si espandono e si contraggono mentre il programma viene eseguito.

- Poiché lo spazio di memoria richiesto da una struttura dati dinamica non può essere fissato a priori, l'allocazione/disallocazione della memoria per tali strutture dinamiche avviene a tempo di esecuzione nello heap (vedi Sez. 5.9) attraverso l'invocazione delle seguenti funzioni disponibili nel file di intestazione di libreria standard `stdlib.h`:

  - `void *malloc(size_t dim)`
    Alloca un blocco di `dim` byte nello heap e restituisce l'indirizzo di tale blocco (`NULL` in caso di fallimento, cioè assenza di un blocco sufficientemente grande). Il blocco allocato viene marcato come occupato nello heap.
  - `void *calloc(size_t num, size_t dim)`
    Alloca `num` blocchi consecutivi di `dim` byte ciascuno nello heap e restituisce l'indirizzo del primo blocco (`NULL` in caso di fallimento). Questa funzione serve per allocare dinamicamente array nel momento in cui il numero dei loro elementi diviene noto a tempo di esecuzione.
  - `void *realloc(void *vecchio_blocco, size_t nuova_dim)`
    Cambia la dimensione di un blocco di memoria nello heap precedentemente allocato con `malloc`/`calloc` (senza modificarne il contenuto) e restituisce l'indirizzo del blocco ridimensionato (`NULL` in caso di fallimento). Quest'ultimo blocco potrebbe trovarsi in una posizione dello heap diversa da quella del blocco originario.
  - `void free(void *blocco)`
    Disalloca un blocco di memoria nello heap precedentemente allocato con `malloc`/`calloc`; prima di applicarla, è bene assicurarsi che non venga più fatto riferimento al blocco tramite puntatori nelle istruzioni da eseguire successivamente. Il blocco disallocato viene marcato come libero nello heap, ritornando quindi nuovamente disponibile per successive allocazioni.

- Esempi:
  - Allocazione dinamica e utilizzo di un array:

```
int n,    /* numero di elementi dell'array */
    i,    /* indice di scorrimento dell'array */
    *a;   /* array da allocare dinamicamente */
do
{
  printf("Digita il numero di elementi (> 0): ");
  scanf("%d",
        &n);
}
while (n <= 0);
a = (int *)calloc(n,              /* a = (int *)calloc(n + 1,        */
                  sizeof(int));   /*                   sizeof(int)); */
...                               /* a[0] = n;                       */
for (i = 0;                       /* for (i = 1;                     */
     (i < n);                     /*      (i <= n);                  */
     i++)                         /*      i++)                       */
  a[i] = 2 * i;                   /*   a[i] = 2 * i;                 */
...
```

    Sarebbe stato un errore dichiarare l'array dinamico nel seguente modo:

```
int n,      /* numero di elementi dell'array */
    i,      /* indice di scorrimento dell'array */
    a[n];   /* array dinamico */
```

    in quanto tutti gli operandi che compaiono nell'espressione che definisce il numero di elementi di un array debbono essere costanti (quindi una variabile come `n` non è ammissibile).

  - Funzione per attraversare una lista ordinata e stamparne i valori:

```
void attraversa_lista(comp_lista_t *testa_p)   /* indirizzo prima componente */
{
  comp_lista_t *punt;

  for (punt = testa_p;
       (punt != NULL);
       punt = punt->succ_p)
    printf("%d\n",
           punt->valore);
}
```

  - Funzione per cercare un valore in una lista ordinata (notare l'importanza della cortocircuitazione dell'operatore logico presente nelle istruzioni `for` e `if`):

```
comp_lista_t *cerca_in_lista(comp_lista_t *testa_p,
                             int          valore)
{
  comp_lista_t *punt;

  for (punt = testa_p;
       ((punt != NULL) && (punt->valore < valore));
       punt = punt->succ_p);
  if ((punt != NULL) && (punt->valore > valore))
    punt = NULL;
  return(punt);
}
```

– Funzione per inserire un valore in una lista ordinata (l'indirizzo della prima componente potrebbe cambiare a seguito dell'inserimento, quindi deve essere passato per indirizzo da cui il doppio *):

```
int inserisci_in_lista(comp_lista_t **testa_p,
                       int          valore)
{
  int          ris;
  comp_lista_t *corr_p,
               *prec_p,
               *nuova_p;

  for (corr_p = prec_p = *testa_p;
       ((corr_p != NULL) && (corr_p->valore < valore));
       prec_p = corr_p, corr_p = corr_p->succ_p);
  if ((corr_p != NULL) && (corr_p->valore == valore))
    ris = 0;
  else
  {
    ris = 1;
    nuova_p = (comp_lista_t *)malloc(sizeof(comp_lista_t));
    nuova_p->valore = valore;
    nuova_p->succ_p = corr_p;
    if (corr_p == *testa_p)
      *testa_p = nuova_p;
    else
      prec_p->succ_p = nuova_p;
  }
  return(ris);
}
```

– Funzione per rimuovere un valore da una lista ordinata (l'indirizzo della prima componente potrebbe cambiare anche a seguito della rimozione, da cui di nuovo il doppio *):

```
int rimuovi_da_lista(comp_lista_t **testa_p,
                     int          valore)
{
  int          ris;
  comp_lista_t *corr_p,
               *prec_p;

  for (corr_p = prec_p = *testa_p;
       ((corr_p != NULL) && (corr_p->valore < valore));
       prec_p = corr_p, corr_p = corr_p->succ_p);
  if ((corr_p == NULL) || (corr_p->valore > valore))
    ris = 0;
  else
  {
    ris = 1;
    if (corr_p == *testa_p)
      *testa_p = corr_p->succ_p;
    else
      prec_p->succ_p = corr_p->succ_p;
    free(corr_p);
  }
  return(ris);
}
```

– Uso pericoloso di `free`:

```
int *i,
    *j;

i = (int *)malloc(sizeof(int));
*i = 24;
j = i;
...
free(i);
...
*j = 18;
...
```

Le variabili `i` e `j` puntano alla stessa locazione di memoria, la quale contiene il valore `24` fino a quando non viene disallocata. Dopo la sua disallocazione, essa potrebbe essere nuovamente utilizzata per allocare qualche altra struttura dati, quindi assegnarle il valore `18` tramite la variabile di tipo puntatore `j` potrebbe causare l'effetto indesiderato di modificare accidentalmente il contenuto di un'altra variabile. ■fltpp_18

Capitolo 7

Correttezza di programmi procedurali

7.1 Triple di Hoare

- Dati un problema e un programma che si suppone risolvere il problema, si pone la questione di verificare se il programma è corretto rispetto al problema, cioè se *per ogni* istanza dei dati di ingresso del problema il programma termina e produce la soluzione corrispondente (testare il programma non è sufficiente).
- Ciò richiede di stabilire formalmente cosa il programma calcola. Nel caso di un programma sequenziale, il suo significato viene tradizionalmente definito mediante una funzione matematica che descrive l'effetto ingresso/uscita dell'esecuzione del programma ignorando gli stati intermedi della computazione.
- Nel paradigma di programmazione imperativo di natura procedurale, per stato della computazione si intende il contenuto della memoria ad un certo punto dell'esecuzione del programma. La funzione che rappresenta il significato del programma descrive quindi quale sia lo stato finale della computazione a fronte dello stato iniziale della computazione determinato da una generica istanza dei dati di ingresso.
- Tra i vari approcci alla verifica di correttezza di programmi imperativi procedurali, l'approccio assiomatico di Hoare si basa sull'idea di annotare i programmi con formule della logica dei predicati (da ora in poi dette semplicemente predicati) che esprimono proprietà valide nei vari stati della computazione. Tali predicati saranno asserzioni sui valori contenuti nelle variabili dei programmi.
- Si dice tripla di Hoare una tripla della seguente forma:
$$\{Q\}\ S\ \{R\}$$
dove Q è un predicato detto precondizione, S è un'istruzione ed R è un predicato detto postcondizione.
- La tripla $\{Q\}\ S\ \{R\}$ è vera sse l'esecuzione dell'istruzione S inizia in uno stato della computazione in cui Q è soddisfatta e termina raggiungendo uno stato della computazione in cui R è soddisfatta.

7.2 Determinazione della precondizione più debole

- Nella pratica, data una tripla di Hoare $\{Q\}\ S\ \{R\}$ in cui S è un intero programma, S è ovviamente noto come pure è nota la postcondizione R, la quale rappresenta in sostanza il risultato che si vuole ottenere alla fine dell'esecuzione del programma. La precondizione Q è invece ignota.
- Verificare la correttezza di un programma S che si prefigge di calcolare un risultato R consiste quindi nel determinare se esiste un predicato Q che risolve la seguente equazione logica:
$$\{Q\}\ S\ \{R\}\ \equiv\ \mathit{vero}$$
- Poiché l'equazione logica riportata sopra potrebbe ammettere più soluzioni, ci si concentra sulla determinazione della precondizione più debole (nel senso di meno vincolante), cioè la precondizione Q' tale che $Q'' \rightarrow Q'$ è vero per ogni precondizione Q'' che risolve l'equazione logica. Dati un programma S e una postcondizione R, denotiamo con $wp(S, R)$ la precondizione più debole rispetto ad S ed R.

- Premesso che *vero* è soddisfatto da ogni stato della computazione mentre *falso* non è soddisfatto da nessuno stato della computazione, dati un programma S e una postcondizione R si hanno i seguenti tre casi:
  - Se $wp(S, R) = \mathit{vero}$, allora qualunque sia la precondizione Q risulta che la tripla di Hoare $\{Q\}\ S\ \{R\}$ è vera. Infatti, $Q \rightarrow \mathit{vero}$ è vero per ogni predicato Q. In tal caso, il programma è sempre corretto rispetto al problema, cioè è corretto a prescindere da quale sia lo stato iniziale della computazione.
  - Se $wp(S, R) = \mathit{falso}$, allora qualunque sia la precondizione Q risulta che la tripla di Hoare $\{Q\}\ S\ \{R\}$ è falsa. Infatti, $Q \rightarrow \mathit{falso}$ è vero se Q non è soddisfatto nello stato iniziale della computazione, mentre $Q \rightarrow \mathit{falso}$ è falso (e quindi Q non può essere una soluzione) se Q è soddisfatto nello stato iniziale della computazione. In tal caso, il programma non è mai corretto rispetto al problema, cioè non è corretto a prescindere da quale sia lo stato iniziale della computazione.
  - Se $wp(S, R) \notin \{\mathit{vero}, \mathit{falso}\}$, cioè $wp(S, R)$ è un predicato non banale, allora la correttezza del programma rispetto al problema potrebbe dipendere dallo stato iniziale della computazione.
- Dato un programma S, $wp(S, \_)$ soddisfa le seguenti proprietà:
$$\begin{aligned} wp(S, \mathit{falso}) &\equiv \mathit{falso} \\ wp(S, R_1 \wedge R_2) &\equiv (wp(S, R_1) \wedge wp(S, R_2)) \\ (R_1 \rightarrow R_2) &\models (wp(S, R_1) \rightarrow wp(S, R_2)) \\ (\{Q\}\ S\ \{R'\} \wedge (R' \rightarrow R)) &\models \{Q\}\ S\ \{R\} \end{aligned}$$
- Dato un programma S privo di iterazione e ricorsione e data una postcondizione R, $wp(S, R)$ può essere determinata per induzione sulla struttura sintattica di S nel seguente modo sviluppato da Dijkstra:
  - Se S è un'istruzione di assegnamento "`x = e;`", si applica la seguente regola di retropropagazione:
$$wp(S, R) = R_{\mathtt{x},\mathtt{e}}$$
dove $R_{\mathtt{x},\mathtt{e}}$ è il predicato ottenuto da R sostituendo tutte le occorrenze di `x` con `e`, cioè $R[{}^{\mathtt{e}}/_{\mathtt{x}}]$.
  - Se S è un'istruzione di selezione "`if` (β) S_1 `else` S_2", si ragiona sulle due parti come segue:
$$wp(S, R) = ((\beta \rightarrow wp(S_1, R)) \wedge (\neg\beta \rightarrow wp(S_2, R)))$$
  - Se S è una sequenza di istruzioni "$S_1 S_2$", si procede a ritroso come segue:
$$wp(S, R) = wp(S_1, wp(S_2, R))$$
- Come suggerito dalla regola per la sequenza di istruzioni, il calcolo della precondizione più debole per un programma procede andando a ritroso a partire dalla postcondizione e dall'ultima istruzione. L'applicazione delle regole suddette è pleonastica se la postcondizione ripete banalmente le istruzioni oggetto di verifica.
- Esempi:
  - La correttezza di un programma può dipendere dallo stato iniziale della computazione, nel qual caso la precondizione più debole non è equivalente né a *vero* né a *falso*. Data l'istruzione di assegnamento:
```
x = x + 1;
```
e data la postcondizione:
$$x < 1$$
l'ottenimento del risultato prefissato dipende ovviamente dal valore di `x` prima che venga eseguita l'istruzione. Infatti, la precondizione più debole risulta essere:
$$(x < 1)_{\mathtt{x},\mathtt{x+1}} = (x + 1 < 1) \equiv (x < 0)$$
  - Non c'è dipendenza dallo stato iniziale della computazione se la precondizione più debole è *vero*. Il seguente brano di codice per determinare quale tra le variabili `x` e `y` contiene il valore minimo:
```
if (x <= y)
  z = x;
else
  z = y;
```

è sempre corretto perché, formalizzando la postcondizione nel seguente modo:

$$z = \min(x, y)$$

la precondizione più debole risulta essere *vero* in quanto:

$$(x \leq y \rightarrow (z = \min(x,y))_{\mathtt{z,x}}) = (x \leq y \rightarrow x = \min(x,y)) \equiv \mathit{vero}$$
$$(x > y \rightarrow (z = \min(x,y))_{\mathtt{z,y}}) = (x > y \rightarrow y = \min(x,y)) \equiv \mathit{vero}$$
$$(\mathit{vero} \wedge \mathit{vero}) \equiv \mathit{vero}$$

Una postcondizione come $(x \leq y \rightarrow z = x) \wedge (x > y \rightarrow z = y)$ sarebbe stata banale, mentre una postcondizione come $z = x \vee z = y$ sarebbe stata imprecisa oltre che ovvia.

– Potrebbe non esserci dipendenza dallo stato iniziale della computazione anche se la precondizione più debole non è *vero*. Il seguente brano di codice per scambiare i valori delle variabili `x` e `y`:

```
tmp = x;
x = y;
y = tmp;
```

è sempre corretto perché, indicando con X ed Y i valori rispettivamente contenuti nelle variabili `x` e `y` all'inizio dell'esecuzione e formalizzando la postcondizione nel seguente modo:

$$x = Y \wedge y = X$$

la precondizione più debole risulta essere proprio il predicato:

$$x = X \wedge y = Y$$

in quanto:

$$(x = Y \wedge y = X)_{\mathtt{y,tmp}} = (x = Y \wedge \mathit{tmp} = X)$$
$$(x = Y \wedge \mathit{tmp} = X)_{\mathtt{x,y}} = (y = Y \wedge \mathit{tmp} = X)$$
$$(y = Y \wedge \mathit{tmp} = X)_{\mathtt{tmp,x}} = (y = Y \wedge x = X) \equiv (x = X \wedge y = Y)$$ ■fltpp_19

7.3 Verifica della correttezza di programmi procedurali iterativi

- Per verificare mediante triple di Hoare la correttezza di un'istruzione di ripetizione, bisogna individuare un invariante di ciclo, cioè un predicato che è soddisfatto sia nello stato iniziale della computazione che nello stato finale della computazione di ciascuna iterazione, assieme ad una funzione descrescente che misura il tempo residuo alla fine dell'esecuzione dell'istruzione di ripetizione basandosi sulle variabili di controllo del ciclo (ricordiamo che Turing dimostrò che il problema della terminazione è indecidibile). Sotto certe ipotesi, l'invariante di ciclo è la precondizione dell'istruzione di ripetizione.
- Teorema dell'invariante di ciclo: Data un'istruzione di ripetizione "`while (`β`)` S", se esistono un predicato P e una funzione intera tr tali che:

$$\{P \wedge \beta\}\ S\ \{P\} \qquad \text{(invarianza)}$$
$$\{P \wedge \beta \wedge tr(i) = t\}\ S\ \{tr(i+1) < t\} \qquad \text{(progresso)}$$
$$(P \wedge tr(i) \leq 0) \rightarrow \neg\beta \qquad \text{(limitatezza)}$$

allora:

$$\{P\}\ \mathtt{while\ (}\beta\mathtt{)}\ S\ \{P \wedge \neg\beta\}$$

- Corollario: Date un'istruzione di ripetizione "`while (`β`)` S" ed una postcondizione R, se esiste un invariante di ciclo P per quell'istruzione tale che:

$$(P \wedge \neg\beta) \rightarrow R$$

allora:

$$\{P\}\ \mathtt{while\ (}\beta\mathtt{)}\ S\ \{R\}$$

- Esempio: il seguente programma per calcolare la somma dei valori contenuti in un array di 10 elementi:

```
somma = 0;
i = 0;
while (i <= 9)
{
  somma = somma + a[i];
  i = i + 1;
}
```

è corretto perché, formalizzando la postcondizione nel seguente modo:

$$R \;=\; (somma = \sum_{j=0}^{9} a[j])$$

si può rendere la tripla vera mettendo come precondizione *vero* in quanto:

– Il predicato:

$$P \;=\; (0 \leq i \leq 10 \wedge somma = \sum_{j=0}^{i-1} a[j])$$

e la funzione:

$$tr(i) \;=\; 10 - i$$

soddisfano le ipotesi del teorema dell'invariante di ciclo in quanto:

∗ L'invarianza:

$$\{P \wedge i \leq 9\}\ \texttt{somma = somma + a[i]; i = i + 1;}\ \{P\}$$

segue da:

$$\begin{aligned} P_{\texttt{i,i+1}} &= (0 \leq i+1 \leq 10 \wedge somma = \sum_{j=0}^{i+1-1} a[j]) \\ &\equiv (0 \leq i+1 \leq 10 \wedge somma = \sum_{j=0}^{i} a[j]) \end{aligned}$$

e, denotato con P' quest'ultimo predicato, da:

$$\begin{aligned} P'_{\texttt{somma,somma+a[i]}} &= (0 \leq i+1 \leq 10 \wedge somma + a[i] = \sum_{j=0}^{i} a[j]) \\ &\equiv (0 \leq i+1 \leq 10 \wedge somma = \sum_{j=0}^{i-1} a[j]) \end{aligned}$$

in quanto, denotato con P'' quest'ultimo predicato, si ha:

$$\begin{aligned} (P \wedge i \leq 9) &= (0 \leq i \leq 10 \wedge somma = \sum_{j=0}^{i-1} a[j] \wedge i \leq 9) \\ &\models P'' \end{aligned}$$

∗ Il progresso è garantito dal fatto che $tr(i)$ decresce di un'unità ad ogni iterazione in quanto `i` viene incrementata di un'unità ad ogni iterazione.

∗ La limitatezza segue da:

$$\begin{aligned} (P \wedge tr(i) \leq 0) &= (0 \leq i \leq 10 \wedge somma = \sum_{j=0}^{i-1} a[j] \wedge 10 - i \leq 0) \\ &\equiv (i = 10 \wedge somma = \sum_{j=0}^{9} a[j]) \\ &\models (i \not\leq 9) \end{aligned}$$

– Poiché:

$$\begin{aligned} (P \wedge i \not\leq 9) &= (0 \leq i \leq 10 \wedge somma = \sum_{j=0}^{i-1} a[j] \wedge i \not\leq 9) \\ &\equiv (i = 10 \wedge somma = \sum_{j=0}^{9} a[j]) \\ &\models R \end{aligned}$$

dal corollario del teorema dell'invariante di ciclo segue che P può essere usato come precondizione dell'intera istruzione di ripetizione.

– Proseguendo infine a ritroso si ottiene prima:

$$P_{\texttt{i,0}} \;=\; (0 \leq 0 \leq 10 \wedge somma = \sum_{j=0}^{0-1} a[j]) \;\equiv\; (somma = 0)$$

e poi, denotato con P''' quest'ultimo predicato, si ha:

$$P'''_{\texttt{somma,0}} \;=\; (0 = 0) \;\equiv\; vero$$

7.4 Verifica della correttezza di programmi procedurali ricorsivi

- Come nel caso dei programmi iterativi, così anche nel caso dei programmi ricorsivi non ci sono regole come quelle di Dijkstra da applicare meccanicamente.

- Per verificare la correttezza di un programma ricorsivo, conviene ricorrere al principio di induzione, avvalendosi eventualmente anche delle triple di Hoare.

- Esempi relativi ad alcune delle funzioni ricorsive della Sez. 5.8:

  - La funzione ricorsiva per calcolare il fattoriale di $n \in \mathbb{N}$:

```
int fattoriale(int n)
{
  int fatt;

  if (n == 0)
    fatt = 1;
  else
    fatt = n * fattoriale(n - 1);
  return(fatt);
}
```

    soddisfa `fattoriale(n)` $= n!$ per ogni $n \in \mathbb{N}$, come si dimostra procedendo per induzione su n:

    * Sia $n = 0$. Risulta `fattoriale(0)` $= 1 = 0!$ e quindi l'asserto è vero per $n = 0$.
    * Dato un certo $n \geq 1$, supponiamo che `fattoriale(n - 1)` $= (n-1)!$. Risulta `fattoriale(n)` $=$ `n * fattoriale(n - 1)` $= n \cdot (n-1)!$ per ipotesi induttiva. Poiché $n \cdot (n-1)! = n!$, l'asserto è vero per n.

  - La funzione ricorsiva per calcolare l'n-esimo numero di Fibonacci ($n \geq 1$):

```
int fibonacci(int n)
{
  int fib;

  if ((n == 1) || (n == 2))
    fib = 1;
  else
    fib = fibonacci(n - 1) + fibonacci(n - 2);
  return(fib);
}
```

    soddisfa `fibonacci(n)` $= fib_n$ per ogni $n \in \mathbb{N}$, come si dimostra procedendo per induzione su n:

    * Sia $n \in \{1, 2\}$. Risulta `fibonacci(n)` $= 1 = fib_n$ e quindi l'asserto è vero per $n \in \{1, 2\}$.
    * Dato un certo $n \geq 3$, supponiamo che `fibonacci(m)` $= fib_m$ per ogni m tale che $1 \leq m < n$. Risulta `fibonacci(n)` $=$ `fibonacci(n - 1) + fibonacci(n - 2)` $= fib_{n-1} + fib_{n-2}$ per ipotesi induttiva. Poiché $fib_{n-1} + fib_{n-2} = fib_n$, l'asserto è vero per n.

- Esempio: il massimo e il submassimo di un insieme I_n di $n \geq 2$ elementi può essere determinato attraverso una funzione ricorsiva che ad ogni invocazione dimezza l'insieme da esaminare e poi ne calcola massimo e submassimo confrontando massimi e submassimi delle sue due metà:

```
coppia_t calcola_max_submax(int a[],
                            int sx,
                            int dx)
{
  coppia_t ms,    /* max e submax da a[sx] ad a[dx] */
           ms1,   /* max e submax da a[sx] ad a[(sx + dx) / 2] */
           ms2;   /* max e submax da a[(sx + dx) / 2 + 1] ad a[dx] */

  if (dx - sx + 1 == 2)
  {
    ...   /* vedi sotto, caso n = 2 */
  }
  else
  {
    ms1 = calcola_max_submax(a,
                             sx,
                             (sx + dx) / 2);
    ms2 = calcola_max_submax(a,
                             (sx + dx) / 2 + 1,
                             dx);
    ...   /* vedi sotto, caso n > 2 */
  }
  return(ms);
}
```

Procedendo per induzione su n si dimostra che `calcola_max_submax(`I_n`)` = max_submax(I_n):

– Sia $n = 2$. Risulta `calcola_max_submax(`I_2`)` = max_submax(I_2) e quindi l'asserto è vero per $n = 2$, perché usando le triple di Hoare e procedendo a ritroso si ha:

```
/* {vero} */
if (a[sx] >= a[dx])
{
  /* {a[sx] = max(I_2) /\ a[dx] = submax(I_2)} */
  ms.max = a[sx];
  /* {ms.max = max(I_2) /\ a[dx] = submax(I_2)} */
  ms.submax = a[dx];
}
else
{
  /* {a[dx] = max(I_2) /\ a[sx] = submax(I_2)} */
  ms.max = a[dx];
  /* {ms.max = max(I_2) /\ a[sx] = submax(I_2)} */
  ms.submax = a[sx];
}
/* {ms.max = max(I_2) /\ ms.submax = submax(I_2)} */
```

– Dato un certo $n \geq 3$, sia vero l'asserto per ogni m tale che $2 \leq m < n$. In questo caso, la funzione invoca ricorsivamente `calcola_max_submax(`$I'_{n/2}$`)` e `calcola_max_submax(`$I''_{n/2}$`)`, quindi per ipotesi induttiva `ms1` = max_submax($I'_{n/2}$) ed `ms2` = max_submax($I''_{n/2}$), rispettivamente. L'asserto è allora vero per n, perché usando le triple di Hoare e procedendo a ritroso si ha:

```
/* {vero} */
if (ms1.max >= ms2.max)
{
  /* {ms1.max = max(I_n) /\
      (ms2.max >= ms1.submax --> ms2.max = submax(I_n)) /\
      (ms2.max < ms1.submax --> ms1.submax = submax(I_n))} */
  ms.max = ms1.max;
  if (ms2.max >= ms1.submax)
    /* {ms.max = max(I_n) /\ ms2.max = submax(I_n)} */
    ms.submax = ms2.max;
  else
    /* {ms.max = max(I_n) /\ ms1.submax = submax(I_n)} */
    ms.submax = ms1.submax;
}
else
{
  /* {ms2.max = max(I_n) /\
      (ms1.max >= ms2.submax --> ms1.max = submax(I_n)) /\
      (ms1.max < ms2.submax --> ms2.submax = submax(I_n))} */
  ms.max = ms2.max;
  if (ms1.max >= ms2.submax)
    /* {ms.max = max(I_n) /\ ms1.max = submax(I_n)} */
    ms.submax = ms1.max;
  else
    /* {ms.max = max(I_n) /\ ms2.submax = submax(I_n)} */
    ms.submax = ms2.submax;
}
/* {ms.max = max(I_n) /\ ms.submax = submax(I_n)} */
```

■fltpp_20

Capitolo 8

Introduzione alla logica matematica

8.1 Cenni di storia della logica

- La logica può essere definita come la disciplina che studia la verità di un'affermazione deducendola (tramite regole) da altre affermazioni che sono state poste essere vere (postulati, assiomi, principi) o che sono già state dedotte essere vere (teoremi). Dal punto di vista concettuale, la logica si contrappone all'intuizione e risponde all'esigenza di trovare un modo di distinguere ciò che è vero da ciò che è falso.
- La storia della logica si articola in tre epoche: logica simbolica (dal 500 a.C. alla metà del 1800), logica algebrica (dalla metà del 1800 alla fine del 1800) e logica matematica (dalla fine del 1800 in poi).
- Nell'antica Grecia, la logica era già ritenuta una materia fondamentale. Infatti, insieme alla grammatica e alla retorica, la logica costituiva una delle tre materie del trivium, la cui padronanza era necessaria per affrontare le quattro materie del quadrivium: aritmetica, geometria, musica e astronomia.
- La logica venne originariamente studiata dai sofisti nel VI secolo a.C. allo scopo di definire un sistema oggettivo di regole per determinare oltre ogni dubbio chi avesse vinto una disputa. I sofisti si reputavano portatori di saggezza e credevano che la cultura e la preparazione necessarie nella vita pubblica dovessero essere acquisite attraverso la discussione ed il dibattito piuttosto che attraverso il pensiero profondo. Per questo motivo, essi insegnavano ai giovani ateniesi come usare la logica (cioè il ragionamento) e la retorica (cioè la capacità oratoria finalizzata alla persuasione) per sconfiggere gli avversari nelle controversie.
- Nel IV secolo a.C. Aristotele diede alla logica la forma rigorosamente deduttiva del sillogismo, cioè del ragionamento concatenato attraverso cui si giunge a conclusioni coerenti con le premesse da cui si parte. In ogni sillogismo, sia le premesse che la conclusione sono proposizioni formate da un soggetto universale o particolare e da un predicato verbale affermativo o negativo, dove la verità della conclusione dipende dalla verità delle premesse. La logica di Aristotele si basa su due valori di verità ed è governata da tre leggi: il principio di identità (ogni oggetto è uguale a se stesso), il principio di non contraddizione (due proposizioni in contraddizione non possono essere entrambe vere) e il principio del terzo escluso (tertium non datur: ogni proposizione deve essere o vera o falsa).
- Esempio di sillogismo: tutti gli uomini sono mortali, Socrate è un uomo, quindi Socrate è mortale. Per fare ragionamenti come questo, bisogna definire quali inferenze si possano ricavare da parole come "tutti" piuttosto che "alcuni", cosicché la dimostrazione possa poi seguire da queste definizioni.
- Inizialmente la logica trattava affermazioni espresse in linguaggio naturale e si basava su regole di ragionamento molto semplici. Sfortunatamente, l'intrinseca ambiguità del linguaggio naturale conduce facilmente a dei paradossi, spesso nella forma di autologie (affermazioni che descrivono se stesse):
  - Paradosso del bugiardo: questa affermazione è una bugia. Se l'affermazione fosse vera, allora essa sarebbe una bugia e quindi sarebbe falsa. Se l'affermazione fosse falsa, allora essa non sarebbe una bugia e quindi sarebbe vera. Non si riesce dunque a stabilire se l'affermazione sia vera o falsa.

  - Un esempio più complesso è il paradosso del sofista. Uno studente si iscrive ad un corso di sofismo insegnato presso una scuola, concordando di pagare la quota di iscrizione quando avrà imparato i principi del sofismo. Al termine del corso, la scuola chiede di essere pagata ma lo studente rifiuta di darle i soldi e quindi la scuola lo cita in giudizio. Se lo studente vincesse la causa, allora non dovrebbe pagare l'iscrizione, ma la sua vittoria implicherebbe che la scuola lo aveva istruito sufficientemente bene e quindi la scuola meriterebbe di ricevere il denaro. Se lo studente perdesse la causa, allora dovrebbe pagare l'iscrizione, ma la sua sconfitta implicherebbe che la scuola non lo aveva istruito sufficientemente bene e quindi la scuola non meriterebbe di ricevere il denaro.

- Nella seconda metà del 1600 Leibniz riteneva che una larga parte del ragionamento umano potesse essere ridotta a calcoli e che questi ultimi potessero essere usati per risolvere molte divergenze di opinioni (da cui il motto "calculemus"). A tale scopo, Leibniz immaginò un linguaggio formale universale detto "characteristica universalis" in grado di esprimere concetti matematici, scientifici e metafisici, da usare nel contesto di un calcolo logico universale detto "calculus ratiocinator". Leibniz enunciò inoltre le principali proprietà di ciò che ora chiamiamo congiunzione, disgiunzione e negazione ed introdusse il sistema di numerazione binario, i cui due simboli di base 0 ed 1 possono essere messi in corrispondenza biunivoca con i due valori di verità falso e vero.

- La prima formulazione della logica in termini di un linguaggio matematico venne espressa nel 1847 da Boole nel suo libro "The Mathematical Analysis of Logic". Il suo obiettivo era quello di investigare le leggi fondamentali delle operazioni che avvengono nella mente umana durante il ragionamento. Boole individuò un'analogia tra le operazioni aritmetiche di addizione e moltiplicazione e le operazioni insiemistiche di unione ed intersezione: ad esempio $a \cdot (b + c) = (a \cdot b) + (a \cdot c)$ così come $A \cap (B \cup C) = (A \cap B) \cup (A \cap C)$. Egli estese questa analogia alla logica introducendo e studiando le proprietà algebriche di un operatore di disgiunzione logica (OR) e un operatore di congiunzione logica (AND) accompagnati da un operatore di negazione logica (NOT) e dai valori di verità falso (0) e vero (1).

- Nello stesso anno De Morgan fornì nel suo libro "Formal Logic" degli importanti contributi allo studio dei sillogismi numericamente definiti. Nel 1880 Venn propose nel suo articolo "On the Diagrammatic and Mechanical Representation of Propositions and Reasonings" l'uso di particolari diagrammi per rappresentare gli insiemi e le loro operazioni che, grazie all'analogia di Boole, potevano essere impiegati anche nella logica. Alla fine del 1800 Schröder anticipò nel suo libro "The Algebra of Logic" l'importanza di trovare metodi rapidi per determinare le conseguenze di premesse arbitrarie.

- Come la logica necessitava della matematica per superare l'ambiguità del linguaggio naturale, così la matematica necessitava di rigorosi meccanismi deduttivi per migliorare la qualità delle dimostrazioni dei teoremi. Ad esempio, nel 1820 Cauchy "provò" che la somma $\sum_{i \in \mathbb{N}} f_i$ di una qualsiasi successione $(f_i)_{i \in \mathbb{N}}$ di funzioni continue è anch'essa continua, ma nel 1826 Abel trovò un controesempio.

- Nel 1879 Frege propose nel suo "Begriffsschrift" la logica come un linguaggio per la matematica funzionale alla dimostrazione dei teoremi, dove eventuali elementi intuitivi o primitivi devono essere isolati come assiomi (o postulati o principi) e la prova deve procedere a partire da questi assiomi in modo puramente logico e senza salti. Frege introdusse inoltre un sistema logico con negazione, implicazione, quantificazione universale e tabelle di verità, che può essere visto come il primo calcolo dei predicati.

- Grazie a questo rinnovato rigore, verso la fine del 1800 Cantor evidenziò come la teoria degli insiemi fosse degna di essere studiata in quanto non c'è un unico concetto di infinito ma esiste una gerarchia di infiniti. Ad esempio, Cantor provò – utilizzando quello che oggi viene chiamato il metodo della diagonale – che l'insieme (infinito) dei numeri reali strettamente compresi tra 0 ed 1 è più numeroso dell'insieme (infinito) dei numeri naturali. Supponiamo che i numeri reali tra 0 ed 1 siano tanti quanti i numeri naturali e che quindi possano essere enumerati come segue dove i valori $d_{i,j}$ sono cifre decimali:

$$\begin{array}{lcl} r_1 & = & 0.d_{1,1}d_{1,2}\ldots d_{1,m}\ldots \\ r_2 & = & 0.d_{2,1}d_{2,2}\ldots d_{2,m}\ldots \\ \ldots & \ldots & \ldots\ldots\ldots\ldots\ldots \\ r_n & = & 0.d_{n,1}d_{n,2}\ldots d_{n,m}\ldots \\ \ldots & \ldots & \ldots\ldots\ldots\ldots\ldots \end{array}$$

Consideriamo ora il numero reale $r = 0.d_1d_2 \dots d_m \dots$ tale che $d_k \neq d_{k,k}$ per ogni k, dove almeno una delle sue cifre dopo il punto decimale è diverso da 0. Poiché r è un numero reale strettamente compreso tra 0 ed 1, esso deve comparire nell'elenco, ma così non è perché $r \neq r_k$ per ogni k in quanto le loro k-esime cifre dopo il punto decimale sono diverse. Quindi l'insieme dei numeri reali tra 0 ed 1 ha cardinalità (indicata con $\aleph_1$) maggiore di quella (indicata con $\aleph_0$) dell'insieme dei numeri naturali. In generale, la gerarchia di infiniti è illimitata e può essere ottenuta costruendo ripetutamente l'insieme delle parti iniziando da $\mathbb{N}$.

- La teoria ingenua degli insiemi formulata da Cantor contiene numerosi paradossi. Denotata con T la collezione di tutti gli insiemi che non sono elementi di se stessi, non si riesce a stabilire se T appartiene o no a se stesso (paradosso di Russell). Infatti, se T appartenesse a se stesso, allora T non potrebbe appartenere a se stesso per definizione di T; se T non appartenesse a se stesso, allora T dovrebbe appartenere a se stesso per definizione di T. Analogamente, se la collezione V di tutti gli insiemi fosse un insieme, allora, poiché ogni sottocollezione di V sarebbe un insieme e quindi apparterrebbe a V, concluderemmo che l'insieme delle parti di V sarebbe contenuto in V, ma ciò non è possibile perché non esiste nessuna funzione suriettiva da un insieme al suo insieme delle parti (paradosso di Cantor).

- Per evitare questo genere di paradossi, nel 1903 Russell riconobbe che era necessario formulare la teoria degli insiemi sulla base di un gruppo di assiomi che evitassero definizioni circolari. Come diventerà chiaro con la teoria assiomatica degli insiemi successivamente sviluppata da Zermelo e Fraenkel, non tutte le collezioni di insiemi definibili nel linguaggio usato per la teoria degli insiemi sono insiemi; terminologicamente, si distingue tra classi e insiemi.

- Come conseguenza, agli inizi del 1900 prese piede il logicismo, una scuola di pensiero basata sulla visione di Frege secondo cui la matematica è riducibile alla logica. In particolare, il logicismo di Russell consisteva in due tesi principali. In primo luogo, tutte le verità matematiche possono essere tradotte in verità logiche, cioè il vocabolario della matematica è incluso nel vocabolario della logica. In secondo luogo, tutte le dimostrazioni dei teoremi della matematica possono essere riformulate come prove logiche, cioè l'insieme dei teoremi della matematica è incluso nell'insieme dei teoremi della logica. A supporto del logicismo, tra il 1910 ed il 1913 Russell e Whitehead dimostrarono formalmente nel loro libro "Principia Mathematica" il grosso delle conoscenze matematiche del loro tempo utilizzando pure manipolazioni simboliche.

- All'apice del logicismo di Russell, nel 1920 Hilbert lanciò un programma per trovare una singola procedura formale per derivare tutti i teoremi della matematica, che equivale a riuscire a ridurre tutta la matematica in forma assiomatica e a provare che tale assiomatizzazione non porta a contraddizioni. Hilbert sosteneva che "una volta stabilito un formalismo logico, ci si può aspettare che sia possibile un trattamento sistematico, diciamo computazionale, delle formule logiche, in parte analogo alla teoria delle equazioni in algebra". Secondo Hilbert, le teorie complesse della matematica potevano essere fondate su teorie più semplici fino a basare l'intera matematica sull'aritmetica, cosicché provando la correttezza di quest'ultima si sarebbe provata la non contraddittorietà di tutta la matematica.

- Il colpo di grazia al programma di Hilbert e quindi al logicismo di Russell venne inflitto nel 1931 dai due teoremi di incompletezza dimostrati da Gödel (usando il metodo della diagonale di Cantor):

  - Primo teorema di incompletezza di Gödel: In ogni teoria assiomatizzabile sufficientemente espressiva da formare affermazioni su ciò che può dimostrare, ci saranno sempre delle affermazioni vere che la teoria può esprimere ma non derivare dai suoi assiomi attraverso le sue regole di inferenza. Ciò significa che in ogni teoria matematica sufficientemente potente ci sono dei teoremi che non sono dimostrabili all'interno della teoria stessa; per dimostrarli, bisognerà ricorrere ad una teoria più potente, ma anch'essa soffrirà della stessa limitazione.
  - Secondo teorema di incompletezza di Gödel: Ogni teoria assiomatizzabile sufficientemente espressiva da formare affermazioni sull'aritmetica non potrà mai dimostrare la propria correttezza. Questo risultato evidenzia come già una branca fondamentale della matematica quale l'aritmetica mostri problemi di incompletezza dal punto di vista di un suo trattamento computazionale.

- Nel 1936 apparvero analoghi risultati negativi di natura computazionale:
  - Church dimostrò che la logica dei predicati non è decidibile, cioè che non esiste alcun algoritmo in grado di stabilire in tempo finito se una formula predicativa arbitraria è vera o falsa.
  - Turing dimostrò che il problema della terminazione non è decidibile, cioè che non esiste alcun algoritmo in grado di stabilire in tempo finito se l'esecuzione di un algoritmo arbitrario su un'istanza arbitraria dei suoi dati di ingresso termina oppure no.

- L'esistenza di teoremi che non possono essere dimostrati in nessun sistema deduttivo e di problemi computazionali che non possono essere risolti da nessun algoritmo determinò l'impossibilità di ridurre la matematica alla logica. Dal 1940 in poi la logica continuò a svilupparsi non più come il fondamento universale di tutta la matematica, ma come una branca di essa, pur rimanendo aperta la possibilità di mettere a punto sistemi deduttivi che servano come fondamenti per parti specifiche della matematica.

- La logica è particolarmente importante anche nell'informatica per via della natura computazionale del concetto di deduzione. Facendo un paragone con il ruolo che il calcolo infinitesimale ha avuto nella fisica, a volte la logica è descritta come il calcolo dell'informatica. A partire dai primi risultati di indecidibilità (anni 1930) e ancor più dall'avvento dei primi elaboratori elettronici (anni 1940) e dei primi linguaggi di programmazione di alto livello (anni 1950), la logica ha avuto un ruolo fondamentale e pervasivo nell'informatica:
  - Nella teoria della calcolabilità e della complessità, la logica ha fornito caratterizzazioni del concetto di risolvibilità di un problema computazionale sin dall'introduzione del modello di macchina astratta di Turing, come pure della visione computazionale del concetto di funzione matematica attraverso il λ-calcolo di Church. Inoltre, la logica è stata cruciale nello sviluppo della teoria della NP-completezza, la quale formalizza il concetto di esplosione combinatoria. Ad esempio, il problema della soddisfacibilità di una formula di logica proposizionale è l'archetipo di problema computazionale che non sappiamo come risolvere in modo efficiente.
  - Nell'architettura degli elaboratori, i circuiti che formano i componenti hardware sono costituiti da porte che implementano le operazioni logiche di base introdotte da Boole. Sebbene Boole intendesse scoprire le leggi del ragionamento della mente umana, l'applicazione principale della sua teoria si trova dunque nell'ingegneria elettronica e nell'industria informatica.
  - Nella programmazione degli elaboratori, è necessario formalizzare la semantica dei linguaggi di programmazione in modo che i loro costrutti siano coerenti in implementazioni diverse dello stesso linguaggio e vengano compresi senza ambiguità da parte dei programmatori. La logica fornisce strumenti per sviluppare tale semantica e per ragionare sulle proprietà dei programmi (vedi le triple di Hoare). Inoltre, esistono paradigmi di programmazione – come quello dichiarativo di natura logica di cui il linguaggio Prolog è il principale esponente – in cui la logica viene usata per esprimere i programmi e i suoi meccanismi di ragionamento vengono sfruttati per eseguire i programmi.
  - Nella progettazione dell'hardware e del software, la logica viene utilizzata per verificare la correttezza di un progetto con un grado di confidenza che va oltre quello del testing. Per la precisione, viene spesso impiegata una variante della logica classica detta logica temporale la quale comprende operatori che consentono di esprimere affermazioni sull'ordine in cui certe computazioni hanno luogo oppure sul fatto che certe computazioni possano o debbano avere luogo.
  - Nella gestione delle basi di dati, un linguaggio comunemente impiegato per reperire e manipolare dati è SQL. Questo linguaggio di interrogazione per basi di dati risulta essere un'implementazione della logica del primo ordine.
  - Nell'intelligenza artificiale, la logica è essenziale per formalizzare conoscenze acquisite dall'uomo e regole di ragionamento sulla base delle quali sistemi esperti riescono a dedurre nuove conoscenze. Lo stesso meccanismo si applica ai sistemi per la dimostrazione automatica di teoremi. ■fltpl_1

8.2 Elementi di teoria degli insiemi

- La teoria degli insiemi svolge il ruolo di teoria fondazionale nella matematica moderna, in quanto permette di interpretare all'interno di una singola teoria affermazioni relative ad oggetti – quali ad esempio numeri, relazioni e funzioni – tratti da tutte le principali aree della matematica.
- Assumiamo come primitivi (cioè non riconducibili ad altri più semplici) i concetti di elemento, insieme ed appartenenza. Scriviamo $a \in A$ per indicare che a è un elemento appartenente all'insieme A e $a \notin A$ per indicare che a non è un elemento appartenente all'insieme A. Denotiamo inoltre con U l'universo del discorso, cioè un insieme a cui appartengono tutti gli oggetti di cui si sta parlando.
- Ogni insieme può essere definito in al più due modi:
  - Se l'insieme è formato da un numero finito di elementi, esso può essere definito elencandone gli elementi, i quali saranno separati da virgole e racchiusi tra parentesi graffe: $\{a_1, a_2, \ldots, a_n\}$. L'ordine in cui gli elementi sono elencati e il numero di volte che un elemento è ripetuto non sono rilevanti.
  - L'insieme può sempre essere definito attraverso una proprietà $\mathcal{Q}$ soddisfatta da tutti e soli gli elementi dell'insieme: $\{u \in U \mid u \text{ soddisfa } \mathcal{Q}\}$. L'insieme è detto essere l'insieme di verità per $\mathcal{Q}$.
- L'insieme vuoto, cioè l'insieme privo di elementi, viene denotato con $\emptyset$.
- Siano A e B due insiemi:
  - A è un sottoinsieme di B, scritto $A \subseteq B$, sse ogni elemento di A è anche elemento di B.
  - A è uguale a B, scritto $A = B$, sse $A \subseteq B$ e $B \subseteq A$.
  - A è un sottoinsieme proprio di B, scritto $A \subset B$ o $A \subsetneq B$ o $A \subsetneqq B$, sse $A \subseteq B$ e $A \neq B$.
- Sia A un insieme:
  - La cardinalità di A, denotata con $|A|$ o $card(A)$, è il numero di elementi di A.
  - L'insieme delle parti di A, denotato con $\mathcal{P}(A)$ o 2^A, è l'insieme di tutti i sottoinsiemi di A.
- Proprietà dell'inclusione e dell'uguaglianza insiemistiche rispetto alla cardinalità:
  - Se $A \subseteq B$, allora $|A| \leq |B|$.
  - Se $A \subset B$ e $|A| \in \mathbb{N}$, allora $|A| < |B|$.
  - Se $A = B$, allora $|A| = |B|$.
  - $|A| < |\mathcal{P}(A)|$.
  - Se $|A| = n \in \mathbb{N}$, allora $|\mathcal{P}(A)| = 2^n$.
- Dato un insieme A, si definiscono le seguenti operazioni unarie su di esso:
  - Complementazione: $\overline{A} = A^{\mathrm{c}} = \mathcal{C}(A) = \complement(A) = \{u \in U \mid u \notin A\}$.
- Dati due insiemi A e B, si definiscono le seguenti operazioni binarie su di essi:
  - Unione: $A \cup B = \{u \in U \mid u \in A \text{ o } u \in B\}$.
  - Intersezione: $A \cap B = \{u \in U \mid u \in A \text{ e } u \in B\}$.
  - Differenza: $A \setminus B = \{u \in U \mid u \in A \text{ e } u \notin B\}$.
  - Differenza simmetrica: $A \,\Delta\, B = \{u \in U \mid u \in A \text{ e } u \notin B, \text{ oppure } u \in B \text{ e } u \notin A\}$.
- Proprietà delle operazioni insiemistiche:
  - $\overline{A} = U \setminus A$, $\overline{\emptyset} = U$, $\overline{U} = \emptyset$.

  - $A \cup B$ è il più piccolo insieme che include sia A che B e quindi:
    * $A \subseteq B$ sse $A \cup B = B$.
    * $A \subseteq B$ sse $\overline{A} \cup B = U$.
  - $A \cap B$ è il più grande insieme incluso sia in A che in B e quindi:
    * $A \subseteq B$ sse $A \cap B = A$.
    * $A \subseteq B$ sse $A \cap \overline{B} = \emptyset$.
  - Le due operazioni di differenza possono essere derivate dalle altre tre operazioni:
    * $A \setminus B = A \cap \overline{B}$.
    * $A \,\Delta\, B = (A \setminus B) \cup (B \setminus A) = (A \cup B) \setminus (A \cap B)$.

- La struttura algebrica $(\mathcal{P}(U), \cup, \cap, \bar{}, \emptyset, U)$ è un reticolo booleano, cioè soddisfa le seguenti leggi:
  - Commutatività:
    * $A \cup B = B \cup A$.
    * $A \cap B = B \cap A$.
  - Associatività:
    * $(A \cup B) \cup C = A \cup (B \cup C)$.
    * $(A \cap B) \cap C = A \cap (B \cap C)$.
  - Assorbimento:
    * $A \cup (A \cap B) = A$.
    * $A \cap (A \cup B) = A$.
  - Distributività:
    * $A \cup (B \cap C) = (A \cup B) \cap (A \cup C)$.
    * $A \cap (B \cup C) = (A \cap B) \cup (A \cap C)$.
  - Elemento neutro:
    * $A \cup \emptyset = A$.
    * $A \cap U = A$.
  - Elemento inverso:
    * $A \cup \overline{A} = U$.
    * $A \cap \overline{A} = \emptyset$.

- La struttura algebrica $(\mathcal{P}(U), \cup, \cap, \bar{}, \emptyset, U)$ possiede inoltre le seguenti proprietà derivate:
  - Elemento assorbente:
    * $A \cup U = U$.
    * $A \cap \emptyset = \emptyset$.
  - Idempotenza:
    * $A \cup A = A$.
    * $A \cap A = A$.
  - Doppia inversione:
    * $\overline{\overline{A}} = A$.
  - Leggi di De Morgan:
    * $\overline{A \cup B} = \overline{A} \cap \overline{B}$.
    * $\overline{A \cap B} = \overline{A} \cup \overline{B}$.

8.3 Relazioni, funzioni, operazioni

- Molto spesso in matematica si cerca di stabilire delle relazioni tra gli elementi di vari insiemi. Queste relazioni possono essere di diversa natura e tra le più importanti citiamo le relazioni d'ordine, le relazioni d'equivalenza e le relazioni funzionali (che includono le operazioni). Esse sono definite sulla base del concetto di coppia ordinata e di un'ulteriore operazione insiemistica detta prodotto cartesiano.

- Dati due elementi a e b, la coppia ordinata (a, b) è definita come l'insieme $\{\{a, b\}, \{a\}\}$. Si noti che i due insiemi $\{a, b\}$ e $\{b, a\}$ sono sempre uguali mentre le due coppie ordinate (a, b) e (b, a) sono diverse quando $a \neq b$.

- Dati due insiemi A e B, il loro prodotto cartesiano è l'insieme di tutte le coppie ordinate formate dai loro elementi, cioè $A \times B = \{(a, b) \mid a \in A \text{ e } b \in B\}$. Questa operazione non è commutativa in generale.

- Esempio: se U è l'insieme dei punti di una linea retta, allora $U \times U$ può essere visto come una rappresentazione del piano cartesiano in cui ogni coppia ordinata (u_1, u_2) denota le coordinate di un punto del piano.

- Una relazione tra l'insieme A e l'insieme B è un sottoinsieme di $A \times B$. Quando $A = B$, diciamo che essa è una relazione su A.

- Sia $\mathcal{R} \subseteq A \times B$:

  - Il dominio di $\mathcal{R}$ è l'insieme $dom(\mathcal{R}) = \{a \in A \mid \text{esiste } b \in B \text{ tale che } (a, b) \in \mathcal{R}\}$ incluso in A.
  - Il codominio di $\mathcal{R}$ è l'insieme $cod(\mathcal{R}) = \{b \in B \mid \text{esiste } a \in A \text{ tale che } (a, b) \in \mathcal{R}\}$ incluso in B.
  - Il campo di $\mathcal{R}$ è l'insieme $campo(\mathcal{R}) = dom(\mathcal{R}) \cup cod(\mathcal{R})$ incluso in $A \cup B$.

- Data una relazione $\mathcal{R} \subseteq A \times A$, diciamo che essa è:

  - Riflessiva sse $(a, a) \in \mathcal{R}$ per ogni $a \in campo(\mathcal{R})$.
  - Antiriflessiva sse $(a, a) \notin \mathcal{R}$ per ogni $a \in campo(\mathcal{R})$.
  - Simmetrica sse $(a_1, a_2) \in \mathcal{R}$ implica $(a_2, a_1) \in \mathcal{R}$ per ogni $a_1, a_2 \in campo(\mathcal{R})$.
  - Antisimmetrica sse $(a_1, a_2) \in \mathcal{R}$ implica $(a_2, a_1) \notin \mathcal{R}$ per ogni $a_1, a_2 \in campo(\mathcal{R})$, $a_1 \neq a_2$.
  - Transitiva sse $(a_1, a_2) \in \mathcal{R}$ e $(a_2, a_3) \in \mathcal{R}$ implicano $(a_1, a_3) \in \mathcal{R}$ per ogni $a_1, a_2, a_3 \in campo(\mathcal{R})$.

- Diciamo che $\mathcal{R} \subseteq A \times A$ è una relazione d'ordine parziale sse $\mathcal{R}$ è riflessiva, antisimmetrica e transitiva. Diciamo che $\mathcal{R}$ è una relazione d'ordine totale o lineare sse $\mathcal{R}$ soddisfa anche la seguente proprietà chiamata dicotomia: $(a_1, a_2) \in \mathcal{R}$ o $(a_2, a_1) \in \mathcal{R}$ per ogni $a_1, a_2 \in campo(\mathcal{R})$.

- Diciamo che $\mathcal{R} \subseteq A \times A$ è una relazione d'equivalenza sse $\mathcal{R}$ è riflessiva, simmetrica e transitiva. In tal caso, chiamiamo insieme quoziente l'insieme $campo(\mathcal{R})/\mathcal{R}$ di tutte le classi d'equivalenza $[a]_{\mathcal{R}} = \{a' \in A \mid (a, a') \in \mathcal{R}\}$ per $a \in campo(\mathcal{R})$, il quale è una partizione di $campo(\mathcal{R})$, cioè:

  - $[a]_{\mathcal{R}} \neq \emptyset$ per ogni $a \in campo(\mathcal{R})$.
  - $[a_1]_{\mathcal{R}} \cap [a_2]_{\mathcal{R}} = \emptyset$ per ogni $a_1, a_2 \in campo(\mathcal{R})$ tali che $(a_1, a_2) \notin \mathcal{R}$.
  - $\bigcup_{a \in campo(\mathcal{R})} [a]_{\mathcal{R}} = campo(\mathcal{R})$.

- Esempi:

  - La relazione $\leq$ è una relazione d'ordine totale su $\mathbb{N}$.
  - La relazione $\subseteq$ è una relazione d'ordine parziale su $\mathcal{P}(U)$.
  - La relazione $=$ è una relazione d'equivalenza su $\mathbb{N}$ e $\mathcal{P}(U)$.

  - La relazione $\mathcal{R}_5 = \{(n_1, n_2) \in \mathbb{N} \times \mathbb{N} \mid n_1 \text{ ed } n_2 \text{ danno lo stesso resto nella divisione per } 5\}$ è una relazione d'equivalenza le cui classi sono $[0]_{\mathcal{R}_5} = \{5 \cdot n \mid n \in \mathbb{N}\}$, $[1]_{\mathcal{R}_5} = \{5 \cdot n + 1 \mid n \in \mathbb{N}\}$, $[2]_{\mathcal{R}_5} = \{5 \cdot n + 2 \mid n \in \mathbb{N}\}$, $[3]_{\mathcal{R}_5} = \{5 \cdot n + 3 \mid n \in \mathbb{N}\}$, $[4]_{\mathcal{R}_5} = \{5 \cdot n + 4 \mid n \in \mathbb{N}\}$.

- Diciamo che $\mathcal{R} \subseteq A \times B$ è una relazione funzionale tra A e B, o equivalentemente che $\mathcal{R}$ è una funzione da A a B, sse per ogni $a \in dom(\mathcal{R})$ esiste esattamente un $b \in B$ tale che $(a, b) \in \mathcal{R}$. In tal caso scriviamo $\mathcal{R} : A \to B$ e poniamo $\mathcal{R}(a) = b$ per indicare che $(a, b) \in \mathcal{R}$.

- Data una funzione $f : A \to B$, diciamo che essa è:

  - Parziale sse $dom(f) \subset A$.
  - Totale sse $dom(f) = A$.
  - Iniettiva sse per ogni $b \in B$ esiste al più un $a \in dom(f)$ tale che $f(a) = b$.
  - Suriettiva sse per ogni $b \in B$ esiste almeno un $a \in dom(f)$ tale che $f(a) = b$.
  - Biiettiva sse per ogni $b \in B$ esiste esattamente un $a \in dom(f)$ tale che $f(a) = b$.

- Le funzioni possono essere usate per caratterizzare gli insiemi:

  - Ogni insieme A può essere individuato attraverso la sua funzione caratteristica $\chi_A : U \to \{0, 1\}$ così definita: $\chi_A(u) = 1$ se $u \in A$, $\chi_A(u) = 0$ se $u \notin A$.
  - Due insiemi A e B sono equipotenti, cioè $|A| = |B|$, sse esiste una funzione totale biiettiva da A a B.
  - Un insieme non vuoto A è finito sse esiste $n \in \mathbb{N}$ tale che A è equipotente ad $\{1, \ldots, n\}$.
  - Un insieme A è infinito sse è equipotente ad un suo sottoinsieme proprio, cioè diverso da $\emptyset$ e A.
  - Un insieme infinito A è numerabile sse è equipotente ad $\mathbb{N}$.

- Tutte le operazioni (come quelle aritmetiche e quelle insiemistiche) sono formalizzate tramite funzioni e vengono classificate come unarie o binarie a seconda che il dominio delle corrispondenti funzioni sia costituito da un solo insieme (inversione di segno, complementazione) o dal prodotto cartesiano di due insiemi (addizione, sottrazione, moltiplicazione, divisione, unione, intersezione, differenza, differenza simmetrica, prodotto cartesiano).

- Le operazioni non vengono espresse con la stessa notazione delle funzioni. Le operazioni unarie sono rappresentate in notazione prefissa – cioè il simbolo dell'operazione viene scritto prima dell'operando senza racchiudere quest'ultimo tra parentesi – mentre le operazioni binarie sono rappresentate in notazione infissa – cioè il simbolo dell'operazione viene scritto in mezzo ai due operandi.

- Un insieme è detto essere chiuso rispetto ad un'operazione sse l'operazione fornisce un risultato appartenente a quell'insieme quando viene applicata ad operandi di quell'insieme. Ad esempio, $\mathbb{N}$ non è chiuso rispetto alla sottrazione, $\mathbb{Z}$ non è chiuso rispetto alla divisione, $\mathbb{Q}$ non è chiuso rispetto all'estrazione di radice di numeri positivi, $\mathbb{R}$ non è chiuso rispetto all'estrazione di radice di numeri negativi (mentre $\mathbb{C}$ lo è) e $\mathcal{P}(U)$ non è chiuso rispetto al prodotto cartesiano.

- Una relazione d'equivalenza su un insieme chiuso rispetto ad un'operazione è detta essere una congruenza rispetto a quell'operazione sse, ogni volta che si sostituisce un operando con un altro operando equivalente al primo, si ottiene un risultato equivalente a quello originario. Formalmente, date una relazione d'equivalenza $\mathcal{R} \subseteq A \times A$ e un'operazione $\odot : A \times A \times \ldots \times A \to A$, la relazione $\mathcal{R}$ è una congruenza rispetto all'operazione $\odot$ sse per ogni $a_1, a_2, \ldots, a_n, a'_1, a'_2, \ldots, a'_n \in A$ risulta che, se $(a_i, a'_i) \in \mathcal{R}$ per ogni $1 \leq i \leq n$, allora $(\odot(a_1, a_2, \ldots, a_n), \odot(a'_1, a'_2, \ldots, a'_n)) \in \mathcal{R}$.

- L'importanza di essere una congruenza è dovuta al fatto che questa proprietà supporta il ragionamento composizionale. Ad esempio, le manipolazioni che si possono apportare a singole parti di un'espressione aritmetica o insiemistica garantiscono la preservazione del valore dell'espressione originaria grazie al fatto che la relazione $=$ è una congruenza rispetto alle operazioni aritmetiche e insiemistiche. ■fltpl_2

8.4 Principio di induzione

- Il principio di induzione è il quinto dei postulati introdotti alla fine del 1800 da Peano per dare una definizione assiomatica di $\mathbb{N}$ come il più piccolo insieme che contiene 0 ed è chiuso rispetto all'operazione unaria di successore:

  1. Esiste un numero $0 \in \mathbb{N}$.
  2. Esiste una funzione totale $succ : \mathbb{N} \to \mathbb{N}$.
  3. Per ogni $n \in \mathbb{N}$, $succ(n) \neq 0$.
  4. Per ogni $n, n' \in \mathbb{N}$, se $n \neq n'$ allora $succ(n) \neq succ(n')$.
  5. Se M è un sottoinsieme di $\mathbb{N}$ tale che:
     (a) $0 \in M$;
     (b) per ogni $n \in \mathbb{N}$, $n \in M$ implica $succ(n) \in M$;
     allora $M = \mathbb{N}$.

- Oltre che $\mathbb{N}$ stesso, il principio di induzione consente di definire *in modo finito* le quattro operazioni aritmetiche su $\mathbb{N}$ avendo a disposizione la funzione $succ : \mathbb{N} \to \mathbb{N}$ ed introducendo la funzione totale $pred : \mathbb{N}_{\neq 0} \to \mathbb{N}$ tale che $pred(succ(n)) = n$ per ogni $n \in \mathbb{N}$ e $succ(pred(n)) = n$ per ogni $n \in \mathbb{N}_{\neq 0}$:

  – addizione: $m \oplus n = \begin{cases} m & \text{se } n = 0 \\ succ(m) \oplus pred(n) & \text{se } n \neq 0 \end{cases}$
  sulla cui base si definisce $n \leq m$ sse esiste $n' \in \mathbb{N}$ tale che $n \oplus n' = m$ e conseguentemente $n < m$ sse $n \leq m$ con $n \neq m$, nonché $m \geq n$ sse $n \leq m$ ed $m > n$ sse $n < m$;

  – sottrazione: $m \ominus n = \begin{cases} m & \text{se } n = 0 \\ pred(m) \ominus pred(n) & \text{se } n > 0 \end{cases}$ dove $n \leq m$;

  – moltiplicazione: $m \otimes n = \begin{cases} 0 & \text{se } n = 0 \\ m \oplus (m \otimes pred(n)) & \text{se } n > 0 \end{cases}$;

  – divisione: $m \oslash n = \begin{cases} 0 & \text{se } m < n \\ succ((m \ominus n) \oslash n) & \text{se } m \geq n \end{cases}$ dove $n \neq 0$.

- Esempi:

  – $5 \oplus 2 = 6 \oplus 1 = 7 \oplus 0 = 7$.
  – $5 \ominus 2 = 4 \ominus 1 = 3 \ominus 0 = 3$.
  – $5 \otimes 2 = 5 \oplus (5 \otimes 1) = 5 \oplus (5 \oplus (5 \otimes 0)) = 5 \oplus (5 \oplus 0) = 5 \oplus 5 = \ldots = 10$.
  – $5 \oslash 2 = succ((5 \ominus 2) \oslash 2) = \ldots = succ(3 \oslash 2) = succ(succ((3 \ominus 2) \oslash 2)) = \ldots = succ(succ(1 \oslash 2)) = succ(succ(0)) = succ(1) = 2$.

- In generale, il principio di induzione fornisce un meccanismo per descrivere *in modo finito* un insieme infinito numerabile. Questo meccanismo prende il nome di definizione ricorsiva e comprende un caso base – nel quale la definizione è espressa in modo diretto – ed un caso induttivo – nel quale la definizione è espressa ricorrendo ad una definizione della stessa natura che è più vicina al caso base.

- Esempi:

  – Il fattoriale di un numero $n \in \mathbb{N}$ tale che $n \geq 1$ viene normalmente definito come $n! = \prod_{i=1}^{n} i$.
  È tuttavia possibile definirlo anche nel seguente modo ricorsivo:
  $$n! = \begin{cases} 1 & \text{se } n = 1 \\ n \cdot (n-1)! & \text{se } n > 1 \end{cases}$$

- La potenza cartesiana n-esima di un insieme A, con $n \in \mathbb{N}$ tale che $n \geq 1$, viene normalmente definita come $A^n = \underbrace{A \times \ldots \times A}_{n \text{ volte}}$. È tuttavia possibile definirla anche nel seguente modo ricorsivo:

$$A^n = \begin{cases} A & \text{se } n = 1 \\ A^{n-1} \times A & \text{se } n > 1 \end{cases}$$

- La chiusura riflessiva e transitiva di una relazione $\mathcal{R}$ su un insieme A è la relazione $\mathcal{R}^*$ ottenuta da $\mathcal{R}$ rendendola riflessiva e transitiva. Se definiamo la composizione di due relazioni $\mathcal{R}_1$ ed $\mathcal{R}_2$ su A ponendo $\mathcal{R}_1 \circ \mathcal{R}_2 = \{(a_1, a_2) \mid \text{esiste } a \in A \text{ tale che } (a_1, a) \in \mathcal{R}_1 \text{ e } (a, a_2) \in \mathcal{R}_2\}$, allora $\mathcal{R}^* = \bigcup_{n \in \mathbb{N}} \mathcal{R}^n$ dove $\mathcal{R}^n = \underbrace{\mathcal{R} \circ \ldots \circ \mathcal{R}}_{n \text{ volte}}$ può essere definito ricorsivamente come segue:

$$\mathcal{R}^n = \begin{cases} \mathcal{I}_A = \{(a, a) \mid a \in A\} & \text{se } n = 0 \\ \mathcal{R}^{n-1} \circ \mathcal{R} & \text{se } n > 0 \end{cases}$$

  Poiché $\mathcal{R}$ è interpretabile come un grafo i cui vertici corrispondono agli elementi di A e i cui archi corrispondono agli elementi di $\mathcal{R}$, la relazione $\mathcal{R}^*$ stabilisce se esiste un percorso tra tutte le coppie di vertici del grafo, con $\mathcal{R}^n$ rappresentante l'esistenza di un percorso composto da $n \in \mathbb{N}$ archi.

- Dato un insieme non vuoto di simboli Σ detto alfabeto, l'insieme Σ^* di tutte le frasi (o sequenze o stringhe o parole) di lunghezza finita basate su Σ viene ottenuto attraverso una definizione ricorsiva analoga a quella della chiusura riflessiva e transitiva di una relazione, che prende il nome di chiusura (o stella) di Kleene. Precisamente $\Sigma^* = \bigcup_{n \in \mathbb{N}} \Sigma^n$ dove:

$$\Sigma^n = \begin{cases} \{\varepsilon\} & \text{se } n = 0 \\ \{\sigma a \mid \sigma \in \Sigma^{n-1} \text{ e } a \in \Sigma\} & \text{se } n > 0 \end{cases}$$

  utilizzando ε per denotare la frase vuota e σa per denotare la concatenazione di σ ed a. Dunque Σ^n rappresenta l'insieme delle frasi composte da $n \in \mathbb{N}$ simboli di Σ. Si dice linguaggio su Σ un qualsiasi sottoinsieme L di Σ^*. Il motivo per cui un linguaggio L su Σ non coincide necessariamente con Σ^* è che L contiene tutte e sole le frasi di Σ^* che soddisfano le regole grammaticali del linguaggio stesso, cioè le frasi sintatticamente ben formate.

- Ci sono due formulazioni equivalenti del principio di induzione:

  - Siano $n_0 \in \mathbb{N}$ e $\mathcal{Q}$ una proprietà definita su ogni $n \in \mathbb{N}$ tale che $n \geq n_0$. Se:

    1. $\mathcal{Q}$ è soddisfatta da n_0;
    2. per ogni $n \in \mathbb{N}$ tale che $n \geq n_0$, $\mathcal{Q}$ soddisfatta da n implica $\mathcal{Q}$ soddisfatta da $succ(n)$;

    allora $\mathcal{Q}$ è soddisfatta da ogni $n \in \mathbb{N}$ tale che $n \geq n_0$.

  - Siano $n_0 \in \mathbb{N}$ e $\mathcal{Q}$ una proprietà definita su ogni $n \in \mathbb{N}$ tale che $n \geq n_0$. Se:

    1. $\mathcal{Q}$ è soddisfatta da n_0;
    2. per ogni $n \in \mathbb{N}$ tale che $n \geq n_0$, $\mathcal{Q}$ soddisfatta da ogni $m \in \mathbb{N}$ tale che $n_0 \leq m \leq n$ implica $\mathcal{Q}$ soddisfatta da $succ(n)$;

    allora $\mathcal{Q}$ è soddisfatta da ogni $n \in \mathbb{N}$ tale che $n \geq n_0$.

- Il principio di induzione fornisce dunque una condizione sufficiente per verificare se una proprietà è soddisfatta da *tutti* gli elementi di un insieme infinito numerabile dotato di una relazione d'ordine totale che ammette l'elemento minimo. La dimostrazione si suddivide in due parti. Nella prima parte (caso base), si verifica direttamente se la proprietà è soddisfatta dall'elemento minimo dell'insieme. Nella seconda parte (caso induttivo), si verifica se la proprietà è soddisfatta da un generico elemento dell'insieme maggiore del minimo, assumendo che la proprietà sia soddisfatta dall'elemento che lo precede o da tutti gli elementi che lo precedono (ipotesi induttiva).

- Esempi:
  - Dimostriamo che $\sum_{i=1}^{n} i = \frac{n\cdot(n+1)}{2}$ per ogni $n \in \mathbb{N}$ tale che $n \geq 1$ procedendo per induzione su n:
    - Sia $n = 1$. Risulta $\sum_{i=1}^{1} i = 1 = \frac{1\cdot(1+1)}{2}$ e quindi la proprietà è vera per $n = 1$.
    - Dato un arbitrario $n \in \mathbb{N}$ tale che $n \geq 1$, supponiamo che $\sum_{i=1}^{n} i = \frac{n\cdot(n+1)}{2}$. Risulta $\sum_{i=1}^{(n+1)} i = (n+1) + \sum_{i=1}^{n} i = (n+1) + \frac{n\cdot(n+1)}{2}$ per ipotesi induttiva. Poiché $(n+1) + \frac{n\cdot(n+1)}{2} = \frac{2\cdot(n+1)+n\cdot(n+1)}{2} = \frac{(n+1)\cdot(n+2)}{2}$, abbiamo $\sum_{i=1}^{(n+1)} i = \frac{(n+1)\cdot((n+1)+1)}{2}$ e quindi la proprietà è vera per $n+1$.
  - Dimostriamo che $|\mathcal{P}(A)| = 2^n$ per ogni insieme finito A tale che $|A| = n$ procedendo per induzione su $n \in \mathbb{N}$:
    - Sia $n = 0$. In questo caso $A = \emptyset$ e risulta $|\mathcal{P}(\emptyset)| = |\{\emptyset\}| = 1 = 2^0$, quindi la proprietà è vera per l'insieme finito di cardinalità $n = 0$.
    - Dato un arbitrario $n \in \mathbb{N}$, supponiamo che per ogni insieme A tale che $|A| = n$ risulti $|\mathcal{P}(A)| = 2^n$ e consideriamo un qualsiasi insieme A' di cardinalità $n+1$, che quindi possiamo scrivere come $A' = A'' \cup \{a\}$ con $a \notin A''$. Da $\mathcal{P}(A') = \{B \mid B \subseteq A''\} \cup \{B \cup \{a\} \mid B \subseteq A''\}$ segue che $|\mathcal{P}(A')| = 2 \cdot |\mathcal{P}(A'')| = 2 \cdot 2^n$ per ipotesi induttiva. Poiché $2 \cdot 2^n = 2^{(n+1)}$, abbiamo $|\mathcal{P}(A')| = 2^{(n+1)}$ e quindi la proprietà è vera per ogni insieme finito di cardinalità $n+1$. ■fltpl_3

Capitolo 9

Logica proposizionale

9.1 Sintassi della logica proposizionale

- La logica proposizionale è una logica a due valori (vero e falso) basata su un insieme di proposizioni e su un insieme di connettivi attraverso i quali è possibile combinare le proposizioni per creare formule arbitrariamente complesse. Per proposizione intendiamo un'affermazione relativa ad un singolo fatto di cui può essere stabilita la verità; di conseguenza sono escluse forme interrogative e dubitative così come affermazioni riguardanti il futuro oppure contenenti parti non completamente specificate.
- I valori di verità sono assimilabili a costanti logiche, le proposizioni sono assimilabili a variabili logiche, i connettivi sono assimilabili ad operatori logici e le formule sono assimilabili ad espressioni logiche.
- Gli elementi sintattici di base della logica proposizionale sono i seguenti:
  - Un insieme *Prop* di proposizioni, le quali verranno indicate con le lettere p, q, r.
  - I seguenti connettivi logici unari:
    - Negazione: $\neg$.
  - I seguenti connettivi logici binari:
    - Disgiunzione: $\vee$.
    - Congiunzione: $\wedge$.
    - Implicazione: $\rightarrow$.
    - Doppia implicazione (o biimplicazione o coimplicazione): $\leftrightarrow$.
  - I simboli ausiliari (e).
- I connettivi logici vanno letti nel seguente modo: $\neg$ corrisponde a "non", $\vee$ corrisponde ad "o" (nel senso inclusivo del "vel" latino), $\wedge$ corrisponde ad "e", $\rightarrow$ corrisponde a "se ... allora ...", $\leftrightarrow$ corrisponde a "se e solo se". L'uso delle parentesi nelle formule ha il solo scopo di rendere non ambigua la lettura di formule che hanno più connettivi.
- Algebricamente, l'insieme *FBF* delle formule ben formate della logica proposizionale è definito come il più piccolo insieme che include *Prop* ed è chiuso rispetto a $\neg$, $\vee$, $\wedge$, $\rightarrow$, $\leftrightarrow$. Alternativamente, *FBF* può essere definito come il più piccolo linguaggio su $Prop \cup \{\neg, \vee, \wedge, \rightarrow, \leftrightarrow, (,)\}$ tale che:
  - $Prop \subseteq FBF$.
  - $(\neg\phi) \in FBF$ per ogni $\phi \in FBF$.
  - $(\phi_1 \vee \phi_2) \in FBF$ per ogni $\phi_1, \phi_2 \in FBF$.
  - $(\phi_1 \wedge \phi_2) \in FBF$ per ogni $\phi_1, \phi_2 \in FBF$.
  - $(\phi_1 \rightarrow \phi_2) \in FBF$ per ogni $\phi_1, \phi_2 \in FBF$.
  - $(\phi_1 \leftrightarrow \phi_2) \in FBF$ per ogni $\phi_1, \phi_2 \in FBF$.

- Esempio: le formule $(p \wedge \vee q)$ e $\neg r$ non sono ben formate, mentre $(p \wedge q)$ e $(\neg r)$ lo sono.

- Teorema: Sia FBF' un linguaggio su $Prop \cup \{\neg, \vee, \wedge, \rightarrow, \leftrightarrow, (,)\}$ definito ponendo $FBF' = \bigcup_{n \in \mathbb{N}} FBF_n$ dove:

$$FBF_n = \begin{cases} Prop & \text{se } n = 0 \\ FBF_{n-1} \cup \{(\neg\phi) \mid \phi \in FBF_{n-1}\} \cup & \text{se } n > 0 \\ \quad \{(\phi_1 \vee \phi_2) \mid \phi_1, \phi_2 \in FBF_{n-1}\} \cup & \\ \quad \{(\phi_1 \wedge \phi_2) \mid \phi_1, \phi_2 \in FBF_{n-1}\} \cup & \\ \quad \{(\phi_1 \rightarrow \phi_2) \mid \phi_1, \phi_2 \in FBF_{n-1}\} \cup & \\ \quad \{(\phi_1 \leftrightarrow \phi_2) \mid \phi_1, \phi_2 \in FBF_{n-1}\} & \end{cases}$$

  Allora $FBF' = FBF$, cioè FBF ammette un'ulteriore caratterizzazione basata sulla massima profondità di annidamento dei connettivi che compaiono nelle sue formule.

- Dimostrazione: Poiché FBF' include $Prop$ ed è chiuso rispetto a $\neg$, $\vee$, $\wedge$, $\rightarrow$, $\leftrightarrow$ (a tal fine è essenziale che FBF_{n-1} faccia parte di FBF_n come si vede per i connettivi binari), dalla minimalità di FBF come insieme che include $Prop$ ed è chiuso rispetto a quegli operatori ricaviamo che $FBF \subseteq FBF'$.
  Sia $\phi \in FBF'$. Dalla definizione di FBF' segue che $\phi \in FBF_n$ per qualche $n \in \mathbb{N}$. Proviamo che da ciò segue che $\phi \in FBF$ procedendo per induzione su n:

  - Sia $n = 0$. In questo caso $\phi \in Prop$ e quindi $\phi \in FBF$.
  - Dato un arbitrario $n \in \mathbb{N}$, supponiamo che per ogni formula $\phi \in FBF_n$ risulti $\phi \in FBF$ e consideriamo una qualsiasi formula $\phi' \in FBF_{n+1}$:
    * Se $\phi' \in FBF_n$ allora $\phi' \in FBF$ per ipotesi induttiva.
    * Se $\phi' \notin FBF_n$ allora ϕ' è della forma $(\neg\phi'')$ con $\phi'' \in FBF_n$ oppure della forma $(\phi_1'' \vee \phi_2'')$, $(\phi_1'' \wedge \phi_2'')$, $(\phi_1'' \rightarrow \phi_2'')$ o $(\phi_1'' \leftrightarrow \phi_2'')$ con $\phi_1'', \phi_2'' \in FBF_n$. Dall'ipotesi induttiva segue che $\phi'' \in FBF$ nel caso della negazione e $\phi_1'', \phi_2'' \in FBF$ nel caso dei connettivi binari. Dalla chiusura di FBF rispetto a tutti connettivi logici segue che $\phi' \in FBF$.

  Di conseguenza $FBF_n \subseteq FBF$ per ogni $n \in \mathbb{N}$ e quindi $FBF' \subseteq FBF$. In conclusione $FBF' = FBF$.

- Date due formule $\phi, \gamma \in FBF$, diciamo che γ è una sottoformula di ϕ sse γ compare in ϕ. Più precisamente, l'insieme $sf(\phi)$ delle sottoformule di ϕ è definito per induzione sulla struttura sintattica di ϕ come segue:

$$sf(\phi) = \{\phi\} \cup \begin{cases} \emptyset & \text{se } \phi \in Prop \\ sf(\phi') & \text{se } \phi \text{ è della forma } (\neg\phi') \\ sf(\phi_1') \cup sf(\phi_2') & \text{se } \phi \text{ è della forma } (\phi_1' \vee \phi_2'),\ (\phi_1' \wedge \phi_2'),\ (\phi_1' \rightarrow \phi_2') \text{ o } (\phi_1' \leftrightarrow \phi_2') \end{cases}$$

- Data una formula $\phi \in FBF$, diciamo che essa è:
  - Atomica sse è una proposizione appartenente a $Prop$.
  - Composta con connettivo primario $\neg$ e sottoformula immediata ϕ' sse è della forma $(\neg\phi')$.
  - Composta con connettivo primario $\vee$ e sottoformule immediate ϕ_1', ϕ_2' sse è della forma $(\phi_1' \vee \phi_2')$.
  - Composta con connettivo primario $\wedge$ e sottoformule immediate ϕ_1', ϕ_2' sse è della forma $(\phi_1' \wedge \phi_2')$.
  - Composta con connettivo primario $\rightarrow$ e sottoformule immediate ϕ_1', ϕ_2' sse è della forma $(\phi_1' \rightarrow \phi_2')$.
  - Composta con connettivo primario $\leftrightarrow$ e sottoformule immediate ϕ_1', ϕ_2' sse è della forma $(\phi_1' \leftrightarrow \phi_2')$.

- Esempio: la formula ben formata ϕ data da $(((\neg p) \wedge (q \vee p)) \rightarrow r)$ ha $\rightarrow$ come connettivo primario, $((\neg p) \wedge (q \vee p))$ ed r come sottoformule immediate e $sf(\phi) = \{\phi, ((\neg p) \wedge (q \vee p)), r, (\neg p), (q \vee p), p, q\}$ come insieme di sottoformule.

- Teorema (di leggibilità univoca): Le formule ben formate non sono ambigue, cioè esiste un unico modo di leggere ogni $\phi \in FBF$.
- Dimostrazione: Segue dal fatto che, grazie all'uso sistematico delle parentesi, tutte le formule composte hanno un unico connettivo primario.
- L'uso sistematico delle parentesi può essere evitato introducendo delle regole di precedenza e associatività per i connettivi logici. È consuetudine assumere che $\neg$ abbia precedenza su $\wedge$, che $\wedge$ abbia precedenza su $\vee$, che $\vee$ abbia precedenza su $\rightarrow$, che $\rightarrow$ abbia precedenza su $\leftrightarrow$ e che tutti questi connettivi logici siano associativi da sinistra ad eccezione di $\neg$.
- Il principio di induzione per $\mathbb{N}$ può essere riformulato per FBF sulla base della struttura sintattica delle formule.
- Teorema (principio di induzione strutturale): Sia $\mathcal{Q}$ una proprietà definita su FBF. Se:
  1. $\mathcal{Q}$ è soddisfatta da ogni formula atomica di FBF;
  2. per ogni $\phi \in FBF$, $\mathcal{Q}$ soddisfatta da ϕ implica $\mathcal{Q}$ soddisfatta da $(\neg\phi)$;
  3. per ogni $\phi_1, \phi_2 \in FBF$, $\mathcal{Q}$ soddisfatta da ϕ_1 e $\mathcal{Q}$ soddisfatta da ϕ_2 implicano $\mathcal{Q}$ soddisfatta da $(\phi_1 \vee \phi_2)$;
  4. per ogni $\phi_1, \phi_2 \in FBF$, $\mathcal{Q}$ soddisfatta da ϕ_1 e $\mathcal{Q}$ soddisfatta da ϕ_2 implicano $\mathcal{Q}$ soddisfatta da $(\phi_1 \wedge \phi_2)$;
  5. per ogni $\phi_1, \phi_2 \in FBF$, $\mathcal{Q}$ soddisfatta da ϕ_1 e $\mathcal{Q}$ soddisfatta da ϕ_2 implicano $\mathcal{Q}$ soddisfatta da $(\phi_1 \rightarrow \phi_2)$;
  6. per ogni $\phi_1, \phi_2 \in FBF$, $\mathcal{Q}$ soddisfatta da ϕ_1 e $\mathcal{Q}$ soddisfatta da ϕ_2 implicano $\mathcal{Q}$ soddisfatta da $(\phi_1 \leftrightarrow \phi_2)$;

  allora $\mathcal{Q}$ è soddisfatta da ogni $\phi \in FBF$.
- Dimostrazione: Sia Φ l'insieme delle formule che soddisfano $\mathcal{Q}$. Ovviamente $\Phi \subseteq FBF$. D'altro canto Φ soddisfa le condizioni della definizione di FBF e quindi $FBF \subseteq \Phi$ per la minimalità di FBF come insieme che include $Prop$ ed è chiuso rispetto a tutti i connettivi. In conclusione $\Phi = FBF$. ■fltpl_4

9.2 Semantica e decidibilità della logica proposizionale

- Stabilita la sintassi della logica proposizionale, cioè la forma delle sue formule, è necessario definirne la semantica, cioè il significato delle sue formule, in termini di valori di verità. Osservato che il valore di verità di una formula non è un concetto assoluto ma dipende dai valori di verità delle proposizioni presenti in quella formula, introduciamo in modo informale il significato dei vari connettivi logici:
  - $(\neg\phi)$ è vera se ϕ è falsa, mentre è falsa se ϕ è vera.
  - $(\phi_1 \vee \phi_2)$ è vera se almeno una tra ϕ_1 e ϕ_2 è vera, altrimenti è falsa.
  - $(\phi_1 \wedge \phi_2)$ è vera se ϕ_1 e ϕ_2 sono entrambe vere, altrimenti è falsa.
  - $(\phi_1 \rightarrow \phi_2)$ è vera se ϕ_1 e ϕ_2 sono entrambe vere o entrambe false oppure ϕ_2 è vera e ϕ_1 è falsa, altrimenti è falsa. In altri termini, essa è vera se ϕ_1 è falsa oppure ϕ_2 è vera, altrimenti è falsa.
  - $(\phi_1 \leftrightarrow \phi_2)$ è vera se $(\phi_1 \rightarrow \phi_2)$ e $(\phi_2 \rightarrow \phi_1)$ sono entrambe vere, altrimenti è falsa. In altri termini, essa è vera se ϕ_1 e ϕ_2 sono entrambe vere o entrambe false, altrimenti è falsa.
- Al fine di comprendere meglio il significato del connettivo di implicazione, consideriamo la seguente affermazione condizionale: se domani c'è il sole, allora andremo al mare. Chi la pronuncia è un bugiardo – e quindi l'affermazione complessiva è falsa – solo nel caso in cui domani ci sia il sole e non si vada al mare. Qualora domani non ci sia il sole, la decisione di andare al mare oppure no non inficia la verità dell'intera affermazione (ex falso sequitur quodlibet). Il significato intuitivo delle affermazioni condizionali nel linguaggio naturale ci induce a leggere $(\phi_1 \rightarrow \phi_2)$ come il fatto che da ϕ_1 siamo in grado di concludere ϕ_2 *mediante un qualche ragionamento logico*. Questo non è corretto perché $\rightarrow$ è semplicemente un simbolo appartenente al linguaggio della logica proposizionale, mentre i meccanismi di inferenza appartengono ai sistemi deduttivi e quindi si collocano ad un livello metalinguistico.

- Per attribuire un valore di verità ad ogni proposizione, è necessaria una funzione da *Prop* a $\{vero, falso\}$. Equivalentemente, chiamiamo assegnamento di verità ciascun elemento A di 2^{Prop}, intendendo che una proposizione p è vera sse $p \in A$. Il valore di verità di una formula ben formata può quindi variare a seconda di quanto stabilito dallo specifico assegnamento di verità utilizzato per le proposizioni presenti nella formula stessa.
- La semantica della logica proposizionale viene formalizzata attraverso la relazione di soddisfacimento (o relazione di verità) $\models$ definita come il più piccolo sottoinsieme di $2^{Prop} \times FBF$ tale che:
  - $A \models p$ se $p \in A$.
  - $A \models (\neg\phi)$ se $A \not\models \phi$.
  - $A \models (\phi_1 \vee \phi_2)$ se $A \models \phi_1$ o $A \models \phi_2$.
  - $A \models (\phi_1 \wedge \phi_2)$ se $A \models \phi_1$ e $A \models \phi_2$.
  - $A \models (\phi_1 \rightarrow \phi_2)$ se $A \not\models \phi_1$ oppure $A \models \phi_2$.
  - $A \models (\phi_1 \leftrightarrow \phi_2)$ se $A \models \phi_1$ e $A \models \phi_2$, oppure $A \not\models \phi_1$ e $A \not\models \phi_2$.

  Quando $A \models \phi$ diciamo che A soddisfa ϕ oppure che A è un modello di ϕ.
- Esempi:
  - $\{p\} \models (p \vee q)$.
  - $\{p\} \not\models (p \wedge q)$.
  - $\{p, q\} \not\models ((p \rightarrow q) \rightarrow r)$.
  - $\emptyset \models (p \leftrightarrow q)$.
- Un modo alternativo di definire la semantica della logica proposizionale si basa sulle tabelle di verità. Potremmo chiamarlo la visione dell'ingegnere elettronico in quanto ogni formula ben formata viene vista come un circuito in cui le proposizioni sono i fili in ingresso, i connettivi logici sono le porte e gli assegnamenti di verità determinano il voltaggio lungo ogni filo di ingresso. Questa visione è interessante perché consente di evidenziare delle analogie tra connettivi logici e operazioni e relazioni aritmetiche.
- Indicato con 1 il valore di verità vero e con 0 il valore di verità falso, dato un assegnamento di verità $A \in 2^{Prop}$ definiamo una funzione di interpretazione $\mathcal{I}_A : FBF \rightarrow \{0, 1\}$ coerente con A – cioè tale che, per ogni $p \in Prop$, $\mathcal{I}_A(p) = 1$ sse $p \in A$ – per induzione sulla struttura sintattica delle formule ben formate attraverso la seguente tabella di verità del connettivo logico $\neg$:

$\mathcal{I}_A(\phi)$	$\mathcal{I}_A(\neg\phi)$
1	0
0	1

  e le seguenti tabelle di verità dei connettivi logici binari:

$\mathcal{I}_A(\phi_1)$	$\mathcal{I}_A(\phi_2)$	$\mathcal{I}_A(\phi_1 \vee \phi_2)$	$\mathcal{I}_A(\phi_1 \wedge \phi_2)$	$\mathcal{I}_A(\phi_1 \rightarrow \phi_2)$	$\mathcal{I}_A(\phi_1 \leftrightarrow \phi_2)$
1	1	1	1	1	1
1	0	1	0	0	0
0	1	1	0	1	0
0	0	0	0	1	1

- Osserviamo che per ogni $\phi, \phi_1, \phi_2 \in FBF$ vale che:
  - $\mathcal{I}_A(\neg\phi) = 1 - \mathcal{I}_A(\phi)$.
  - $\mathcal{I}_A(\phi_1 \vee \phi_2) = \max(\mathcal{I}_A(\phi_1), \mathcal{I}_A(\phi_2)) = \min(\mathcal{I}_A(\phi_1) + \mathcal{I}_A(\phi_2), 1)$.
  - $\mathcal{I}_A(\phi_1 \wedge \phi_2) = \min(\mathcal{I}_A(\phi_1), \mathcal{I}_A(\phi_2)) = \mathcal{I}_A(\phi_1) \cdot \mathcal{I}_A(\phi_2)$.
  - $\mathcal{I}_A(\phi_1 \rightarrow \phi_2) = 1$ sse $\mathcal{I}_A(\phi_1) \leq \mathcal{I}_A(\phi_2)$.
  - $\mathcal{I}_A(\phi_1 \leftrightarrow \phi_2) = 1$ sse $\mathcal{I}_A(\phi_1) = \mathcal{I}_A(\phi_2)$.

- Teorema: Per ogni $\phi \in FBF$ ed $A \in 2^{Prop}$ risulta $\mathcal{I}_A(\phi) = 1$ sse $A \models \phi$.
- Dimostrazione: Procediamo per induzione sulla struttura sintattica di ϕ:
  - Se $\phi \in Prop$ allora $\mathcal{I}_A(\phi) = 1$ sse $\phi \in A$ sfruttando la coerenza di $\mathcal{I}_A$ con A, cioè sse $A \models \phi$.
  - Sia ϕ della forma $(\neg\phi')$ e supponiamo che $\mathcal{I}_A(\phi')=1$ sse $A \models \phi'$. Allora $\mathcal{I}_A(\phi)=1$ sse $1-\mathcal{I}_A(\phi')=1$ sse $\mathcal{I}_A(\phi')=0$, cioè sse $A \not\models \phi'$ sfruttando l'ipotesi induttiva. Di conseguenza $\mathcal{I}_A(\phi)=1$ sse $A \models \phi$.
  - Sia ϕ della forma $(\phi'_1 \vee \phi'_2)$ e supponiamo che $\mathcal{I}_A(\phi'_i) = 1$ sse $A \models \phi'_i$ per $i \in \{1,2\}$. Allora $\mathcal{I}_A(\phi) = 1$ sse $\mathcal{I}_A(\phi'_1) = 1$ o $\mathcal{I}_A(\phi'_2) = 1$, cioè sse $A \models \phi'_1$ o $A \models \phi'_2$ sfruttando l'ipotesi induttiva. Di conseguenza $\mathcal{I}_A(\phi) = 1$ sse $A \models \phi$.
  - Sia ϕ della forma $(\phi'_1 \wedge \phi'_2)$ e supponiamo che $\mathcal{I}_A(\phi'_i) = 1$ sse $A \models \phi'_i$ per $i \in \{1,2\}$. Allora $\mathcal{I}_A(\phi) = 1$ sse $\mathcal{I}_A(\phi'_1) = 1$ e $\mathcal{I}_A(\phi'_2) = 1$, cioè sse $A \models \phi'_1$ e $A \models \phi'_2$ sfruttando l'ipotesi induttiva. Di conseguenza $\mathcal{I}_A(\phi) = 1$ sse $A \models \phi$.
  - Sia ϕ della forma $(\phi'_1 \rightarrow \phi'_2)$ e supponiamo che $\mathcal{I}_A(\phi'_i) = 1$ sse $A \models \phi'_i$ per $i \in \{1,2\}$. Allora $\mathcal{I}_A(\phi) = 1$ sse $\mathcal{I}_A(\phi'_1) = 0$ oppure $\mathcal{I}_A(\phi'_2) = 1$, cioè sse $A \not\models \phi'_1$ oppure $A \models \phi'_2$ sfruttando l'ipotesi induttiva. Di conseguenza $\mathcal{I}_A(\phi) = 1$ sse $A \models \phi$.
  - Sia ϕ della forma $(\phi'_1 \leftrightarrow \phi'_2)$ e supponiamo che $\mathcal{I}_A(\phi'_i) = 1$ sse $A \models \phi'_i$ per $i \in \{1,2\}$. Allora $\mathcal{I}_A(\phi) = 1$ sse $\mathcal{I}_A(\phi'_1) = 1$ e $\mathcal{I}_A(\phi'_2) = 1$ oppure $\mathcal{I}_A(\phi'_1) = 0$ e $\mathcal{I}_A(\phi'_2) = 0$, cioè sse $A \models \phi'_1$ e $A \models \phi'_2$ oppure $A \not\models \phi'_1$ e $A \not\models \phi'_2$ sfruttando l'ipotesi induttiva. Di conseguenza $\mathcal{I}_A(\phi)=1$ sse $A \models \phi$.
- Un altro modo alternativo di definire la semantica della logica proposizionale si basa sull'individuazione di tutti i possibili modelli per ogni formula ben formata. Potremmo chiamarlo la visione dell'ingegnere del software in quanto questi è interessato a descrivere tutti i possibili scenari in cui il sistema software che ha sviluppato funziona correttamente. Questa visione è interessante perché consente di evidenziare delle analogie tra connettivi logici e operazioni insiemistiche.
- La funzione $modelli : FBF \rightarrow 2^{2^{Prop}}$ è definita per induzione sulla struttura sintattica delle formule ben formate come segue (dove il complementare è preso rispetto a 2^{Prop}):
  - $modelli(p) = \{A \in 2^{Prop} \mid p \in A\}$.
  - $modelli(\neg\phi) = \complement(modelli(\phi))$.
  - $modelli(\phi_1 \vee \phi_2) = modelli(\phi_1) \cup modelli(\phi_2)$.
  - $modelli(\phi_1 \wedge \phi_2) = modelli(\phi_1) \cap modelli(\phi_2)$.
  - $modelli(\phi_1 \rightarrow \phi_2) = \complement(modelli(\phi_1)) \cup modelli(\phi_2)$.
  - $modelli(\phi_1 \leftrightarrow \phi_2) = (modelli(\phi_1) \cap modelli(\phi_2)) \cup (\complement(modelli(\phi_1)) \cap \complement(modelli(\phi_2)))$.
- Teorema: Per ogni $\phi \in FBF$ ed $A \in 2^{Prop}$ risulta $A \in modelli(\phi)$ sse $A \models \phi$.
- Dimostrazione: Procediamo per induzione sulla struttura sintattica di ϕ:
  - Se $\phi \in Prop$ allora $A \in modelli(\phi)$ sse $\phi \in A$, cioè sse $A \models \phi$.
  - Sia ϕ della forma $(\neg\phi')$ e supponiamo che $A \in modelli(\phi')$ sse $A \models \phi'$. Allora $A \in modelli(\phi) = 2^{Prop} \setminus modelli(\phi')$ sse $A \not\models \phi'$ sfruttando l'ipotesi induttiva. Di conseguenza $A \in modelli(\phi)$ sse $A \models \phi$.
  - Sia ϕ della forma $(\phi'_1 \vee \phi'_2)$ e supponiamo che $A \in modelli(\phi'_i)$ sse $A \models \phi'_i$ per $i \in \{1,2\}$. Allora $A \in modelli(\phi)$ sse $A \in modelli(\phi'_1)$ o $A \in modelli(\phi'_2)$, cioè sse $A \models \phi'_1$ o $A \models \phi'_2$ sfruttando l'ipotesi induttiva. Di conseguenza $A \in modelli(\phi)$ sse $A \models \phi$.
  - Sia ϕ della forma $(\phi'_1 \wedge \phi'_2)$ e supponiamo che $A \in modelli(\phi'_i)$ sse $A \models \phi'_i$ per $i \in \{1,2\}$. Allora $A \in modelli(\phi)$ sse $A \in modelli(\phi'_1)$ e $A \in modelli(\phi'_2)$, cioè sse $A \models \phi'_1$ e $A \models \phi'_2$ sfruttando l'ipotesi induttiva. Di conseguenza $A \in modelli(\phi)$ sse $A \models \phi$.
  - Sia ϕ della forma $(\phi'_1 \rightarrow \phi'_2)$ e supponiamo che $A \in modelli(\phi'_i)$ sse $A \models \phi'_i$ per $i \in \{1,2\}$. Allora $A \in modelli(\phi)$ sse $A \in 2^{Prop} \setminus modelli(\phi'_1)$ oppure $A \in modelli(\phi'_2)$, cioè sse $A \not\models \phi'_1$ oppure $A \models \phi'_2$ sfruttando l'ipotesi induttiva. Di conseguenza $A \in modelli(\phi)$ sse $A \models \phi$.
  - Sia ϕ della forma $(\phi'_1 \leftrightarrow \phi'_2)$ e supponiamo che $A \in modelli(\phi'_i)$ sse $A \models \phi'_i$ per $i \in \{1,2\}$. Allora $A \in modelli(\phi)$ sse $A \in modelli(\phi'_1)$ e $A \in modelli(\phi'_2)$ oppure $A \in 2^{Prop} \setminus modelli(\phi'_1)$ e $A \in 2^{Prop} \setminus modelli(\phi'_2)$, cioè sse $A \models \phi'_1$ e $A \models \phi'_2$ oppure $A \not\models \phi'_1$ e $A \not\models \phi'_2$ sfruttando l'ipotesi induttiva. Di conseguenza $A \in modelli(\phi)$ sse $A \models \phi$.

- Data una formula $\phi \in FBF$, il suo valore di verità non dipende da ciò che non sta nell'insieme $sfa(\phi)$ delle sue sottoformule atomiche, il quale è definito per induzione sulla struttura sintattica di ϕ come segue:
$$sfa(\phi) = \begin{cases} \{\phi\} & \text{se } \phi \in Prop \\ sfa(\phi') & \text{se } \phi \text{ è della forma } (\neg\phi') \\ sfa(\phi'_1) \cup sfa(\phi'_2) & \text{se } \phi \text{ è della forma } (\phi'_1 \vee \phi'_2),\ (\phi'_1 \wedge \phi'_2),\ (\phi'_1 \to \phi'_2) \text{ o } (\phi'_1 \leftrightarrow \phi'_2) \end{cases}$$

- Teorema: Siano $\phi \in FBF$ e $A_1, A_2 \in 2^{Prop}$ tali che per ogni $p \in sfa(\phi)$ risulta $p \in A_1$ sse $p \in A_2$. Allora $A_1 \models \phi$ sse $A_2 \models \phi$.

- Dimostrazione: Procediamo per induzione sulla struttura sintattica di ϕ:
  - Se $\phi \in Prop$ allora $\phi \in sfa(\phi)$ e quindi $\phi \in A_1$ sse $\phi \in A_2$. Perciò $A_1 \models \phi$ sse $A_2 \models \phi$.
  - Sia ϕ della forma $(\neg\phi')$ e supponiamo che $A_1 \models \phi'$ sse $A_2 \models \phi'$. Allora $A_1 \models \phi$ sse $A_1 \not\models \phi'$, cioè sse $A_2 \not\models \phi'$ sfruttando l'ipotesi induttiva, cioè sse $A_2 \models \phi$.
  - Sia ϕ della forma $(\phi'_1 \vee \phi'_2)$ e supponiamo che $A_1 \models \phi'_i$ sse $A_2 \models \phi'_i$ per $i \in \{1, 2\}$. Allora $A_1 \models \phi$ sse $A_1 \models \phi'_1$ o $A_1 \models \phi'_2$, cioè sse $A_2 \models \phi'_1$ o $A_2 \models \phi'_2$ sfruttando l'ipotesi induttiva, cioè sse $A_2 \models \phi$.
  - Sia ϕ della forma $(\phi'_1 \wedge \phi'_2)$ e supponiamo che $A_1 \models \phi'_i$ sse $A_2 \models \phi'_i$ per $i \in \{1, 2\}$. Allora $A_1 \models \phi$ sse $A_1 \models \phi'_1$ e $A_1 \models \phi'_2$, cioè sse $A_2 \models \phi'_1$ e $A_2 \models \phi'_2$ sfruttando l'ipotesi induttiva, cioè sse $A_2 \models \phi$.
  - Sia ϕ della forma $(\phi'_1 \to \phi'_2)$ e supponiamo che $A_1 \models \phi'_i$ sse $A_2 \models \phi'_i$ per $i \in \{1, 2\}$. Allora $A_1 \models \phi$ sse $A_1 \not\models \phi'_1$ oppure $A_1 \models \phi'_2$, cioè sse $A_2 \not\models \phi'_1$ e $A_2 \models \phi'_2$ sfruttando l'ipotesi induttiva, cioè sse $A_2 \models \phi$.
  - Sia ϕ della forma $(\phi'_1 \leftrightarrow \phi'_2)$ e supponiamo che $A_1 \models \phi'_i$ sse $A_2 \models \phi'_i$ per $i \in \{1, 2\}$. Allora $A_1 \models \phi$ sse $A_1 \models \phi'_1$ e $A_1 \models \phi'_2$ oppure $A_1 \not\models \phi'_1$ e $A_1 \not\models \phi'_2$, cioè sse $A_2 \models \phi'_1$ e $A_2 \models \phi'_2$ oppure $A_2 \not\models \phi'_1$ e $A_2 \not\models \phi'_2$ sfruttando l'ipotesi induttiva, cioè sse $A_2 \models \phi$.

- Data una formula $\phi \in FBF$, diciamo che essa è:
  - Soddisfacibile sse esiste $A \in 2^{Prop}$ tale che $A \models \phi$.
  - Una tautologia (o valida) sse $A \models \phi$ per ogni $A \in 2^{Prop}$.
  - Una contraddizione (o insoddisfacibile) sse $A \not\models \phi$ per ogni $A \in 2^{Prop}$.

- Teorema: Sia $\phi \in FBF$. Allora:
  - ϕ è una tautologia sse $(\neg\phi)$ è una contraddizione.
  - ϕ non è una tautologia sse $(\neg\phi)$ è soddisfacibile.
  - ϕ è soddisfacibile sse ϕ non è una contraddizione.

- Dimostrazione:
  - ϕ è una tautologia sse $A \models \phi$ per ogni $A \in 2^{Prop}$, cioè sse $A \not\models (\neg\phi)$ per ogni $A \in 2^{Prop}$, cioè sse $(\neg\phi)$ è una contraddizione.
  - ϕ non è una tautologia sse non è vero che $A \models \phi$ per ogni $A \in 2^{Prop}$, cioè sse esiste $A \in 2^{Prop}$ tale che $A \not\models \phi$, cioè sse esiste $A \in 2^{Prop}$ tale che $A \models (\neg\phi)$, cioè sse $(\neg\phi)$ è soddisfacibile.
  - ϕ è soddisfacibile sse esiste $A \in 2^{Prop}$ tale che $A \models \phi$, cioè sse non è vero che $A \not\models \phi$ per ogni $A \in 2^{Prop}$, cioè sse ϕ non è una contraddizione.

- Esempi di tautologie e contraddizioni (per semplicità omettiamo la maggior parte delle parentesi):

p	q	$\neg p$	$\neg q$	$p \to q$ $\neg p \vee q$ $\neg q \to \neg p$	$p \vee \neg p$	$q \wedge \neg q$	$(p \to q) \leftrightarrow (\neg p \vee q)$	$(p \to q) \leftrightarrow (\neg q \to \neg p)$
1	1	0	0	1	1	0	1	1
1	0	0	1	0	1	0	1	1
0	1	1	0	1	1	0	1	1
0	0	1	1	1	1	0	1	1

- Dati $\phi \in FBF$ e $A \in 2^{Prop}$, è possibile stabilire se $A \models \phi$ impiegando un tempo che cresce linearmente con il numero di occorrenze di proposizioni e connettivi logici presenti in ϕ.
- Data $\phi \in FBF$, un algoritmo per decidere se ϕ è soddisfacibile, una tautologia o una contraddizione consiste nel costruire la corrispondente tabella di verità e nel verificare se la sua ultima colonna contiene almeno un 1, tutti 1 o tutti 0, rispettivamente. Il tempo di esecuzione di questo algoritmo cresce esponenzialmente al crescere di $|sfa(\phi)|$ perché la tabella di verità ha $2^{|sfa(\phi)|}$ righe, che sono tante quante le interpretazioni coerenti con le classi di assegnamenti di verità che coincidono sull'insieme $sfa(\phi)$. Visto che molti problemi computazionali possono essere ricondotti a quello della soddisfacibilità, una delle più grandi questioni aperte nella teoria della complessità computazionale è quella di stabilire se esiste un algoritmo che, diversamente dal precedente, sia in grado di risolvere il problema della soddisfacibilità in tempo polinomiale. ■fltpl_5

9.3 Conseguenza ed equivalenza nella logica proposizionale

- Sulla base della semantica precedentemente definita, possiamo introdurre una relazione d'ordine ed una relazione d'equivalenza sull'insieme FBF. La seconda relazione è utile per stabilire quando due formule sintatticamente diverse hanno lo stesso valore di verità.
- Dato un insieme di formule $\Phi \in 2^{FBF}$, denotiamo con $modelli(\Phi) = \bigcap_{\phi \in \Phi} modelli(\phi)$ l'insieme degli assegnamenti di verità che soddisfano tutte le formule di Φ. Per $\Phi = \emptyset$, poniamo $modelli(\emptyset) = 2^{Prop}$.
- La relazione di soddisfacimento $\models$ può essere trasformata in una relazione su 2^{FBF} detta relazione di conseguenza logica ponendo $\Phi \models \Gamma$ (o $\Phi \Rightarrow \Gamma$) sse $modelli(\Phi) \subseteq modelli(\Gamma)$ per ogni $\Phi, \Gamma \in 2^{FBF}$. In altri termini, Γ è una conseguenza logica di Φ sse ogni assegnamento di verità che soddisfa tutte le formule di Φ soddisfa anche tutte le formule di Γ.
- Il fatto che Γ sia una conseguenza logica di Φ significa che, qualora ci si trovi in una condizione che soddisfa tutte le formule di Φ, allora anche tutte le formule di Γ sono soddisfatte in quella condizione senza doverlo verificare. In particolare, $\models \Gamma$ significa che tutte le formule di Γ sono delle tautologie.
- Osserviamo che la relazione di conseguenza logica è riflessiva e transitiva – perché tale è la relazione di inclusione insiemistica – ma non è una relazione d'ordine parziale. Ad esempio $\{(p \to q)\} \neq \{((\neg p) \vee q)\}$ ma $modelli(\{(p \to q)\}) = \{A \in 2^{Prop} \mid p \notin A \text{ oppure } q \in A\} = modelli(\{((\neg p) \vee q)\})$. Inoltre la relazione $\models$ (o $\Rightarrow$) è strettamente legata al connettivo $\to$.
- Teorema (di conseguenza logica): Dato $n \in \mathbb{N}$, siano $\Phi = \{\phi_1, \ldots, \phi_n\} \in 2^{FBF}$ e $\gamma \in FBF$. Allora $\Phi \models \{\gamma\}$ sse $((\bigwedge_{i=1}^{n} \phi_i) \to \gamma)$ è una tautologia, ovvero sse $((\bigwedge_{i=1}^{n} \phi_i) \wedge (\neg\gamma))$ è una contraddizione.
- Dimostrazione: In primo luogo proviamo che da $\Phi \models \{\gamma\}$ segue che $((\bigwedge_{i=1}^{n} \phi_i) \to \gamma)$ è una tautologia. Supponiamo che esista $A \in 2^{Prop}$ tale che $A \not\models ((\bigwedge_{i=1}^{n} \phi_i) \to \gamma)$. Ciò significa che $A \models (\bigwedge_{i=1}^{n} \phi_i)$ e $A \not\models \gamma$. Dunque esiste $A \in 2^{Prop}$ tale che $A \in modelli(\Phi)$ e $A \notin modelli(\gamma)$, che contraddice l'ipotesi $\Phi \models \{\gamma\}$. In secondo luogo proviamo che dal fatto che $((\bigwedge_{i=1}^{n} \phi_i) \to \gamma)$ sia una tautologia segue che $((\bigwedge_{i=1}^{n} \phi_i) \wedge (\neg\gamma))$ è una contraddizione. Supponiamo che esista $A \in 2^{Prop}$ tale che $A \models ((\bigwedge_{i=1}^{n} \phi_i) \wedge (\neg\gamma))$. Ciò significa che $A \models (\bigwedge_{i=1}^{n} \phi_i)$ e $A \not\models \gamma$. Dunque esiste $A \in 2^{Prop}$ tale che $A \not\models ((\bigwedge_{i=1}^{n} \phi_i) \to \gamma)$, che contraddice l'ipotesi secondo cui $((\bigwedge_{i=1}^{n} \phi_i) \to \gamma)$ è una tautologia.

  In terzo luogo proviamo che dal fatto che $((\bigwedge_{i=1}^{n} \phi_i) \wedge (\neg\gamma))$ sia una contraddizione segue che $\Phi \models \{\gamma\}$. Supponiamo che esista $A \in 2^{Prop}$ tale che $A \in modelli(\Phi)$ e $A \notin modelli(\gamma)$. Ciò significa che $A \models (\bigwedge_{i=1}^{n} \phi_i)$ e $A \not\models \gamma$. Dunque esiste $A \in 2^{Prop}$ tale che $A \models ((\bigwedge_{i=1}^{n} \phi_i) \wedge (\neg\gamma))$, che contraddice l'ipotesi secondo cui $((\bigwedge_{i=1}^{n} \phi_i) \wedge (\neg\gamma))$ è una contraddizione.
- Corollario: Siano $\phi, \gamma \in FBF$. Allora $\{\phi\} \models \{\gamma\}$ sse $\models \{(\phi \to \gamma)\}$.

- La relazione di equivalenza logica è una relazione su 2^{FBF} definita ponendo $\Phi \equiv \Gamma$ (o $\Phi \Leftrightarrow \Gamma$) sse $\Phi \models \Gamma$ e $\Gamma \models \Phi$. In altri termini, Φ e Γ sono logicamente equivalenti sse $modelli(\Phi) = modelli(\Gamma)$. Questa relazione è una relazione d'equivalenza, cioè è riflessiva, simmetrica e transitiva. Inoltre la relazione $\equiv$ (o $\Leftrightarrow$) è strettamente legata al connettivo $\leftrightarrow$.
- Teorema (di equivalenza logica): Dato $n \in \mathbb{N}$, siano $\Phi = \{\phi_1, \ldots, \phi_n\} \in 2^{FBF}$ e $\gamma \in FBF$. Allora $\Phi \equiv \{\gamma\}$ sse $((\bigwedge_{i=1}^{n} \phi_i) \leftrightarrow \gamma)$ è una tautologia.
- Dimostrazione: $\Phi \equiv \{\gamma\}$ sse $modelli(\Phi) = modelli(\{\gamma\})$, cioè sse per ogni $A \in 2^{Prop}$ risulta $A \models (\bigwedge_{i=1}^{n} \phi_i)$ e $A \models \gamma$ oppure $A \not\models (\bigwedge_{i=1}^{n} \phi_i)$ e $A \not\models \gamma$, cioè sse per ogni $A \in 2^{Prop}$ risulta $A \models ((\bigwedge_{i=1}^{n} \phi_i) \leftrightarrow \gamma)$, cioè sse $((\bigwedge_{i=1}^{n} \phi_i) \leftrightarrow \gamma)$ è una tautologia.
- Corollario: Siano $\phi, \gamma \in FBF$. Allora $\{\phi\} \equiv \{\gamma\}$ sse $\models \{(\phi \leftrightarrow \gamma)\}$.
- Il corollario al teorema di equivalenza logica fornisce una caratterizzazione alternativa di $\{\phi\} \equiv \{\gamma\}$ che è utile dal punto di vista computazionale, nel senso che invece di costruire $modelli(\{\phi\})$ e $modelli(\{\gamma\})$, che potrebbero essere insiemi infiniti, basta verificare che $(\phi \leftrightarrow \gamma)$ sia una tautologia attraverso la costruzione della sua tabella di verità.
- La relazione di equivalenza logica ristretta ad FBF, cioè ad insiemi singoletto (che verranno da ora in poi denotati senza parentesi graffe), risulta essere una congruenza rispetto a tutti i connettivi logici. Questo è molto importante perché consente di manipolare composizionalmente le formule. In altri termini, data una formula ben formata, qualora una sua sottoformula venga sostituita con un'altra formula ad essa logicamente equivalente, la formula ben formata risultante è logicamente equivalente a quella originaria. Dunque la sostituzione di sottoformule con formule logicamente equivalenti ad esse non altera il valore di verità della formula ben formata originaria.
- Teorema (di sostituzione): Siano $\phi_1, \phi_2 \in FBF$. Se $\phi_1 \equiv \phi_2$ allora:
  - $(\neg\phi_1) \equiv (\neg\phi_2)$.
  - $(\phi_1 \vee \gamma) \equiv (\phi_2 \vee \gamma)$ e $(\gamma \vee \phi_1) \equiv (\gamma \vee \phi_2)$ per ogni $\gamma \in FBF$.
  - $(\phi_1 \wedge \gamma) \equiv (\phi_2 \wedge \gamma)$ e $(\gamma \wedge \phi_1) \equiv (\gamma \wedge \phi_2)$ per ogni $\gamma \in FBF$.
  - $(\phi_1 \rightarrow \gamma) \equiv (\phi_2 \rightarrow \gamma)$ e $(\gamma \rightarrow \phi_1) \equiv (\gamma \rightarrow \phi_2)$ per ogni $\gamma \in FBF$.
  - $(\phi_1 \leftrightarrow \gamma) \equiv (\phi_2 \leftrightarrow \gamma)$ e $(\gamma \leftrightarrow \phi_1) \equiv (\gamma \leftrightarrow \phi_2)$ per ogni $\gamma \in FBF$.
- Dimostrazione: Siano $\phi_1, \phi_2 \in FBF$ tali che $\phi_1 \equiv \phi_2$ e quindi $modelli(\phi_1) = modelli(\phi_2)$:
  - $(\neg\phi_1) \equiv (\neg\phi_2)$ segue da $modelli(\neg\phi_1)) = 2^{Prop} \setminus modelli(\phi_1) = 2^{Prop} \setminus modelli(\phi_2) = modelli(\neg\phi_2))$.
  - $(\phi_1 \vee \gamma) \equiv (\phi_2 \vee \gamma)$ segue da $modelli(\phi_1 \vee \gamma) = modelli(\phi_1) \cup modelli(\gamma) = modelli(\phi_2) \cup modelli(\gamma) = modelli(\phi_2 \vee \gamma)$. La dimostrazione di $(\gamma \vee \phi_1) \equiv (\gamma \vee \phi_2)$ è simile.
  - $(\phi_1 \wedge \gamma) \equiv (\phi_2 \wedge \gamma)$ segue da $modelli(\phi_1 \wedge \gamma) = modelli(\phi_1) \cap modelli(\gamma) = modelli(\phi_2) \cap modelli(\gamma) = modelli(\phi_2 \wedge \gamma)$. La dimostrazione di $(\gamma \wedge \phi_1) \equiv (\gamma \wedge \phi_2)$ è simile.
  - $(\phi_1 \rightarrow \gamma) \equiv (\phi_2 \rightarrow \gamma)$ segue da $modelli(\phi_1 \rightarrow \gamma) = (2^{Prop} \setminus modelli(\phi_1)) \cup modelli(\gamma) = (2^{Prop} \setminus modelli(\phi_2)) \cup modelli(\gamma) = modelli(\phi_2 \rightarrow \gamma)$. La dimostrazione di $(\gamma \rightarrow \phi_1) \equiv (\gamma \rightarrow \phi_2)$ è simile.
  - $(\phi_1 \leftrightarrow \gamma) \equiv (\phi_2 \leftrightarrow \gamma)$ segue da $modelli(\phi_1 \leftrightarrow \gamma) = (modelli(\phi_1) \cap modelli(\gamma)) \cup ((2^{Prop} \setminus modelli(\phi_1)) \cap (2^{Prop} \setminus modelli(\gamma))) = (modelli(\phi_2) \cap modelli(\gamma)) \cup ((2^{Prop} \setminus modelli(\phi_2)) \cap (2^{Prop} \setminus modelli(\gamma))) = modelli(\phi_2 \leftrightarrow \gamma)$. La dimostrazione di $(\gamma \leftrightarrow \phi_1) \equiv (\gamma \leftrightarrow \phi_2)$ è simile.
- L'equivalenza logica $\equiv$ è un concetto semantico che non va confuso con l'uguaglianza sintattica $=$. Formule ben formate sintatticamente diverse possono essere logicamente equivalenti, come ad esempio $(p \rightarrow q)$, $(\neg p \vee q)$ e $(\neg q \rightarrow \neg p)$.

9.4 Proprietà algebriche dei connettivi logici

- Teorema: Siano $\mathbb{O}$ una qualsiasi contraddizione ed $\mathbb{I}$ una qualsiasi tautologia. La struttura algebrica $(FBF, \vee, \wedge, \neg, \mathbb{O}, \mathbb{I})$ è un reticolo booleano, cioè soddisfa le seguenti leggi:
  - Commutatività:
    - $(\phi \vee \gamma) \equiv (\gamma \vee \phi)$.
    - $(\phi \wedge \gamma) \equiv (\gamma \wedge \phi)$.
  - Associatività:
    - $((\phi \vee \gamma) \vee \delta) \equiv (\phi \vee (\gamma \vee \delta))$.
    - $((\phi \wedge \gamma) \wedge \delta) \equiv (\phi \wedge (\gamma \wedge \delta))$.
  - Assorbimento:
    - $(\phi \vee (\phi \wedge \gamma)) \equiv \phi$.
    - $(\phi \wedge (\phi \vee \gamma)) \equiv \phi$.
  - Distributività:
    - $(\phi \vee (\gamma \wedge \delta)) \equiv ((\phi \vee \gamma) \wedge (\phi \vee \delta))$.
    - $(\phi \wedge (\gamma \vee \delta)) \equiv ((\phi \wedge \gamma) \vee (\phi \wedge \delta))$.
  - Elemento neutro:
    - $(\phi \vee \mathbb{O}) \equiv \phi$.
    - $(\phi \wedge \mathbb{I}) \equiv \phi$.
  - Elemento inverso:
    - $(\phi \vee (\neg\phi)) \equiv \mathbb{I}$.
    - $(\phi \wedge (\neg\phi)) \equiv \mathbb{O}$.
- Dimostrazione: In virtù del corollario al teorema di equivalenza logica, per ogni legge è sufficiente verificare che la biimplicazione tra la formula di sinistra e la formula di destra è una tautologia costruendo la corrispondente tabella di verità.
- Teorema: La struttura algebrica $(FBF, \vee, \wedge, \neg, \mathbb{O}, \mathbb{I})$ possiede inoltre le seguenti proprietà derivate:
  - Elemento assorbente:
    - $(\phi \vee \mathbb{I}) \equiv \mathbb{I}$.
    - $(\phi \wedge \mathbb{O}) \equiv \mathbb{O}$.
  - Idempotenza:
    - $(\phi \vee \phi) \equiv \phi$.
    - $(\phi \wedge \phi) \equiv \phi$.
  - Doppia inversione:
    - $(\neg(\neg\phi)) \equiv \phi$.
  - Leggi di De Morgan:
    - $(\neg(\phi \vee \gamma)) \equiv ((\neg\phi) \wedge (\neg\gamma))$.
    - $(\neg(\phi \wedge \gamma)) \equiv ((\neg\phi) \vee (\neg\gamma))$.
- Dimostrazione: In virtù del corollario al teorema di equivalenza logica, per ogni proprietà è sufficiente verificare che la biimplicazione tra la formula di sinistra e la formula di destra è una tautologia costruendo la corrispondente tabella di verità.

- Poiché $(\phi_1 \vee \phi_2)$ è falsa sse ϕ_1 e ϕ_2 sono entrambe false mentre $(\phi_1 \wedge \phi_2)$ è vera sse ϕ_1 e ϕ_2 sono entrambe vere, i due connettivi logici binari del reticolo booleano sono operazioni duali, cioè sono derivabili l'uno dall'altro scambiando i ruoli tra vero e falso. Peraltro, dalle leggi di De Morgan deriviamo che $(\phi \vee \gamma) \equiv (\neg((\neg\phi) \wedge (\neg\gamma)))$ e $(\phi \wedge \gamma) \equiv (\neg((\neg\phi) \vee (\neg\gamma)))$.
- In generale, dati un insieme C di connettivi logici ed un connettivo logico $\odot \notin C$, diciamo che $\odot$ è semanticamente derivabile da C sse ogni formula ben formata di cui $\odot$ è il connettivo primario è equivalente ad una formula ben formata i cui connettivi logici appartengono tutti a C.
- Un insieme C di connettivi logici è funzionalmente completo sse ogni altro possibile connettivo logico è semanticamente derivabile da C. Oltre ai quattro connettivi logici binari, è possibile introdurne altri, stabilire delle leggi di derivabilità e individuare degli insiemi funzionalmente completi.
- Presentiamo le tabelle di verità di tre ulteriori connettivi logici binari chiamati xor $\veebar$ ("o ... o ..." come l'"aut" latino), nor $\downarrow$ ("né ... né ...") e nand $\uparrow$ ("... o ... ma non entrambi"):

$\mathcal{I}_A(\phi_1)$	$\mathcal{I}_A(\phi_2)$	$\mathcal{I}_A(\phi_1 \veebar \phi_2)$	$\mathcal{I}_A(\phi_1 \downarrow \phi_2)$	$\mathcal{I}_A(\phi_1 \uparrow \phi_2)$
1	1	0	0	0
1	0	1	0	1
0	1	1	0	1
0	0	0	1	1

- Teorema: Valgono le seguenti leggi di derivabilità semantica per i connettivi logici diversi da $\vee$, $\wedge$, $\neg$:
  - $(\phi \to \gamma) \equiv ((\neg\phi) \vee \gamma)$.
  - $(\phi \leftrightarrow \gamma) \equiv ((\phi \to \gamma) \wedge (\gamma \to \phi))$.
  - $(\phi \leftrightarrow \gamma) \equiv ((\phi \wedge \gamma) \vee ((\neg\phi) \wedge (\neg\gamma)))$.
  - $(\phi \leftrightarrow \gamma) \equiv (\neg(\phi \veebar \gamma))$.
  - $(\phi \veebar \gamma) \equiv (\neg(\phi \leftrightarrow \gamma))$.
  - $(\phi \veebar \gamma) \equiv ((\phi \wedge (\neg\gamma)) \vee ((\neg\phi) \wedge \gamma))$.
  - $(\phi \downarrow \gamma) \equiv (\neg(\phi \vee \gamma))$.
  - $(\phi \downarrow \gamma) \equiv ((\neg\phi) \wedge (\neg\gamma))$.
  - $(\phi \uparrow \gamma) \equiv (\neg(\phi \wedge \gamma))$.
  - $(\phi \uparrow \gamma) \equiv ((\neg\phi) \vee (\neg\gamma))$.

  Inoltre valgono le seguenti leggi di derivabilità semantica per i connettivi logici $\vee$, $\wedge$, $\neg$:
  - $(\phi \vee \gamma) \equiv (\neg((\neg\phi) \wedge (\neg\gamma)))$.
  - $(\phi \vee \gamma) \equiv ((\neg\phi) \to \gamma)$.
  - $(\phi \vee \gamma) \equiv (\neg(\phi \downarrow \gamma)) \equiv ((\phi \downarrow \gamma) \downarrow (\phi \downarrow \gamma))$.
  - $(\phi \vee \gamma) \equiv ((\neg\phi) \uparrow (\neg\gamma)) \equiv ((\phi \uparrow \phi) \uparrow (\gamma \uparrow \gamma))$.
  - $(\phi \wedge \gamma) \equiv (\neg((\neg\phi) \vee (\neg\gamma)))$.
  - $(\phi \wedge \gamma) \equiv ((((\phi \to \mathbb{0}) \to \mathbb{0}) \to (\gamma \to \mathbb{0})) \to \mathbb{0})$.
  - $(\phi \wedge \gamma) \equiv ((\neg\phi) \downarrow (\neg\gamma)) \equiv ((\phi \downarrow \phi) \downarrow (\gamma \downarrow \gamma))$.
  - $(\phi \wedge \gamma) \equiv (\neg(\phi \uparrow \gamma)) \equiv ((\phi \uparrow \gamma) \uparrow (\phi \uparrow \gamma))$.
  - $(\neg\phi) \equiv (\phi \to \mathbb{0})$.
  - $(\neg\phi) \equiv (\phi \downarrow \phi)$.
  - $(\neg\phi) \equiv (\phi \uparrow \phi)$.
- Dimostrazione: In virtù del corollario al teorema di equivalenza logica, per ogni legge è sufficiente verificare che la biimplicazione tra la formula di sinistra e la formula di destra è una tautologia costruendo la corrispondente tabella di verità.

- Corollario: I seguenti insiemi di connettivi logici sono funzionalmente completi:
  - $\{\vee, \wedge, \neg\}$.
  - $\{\vee, \neg\}$.
  - $\{\wedge, \neg\}$.
  - $\{\downarrow\}$.
  - $\{\uparrow\}$. ■fltpl_6

9.5 Sistemi deduttivi per la logica proposizionale

- La logica matematica si fonda su un livello linguistico ed un livello metalinguistico. Il primo livello è formato da una sintassi che include determinati connettivi logici e da una semantica attraverso cui viene attribuito un valore di verità ad ogni formula. Il secondo livello comprende meccanismi di ragionamento per stabilire la verità di una formula deducendola da altre formule che sono state poste essere vere o dedotte essere vere, *attraverso pure manipolazioni simboliche*. Qual è il legame tra i due livelli?
- I connettivi logici che danno luogo alla sintassi della logica proposizionale sono funzionalmente completi e concorrono a formare gli enunciati dei teoremi della matematica. Tali teoremi sono di solito espressi nella forma $(\phi \to \gamma)$ oppure $(\phi \leftrightarrow \gamma)$ e rappresentano delle tautologie, cioè $\phi \Rightarrow \gamma$ oppure $\phi \Leftrightarrow \gamma$.
- Dato un teorema della forma $(\phi \to \gamma)$, diciamo che:
  - ϕ è condizione sufficiente per γ, nel senso che affinché γ sia vera è sufficiente che ϕ sia vera; se ϕ è falsa non possiamo concludere nulla su γ.
  - γ è condizione necessaria per ϕ, nel senso che – sfruttando la proprietà $(\phi \to \gamma) \equiv ((\neg\gamma) \to (\neg\phi))$ – affinché ϕ sia vera è necessario che γ sia vera; se γ è falsa sicuramente anche ϕ è falsa altrimenti il teorema non vale.
- Dato un teorema della forma $(\phi \leftrightarrow \gamma)$, diciamo che γ è condizione necessaria e sufficiente per ϕ, perché basta che una delle due formule sia vera (rispettivamente falsa) per concludere che anche l'altra lo è.
- Per dimostrare un teorema della matematica di solito non si ragiona in termini semantici mediante funzioni di interpretazione basate su tabelle di verità o insiemi di modelli, ma si procede per deduzione mediante l'applicazione di regole di inferenza a risultati che sono già noti.
- Un teorema della forma $(\phi \to \gamma)$ può essere dimostrato:
  - In modo diretto, cioè assumendo che ϕ sia vera e cercando di derivare che anche γ è vera; se ϕ fosse falsa allora $(\phi \to \gamma)$ sarebbe banalmente vera.
  - In modo indiretto attraverso l'equivalente enunciato contronominale della forma $((\neg\gamma) \to (\neg\phi))$, cioè assumendo che $(\neg\gamma)$ sia vera e cercando di derivare che anche $(\neg\phi)$ è vera.
  - Per assurdo, cioè assumendo che ϕ e $(\neg\gamma)$ siano entrambe vere e cercando di derivare una contraddizione; si noti che $(\phi \wedge (\neg\gamma)) \equiv (\neg((\neg\phi) \vee \gamma)) \equiv (\neg(\phi \to \gamma))$.

  Un teorema della forma $(\phi \leftrightarrow \gamma)$ viene dimostrato derivando che $(\phi \to \gamma)$ e $(\gamma \to \phi)$ sono vere, dato che $(\phi \leftrightarrow \gamma) \equiv ((\phi \to \gamma) \wedge (\gamma \to \phi))$.
- Esempio: Per dimostrare "se $(p \to q)$ e $(p \to (q \to r))$ allora $(p \to r)$" si può costruire la tabella di verità di $(((p \to q) \wedge (p \to (q \to r))) \to (p \to r))$ e verificare che essa sia una tautologia. Tuttavia, in un testo di matematica troveremmo una dimostrazione come la seguente. Supponiamo che $(p \to q)$ e $(p \to (q \to r))$ siano vere. Concentrandoci su $(p \to r)$, assumiamo che p sia vera. Poiché p è vera e $(p \to q)$ è vera, anche q deve essere vera. Poiché p e q sono vere e $(p \to (q \to r))$ è vera, anche r deve essere vera. Dunque $(p \to r)$ è vera. Di conseguenza $(((p \to q) \wedge (p \to (q \to r))) \to (p \to r))$ è vera.
- Quando si esamina la validità di una formula ben formata, invece di considerare la relazione semantica di conseguenza logica $\models$ propria della teoria dei modelli per stabilire se la formula è una tautologia, si può dunque considerare una relazione sintattica di derivabilità logica $\vdash$ propria della teoria delle dimostrazioni per stabilire se la formula è un teorema.

- Un sistema deduttivo (o sistema di dimostrazione o sistema formale) è un insieme $\Delta \subseteq FBF^*$ di sequenze non vuote di lunghezza finita di formule ben formate, in cui:
  - Ogni sequenza $\phi \in \Delta$ di lunghezza uno è detta assioma ed è rappresentata come segue:
    $$\frac{}{\phi}$$
  - Ogni sequenza $\phi_1 \dots \phi_k \phi \in \Delta$ con $k \geq 1$ è detta regola di inferenza ed è rappresentata come segue:
    $$\frac{\phi_1 \dots \phi_k}{\phi}$$
    dove $\phi_1, \dots, \phi_k$ sono le premesse mentre ϕ è la conclusione ottenuta per deduzione dalle premesse.
- Sia $\Delta \subseteq FBF^*$ un sistema deduttivo. Diciamo che $\phi \in FBF$ è un teorema in Δ, scritto $\vdash_\Delta \phi$, sse esiste una sequenza non vuota di lunghezza finita di formule ben formate $\gamma_1 \dots \gamma_n \in FBF^*$ tale che:
  - Ogni formula γ_i della sequenza:
    - o è un assioma di Δ, cioè $\gamma_i \in \Delta$;
    - o è la conclusione di una regola di inferenza di Δ applicata a premesse costituite da alcune delle formule che appartengono a $\{\gamma_1, \dots, \gamma_{i-1}\}$.
  - L'ultima formula γ_n della sequenza è ϕ.

  In questo caso la sequenza è detta essere una dimostrazione (o prova) di ϕ in Δ. In generale, dato $\Phi \in 2^{FBF}$, scriviamo $\Phi \vdash_\Delta \phi$ per indicare che le formule della sequenza possono appartenere a Φ; diciamo che Φ è una teoria in Δ sse $\Phi \vdash_\Delta \phi$ implica $\phi \in \Phi$ per ogni $\phi \in FBF$.
- Dato un sistema deduttivo $\Delta \subseteq FBF^*$, diciamo che esso è:
  - Corretto rispetto alla logica considerata sse per ogni $\phi \in FBF$ da $\vdash_\Delta \phi$ segue che $\models \phi$, cioè ogni teorema in Δ corrisponde ad una tautologia della logica considerata.
  - Completo rispetto alla logica considerata sse per ogni $\phi \in FBF$ da $\models \phi$ segue che $\vdash_\Delta \phi$, cioè ogni tautologia della logica considerata è derivabile come teorema in Δ.
  - Decidibile sse è possibile stabilire in tempo finito se un'arbitraria formula ben formata della logica considerata è un teorema in Δ oppure no.
- Il sistema deduttivo sviluppato da Gentzen sotto il nome di deduzione naturale si basa sui seguenti tre gruppi di regole di inferenza il cui significato è molto intuitivo:
  - Regole elementari che introducono connettivi logici rispetto alle premesse:
    $$\frac{p}{(p \vee q)} \qquad \frac{q}{(p \vee q)} \qquad \frac{p \quad q}{(p \wedge q)}$$
  - Regole elementari che eliminano connettivi logici rispetto alle premesse:
    $$\frac{(p \wedge q)}{p} \qquad \frac{(p \wedge q)}{q} \qquad \frac{p \quad (p \to q)}{q} \qquad \frac{(\neg q) \quad (p \to q)}{(\neg p)} \qquad \frac{p \quad (\neg p)}{\mathbb{O}} \qquad \frac{\mathbb{O}}{p}$$
    La terza regola è detta modus ponens, la quarta regola è detta modus tollens e la sesta regola formalizza il principio secondo cui ex falso sequitur quodlibet.
  - Regole condizionali in cui alcune premesse dipendono da ipotesi racchiuse tra parentesi quadre:
    $$\frac{[p]r \quad [q]r \quad (p \vee q)}{r} \qquad \frac{[p]q}{(p \to q)} \qquad \frac{[p]\mathbb{O}}{(\neg p)} \qquad \frac{[(\neg p)]\mathbb{O}}{p}$$
    La prima regola esprime il fatto che da $(p \vee q)$ non si possa inferire né p né q singolarmente ma ogni proposizione r valida tanto sotto p quanto sotto q, mentre la quarta regola è detta reductio ad absurdum.

- Nella deduzione naturale le dimostrazioni prendono la forma di alberi. Il motivo è che, ogni volta che la conclusione di una regola coincide con la premessa di un'altra, le due regole possono essere composte. Ciò deriva dal principio, detta regola di taglio, secondo cui se p deriva da un insieme di ipotesi Φ e se da $\Phi \cup \{p\}$ si deriva q, allora la composizione delle due dimostrazioni è una dimostrazione che q deriva da Φ. In altri termini, per dimostrare un teorema complesso (q) a partire da una serie di ipotesi (Φ), si può cominciare derivando dei lemmi più semplici (p) da quelle stesse ipotesi.

- Esempio: Il seguente albero rappresenta una dimostrazione di $\{(p \wedge q), ((q \wedge p) \to r)\} \vdash_{\mathrm{DNG}} (r \vee s)$:

$$\dfrac{\dfrac{\dfrac{(p \wedge q)}{q} \quad \dfrac{(p \wedge q)}{p}}{(q \wedge p)} \qquad ((q \wedge p) \to r)}{\dfrac{r}{(r \vee s)}}$$

- Teorema: Il sistema di deduzione naturale di Gentzen è corretto e completo rispetto alla logica proposizionale.

- I sistemi deduttivi alla Hilbert accettano come unica regola di inferenza il modus ponens (essendo quella più utile per dimostrare i teoremi della matematica):

$$\dfrac{p \quad (p \to q)}{q}$$

  e sostituiscono le altre regole della deduzione naturale con oppurtuni assiomi che contengono l'implicazione come unico connettivo binario (coerentemente con l'adozione del modus ponens).

- Osservato che la seconda regola condizionale della deduzione naturale è derivabile dai seguenti due assiomi:

$$\dfrac{}{(p \to (q \to p))} \qquad \dfrac{}{((p \to (q \to r)) \to ((p \to q) \to (p \to r)))}$$

  tutte le altre regole di inferenza della deduzione naturale possono essere derivate dai seguenti assiomi in cui le premesse delle regole originarie sono codificate attraverso il connettivo di implicazione:

$$\dfrac{}{(p \to (p \vee q))} \qquad \dfrac{}{(q \to (p \vee q))} \qquad \dfrac{}{(p \to (q \to (p \wedge q)))}$$

$$\dfrac{}{((p \wedge q) \to p)} \qquad \dfrac{}{((p \wedge q) \to q)} \qquad \dfrac{}{(p \to ((\neg p) \to \mathbb{O}))} \qquad \dfrac{}{(\mathbb{O} \to p)}$$

$$\dfrac{}{((p \to r) \to ((q \to r) \to ((p \vee q) \to r)))} \qquad \dfrac{}{((p \to \mathbb{O}) \to (\neg p))} \qquad \dfrac{}{(((\neg p) \to \mathbb{O}) \to p)}$$

- Esempio: Il seguente albero rappresenta una dimostrazione di $\vdash_{\mathrm{H}} (p \to p)$ in cui utilizziamo soltanto i primi due assiomi riportati sopra, dove il secondo assioma viene usato una sola volta con q sostituito da $(p \to p)$ ed r sostituito da p mentre il primo assioma viene usato due volte con analoghe sostituzioni:

$$\dfrac{\dfrac{((p \to ((p \to p) \to p)) \to ((p \to (p \to p)) \to (p \to p))) \qquad (p \to ((p \to p) \to p))}{((p \to (p \to p)) \to (p \to p))} \qquad (p \to (p \to p))}{(p \to p)}$$

- Teorema: Il sistema deduttivo alla Hilbert formato dal modus ponens e dai seguenti tre assiomi:

$$\frac{}{(p \to (q \to p))} \qquad \frac{}{((p \to (q \to r)) \to ((p \to q) \to (p \to r)))} \qquad \frac{}{(((\neg p) \to (\neg q)) \to (q \to p))}$$

  è corretto e completo rispetto alla logica proposizionale, nonché decidibile.

- Corollario: L'insieme di connettivi logici $\{\neg, \to\}$ è funzionalmente completo per le tautologie.

- Teorema (di deduzione): Siano $\Phi \in 2^{FBF}$ finito e $\phi, \gamma \in FBF$. Se $\Phi \cup \{\phi\} \vdash_{\mathrm{H}} \gamma$, allora $\Phi \vdash_{\mathrm{H}} (\phi \to \gamma)$.

- Nei sistemi deduttivi alla Hilbert è interessante ridurre il numero di assiomi preservando il potere deduttivo. Il minimo sistema deduttivo alla Hilbert è formato dal modus ponens e dal seguente assioma detto assioma di Meredith:

$$\frac{}{(((((p \to q) \to ((\neg r) \to (\neg s))) \to r) \to t) \to ((t \to p) \to (s \to p)))}$$

- Ad essere precisi, nei sistema deduttivi alla Hilbert gli assiomi sono in realtà schemi di assiomi, nel senso che da ognuno di essi è possibile ottenere un assioma equivalente a quello originario sostituendo tutte le occorrenze di almeno una proposizione con la stessa formula ben formata (come fatto nell'ultimo esempio relativo a $\vdash_{\mathrm{H}}$). Il motivo per cui il nuovo assioma è equivalente a quello originario discende dal teorema di sostituzione. ■fltpl_7

Capitolo 10

Logica dei predicati

10.1 Sintassi della logica dei predicati

- La logica proposizionale non è sufficientemente espressiva per rappresentare ragionamenti relativi ad una moltitudine di oggetti, come "tutti gli oggetti godono di una certa proprietà" oppure "esiste almeno un oggetto che gode di una certa proprietà". La logica proposizionale si concentra infatti sui connettivi logici e su come il valore di verità di una formula ben formata dipenda dal valore di verità delle sue sottoformule immediate, senza considerare l'eventuale struttura interna delle proposizioni, presenti nella formula, che di solito è della forma soggetto-predicato.
- La logica dei predicati ovvia a questi inconvenienti includendo degli ulteriori operatori detti quantificatori in aggiunta ai connettivi logici della logica proposizionale e sostituendo le proposizioni con predicati applicati a termini. I termini, che individuano gli oggetti di interesse, sono espressi tramite costanti, variabili e funzioni definite sui termini. I predicati, che specificano le proprietà di interesse per gli oggetti precedentemente individuati, sono espressi tramite relazioni su insiemi di termini. I predicati sono assimilabili a funzioni logiche.
- La logica dei predicati del primo ordine ha un quantificatore universale denotato con $\forall$ ("per ogni") ed un quantificatore esistenziale denotato con $\exists$ ("esiste"), i quali possono fare riferimento solo a termini variabili che compaiono come argomenti di funzioni e predicati. Da ora in poi per logica dei predicati intenderemo sempre quella del primo ordine, escludendo quindi variabili funzionali e variabili predicative assieme alle relative quantificazioni.
- Gli elementi sintattici di base della logica dei predicati sono i seguenti:
  - Un insieme Cos di simboli di costante, i quali verranno indicati con le lettere a, b, c.
  - Un insieme numerabile Var di simboli di variabile, i quali verranno indicati con le lettere x, y, z.
  - Un insieme Fun di simboli di funzione, i quali verranno indicati con le lettere f, g, h.
  - Un insieme $Pred$ di simboli di predicato, i quali verranno indicati con le lettere P, Q, R.
  - I connettivi logici $\neg$, $\vee$, $\wedge$, $\rightarrow$, $\leftrightarrow$.
  - I quantificatori $\forall$, $\exists$.
  - I simboli ausiliari (e).
- L'insieme Ter dei termini è il più piccolo linguaggio su $Cos \cup Var \cup Fun \cup \{(,)\}$ tale che:
  - $Cos \subseteq Ter$.
  - $Var \subseteq Ter$.
  - $f(t_1, \ldots, t_n) \in Ter$ per ogni $f \in Fun$ con $n \geq 1$ argomenti e $t_1, \ldots, t_n \in Ter$.

  Gli elementi di Ter non hanno valori di verità ad essi associati, in quanto sono oggetti *extra logici* introdotti nella sintassi soltanto per arricchire l'espressività rispetto alla logica proposizionale.

- L'insieme FBF delle formule ben formate della logica dei predicati è il più piccolo linguaggio su $Ter \cup Pred \cup \{\neg, \vee, \wedge, \rightarrow, \leftrightarrow, \forall, \exists, (,)\}$ tale che:
  - $P(t_1, \ldots, t_n) \in FBF$ per ogni $P \in Pred$ con $n \geq 1$ argomenti e $t_1, \ldots, t_n \in Ter$.
  - $(\neg\phi) \in FBF$ per ogni $\phi \in FBF$.
  - $(\phi_1 \vee \phi_2) \in FBF$ per ogni $\phi_1, \phi_2 \in FBF$.
  - $(\phi_1 \wedge \phi_2) \in FBF$ per ogni $\phi_1, \phi_2 \in FBF$.
  - $(\phi_1 \rightarrow \phi_2) \in FBF$ per ogni $\phi_1, \phi_2 \in FBF$.
  - $(\phi_1 \leftrightarrow \phi_2) \in FBF$ per ogni $\phi_1, \phi_2 \in FBF$.
  - $(\forall x)(\phi) \in FBF$ per ogni $x \in Var$ e $\phi \in FBF$.
  - $(\exists x)(\phi) \in FBF$ per ogni $x \in Var$ e $\phi \in FBF$.
- Teorema: Sia FBF' un linguaggio su $Ter \cup Pred \cup \{\neg, \vee, \wedge, \rightarrow, \leftrightarrow, \forall, \exists, (,)\}$ definito ponendo $FBF' = \bigcup_{n \in \mathbb{N}} FBF_n$ dove:
$$FBF_n = \begin{cases} \{P(t_1, \ldots, t_n) \mid P \in Pred \text{ con } n \geq 1 \text{ argomenti e } t_1, \ldots, t_n \in Ter\} & \text{se } n = 0 \\ FBF_{n-1} \cup \{(\neg\phi) \mid \phi \in FBF_{n-1}\} \cup & \text{se } n > 0 \\ \quad \{(\phi_1 \vee \phi_2) \mid \phi_1, \phi_2 \in FBF_{n-1}\} \cup \\ \quad \{(\phi_1 \wedge \phi_2) \mid \phi_1, \phi_2 \in FBF_{n-1}\} \cup \\ \quad \{(\phi_1 \rightarrow \phi_2) \mid \phi_1, \phi_2 \in FBF_{n-1}\} \cup \\ \quad \{(\phi_1 \leftrightarrow \phi_2) \mid \phi_1, \phi_2 \in FBF_{n-1}\} \cup \\ \quad \{(\forall x)(\phi) \mid x \in Var \text{ e } \phi \in FBF_{n-1}\} \cup \\ \quad \{(\exists x)(\phi) \mid x \in Var \text{ e } \phi \in FBF_{n-1}\} \end{cases}$$
Allora $FBF' = FBF$.
- L'insieme $sf(\phi)$ delle sottoformule di $\phi \in FBF$ è definito per induzione sulla struttura sintattica di ϕ come segue:
$$sf(\phi) = \{\phi\} \cup \begin{cases} \emptyset & \text{se } \phi \text{ è della forma } P(t_1, \ldots, t_n) \text{ con } P \in Pred \text{ e } t_1, \ldots, t_n \in Ter \\ sf(\phi') & \text{se } \phi \text{ è della forma } (\neg\phi'), (\forall x)(\phi') \text{ o } (\exists x)(\phi') \\ sf(\phi'_1) \cup sf(\phi'_2) & \text{se } \phi \text{ è della forma } (\phi'_1 \vee \phi'_2), (\phi'_1 \wedge \phi'_2), (\phi'_1 \rightarrow \phi'_2) \text{ o } (\phi'_1 \leftrightarrow \phi'_2) \end{cases}$$
- Data una formula $\phi \in FBF$, diciamo che essa è:
  - Atomica sse è della forma $P(t_1, \ldots, t_n)$ con $P \in Pred$ e $t_1, \ldots, t_n \in Ter$.
  - Composta con connettivo primario $\neg$ e sottoformula immediata ϕ' sse è della forma $(\neg\phi')$.
  - Composta con connettivo primario $\vee$ e sottoformule immediate ϕ'_1, ϕ'_2 sse è della forma $(\phi'_1 \vee \phi'_2)$.
  - Composta con connettivo primario $\wedge$ e sottoformule immediate ϕ'_1, ϕ'_2 sse è della forma $(\phi'_1 \wedge \phi'_2)$.
  - Composta con connettivo primario $\rightarrow$ e sottoformule immediate ϕ'_1, ϕ'_2 sse è della forma $(\phi'_1 \rightarrow \phi'_2)$.
  - Composta con connettivo primario $\leftrightarrow$ e sottoformule immediate ϕ'_1, ϕ'_2 sse è della forma $(\phi'_1 \leftrightarrow \phi'_2)$.
  - Quantificata universalmente rispetto a x con sottoformula immediata ϕ' sse è della forma $(\forall x)(\phi')$.
  - Quantificata esistenzialmente rispetto a x con sottoformula immediata ϕ' sse è della forma $(\exists x)(\phi')$.
- Esempio: "tutte le persone sono mortali" si esprime come $(\forall x)(Persona(x) \rightarrow Mortale(x))$.
- Teorema (di leggibilità univoca): Le formule ben formate non sono ambigue, cioè esiste un unico modo di leggere ogni $\phi \in FBF$.
- L'uso sistematico delle parentesi può essere evitato introducendo delle regole di precedenza e associatività per i connettivi logici e i quantificatori. È consuetudine assumere che $\forall$, $\exists$ e $\neg$ abbiano precedenza su $\wedge$, che $\wedge$ abbia precedenza su $\vee$, che $\vee$ abbia precedenza su $\rightarrow$, che $\rightarrow$ abbia precedenza su $\leftrightarrow$ e che tutti questi operatori logici siano associativi da sinistra ad eccezione di $\forall$, $\exists$ e $\neg$.

- Teorema (principio di induzione strutturale per i termini): Sia $\mathcal{Q}$ una proprietà definita su *Ter*. Se:
  1. $\mathcal{Q}$ è soddisfatta da ogni costante di *Cos*;
  2. $\mathcal{Q}$ è soddisfatta da ogni variabile di *Var*;
  3. per ogni $t_1, \ldots, t_n \in Ter$ con $n \geq 1$, $\mathcal{Q}$ soddisfatta da ogni t_i tale che $1 \leq i \leq n$ implica $\mathcal{Q}$ soddisfatta da $f(t_1, \ldots, t_n)$ per ogni $f \in Fun$ con $n \geq 1$ argomenti;

  allora $\mathcal{Q}$ è soddisfatta da ogni $t \in Ter$.

- Teorema (principio di induzione strutturale per le formule ben formate): Sia $\mathcal{Q}$ una proprietà definita su *FBF*. Se:
  1. $\mathcal{Q}$ è soddisfatta da ogni formula atomica di *FBF*;
  2. per ogni $\phi \in FBF$, $\mathcal{Q}$ soddisfatta da ϕ implica $\mathcal{Q}$ soddisfatta da $(\neg\phi)$;
  3. per ogni $\phi_1, \phi_2 \in FBF$, $\mathcal{Q}$ soddisfatta da ϕ_1 e $\mathcal{Q}$ soddisfatta da ϕ_2 implicano $\mathcal{Q}$ soddisfatta da $(\phi_1 \vee \phi_2)$;
  4. per ogni $\phi_1, \phi_2 \in FBF$, $\mathcal{Q}$ soddisfatta da ϕ_1 e $\mathcal{Q}$ soddisfatta da ϕ_2 implicano $\mathcal{Q}$ soddisfatta da $(\phi_1 \wedge \phi_2)$;
  5. per ogni $\phi_1, \phi_2 \in FBF$, $\mathcal{Q}$ soddisfatta da ϕ_1 e $\mathcal{Q}$ soddisfatta da ϕ_2 implicano $\mathcal{Q}$ soddisfatta da $(\phi_1 \rightarrow \phi_2)$;
  6. per ogni $\phi_1, \phi_2 \in FBF$, $\mathcal{Q}$ soddisfatta da ϕ_1 e $\mathcal{Q}$ soddisfatta da ϕ_2 implicano $\mathcal{Q}$ soddisfatta da $(\phi_1 \leftrightarrow \phi_2)$;
  7. per ogni $\phi \in FBF$, $\mathcal{Q}$ soddisfatta da ϕ implica $\mathcal{Q}$ soddisfatta da $(\forall x)(\phi)$ per ogni $x \in Var$;
  8. per ogni $\phi \in FBF$, $\mathcal{Q}$ soddisfatta da ϕ implica $\mathcal{Q}$ soddisfatta da $(\exists x)(\phi)$ per ogni $x \in Var$;

  allora $\mathcal{Q}$ è soddisfatta da ogni $\phi \in FBF$. ■fltpl_8

- La logica dei predicati comprende delle ulteriori nozioni che riguardano le occorrenze di variabili e le sostituzioni sintattiche. Date una formula $\phi \in FBF$ ed una variabile $x \in Var$, diciamo che:
  - x occorre in ϕ sse x compare in una sottoformula di ϕ che non è quantificata.
  - Un'occorrenza di x in ϕ è legata (ad un quantificatore) se compare in una sottoformula di ϕ della forma $(\forall x)(\phi')$ o $(\exists x)(\phi')$, altrimenti quell'occorrenza di x in ϕ è libera.

  Se ϕ è della forma $(\forall x)(\phi')$ o $(\exists x)(\phi')$ diciamo che ϕ' è il campo d'azione (o scope) del quantificatore.

- Diciamo che una formula $\phi \in FBF$ è chiusa se ogni occorrenza di ogni variabile che occorre in ϕ è legata, altrimenti la formula ϕ è aperta.

- Esempi:
  - La variabile x non occorre nella formula $(\forall x)(P(a))$.
  - La formula chiusa $((\forall x)(P(x)) \wedge (\forall x)(Q(x)))$ mostra che due occorrenze di una stessa variabile possono ricadere nel campo d'azione di due quantificatori che compaiono in due sottoformule indipendenti della formula originaria.
  - La formula chiusa $(\forall x)(P(x) \rightarrow (\forall x)(Q(x)))$ mostra che due occorrenze di una stessa variabile possono ricadere nel campo d'azione di due quantificatori che compaiono in due sottoformule tali che una è sottoformula dell'altra. In tal caso, ogni occorrenza della variabile è legata al quantificatore più interno tra quelli nel cui campo d'azione essa si trova; quindi la x di $P(x)$ è legata al quantificatore che sta a sinistra mentre la x di $Q(x)$ è legata al quantificatore che sta a destra.

- Dato un termine $t \in Ter$, l'insieme $var(t)$ delle variabili che occorrono in t è definito per induzione sulla struttura sintattica di t come segue:

$$var(t) = \begin{cases} \emptyset & \text{se } t \in Cos \\ \{t\} & \text{se } t \in Var \\ var(t_1) \cup \ldots \cup var(t_n) & \text{se } t \text{ è della forma } f(t_1, \ldots, t_n) \text{ con } f \in Fun \text{ e } t_1, \ldots, t_n \in Ter \end{cases}$$

- Data una formula $\phi \in FBF$:
  - L'insieme $var(\phi)$ delle variabili che occorrono in ϕ è definito per induzione sulla struttura sintattica di ϕ come segue:
    $$var(\phi) = \begin{cases} var(t_1) \cup \ldots \cup var(t_n) & \text{se } \phi \text{ è della forma } P(t_1, \ldots, t_n) \text{ con } P \in Pred \text{ e } t_1, \ldots, t_n \in Ter \\ var(\phi') & \text{se } \phi \text{ è della forma } (\neg\phi'), (\forall x)(\phi') \text{ o } (\exists x)(\phi') \\ var(\phi'_1) \cup var(\phi'_2) & \text{se } \phi \text{ è della forma } (\phi'_1 \vee \phi'_2), (\phi'_1 \wedge \phi'_2), (\phi'_1 \rightarrow \phi'_2) \text{ o } (\phi'_1 \leftrightarrow \phi'_2) \end{cases}$$
  - L'insieme $varleg(\phi)$ delle variabili che occorrono legate in ϕ è definito per induzione sulla struttura sintattica di ϕ come segue:
    $$varleg(\phi) = \begin{cases} \emptyset & \text{se } \phi \text{ è della forma } P(t_1, \ldots, t_n) \text{ con } P \in Pred \text{ e } t_1, \ldots, t_n \in Ter \\ varleg(\phi') & \text{se } \phi \text{ è della forma } (\neg\phi') \\ varleg(\phi'_1) \cup varleg(\phi'_2) & \text{se } \phi \text{ è della forma } (\phi'_1 \vee \phi'_2), (\phi'_1 \wedge \phi'_2), (\phi'_1 \rightarrow \phi'_2) \text{ o } (\phi'_1 \leftrightarrow \phi'_2) \\ varleg(\phi') & \text{se } \phi \text{ è della forma } (\forall x)(\phi') \text{ o } (\exists x)(\phi') \text{ e } x \notin var(\phi') \\ varleg(\phi') \cup \{x\} & \text{se } \phi \text{ è della forma } (\forall x)(\phi') \text{ o } (\exists x)(\phi') \text{ e } x \in var(\phi') \end{cases}$$
  - L'insieme $varlib(\phi)$ delle variabili che occorrono libere in ϕ è definito per induzione sulla struttura sintattica di ϕ come segue:
    $$varlib(\phi) = \begin{cases} var(t_1) \cup \ldots \cup var(t_n) & \text{se } \phi \text{ è della forma } P(t_1, \ldots, t_n) \text{ con } P \in Pred \text{ e } t_1, \ldots, t_n \in Ter \\ varlib(\phi') & \text{se } \phi \text{ è della forma } (\neg\phi') \\ varlib(\phi'_1) \cup varlib(\phi'_2) & \text{se } \phi \text{ è della forma } (\phi'_1 \vee \phi'_2), (\phi'_1 \wedge \phi'_2), (\phi'_1 \rightarrow \phi'_2) \text{ o } (\phi'_1 \leftrightarrow \phi'_2) \\ varlib(\phi') \setminus \{x\} & \text{se } \phi \text{ è della forma } (\forall x)(\phi') \text{ o } (\exists x)(\phi') \end{cases}$$
- Esempio:
  - La formula aperta $((\exists x)(R(a, b, x)) \wedge (\neg Q(x)))$ mostra che la stessa variabile può occorrere sia legata che libera nella medesima formula; infatti la x di $R(a, b, x)$ è legata mentra la x di $Q(x)$ è libera. Ciò significa che in generale $varleg(\phi) \cap varlib(\phi) \neq \emptyset$ quando $\phi \in FBF$ è aperta. In tal caso, per evitare confusione è conveniente sostituire le occorrenze libere oppure le occorrenze legate di quella variabile con un nuovo simbolo di variabile; ad esempio la formula data può diventare $((\exists x)(R(a, b, x)) \wedge (\neg Q(y)))$ oppure $((\exists z)(R(a, b, z)) \wedge (\neg Q(x)))$.
- Dati due termini $t, t' \in Ter$ ed una variabile $y \in Var$, il termine ottenuto da t sostituendo ogni occorrenza di y con t' è definito per induzione sulla struttura sintattica di t come segue:
  $$t[^{t'}/_{y}] = \begin{cases} t & \text{se } t \in Cos \\ t' & \text{se } t \in Var \text{ e } t = y \\ t & \text{se } t \in Var \text{ e } t \neq y \\ f(t_1[^{t'}/_{y}], \ldots, t_n[^{t'}/_{y}]) & \text{se } t \text{ è della forma } f(t_1, \ldots, t_n) \text{ con } f \in Fun \text{ e } t_1, \ldots, t_n \in Ter \end{cases}$$
- Dati una formula $\phi \in FBF$, un termine $t \in Ter$ ed una variabile $y \in Var$, la formula ottenuta da ϕ sostituendo ogni occorrenza di y con t' è definita per induzione sulla struttura sintattica di ϕ come segue:
  $$\phi[^{t'}/_{y}] = \begin{cases} P(t_1[^{t'}/_{y}], \ldots, t_n[^{t'}/_{y}]) & \text{se } \phi \text{ è della forma } P(t_1, \ldots, t_n) \text{ con } P \in Pred \text{ e } t_1, \ldots, t_n \in Ter \\ (\neg\phi'[^{t'}/_{y}]) & \text{se } \phi \text{ è della forma } (\neg\phi') \\ (\phi'_1[^{t'}/_{y}] \vee \phi'_2[^{t'}/_{y}]) & \text{se } \phi \text{ è della forma } (\phi'_1 \vee \phi'_2) \\ (\phi'_1[^{t'}/_{y}] \wedge \phi'_2[^{t'}/_{y}]) & \text{se } \phi \text{ è della forma } (\phi'_1 \wedge \phi'_2) \\ (\phi'_1[^{t'}/_{y}] \rightarrow \phi'_2[^{t'}/_{y}]) & \text{se } \phi \text{ è della forma } (\phi'_1 \rightarrow \phi'_2) \\ (\phi'_1[^{t'}/_{y}] \leftrightarrow \phi'_2[^{t'}/_{y}]) & \text{se } \phi \text{ è della forma } (\phi'_1 \leftrightarrow \phi'_2) \\ \phi & \text{se } \phi \text{ è della forma } (\forall y)(\phi') \text{ o } (\exists y)(\phi') \\ (\forall x)(\phi'[^{t'}/_{y}]) & \text{se } \phi \text{ è della forma } (\forall x)(\phi') \text{ e } x \neq y \text{ e } x \notin var(t') \\ (\forall z)(\phi'[^{z}/_{x}][^{t'}/_{y}]) & \text{se } \phi \text{ è della forma } (\forall x)(\phi') \text{ e } x \neq y \text{ e } x \in var(t') \text{ con } z \notin \{y\} \cup var(t') \cup var(\phi') \\ (\exists x)(\phi'[^{t'}/_{y}]) & \text{se } \phi \text{ è della forma } (\exists x)(\phi') \text{ e } x \neq y \text{ e } x \notin var(t') \\ (\exists z)(\phi'[^{z}/_{x}][^{t'}/_{y}]) & \text{se } \phi \text{ è della forma } (\exists x)(\phi') \text{ e } x \neq y \text{ e } x \in var(t') \text{ con } z \notin \{y\} \cup var(t') \cup var(\phi') \end{cases}$$
- Osserviamo che la settima clausola della definizione stabilisce che le occorrenze legate di una variabile non possono essere sostituite. Inoltre la nona e l'undicesima clausola stabiliscono che è necessaria una sostituzione preliminare della forma $[^{z}/_{x}]$ quando il termine t' contiene occorrenze della variabile quantificata x, altrimenti quelle occorrenze verrebbero erroneamente legate (ad esempio, la formula aperta $(\forall x)(P(y))$ diventerebbe erroneamente chiusa applicandole la sostituzione $[^{x}/_{y}]$). La variabile z deve essere nuova al fine di evitare collisioni con y e con le variabili che occorrono in t' o ϕ'. ■fltpl_9

10.2 Semantica e indecidibilità della logica dei predicati

- Per stabilire il valore di verità di una formula ben formata della logica dei predicati, è necessario fissare un dominio, interpretare in quel dominio i simboli di costante, funzione e predicato che compaiono nella formula ed assegnare un valore del dominio a ciascuna variabile che occorre libera nella formula (quindi il valore di verità della formula non è assoluto ma dipende dal dominio e dall'interpretazione). Il significato dei connettivi logici è lo stesso della logica proposizionale, mentre il significato dei quantificatori può essere informalmente descritto tramite sostituzioni sintattiche come segue:
  - $(\forall x)(\phi)$ è vera sse $\phi[^t/_x]$ è vera per ogni $t \in \mathit{Ter}$ con $\mathit{var}(t) = \emptyset$.
  - $(\exists x)(\phi)$ è vera sse esiste $t \in \mathit{Ter}$ con $\mathit{var}(t) = \emptyset$ tale che $\phi[^t/_x]$ è vera.
- Chiamiamo ambiente ogni tripla $\mathcal{E} = (D, \mathcal{I}, \mathcal{V})$ dove:
  - D è un insieme non vuoto di valori detto dominio (o universo).
  - $\mathcal{I}$ è una funzione di interpretazione che associa ad ogni simbolo di:
    - costante $a \in \mathit{Cos}$ un valore $\mathcal{I}(a) \in D$;
    - funzione $f \in \mathit{Fun}$ con $n \geq 1$ argomenti una funzione $\mathcal{I}(f) : D^n \to D$;
    - predicato $P \in \mathit{Pred}$ con $n \geq 1$ argomenti una relazione $\mathcal{I}(P) \subseteq D^n$.
  - $\mathcal{V}$ è una funzione che assegna ad ogni occorrenza libera di una variabile $x \in \mathit{Var}$ un valore $\mathcal{V}(x) \in D$.
- Dato un ambiente $\mathcal{E} = (D, \mathcal{I}, \mathcal{V})$, definiamo l'interpretazione di un termine non costante $t \in \mathit{Ter} \setminus \mathit{Cos}$ estendendo la funzione di interpretazione $\mathcal{I}$ per induzione sulla struttura sintattica di t come segue:
$$\mathcal{I}(t) = \begin{cases} \mathcal{V}(t) & \text{se } t \in \mathit{Var} \\ \mathcal{I}(f)(\mathcal{I}(t_1), \ldots, \mathcal{I}(t_n)) & \text{se } t \text{ è della forma } f(t_1, \ldots, t_n) \text{ con } f \in \mathit{Fun} \text{ e } t_1, \ldots, t_n \in \mathit{Ter} \end{cases}$$
- Sia Amb l'insieme di tutti gli ambienti. La semantica della logica dei predicati viene formalizzata attraverso la relazione di soddisfacimento (o relazione di verità) $\models$ definita come il più piccolo sottoinsieme di $\mathit{Amb} \times \mathit{FBF}$ tale che:
  - $(D, \mathcal{I}, \mathcal{V}) \models P(t_1, \ldots, t_n)$ se $(\mathcal{I}(t_1), \ldots, \mathcal{I}(t_n)) \in \mathcal{I}(P)$.
  - $(D, \mathcal{I}, \mathcal{V}) \models (\neg\phi)$ se $(D, \mathcal{I}, \mathcal{V}) \not\models \phi$.
  - $(D, \mathcal{I}, \mathcal{V}) \models (\phi_1 \vee \phi_2)$ se $(D, \mathcal{I}, \mathcal{V}) \models \phi_1$ o $(D, \mathcal{I}, \mathcal{V}) \models \phi_2$.
  - $(D, \mathcal{I}, \mathcal{V}) \models (\phi_1 \wedge \phi_2)$ se $(D, \mathcal{I}, \mathcal{V}) \models \phi_1$ e $(D, \mathcal{I}, \mathcal{V}) \models \phi_2$.
  - $(D, \mathcal{I}, \mathcal{V}) \models (\phi_1 \to \phi_2)$ se $(D, \mathcal{I}, \mathcal{V}) \not\models \phi_1$ oppure $(D, \mathcal{I}, \mathcal{V}) \models \phi_2$.
  - $(D, \mathcal{I}, \mathcal{V}) \models (\phi_1 \leftrightarrow \phi_2)$ se $(D, \mathcal{I}, \mathcal{V}) \models \phi_1$ e $(D, \mathcal{I}, \mathcal{V}) \models \phi_2$, oppure $(D, \mathcal{I}, \mathcal{V}) \not\models \phi_1$ e $(D, \mathcal{I}, \mathcal{V}) \not\models \phi_2$.
  - $(D, \mathcal{I}, \mathcal{V}) \models (\forall x)(\phi)$ se $(D, \mathcal{I}, \mathcal{V} \setminus \{(x, \mathcal{V}(x))\} \cup \{(x, d)\}) \models \phi$ per ogni $d \in D$.
  - $(D, \mathcal{I}, \mathcal{V}) \models (\exists x)(\phi)$ se esiste $d \in D$ tale che $(D, \mathcal{I}, \mathcal{V} \setminus \{(x, \mathcal{V}(x))\} \cup \{(x, d)\}) \models \phi$.

  Dato $\mathcal{E} = (D, \mathcal{I}, \mathcal{V})$, quando $\mathcal{E} \models \phi$ diciamo che $\mathcal{E}$ soddisfa ϕ oppure che $\mathcal{E}$ è un modello di ϕ.
- Esempio: Consideriamo la formula $((\forall x)(P(x, f(x))) \wedge Q(g(a, z)))$. Se prendiamo un ambiente in cui il dominio è $\mathbb{N}$, la costante a è interpretata come il numero 2, la funzione f è interpretata come la funzione successore, la funzione g è interpretata come l'operazione $+$, il predicato P è interpretato come la relazione $<$, il predicato Q è interpretato come il fatto di essere un numero primo e all'occorrenza libera della variabile z viene assegnato valore 1, allora l'ambiente soddisfa la formula.
- Teorema: Siano $\phi \in \mathit{FBF}$ ed $\mathcal{E}_1 = (D, \mathcal{I}, \mathcal{V}_1), \mathcal{E}_2 = (D, \mathcal{I}, \mathcal{V}_2)$ due ambienti tali che per ogni $x \in \mathit{varlib}(\phi)$ risulta $\mathcal{V}_1(x) = \mathcal{V}_2(x)$. Allora $\mathcal{E}_1 \models \phi$ sse $\mathcal{E}_2 \models \phi$.
- Corollario: Siano $\phi \in \mathit{FBF}$ ed $\mathcal{E}_1 = (D, \mathcal{I}, \mathcal{V}_1)$, $\mathcal{E}_2 = (D, \mathcal{I}, \mathcal{V}_2)$ due ambienti. Se ϕ è chiusa allora $\mathcal{E}_1 \models \phi$ sse $\mathcal{E}_2 \models \phi$.

- Esempi (i nomi attribuiti a costanti, funzioni e predicati evocano le loro interpretazioni come accade normalmente nella pratica):

  – Un predicato molto importante in matematica è l'uguaglianza. Le sue proprietà di riflessività, simmetria, transitività e congruenza rispetto ad un operatore f con $n \geq 1$ argomenti possono essere definite attraverso le seguenti formule ben formate della logica dei predicati:

  $(\forall x)(\mathit{Uguale}(x,x))$
  $(\forall x \forall y)(\mathit{Uguale}(x,y) \rightarrow \mathit{Uguale}(y,x))$
  $(\forall x \forall y \forall z)((\mathit{Uguale}(x,y) \wedge \mathit{Uguale}(y,z)) \rightarrow \mathit{Uguale}(x,z))$
  $(\forall x_1 \ldots \forall x_n \forall y_1 \ldots \forall y_n)((\mathit{Uguale}(x_1,y_1) \wedge \ldots \wedge \mathit{Uguale}(x_n,y_n)) \rightarrow \mathit{Uguale}(f(x_1,\ldots,x_n), f(y_1,\ldots,y_n)))$

  – L'aritmetica può essere definita a partire da una costante 0, una funzione unaria *succ*, due funzioni binarie *somma* e *prod*, un generico predicato P per enunciare il principio di induzione e il predicato binario *Uguale*, attraverso le seguenti formule ben formate della logica dei predicati sviluppate da Peano:

  $(\forall x)(\neg \mathit{Uguale}(\mathit{succ}(x), 0))$
  $(\forall x \forall y)(\mathit{Uguale}(\mathit{succ}(x), \mathit{succ}(y)) \rightarrow \mathit{Uguale}(x,y))$
  $(\forall x)(\mathit{Uguale}(\mathit{somma}(x,0), x))$
  $(\forall x \forall y)(\mathit{Uguale}(\mathit{somma}(x, \mathit{succ}(y)), \mathit{somma}(\mathit{succ}(x), y)))$
  $(\forall x)(\mathit{Uguale}(\mathit{prod}(x,0), 0))$
  $(\forall x \forall y)(\mathit{Uguale}(\mathit{prod}(x, \mathit{succ}(y)), \mathit{somma}(x, \mathit{prod}(x,y))))$
  $((P(0) \wedge (\forall x)(P(x) \rightarrow P(\mathit{succ}(x)))) \rightarrow (\forall x)(P(x)))$

  – La teoria degli insiemi può essere definita a partire da una costante $\emptyset$, una funzione unaria *parti*, una funzione binaria *coppia*, due predicati binari *App* e *Incl* e il predicato binario *Uguale*, attraverso le seguenti formule ben formate della logica dei predicati sviluppate da Skolem per formalizzare gli assiomi introdotti da Zermelo e Fraenkel per superare paradossi quali quelli di Cantor e di Russell nella teoria ingenua degli insiemi formulata da Cantor:

$(\forall x \forall y)((\forall z)(\mathit{App}(z,x) \leftrightarrow \mathit{App}(z,y)) \rightarrow \mathit{Uguale}(x,y))$	estensionalità
$(\forall z)(\neg \mathit{App}(z, \emptyset))$	insieme vuoto
$(\forall x \forall y \forall z)(\mathit{App}(z, \mathit{coppia}(x,y)) \leftrightarrow (\mathit{Uguale}(z,x) \vee \mathit{Uguale}(z,y)))$	coppia non ordinata
$(\forall x \forall y)(\mathit{Incl}(x,y) \leftrightarrow (\forall z)(\mathit{App}(z,x) \rightarrow \mathit{App}(z,y)))$	inclusione
$(\forall x \forall z)(\mathit{App}(z, \mathit{parti}(x)) \leftrightarrow \mathit{Incl}(z,x))$	insieme delle parti
$(\forall x \exists y \forall z)(\mathit{App}(z,y) \leftrightarrow (\mathit{App}(z,x) \wedge P(z)))$	separazione
$(\exists x)(\mathit{App}(\emptyset, x) \wedge (\forall z)(\mathit{App}(z,x) \rightarrow \mathit{App}(\mathit{coppia}(z,z), x)))$	insieme infinito
$((\forall x \exists! y)(Q(x,y)) \rightarrow (\forall x' \exists y' \forall y'')(\mathit{App}(y',y') \leftrightarrow (\exists x'')(\mathit{App}(x'',x') \wedge Q(x'',y''))))$	rimpiazzamento
$(\forall x)((\neg \mathit{Uguale}(x, \emptyset)) \rightarrow (\exists y)(\mathit{App}(y,x) \wedge (\forall z)(\mathit{App}(z,x) \rightarrow (\neg \mathit{App}(z,y)))))$	regolarità

  dove $(\exists! z)(R(z))$ sta per $(\exists z)(R(z) \wedge (\forall z')(R(z') \rightarrow \mathit{Uguale}(z',z)))$.

  – Le strutture algebriche possono essere definite attraverso la logica dei predicati. Per esempio, un gruppo abeliano può essere definito a partire da una costante ν che indica l'elemento neutro, una funzione unaria *inv* che indica l'operazione di inversione, una funzione binaria *comp* che indica l'operazione di composizione e il predicato binario *Uguale*, attraverso le seguenti formule ben formate della logica dei predicati:

  $(\forall x \forall y \forall z)(\mathit{Uguale}(\mathit{comp}(\mathit{comp}(x,y),z), \mathit{comp}(x, \mathit{comp}(y,z))))$
  $(\forall x \forall y)(\mathit{Uguale}(\mathit{comp}(x,y), \mathit{comp}(y,x)))$
  $(\forall x)(\mathit{Uguale}(\mathit{comp}(x,\nu),x) \wedge \mathit{Uguale}(\mathit{comp}(\nu,x),x))$
  $(\forall x)(\mathit{Uguale}(\mathit{comp}(x, \mathit{inv}(x)),\nu) \wedge \mathit{Uguale}(\mathit{comp}(\mathit{inv}(x),x),\nu))$

  - Le strutture dati possono essere definite attraverso la logica dei predicati. Per esempio, una pila può essere definita a partire da una costante ε che indica la pila vuota, una funzione unaria *cima* che restituisce l'elemento che sta in cima alla pila, una funzione unaria *togli* che restituisce la pila dopo aver tolto l'elemento che sta in cima, una funzione binaria *metti* che restituisce la pila dopo aver messo un nuovo elemento in cima e il predicato binario *Uguale*, attraverso le seguenti formule ben formate della logica dei predicati:
$$(\forall x \forall y)(Uguale(cima(metti(x,y)),y))$$
$$(\forall x \forall y)(Uguale(togli(metti(x,y)),x))$$
$$(\forall x)(Uguale(metti(togli(x), cima(x)),x))$$
$$(\forall x)(Uguale(x,\varepsilon) \rightarrow Uguale(togli(x),x))$$

- Data una formula $\phi \in FBF$, diciamo che essa è:
  - Soddisfacibile sse esiste un ambiente $\mathcal{E}$ tale che $\mathcal{E} \models \phi$.
  - Una tautologia (o valida) sse $\mathcal{E} \models \phi$ per ogni ambiente $\mathcal{E}$.
  - Una contraddizione (o insoddisfacibile) sse $\mathcal{E} \not\models \phi$ per ogni ambiente $\mathcal{E}$.

- Teorema: Sia $\phi \in FBF$. Allora:
  - ϕ è una tautologia sse $(\neg\phi)$ è una contraddizione.
  - ϕ non è una tautologia sse $(\neg\phi)$ è soddisfacibile.
  - ϕ è soddisfacibile sse ϕ non è una contraddizione.

- Siano $\phi \in FBF$ e $varlib(\phi) = \{x_1, \ldots, x_k\}$ con $k \in \mathbb{N}$:
  - La chiusura universale di ϕ è definita come $cu(\phi) = (\forall x_1 \ldots \forall x_k)(\phi)$.
  - La chiusura esistenziale di ϕ è definita come $ce(\phi) = (\exists x_1 \ldots \exists x_k)(\phi)$.

- Teorema: Sia $\phi \in FBF$. Allora:
  - ϕ è una tautologia sse $cu(\phi)$ è una tautologia.
  - ϕ è soddisfacibile sse $ce(\phi)$ è soddisfacibile.

- Esempi:
  - La formula $((\exists x \forall y)(P(x,y)) \rightarrow (\forall y \exists x)(P(x,y)))$ è una tautologia sse per ogni ambiente $(D,\mathcal{I},\mathcal{V})$ risulta che $(D,\mathcal{I},\mathcal{V}) \models (\exists x \forall y)(P(x,y))$ implica $(D,\mathcal{I},\mathcal{V}) \models (\forall y \exists x)(P(x,y))$. Prendiamo dunque un ambiente $(D,\mathcal{I},\mathcal{V})$ tale che $(D,\mathcal{I},\mathcal{V}) \models (\exists x \forall y)(P(x,y))$, cioè tale che esiste $d \in D$ per cui $(D,\mathcal{I},\mathcal{V} \setminus \{(x,\mathcal{V}(x)),(y,\mathcal{V}(y))\} \cup \{(x,d),(y,d')\}) \models P(x,y)$ per ogni $d' \in D$. Allora per ogni $d' \in D$ esiste $d'' \in D$ tale che $(D,\mathcal{I},\mathcal{V} \setminus \{(y,\mathcal{V}(y)),(x,\mathcal{V}(x))\} \cup \{(y,d'),(x,d'')\}) \models P(x,y)$ come si vede ponendo $d'' = d$ e quindi $(D,\mathcal{I},\mathcal{V}) \models (\forall y \exists x)(P(x,y))$. Dunque $(D,\mathcal{I},\mathcal{V})$ è un modello per l'intera formula e perciò quest'ultima è una tautologia in virtù della generalità del modello.
  - Verificare l'insoddisfacibilità per via semantica è spesso più facile che verificare la validità. Supponiamo che in un piccolo paese ci sia un barbiere che rade tutti e solo coloro che non si radono da soli. Indicato il barbiere con la costante b, il fatto che egli rade tutti coloro che non si radono da soli è formalizzato dalla formula $(\forall x)((\neg Rade(x,x)) \rightarrow Rade(b,x))$, mentre il fatto che egli rade solo coloro che non si radono da soli è formalizzato come $(\forall x)(Rade(b,x) \rightarrow (\neg Rade(x,x)))$. Dunque lo scenario è descritto dalla formula $(\forall x)(Rade(b,x) \leftrightarrow (\neg Rade(x,x)))$. La domanda è: chi rade il barbiere? Se il barbiere si rade da solo, allora il barbiere non può radersi perché egli rade solo coloro che non si radono da soli. Se il barbiere non si rade da solo, allora il barbiere deve radersi perché egli rade tutti coloro che non si radono da soli. Abbiamo dunque un paradosso. In effetti la formula $(\forall x)(Rade(b,x) \leftrightarrow (\neg Rade(x,x)))$ è una contraddizione perchè, preso un qualsiasi ambiente $(D,\mathcal{I},\mathcal{V})$, questo non può mai essere un modello della formula in quanto per $x = b$ abbiamo $(D,\mathcal{I},\mathcal{V} \setminus \{(x,\mathcal{V}(x))\} \cup \{(x,\mathcal{I}(b))\}) \not\models (Rade(b,x) \leftrightarrow (\neg Rade(x,x)))$.

- Il problema di stabilire se una formula ben formata della logica dei predicati sia soddisfacibile, una tautologia o una contraddizione è molto più complicato rispetto al caso della logica proposizionale. Questo è dovuto non solo al fatto che esistono infinite interpretazioni di una formula predicativa, ma anche al fatto che i domini delle interpretazioni possono essere insiemi infiniti e quindi stabilire la verità di formule quantificate può richiedere un numero illimitato di controlli. Il teorema di Church stabilisce che nella logica dei predicati il problema della validità non è decidibile ed il successivo teorema di Church, Herbrand, Skolem precisa che lo stesso problema è semi-decidibile (se una formula arbitraria è una tautologia allora è possibile stabilirlo in tempo finito tramite un algoritmo, ma se *non* lo è l'algoritmo potrebbe *non* terminare). Da ciò segue che il problema della insoddisfacibilità è pure semi-decidibile (perché ϕ è una contraddizione sse $(\neg\phi)$ è una tautologia) mentre il problema della soddisfacibilità non è nemmeno semi-decidibile (perché ϕ è soddisfacibile sse $(\neg\phi)$ *non* è una tautologia, ovvero sse ϕ *non* è una contraddizione). ■ftpl_10

10.3 Conseguenza ed equivalenza nella logica dei predicati

- Le nozioni di conseguenza ed equivalenza per la logica dei predicati sono definite come le corrispondenti nozioni per la logica proposizionale e quindi godono delle stesse proprietà.
- Dato un insieme di formule $\Phi \in 2^{FBF}$, denotiamo con $modelli(\Phi)$ l'insieme degli ambienti che soddisfano tutte le formule di Φ.
- La relazione di soddisfacimento $\models$ può essere trasformata in una relazione su 2^{FBF} detta relazione di conseguenza logica ponendo $\Phi \models \Gamma$ (o $\Phi \Rightarrow \Gamma$) sse $modelli(\Phi) \subseteq modelli(\Gamma)$ per ogni $\Phi, \Gamma \in 2^{FBF}$. Indichiamo con $\models \Gamma$ il fatto che tutte le formule di Γ siano delle tautologie.
- Teorema (di conseguenza logica): Dato $n \in \mathbb{N}$, siano $\Phi = \{\phi_1, \ldots, \phi_n\} \in 2^{FBF}$ e $\gamma \in FBF$. Allora $\Phi \models \{\gamma\}$ sse $((\bigwedge_{i=1}^{n} \phi_i) \rightarrow \gamma)$ è una tautologia, ovvero sse $((\bigwedge_{i=1}^{n} \phi_i) \wedge (\neg\gamma))$ è una contraddizione.
- Corollario: Siano $\phi, \gamma \in FBF$. Allora $\{\phi\} \models \{\gamma\}$ sse $\models \{(\phi \rightarrow \gamma)\}$.
- La relazione di equivalenza logica è una relazione su 2^{FBF} definita ponendo $\Phi \equiv \Gamma$ (o $\Phi \Leftrightarrow \Gamma$) sse $\Phi \models \Gamma$ e $\Gamma \models \Phi$, cioè sse $modelli(\Phi) = modelli(\Gamma)$.
- Teorema (di equivalenza logica): Dato $n \in \mathbb{N}$, siano $\Phi = \{\phi_1, \ldots, \phi_n\} \in 2^{FBF}$ e $\gamma \in FBF$. Allora $\Phi \equiv \{\gamma\}$ sse $((\bigwedge_{i=1}^{n} \phi_i) \leftrightarrow \gamma)$ è una tautologia.
- Corollario: Siano $\phi, \gamma \in FBF$. Allora $\{\phi\} \equiv \{\gamma\}$ sse $\models \{(\phi \leftrightarrow \gamma)\}$.
- La relazione di equivalenza logica ristretta ad FBF, cioè ad insiemi singoletto (che verranno da ora in poi denotati senza parentesi graffe), risulta essere una congruenza rispetto a tutti i connettivi logici e ai quantificatori. Per questi ultimi vale anche una proprietà di ridenominazione delle variabili legate.
- Teorema (di sostituzione): Siano $\phi_1, \phi_2 \in FBF$. Se $\phi_1 \equiv \phi_2$ allora:
  - $(\neg\phi_1) \equiv (\neg\phi_2)$.
  - $(\phi_1 \vee \gamma) \equiv (\phi_2 \vee \gamma)$ e $(\gamma \vee \phi_1) \equiv (\gamma \vee \phi_2)$ per ogni $\gamma \in FBF$.
  - $(\phi_1 \wedge \gamma) \equiv (\phi_2 \wedge \gamma)$ e $(\gamma \wedge \phi_1) \equiv (\gamma \wedge \phi_2)$ per ogni $\gamma \in FBF$.
  - $(\phi_1 \rightarrow \gamma) \equiv (\phi_2 \rightarrow \gamma)$ e $(\gamma \rightarrow \phi_1) \equiv (\gamma \rightarrow \phi_2)$ per ogni $\gamma \in FBF$.
  - $(\phi_1 \leftrightarrow \gamma) \equiv (\phi_2 \leftrightarrow \gamma)$ e $(\gamma \leftrightarrow \phi_1) \equiv (\gamma \leftrightarrow \phi_2)$ per ogni $\gamma \in FBF$.
  - $(\forall x)(\phi_1) \equiv (\forall x)(\phi_2)$ per ogni $x \in Var$.
  - $(\exists x)(\phi_1) \equiv (\exists x)(\phi_2)$ per ogni $x \in Var$.
- Teorema: Siano $\phi \in FBF$ e $y \in Var$. Se $y \notin varlib(\phi)$ allora per ogni $x \in Var$:
  - $(\forall x)(\phi) \equiv (\forall y)(\phi[^{y}/_{x}])$.
  - $(\exists x)(\phi) \equiv (\exists y)(\phi[^{y}/_{x}])$.

- Dimostrazione: Il risultato segue dal fatto che per ogni ambiente $(D,\mathcal{I},\mathcal{V})$ e per ogni $d \in D$ risulta $(D,\mathcal{I},\mathcal{V}\setminus\{(x,\mathcal{V}(x))\}\cup\{(x,d)\}) \models \phi$ sse $(D,\mathcal{I},\mathcal{V}\setminus\{(y,\mathcal{V}(y))\}\cup\{(y,d)\}) \models \phi[^y/_x]$ qualora $y \notin varlib(\phi)$. Dimostriamo questo fatto procedendo per induzione sulla struttura sintattica di ϕ tale che $y \notin varlib(\phi)$, dopo aver osservato che se ϕ non è una formula quantificata allora $y \notin varlib(\phi)$ implica $y \notin varlib(\phi')$ per ogni sottoformula immediata ϕ' di ϕ:
  - Se ϕ è della forma $P(t_1,\ldots,t_n)$ con $P \in Pred$ e $t_1,\ldots,t_n \in Ter$ allora per ogni ambiente $(D,\mathcal{I},\mathcal{V})$ e per ogni $d \in D$ risulta $(D,\mathcal{I},\mathcal{V}\setminus\{(x,\mathcal{V}(x))\}\cup\{(x,d)\}) \models \phi$ sse $(D,\mathcal{I},\mathcal{V}\setminus\{(y,\mathcal{V}(y))\}\cup\{(y,d)\}) \models \phi[^y/_x]$ perché $y \notin varlib(\phi)$ e quindi $y \notin var(\phi)$ essendo $var(\phi) = varlib(\phi)$.
  - Sia ϕ della forma $(\neg\phi')$ e supponiamo che per ogni ambiente $(D,\mathcal{I},\mathcal{V})$ e per ogni $d \in D$ risulti $(D,\mathcal{I},\mathcal{V}\setminus\{(x,\mathcal{V}(x))\} \cup \{(x,d)\}) \models \phi'$ sse $(D,\mathcal{I},\mathcal{V}\setminus\{(y,\mathcal{V}(y))\} \cup \{(y,d)\}) \models \phi'[^y/_x]$. Allora per ogni ambiente $(D,\mathcal{I},\mathcal{V})$ e per ogni $d \in D$ abbiamo che $(D,\mathcal{I},\mathcal{V}\setminus\{(x,\mathcal{V}(x))\} \cup \{(x,d)\}) \models \phi$ sse $(D,\mathcal{I},\mathcal{V}\setminus\{(x,\mathcal{V}(x))\}\cup\{(x,d)\}) \not\models \phi'$, cioè sse $(D,\mathcal{I},\mathcal{V}\setminus\{(y,\mathcal{V}(y))\}\cup\{(y,d)\}) \not\models \phi'[^y/_x]$ sfruttando l'ipotesi induttiva, cioè sse $(D,\mathcal{I},\mathcal{V}\setminus\{(y,\mathcal{V}(y))\} \cup \{(y,d)\}) \models \phi[^y/_x]$.
  - Sia ϕ della forma $(\phi'_1 \vee \phi'_2)$ e supponiamo che per ogni ambiente $(D,\mathcal{I},\mathcal{V})$ e per ogni $d \in D$ risulti $(D,\mathcal{I},\mathcal{V}\setminus\{(x,\mathcal{V}(x))\} \cup \{(x,d)\}) \models \phi'_i$ sse $(D,\mathcal{I},\mathcal{V}\setminus\{(y,\mathcal{V}(y))\} \cup \{(y,d)\}) \models \phi'_i[^y/_x]$ per $i \in \{1,2\}$. Allora per ogni ambiente $(D,\mathcal{I},\mathcal{V})$ e per ogni $d \in D$ abbiamo che $(D,\mathcal{I},\mathcal{V}\setminus\{(x,\mathcal{V}(x))\}\cup\{(x,d)\}) \models \phi$ sse $(D,\mathcal{I},\mathcal{V}\setminus\{(x,\mathcal{V}(x))\} \cup \{(x,d)\}) \models \phi'_1$ o $(D,\mathcal{I},\mathcal{V}\setminus\{(x,\mathcal{V}(x))\} \cup \{(x,d)\}) \models \phi'_2$, cioè sse $(D,\mathcal{I},\mathcal{V}\setminus\{(y,\mathcal{V}(y))\} \cup \{(y,d)\}) \models \phi'_1[^y/_x]$ o $(D,\mathcal{I},\mathcal{V}\setminus\{(y,\mathcal{V}(y))\} \cup \{(y,d)\}) \models \phi'_2[^y/_x]$ sfruttando l'ipotesi induttiva, cioè sse $(D,\mathcal{I},\mathcal{V}\setminus\{(y,\mathcal{V}(y))\} \cup \{(y,d)\}) \models \phi[^y/_x]$.
  - Sia ϕ della forma $(\phi'_1 \wedge \phi'_2)$ e supponiamo che per ogni ambiente $(D,\mathcal{I},\mathcal{V})$ e per ogni $d \in D$ risulti $(D,\mathcal{I},\mathcal{V}\setminus\{(x,\mathcal{V}(x))\} \cup \{(x,d)\}) \models \phi'_i$ sse $(D,\mathcal{I},\mathcal{V}\setminus\{(y,\mathcal{V}(y))\} \cup \{(y,d)\}) \models \phi'_i[^y/_x]$ per $i \in \{1,2\}$. Allora per ogni ambiente $(D,\mathcal{I},\mathcal{V})$ e per ogni $d \in D$ abbiamo che $(D,\mathcal{I},\mathcal{V}\setminus\{(x,\mathcal{V}(x))\}\cup\{(x,d)\}) \models \phi$ sse $(D,\mathcal{I},\mathcal{V}\setminus\{(x,\mathcal{V}(x))\} \cup \{(x,d)\}) \models \phi'_1$ e $(D,\mathcal{I},\mathcal{V}\setminus\{(x,\mathcal{V}(x))\} \cup \{(x,d)\}) \models \phi'_2$, cioè sse $(D,\mathcal{I},\mathcal{V}\setminus\{(y,\mathcal{V}(y))\} \cup \{(y,d)\}) \models \phi'_1[^y/_x]$ e $(D,\mathcal{I},\mathcal{V}\setminus\{(y,\mathcal{V}(y))\} \cup \{(y,d)\}) \models \phi'_2[^y/_x]$ sfruttando l'ipotesi induttiva, cioè sse $(D,\mathcal{I},\mathcal{V}\setminus\{(y,\mathcal{V}(y))\} \cup \{(y,d)\}) \models \phi[^y/_x]$.
  - Sia ϕ della forma $(\phi'_1 \to \phi'_2)$ e supponiamo che per ogni ambiente $(D,\mathcal{I},\mathcal{V})$ e per ogni $d \in D$ risulti $(D,\mathcal{I},\mathcal{V}\setminus\{(x,\mathcal{V}(x))\} \cup \{(x,d)\}) \models \phi'_i$ sse $(D,\mathcal{I},\mathcal{V}\setminus\{(y,\mathcal{V}(y))\} \cup \{(y,d)\}) \models \phi'_i[^y/_x]$ per $i \in \{1,2\}$. Allora per ogni ambiente $(D,\mathcal{I},\mathcal{V})$ e per ogni $d \in D$ abbiamo che $(D,\mathcal{I},\mathcal{V}\setminus\{(x,\mathcal{V}(x))\}\cup\{(x,d)\}) \models \phi$ sse $(D,\mathcal{I},\mathcal{V}\setminus\{(x,\mathcal{V}(x))\} \cup \{(x,d)\}) \not\models \phi'_1$ oppure $(D,\mathcal{I},\mathcal{V}\setminus\{(x,\mathcal{V}(x))\} \cup \{(x,d)\}) \models \phi'_2$, cioè sse $(D,\mathcal{I},\mathcal{V}\setminus\{(y,\mathcal{V}(y))\} \cup \{(y,d)\}) \not\models \phi'_1[^y/_x]$ oppure $(D,\mathcal{I},\mathcal{V}\setminus\{(y,\mathcal{V}(y))\} \cup \{(y,d)\}) \models \phi'_2[^y/_x]$ sfruttando l'ipotesi induttiva, cioè sse $(D,\mathcal{I},\mathcal{V}\setminus\{(y,\mathcal{V}(y))\} \cup \{(y,d)\}) \models \phi[^y/_x]$.
  - Sia ϕ della forma $(\phi'_1 \leftrightarrow \phi'_2)$ e supponiamo che per ogni ambiente $(D,\mathcal{I},\mathcal{V})$ e per ogni $d \in D$ risulti $(D,\mathcal{I},\mathcal{V}\setminus\{(x,\mathcal{V}(x))\} \cup \{(x,d)\}) \models \phi'_i$ sse $(D,\mathcal{I},\mathcal{V}\setminus\{(y,\mathcal{V}(y))\} \cup \{(y,d)\}) \models \phi'_i[^y/_x]$ per $i \in \{1,2\}$. Allora per ogni ambiente $(D,\mathcal{I},\mathcal{V})$ e per ogni $d \in D$ abbiamo che $(D,\mathcal{I},\mathcal{V}\setminus\{(x,\mathcal{V}(x))\}\cup\{(x,d)\}) \models \phi$ sse $(D,\mathcal{I},\mathcal{V}\setminus\{(x,\mathcal{V}(x))\} \cup \{(x,d)\}) \models \phi'_1$ e $(D,\mathcal{I},\mathcal{V}\setminus\{(x,\mathcal{V}(x))\} \cup \{(x,d)\}) \models \phi'_2$ oppure $(D,\mathcal{I},\mathcal{V}\setminus\{(x,\mathcal{V}(x))\} \cup \{(x,d)\}) \not\models \phi'_1$ e $(D,\mathcal{I},\mathcal{V}\setminus\{(x,\mathcal{V}(x))\} \cup \{(x,d)\}) \not\models \phi'_2$, cioè sse $(D,\mathcal{I},\mathcal{V}\setminus\{(y,\mathcal{V}(y))\} \cup \{(y,d)\}) \models \phi'_1[^y/_x]$ e $(D,\mathcal{I},\mathcal{V}\setminus\{(y,\mathcal{V}(y))\} \cup \{(y,d)\}) \models \phi'_2[^y/_x]$ oppure $(D,\mathcal{I},\mathcal{V}\setminus\{(y,\mathcal{V}(y))\} \cup \{(y,d)\}) \not\models \phi'_1[^y/_x]$ e $(D,\mathcal{I},\mathcal{V}\setminus\{(y,\mathcal{V}(y))\} \cup \{(y,d)\}) \not\models \phi'_2[^y/_x]$ sfruttando l'ipotesi induttiva, cioè sse $(D,\mathcal{I},\mathcal{V}\setminus\{(y,\mathcal{V}(y))\} \cup \{(y,d)\}) \models \phi[^y/_x]$.
  - Sia ϕ della forma $(\forall z)(\phi')$ oppure $(\exists z)(\phi')$. Ci sono tre casi:
    - Se $x = z$ allora $\phi[^y/_x] = \phi$ e quindi per ogni ambiente $(D,\mathcal{I},\mathcal{V})$ e per ogni $d \in D$ risulta $(D,\mathcal{I},\mathcal{V}\setminus\{(x,\mathcal{V}(x))\} \cup \{(x,d)\}) \models \phi$ sse $(D,\mathcal{I},\mathcal{V}\setminus\{(y,\mathcal{V}(y))\} \cup \{(y,d)\}) \models \phi[^y/_x]$ perché il terzo elemento dell'ambiente riguarda le occorrenze libere di variabili mentre $x, y \notin varlib(\phi)$.
    - Siano $x \neq z$ e $z \neq y$, cosicché $y \notin varlib(\phi')$, e supponiamo che per ogni ambiente $(D,\mathcal{I},\mathcal{V})$ e per ogni $d \in D$ risulti $(D,\mathcal{I},\mathcal{V}\setminus\{(x,\mathcal{V}(x))\} \cup \{(x,d)\}) \models \phi'$ sse $(D,\mathcal{I},\mathcal{V}\setminus\{(y,\mathcal{V}(y))\} \cup \{(y,d)\}) \models \phi'[^y/_x]$. Ci sono due sottocasi:
      - Se ϕ è della forma $(\forall z)(\phi')$ allora per ogni ambiente $(D,\mathcal{I},\mathcal{V})$ e per ogni $d \in D$ risulta $(D,\mathcal{I},\mathcal{V}\setminus\{(x,\mathcal{V}(x))\} \cup \{(x,d)\}) \models \phi$ sse $(D,\mathcal{I},\mathcal{V}_{d'}\setminus\{(x,\mathcal{V}_{d'}(x))\} \cup \{(x,d)\}) \models \phi'$ per ogni $d' \in D$ con $\mathcal{V}_{d'} = \mathcal{V}\setminus\{(z,\mathcal{V}(z))\} \cup \{(z,d')\}$, cioè sse $(D,\mathcal{I},\mathcal{V}_{d'}\setminus\{(y,\mathcal{V}_{d'}(y))\} \cup \{(y,d)\}) \models \phi'[^y/_x]$ per ogni $d' \in D$ sfruttando l'ipotesi induttiva, cioè sse $(D,\mathcal{I},\mathcal{V}\setminus\{(y,\mathcal{V}(y))\} \cup \{(y,d)\}) \models \phi[^y/_x]$.

· Se ϕ è della forma $(\exists z)(\phi')$ allora per ogni ambiente $(D,\mathcal{I},\mathcal{V})$ e per ogni $d \in D$ risulta $(D,\mathcal{I},\mathcal{V}\setminus\{(x,\mathcal{V}(x))\}\cup\{(x,d)\}) \models \phi$ sse esiste $d' \in D$ tale che $(D,\mathcal{I},\mathcal{V}_{d'}\setminus\{(x,\mathcal{V}_{d'}(x))\}\cup\{(x,d)\}) \models \phi'$ con $\mathcal{V}_{d'} = \mathcal{V}\setminus\{(z,\mathcal{V}(z))\}\cup\{(z,d')\}$, cioè sse esiste $d' \in D$ tale che $(D,\mathcal{I},\mathcal{V}_{d'}\setminus\{(y,\mathcal{V}_{d'}(y))\}\cup\{(y,d)\}) \models \phi'[{}^{y}/_{x}]$ sfruttando l'ipotesi induttiva, cioè sse $(D,\mathcal{I},\mathcal{V}\setminus\{(y,\mathcal{V}(y))\}\cup\{(y,d)\}) \models \phi[{}^{y}/_{x}]$.

∗ Se $x \neq z$ e $z = y$ allora $(\forall z)(\phi')[{}^{y}/_{x}] = (\forall z')(\phi'_{z'}[{}^{y}/_{x}])$ ed $(\exists z)(\phi')[{}^{y}/_{x}] = (\exists z')(\phi'_{z'}[{}^{y}/_{x}])$ dove $\phi'_{z'} = \phi'[{}^{z'}/_{z}]$ con $z' \notin \{x,y\}\cup var(\phi')$. Indicata con $\phi_{z'}$ la formula $(\forall z')(\phi'_{z'})$ oppure $(\exists z')(\phi'_{z'})$, si procede come in uno dei due casi precedenti per dimostrare che per ogni ambiente $(D,\mathcal{I},\mathcal{V})$ e per ogni $d \in D$ risulta:

· $(D,\mathcal{I},\mathcal{V}\setminus\{(z,\mathcal{V}(z))\}\cup\{(z,d)\}) \models \phi$ sse $(D,\mathcal{I},\mathcal{V}\setminus\{(z',\mathcal{V}(z'))\}\cup\{(z',d)\}) \models \phi_{z'}$ perché $z' \notin var(\phi')$ implica $z' \notin varlib(\phi')$ e $z = y \neq z'$ implica $z \neq z'$.

· $(D,\mathcal{I},\mathcal{V}\setminus\{(x,\mathcal{V}(x))\}\cup\{(x,d)\}) \models \phi_{z'}$ sse $(D,\mathcal{I},\mathcal{V}\setminus\{(y,\mathcal{V}(y))\}\cup\{(y,d)\}) \models \phi_{z'}[{}^{y}/_{x}]$ perché $y \notin varlib(\phi'_{z'})$ e $z' \neq y$.

Osservato che $z' \notin \{x,y\}$ implica $\phi_{z'}[{}^{y}/_{x}] = \phi[{}^{y}/_{x}]$, il risultato segue dai due fatti sopra.

10.4 Proprietà algebriche dei quantificatori

- Nella logica dei predicati i connettivi logici godono delle stesse proprietà algebriche di cui godono nella logica proposizionale. Inoltre vi sono ulteriori proprietà algebriche che riguardano i quantificatori.
- Teorema: Valgono le seguenti proprietà algebriche:
  - $(\forall x)(\phi) \equiv \bigwedge_{t\in Ter}^{var(t)=\emptyset} \phi[{}^{t}/_{x}]$.
  - $(\exists x)(\phi) \equiv \bigvee_{t\in Ter}^{var(t)=\emptyset} \phi[{}^{t}/_{x}]$.
  - $(\neg(\forall x)(\phi)) \equiv (\exists x)(\neg\phi)$.
  - $(\neg(\exists x)(\phi)) \equiv (\forall x)(\neg\phi)$.
  - $(\forall x)(\phi) \equiv (\neg(\exists x)(\neg\phi))$.
  - $(\exists x)(\phi) \equiv (\neg(\forall x)(\neg\phi))$.
  - $(\forall x\forall y)(\phi) \equiv (\forall y\forall x)(\phi)$.
  - $(\exists x\exists y)(\phi) \equiv (\exists y\exists x)(\phi)$.
  - $(\forall x)(\phi) \equiv \phi$ se $x \notin varlib(\phi)$.
  - $(\exists x)(\phi) \equiv \phi$ se $x \notin varlib(\phi)$.
  - $(\forall x)(\phi_1 \wedge \phi_2) \equiv ((\forall x)(\phi_1) \wedge (\forall x)(\phi_2))$.
  - $(\exists x)(\phi_1 \vee \phi_2) \equiv ((\exists x)(\phi_1) \vee (\exists x)(\phi_2))$.
  - $(\forall x)(\phi_1 \vee \phi_2) \equiv ((\forall x)(\phi_1) \vee \phi_2)$ se $x \notin varlib(\phi_2)$.
  - $(\exists x)(\phi_1 \wedge \phi_2) \equiv ((\exists x)(\phi_1) \wedge \phi_2)$ se $x \notin varlib(\phi_2)$.
  - $((\natural_1 x)(\phi_1) \vee (\natural_2 x)(\phi_2)) \equiv (\natural_1 x\natural_2 y)(\phi_1 \vee \phi_2[{}^{y}/_{x}])$ se $y \notin varlib(\phi_1)\cup varlib(\phi_2)$, con $\natural_1,\natural_2 \in \{\forall,\exists\}$.
  - $((\natural_1 x)(\phi_1) \wedge (\natural_2 x)(\phi_2)) \equiv (\natural_1 x\natural_2 y)(\phi_1 \wedge \phi_2[{}^{y}/_{x}])$ se $y \notin varlib(\phi_1)\cup varlib(\phi_2)$, con $\natural_1,\natural_2 \in \{\forall,\exists\}$.
  - $((\forall x)(\phi_1) \to \phi_2) \equiv (\exists x)(\phi_1 \to \phi_2)$ se $x \notin varlib(\phi_2)$.
  - $((\exists x)(\phi_1) \to \phi_2) \equiv (\forall x)(\phi_1 \to \phi_2)$ se $x \notin varlib(\phi_2)$.
  - $(\phi_1 \to (\forall x)(\phi_2)) \equiv (\forall x)(\phi_1 \to \phi_2)$ se $x \notin varlib(\phi_1)$.
  - $(\phi_1 \to (\exists x)(\phi_2)) \equiv (\exists x)(\phi_1 \to \phi_2)$ se $x \notin varlib(\phi_1)$.
- Dimostrazione: In virtù del teorema di equivalenza logica, per ogni proprietà si tratta di verificare tramite l'applicazione della definizione della relazione di soddisfacimento che un generico ambiente soddisfa la formula di sinistra sse soddisfa la formula di destra.

- Esempi:
  - La proprietà $(\forall x)(\phi_1 \vee \phi_2) \equiv ((\forall x)(\phi_1) \vee (\forall x)(\phi_2))$ non vale. Si ottiene un controesempio nel dominio dei numeri naturali prendendo $Pari(x)$ come ϕ_1 e $Dispari(x)$ come ϕ_2: mentre è vero che ogni numero è pari o dispari, non è vero che ogni numero è pari oppure che ogni numero è dispari.
  - La proprietà $(\exists x)(\phi_1 \wedge \phi_2) \equiv ((\exists x)(\phi_1) \wedge (\exists x)(\phi_2))$ non vale. Si ottiene un controesempio nel dominio dei numeri naturali prendendo di nuovo $Pari(x)$ come ϕ_1 e $Dispari(x)$ come ϕ_2: mentre non è vero che esiste un numero che è sia pari che dispari, è vero che esiste un numero pari e che esiste un numero dispari.

10.5 Sistemi deduttivi per la logica dei predicati

- Il sistema di deduzione naturale di Gentzen si estende alla logica dei predicati attraverso le seguenti regole di inferenza aggiuntive:
  - Regole che introducono quantificatori rispetto alle premesse:
    $$\frac{P[^{y}/_{x}]}{(\forall x)(P)} \qquad \frac{P[^{t}/_{x}]}{(\exists x)(P)}$$
    dove il simbolo di predicato P denota una formula atomica in cui y non occorre libera. La prima regola introduce un quantificatore universale perché y è una generica variabile che sostituisce x, mentre la seconda regola introduce una quantificazione esistenziale perché t è uno specifico termine che sostituisce x.
  - Regole che eliminano quantificatori rispetto alle premesse:
    $$\frac{(\forall x)(P)}{P[^{t}/_{x}]} \qquad \frac{(\exists x)(P) \quad [P[^{y}/_{x}]]Q}{Q}$$
    dove i simboli di predicato P e Q denotano due formule atomiche in cui y non occorre libera. La prima regola è simile a quella per eliminare $\wedge$, mentre la seconda regola è simile a quella per eliminare $\vee$ ed è condizionale perché da $(\exists x)(P)$ non si può inferire una specifica variante di P, ma ogni Q valida sotto qualsiasi variante di P.
- Teorema: Il sistema di deduzione naturale di Gentzen esteso è corretto e completo rispetto alla logica dei predicati.
- Un sistema deduttivo alla Hilbert si estende alla logica dei predicati attraverso la seguente regola di inferenza aggiuntiva detta regola di generalizzazione:
  $$\frac{P[^{y}/_{x}]}{(\forall x)(P)}$$
  dove il simbolo di predicato P denota una formula atomica in cui y non occorre libera, assieme ai seguenti assiomi aggiuntivi:
  $$\frac{}{(P[^{t}/_{x}] \rightarrow (\exists x)(P))}$$
  $$\frac{}{((\forall x)(P) \rightarrow P[^{t}/_{x}])}$$
  $$\frac{}{((\forall x)(P \rightarrow Q) \rightarrow ((\exists x)(P) \rightarrow Q))}$$
  $$\frac{}{((\forall x)(Q \rightarrow P) \rightarrow (Q \rightarrow (\forall x)(P)))}$$
  dove i simboli di predicato P e Q denotano due formule atomiche tali che x non occorre libera in Q.

- Teorema: Il sistema deduttivo alla Hilbert esteso è corretto e completo rispetto alla logica dei predicati.
- Teorema (di deduzione): Siano $\Phi \in 2^{FBF}$ finito e $\phi, \gamma \in FBF$. Se $\Phi \cup \{\phi\} \vdash_{\text{HE}} \gamma$, allora $\Phi \vdash_{\text{HE}} (\phi \to \gamma)$.

■ftpl_11

Capitolo 11

Programmazione logica: il linguaggio Prolog

11.1 Forme normali per logica proposizionale e dei predicati

- Il problema di stabilire se una formula ben formata è soddisfacibile, valida (tautologia) o insoddisfacibile (contraddizione) è risolvibile in tempo esponenziale nel caso della logica proposizionale ed è addirittura indecidibile nel caso della logica dei predicati. Una tecnica per affrontare l'elevato costo computazionale o l'indecidibilità che caratterizzano l'analisi della verità di una formula ben formata consiste nel trasformare la formula in una formula ad essa equivalente che è espressa in una forma più semplice da studiare, la quale di solito prende il nome di forma normale.

- Denotiamo con FBF_{prop} ed FBF_{pred} l'insieme delle formule ben formate della logica proposizionale e della logica dei predicati, rispettivamente. Limitiamo inoltre l'uso delle parentesi al minimo.

- Data una formula $\phi \in FBF_{\text{prop}}$, diciamo che essa è in:

  - forma normale disgiuntiva sse è della forma $\bigvee_{i=1}^{n} (\bigwedge_{j=1}^{m_i} \lambda_{i,j})$ con $n \geq 1$ ed $m_i \geq 1$ per ogni $1 \leq i \leq n$
  - forma normale congiuntiva sse è della forma $\bigwedge_{i=1}^{n} (\bigvee_{j=1}^{m_i} \lambda_{i,j})$ con $n \geq 1$ ed $m_i \geq 1$ per ogni $1 \leq i \leq n$

  dove ciascun letterale $\lambda_{i,j}$ è della forma p oppure $\neg p$ con $p \in \mathit{Prop}$.

- Grazie all'associatività e alla commutatività dei connettivi logici $\vee$ e $\wedge$ (vedi Sez. 9.4), l'ordine dei letterali all'interno di una forma normale disgiuntiva o congiuntiva è irrilevante. Poiché l'insieme di connettivi logici $\{\vee, \wedge, \neg\}$ è funzionalmente completo (vedi Sez. 9.4), ogni formula ben formata della logica proposizionale può essere trasformata in forma normale disgiuntiva e in forma normale congiuntiva. La trasformazione sfrutta il teorema di sostituzione (vedi Sez. 9.3) e procede nel seguente modo:

  1. Rimpiazzare tutti i connettivi logici diversi da $\vee$, $\wedge$, $\neg$ applicando le leggi di derivabilità semantica (vedi Sez. 9.4).
  2. Usare ripetutamente la legge di doppia inversione e le leggi di De Morgan (vedi Sez. 9.4) per portare tutti i connettivi logici $\neg$ immediatamente davanti alle proposizioni.
  3. Usare ripetutamente la distributività dei connettivi logici $\vee$ e $\wedge$ (vedi Sez. 9.4) per ottenere una disgiunzione di congiunzioni o una congiunzione di disgiunzioni a seconda della forma normale.

- Esempio: La formula $(p \vee \neg q) \rightarrow r$ viene trasformata prima in $\neg(p \vee \neg q) \vee r$ e poi in $(\neg p \wedge q) \vee (r)$ che è già in forma normale disgiuntiva. Applicando la proprietà distributiva si ottiene l'equivalente forma normale congiuntiva $(\neg p \vee r) \wedge (q \vee r)$.

- Un algoritmo alternativo per trasformare una formula ben formata della logica proposizionale in forma normale disgiuntiva o congiuntiva è il seguente:

  1. Costruire la tabella di verità della formula.

  $2^{\vee}$. Nel caso della forma normale disgiuntiva, per ogni riga il cui ultimo elemento è 1 formare una congiunzione considerando per ogni proposizione p come letterale p stesso oppure $\neg p$ a seconda che il valore di verità assegnato a p in quella riga sia 1 oppure 0, poi prendere la disgiunzione di tutte queste congiunzioni.

  $2^{\wedge}$. Nel caso della forma normale congiuntiva, per ogni riga il cui ultimo elemento è 0 formare una disgiunzione considerando per ogni proposizione p come letterale p stesso oppure $\neg p$ a seconda che il valore di verità assegnato a p in quella riga sia 0 oppure 1, poi prendere la congiunzione di tutte queste disgiunzioni.

- Esempio: Costruita la seguente tabella di verità per $(p \vee \neg q) \rightarrow r$:

$\mathcal{I}_A(p)$	$\mathcal{I}_A(q)$	$\mathcal{I}_A(r)$	$\mathcal{I}_A(p \vee \neg q)$	$\mathcal{I}_A((p \vee \neg q) \rightarrow r)$
1	1	1	1	1
1	1	0	1	0
1	0	1	1	1
1	0	0	1	0
0	1	1	0	1
0	1	0	0	1
0	0	1	1	1
0	0	0	1	0

  la forma normale disgiuntiva è $(p \wedge q \wedge r) \vee (p \wedge \neg q \wedge r) \vee (\neg p \wedge q \wedge r) \vee (\neg p \wedge q \wedge \neg r) \vee (\neg p \wedge \neg q \wedge r)$ mentre la forma normale congiuntiva è $(\neg p \vee \neg q \vee r) \wedge (\neg p \vee q \vee r) \wedge (p \vee q \vee r)$. Queste forme normali sono più complesse di quelle ottenute nell'esempio precedente.

- Sebbene il problema della (in)soddisfacibilità continui ad essere risolvibile in tempo esponenziale per formule in forma normale congiuntiva, *la (in)soddisfacibilità di una formula in forma normale disgiuntiva può essere stabilita in tempo polinomiale*. Infatti una formula siffatta è soddisfacibile sse contiene almeno una congiunzione di letterali che è soddisfacibile, cioè che non comprende due letterali della forma p e $\neg p$. Questo fatto può certamente essere verificato impiegando un tempo che cresce linearmente con il numero di letterali presenti nella formula, senza costruire la tabella di verità.

- Il problema della validità rimane risolvibile in tempo esponenziale per le formule in forma normale disgiuntiva. Infatti una formula siffatta è valida sse la sua negazione è insoddisfacibile, ma quest'ultima formula diventa immediatamente in forma normale congiuntiva applicando le leggi di De Morgan e quindi sappiamo stabilirne la (in)soddisfacibilità solo in tempo esponenziale. Invece *la validità di una formula in forma normale congiuntiva può essere stabilita in tempo polinomiale*. Infatti una formula siffatta è valida sse ogni sua disgiunzione di letterali è valida, cioè comprende due letterali della forma p e $\neg p$. Questo fatto può certamente essere verificato impiegando un tempo che cresce linearmente con il numero di letterali presenti nella formula, senza costruire la tabella di verità.

- Diciamo che un letterale λ è positivo oppure negativo a seconda che sia della forma p oppure $\neg p$ con $p \in \mathit{Prop}$. Chiamiamo clausola una disgiunzione di letterali. Chiamiamo poi clausola di Horn una clausola che contiene al più un letterale positivo, la quale viene classificata come:

  – Un fatto sse è della forma p, che è equivalente a $\mathbb{I} \rightarrow p$.

  – Un vincolo sse è della forma $\bigvee_{i=1}^{n} \neg p_i$, che è equivalente a $(\bigwedge_{i=1}^{n} p_i) \rightarrow \mathbb{O}$.

  – Una regola sse è della forma $p \vee (\bigvee_{i=1}^{n} \neg p_i)$, che è equivalente a $(\bigwedge_{i=1}^{n} p_i) \rightarrow p$.

- Una formula in forma normale congiuntiva è detta essere una formula di Horn sse ogni sua clausola è una clausola di Horn. Osserviamo che:
  - Una formula di Horn priva di fatti è soddisfacibile, come si vede prendendo un qualsiasi assegnamento di verità che non contiene nessuna delle proposizioni che compaiono nella formula.
  - Una formula di Horn priva di vincoli è soddisfacibile, come si vede prendendo un qualsiasi assegnamento di verità che contiene tutte le proposizioni che compaiono nei letterali positivi della formula.
- *La (in)soddisfacibilità di una formula in forma normale congiuntiva che è anche una formula di Horn può essere stabilita in tempo polinomiale* attraverso il seguente algoritmo:
  1. Trasformare la formula ϕ nella formula equivalente in cui ogni clausola viene ad avere il connettivo logico $\rightarrow$ come connettivo primario e tutti i letterali in forma positiva.
  2. Prendere un assegnamento di verità A inizialmente vuoto.
  3. Aggiungere ad A ogni proposizione p che compare come fatto in ϕ.
  4. Finché A non è stabile, per ogni regola $(\bigwedge_{i=1}^{n} p_i) \rightarrow p$ presente in ϕ tale che ogni $p_i \in A$ ma $p \notin A$, aggiungere p ad A.
  5. Se esiste un vincolo $(\bigwedge_{i=1}^{n} p_i) \rightarrow \mathbb{O}$ in ϕ tale che ogni $p_i \in A$, allora ϕ è insoddisfacibile, altrimenti ϕ è soddisfatta da A.

  Se ϕ è soddisfacibile, allora A è il minimo assegnamento di verità che soddisfa ϕ.
- Esempio: La formula di Horn $(p \vee \neg q \vee \neg r) \wedge (\neg s \vee q) \wedge (\neg s \vee r) \wedge (s) \wedge (\neg p' \vee \neg s)$, che è equivalente a $((q \wedge r) \rightarrow p) \wedge (s \rightarrow q) \wedge (s \rightarrow r) \wedge (\mathbb{I} \rightarrow s) \wedge ((p' \wedge s) \rightarrow \mathbb{O})$, è soddisfacibile perché risulta:
  - $A = \{s\}$ dopo aver esaminato l'unico fatto presente nella formula.
  - $A = \{s, q, r\}$ dopo aver esaminato per la prima volta le tre regole presenti nella formula.
  - $A = \{s, q, r, p\}$ dopo aver esaminato per la seconda volta le tre regole presenti nella formula.
  - $p' \notin A$ per quanto riguarda l'unico vincolo presente nella formula.
- Data una formula $\phi \in FBF_{\text{pred}}$, diciamo che essa è in forma normale rettificata sse non ha variabili che occorrono sia legate che libere – cioè $varleg(\phi) \cap varlib(\phi) = \emptyset$ – e non ha molteplici quantificatori che si riferiscono alla stessa variabile. In virtù dell'ultimo teorema della Sez. 10.3, ogni formula ben formata della logica dei predicati può essere trasformata in forma normale rettificata, così da evitare confusione tra le variabili.
- Data una formula $\phi \in FBF_{\text{pred}}$, diciamo che essa è in forma normale prenessa sse è della forma $\natural_1 x_1 \ldots \natural_n x_n \, \phi'$ dove $n \geq 0$, ogni $\natural_i \in \{\forall, \exists\}$ e ϕ' è priva di quantificatori. La sequenza di quantificatori è detto il prefisso della formula, mentre ϕ' è detta la matrice della formula. In virtù delle proprietà algebriche dei quantificatori (vedi Sez. 10.4) e del teorema di sostituzione (vedi Sez. 10.3), ogni formula ben formata della logica dei predicati può essere trasformata in forma normale prenessa, così da raccogliere in testa tutti i quantificatori.
- Esempio: La formula $\forall x \, P(x) \rightarrow \neg \forall y \, Q(y)$ è equivalente a $\forall x \, P(x) \rightarrow \exists y \, \neg Q(y)$, che è equivalente a $\exists y \, (\forall x \, P(x) \rightarrow \neg Q(y))$, che è equivalente a $\exists y \exists x \, (P(x) \rightarrow \neg Q(y))$, che è in forma normale prenessa.
- Data una formula $\phi \in FBF_{\text{pred}}$, diciamo che essa è in forma normale di Skolem sse è della forma $\forall x_1 \ldots \forall x_n \, \phi'$ dove $n \geq 0$ e ϕ' è priva di quantificatori. Ogni formula in forma normale prenessa può essere trasformata in forma normale di Skolem – eliminando quindi tutti i suoi quantificatori esistenziali – introducendo degli opportuni simboli di costante e di funzione detti costanti e funzioni di Skolem. Diversamente dalle precedenti forme normali, la formula ottenuta non è equivalente a quella originaria ma preserva la (in)soddisfacibilità di quest'ultima, cioè è (in)soddisfacibile sse quella originaria lo è.

- L'algoritmo di skolemizzazione applicato ad una formula $\phi \in FBF_{\text{pred}}$ in forma normale prenessa procede come segue:
  - Ogni quantificatore $\exists x$ presente in ϕ che non è nel campo d'azione di un quantificatore universale viene rimosso da ϕ dopo aver applicato alla sua matrice ϕ' la sostituzione $[^{a}/_{x}]$ dove a è una costante che non compare in ϕ'.
  - Ogni quantificatore $\exists x$ presente in ϕ che è nel campo d'azione dei quantificatori universali $\forall x_1 ... \forall x_m$ viene rimosso da ϕ dopo aver applicato alla sua matrice ϕ' la sostituzione $[^{f(x_1,...,x_m)}/_{x}]$ dove f è un simbolo di funzione che non compare in ϕ'.
- Esempio: La formula in forma normale prenessa $\exists x \forall y \exists z\, P(x, y, z, f(a))$ ha come forma normale di Skolem $\forall y\, P(b, y, g(y), f(a))$. Osserviamo che $(D, \mathcal{I}, \mathcal{V}) \models \exists x \forall y \exists z\, P(x, y, z, f(a))$ sse esiste $d_1 \in D$ tale che per ogni $d_2 \in D$ esiste $d_3 \in D$ tale che $(D, \mathcal{I}, \mathcal{V} \setminus \{(x, \mathcal{V}(x)), (y, \mathcal{V}(y)), (z, \mathcal{V}(z))\} \cup \{(x, d_1), (y, d_2), (z, d_3)\}) \models P(x, y, z, f(a))$, cioè sse per ogni $d_2 \in D$ risulta $(D, \mathcal{I}, \mathcal{V} \setminus \{(y, \mathcal{V}(y))\} \cup \{(y, d_2)\}) \models P(b, y, g(y), f(a))$ ponendo $\mathcal{I}(b) = d_1$ e $\mathcal{I}(g(y)) = d_3$, cioè sse $(D, \mathcal{I}, \mathcal{V}) \models \forall y\, P(b, y, g(y), f(a))$.
- Data una formula $\phi \in FBF_{\text{pred}}$, al fine di agevolare la verifica della sua (in)soddisfacibilità conviene eseguire i seguenti passi preliminari:
  1. Trasformare ϕ in una formula ρ equivalente in forma normale rettificata.
  2. Calcolare la chiusura esistenziale di ρ, che è una formula chiusa soddisfacibile sse ρ è soddisfacibile.
  3. Trasformare $ce(\rho)$ in una formula ξ equivalente in forma normale prenessa.
  4. Trasformare ξ in una formula κ in forma normale di Skolem, che è soddisfacibile sse lo è ξ (e quindi ϕ).

  Denotiamo con $FBF_{\text{pred,crs}}$ l'insieme delle formule di FBF_{pred} che sono chiuse e in forma normale rettificata e di Skolem. ■fltpl_12

11.2 Teoria di Herbrand e algoritmo di refutazione

- Per stabilire la validità di una formula ben formata della logica dei predicati procedendo per via semantica, bisogna assicurare che la formula sia soddisfatta da ogni possibile ambiente. In generale, verificare la validità di una formula della logica dei predicati è più complicato che verificarne l'insoddisfacibilità. Di conseguenza, data una formula $\phi \in FBF_{\text{pred}}$, per stabilirne la validità conviene procedere per refutazione, cioè controllare che nessun ambiente soddisfa $\neg\phi$ lavorando sulla corrispondente formula appartenente ad $FBF_{\text{pred,crs}}$ che preserva l'insoddisfacibilità.
- Poichè il numero di domini è illimitato così come il numero di interpretazioni definibili su un dominio specie se infinito, Herbrand osservò che il controllo è fattibile solo se esiste un dominio canonico, cioè un dominio tale che una formula della logica dei predicati è insoddisfacibile sse è falsa in ogni interpretazione su tale dominio. Herbrand scoprì che tale dominio esiste ed è costruibile a partire dai simboli di costante e di funzione presenti nella formula.
- Diciamo che un termine $t \in Ter$ è ground sse $var(t) = \emptyset$ e indichiamo con Ter_{g} l'insieme dei termini ground. Analogamente diciamo che una formula $\phi \in FBF_{\text{pred}}$ è ground sse $var(\phi) = \emptyset$ e indichiamo con $FBF_{\text{pred,g}}$ l'insieme delle formule ground e con $FBF_{\text{pred,a,g}}$ l'insieme delle formule atomiche ground.
- Data una formula $\phi \in FBF_{\text{pred}}$, chiamiamo:
  - Universo di Herbrand di ϕ il più piccolo sottoinsieme $U_{\text{H}}(\phi)$ di Ter_{g} tale che:
    - $a \in U_{\text{H}}(\phi)$ per ogni $a \in Cos$ presente in ϕ; se ϕ non contiene costanti, scegliamo una costante $c \in Cos$ e poniamo $c \in U_{\text{H}}(\phi)$.
    - $f(u_1, ..., u_n) \in U_{\text{H}}(\phi)$ per ogni $f \in Fun$ presente in ϕ con $n \geq 1$ argomenti e $u_1, ..., u_n \in U_{\text{H}}(\phi)$.
  - Base di Herbrand di ϕ il più piccolo sottoinsieme $B_{\text{H}}(\phi)$ di $FBF_{\text{pred,a,g}}$ tale che:
    - $P(u_1, ..., u_n) \in B_{\text{H}}(\phi)$ per ogni $P \in Pred$ presente in ϕ con $n \geq 1$ argomenti e $u_1, ..., u_n \in U_{\text{H}}(\phi)$.
- Esempio: Se ϕ è la formula $\forall x\, P(f(x))$, allora $U_{\text{H}}(\phi)$ comprende c, $f(c)$, $f(f(c))$, $f(f(f(c)))$ e così via, mentre $B_{\text{H}}(\phi)$ comprende $P(c)$, $P(f(c))$, $P(f(f(c)))$, $P(f(f(f(c))))$ e così via.

- Data una formula $\phi \in FBF_{\text{pred}}$, chiamiamo interpretazione di Herbrand di ϕ ogni ambiente $(U_{\text{H}}(\phi), \mathcal{I}, \mathcal{V})$ tale che:
  - $\mathcal{I}(a) = a$ per ogni $a \in Cos$ appartenente a $U_{\text{H}}(\phi)$.
  - $\mathcal{I}(f) = f$ per ogni $f \in Fun$ presente in ϕ.

  Diciamo che $(U_{\text{H}}(\phi), \mathcal{I}, \mathcal{V})$ è un modello di Herbrand di ϕ sse $(U_{\text{H}}(\phi), \mathcal{I}, \mathcal{V}) \models \phi$.
- L'interpretazione di Herbrand di una formula $\phi \in FBF_{\text{pred}}$ non deve soggiacere a nessuna restrizione su nessun predicato P presente in ϕ, quindi due diverse interpretazioni di Herbrand di ϕ differiscono soltanto per il modo in cui vengono interpretati i simboli di predicato. Poiché gli elementi di $B_{\text{H}}(\phi)$ sono assimilabili a formule atomiche di logica proposizionale, ogni interpretazione di Herbrand di ϕ può essere vista come un sottoinsieme A di $B_{\text{H}}(\phi)$ assumendo che una sottoformula atomica di ϕ è vera sse ogni sua istanza ground appartiene ad A. Il seguente teorema e il suo corollario stabiliscono che $U_{\text{H}}(\phi)$ è un dominio canonico.
- Teorema: Sia $\phi \in FBF_{\text{pred,crs}}$. Per ogni ambiente esiste un'interpretazione di Herbrand di ϕ tale che se l'ambiente soddisfa ϕ allora anche l'interpretazione di Herbrand soddisfa ϕ.
- Dimostrazione: Sia $\phi \in FBF_{\text{pred,crs}}$. Dato un ambiente $(D, \mathcal{I}, \mathcal{V})$, l'interpretazione di Herbrand $(U_{\text{H}}(\phi), \mathcal{I}', \mathcal{V})$ di ϕ corrispondente a quell'ambiente è definita ponendo:
  - $\mathcal{I}'(a) = a$ per ogni $a \in Cos$ in $U_{\text{H}}(\phi)$ come da definizione di interpretazione di Herbrand.
  - $\mathcal{I}'(f) = f$ per ogni $f \in Fun$ presente in ϕ come da definizione di interpretazione di Herbrand.
  - $\mathcal{I}'(P) = \{(u_1, \ldots, u_n) \in (U_{\text{H}}(\phi))^n \mid (D, \mathcal{I}, \mathcal{V}) \models P(u_1, \ldots, u_n)\}$ per ogni $P \in Pred$ presente in ϕ.

  Osservato che per ogni sottoformula immediata ϕ' di ϕ risulta $\phi' \in FBF_{\text{pred,crs}}$ se ϕ non è quantificata, dimostriamo per induzione sulla struttura sintattica di ϕ che se $(D, \mathcal{I}, \mathcal{V}) \models \phi$ allora $(U_{\text{H}}(\phi), \mathcal{I}', \mathcal{V}) \models \phi$:
  - Sia ϕ della forma $P(t_1, \ldots, t_n)$ con $P \in Pred$ e $t_1, \ldots, t_n \in Ter$. Poichè ϕ è chiusa e quindi $t_1, \ldots, t_n$ sono termini ground appartenenti a $U_{\text{H}}(\phi)$, se $(D, \mathcal{I}, \mathcal{V}) \models \phi$ allora $(t_1, \ldots, t_n) \in \mathcal{I}'(P)$ per definizione di $\mathcal{I}'(P)$, cioè $(U_{\text{H}}(\phi), \mathcal{I}', \mathcal{V}) \models \phi$.
  - Sia ϕ della forma $(\neg\phi')$ e supponiamo che se $(D, \mathcal{I}, \mathcal{V}) \models \phi'$ allora $(U_{\text{H}}(\phi'), \mathcal{I}', \mathcal{V}) \models \phi'$. Se $(D, \mathcal{I}, \mathcal{V}) \models \phi$, cioè $(D, \mathcal{I}, \mathcal{V}) \not\models \phi'$, allora $(U_{\text{H}}(\phi'), \mathcal{I}', \mathcal{V}) \not\models \phi'$ sfruttando l'ipotesi induttiva, cioè $(U_{\text{H}}(\phi), \mathcal{I}', \mathcal{V}) \models \phi$.
  - Sia ϕ della forma $(\phi'_1 \vee \phi'_2)$ e supponiamo che se $(D, \mathcal{I}, \mathcal{V}) \models \phi'_i$ allora $(U_{\text{H}}(\phi'_i), \mathcal{I}', \mathcal{V}) \models \phi'_i$ per $i \in \{1, 2\}$. Se $(D, \mathcal{I}, \mathcal{V}) \models \phi$, cioè $(D, \mathcal{I}, \mathcal{V}) \models \phi'_1$ o $(D, \mathcal{I}, \mathcal{V}) \models \phi'_2$, allora $(U_{\text{H}}(\phi'_1), \mathcal{I}', \mathcal{V}) \models \phi'_1$ o $(U_{\text{H}}(\phi'_2), \mathcal{I}', \mathcal{V}) \models \phi'_2$ sfruttando l'ipotesi induttiva, cioè $(U_{\text{H}}(\phi), \mathcal{I}', \mathcal{V}) \models \phi$.
  - Sia ϕ della forma $(\phi'_1 \wedge \phi'_2)$ e supponiamo che se $(D, \mathcal{I}, \mathcal{V}) \models \phi'_i$ allora $(U_{\text{H}}(\phi'_i), \mathcal{I}', \mathcal{V}) \models \phi'_i$ per $i \in \{1, 2\}$. Se $(D, \mathcal{I}, \mathcal{V}) \models \phi$, cioè $(D, \mathcal{I}, \mathcal{V}) \models \phi'_1$ e $(D, \mathcal{I}, \mathcal{V}) \models \phi'_2$, allora $(U_{\text{H}}(\phi'_1), \mathcal{I}', \mathcal{V}) \models \phi'_1$ e $(U_{\text{H}}(\phi'_2), \mathcal{I}', \mathcal{V}) \models \phi'_2$ sfruttando l'ipotesi induttiva, cioè $(U_{\text{H}}(\phi), \mathcal{I}', \mathcal{V}) \models \phi$.
  - Sia ϕ della forma $(\phi'_1 \rightarrow \phi'_2)$ e supponiamo che se $(D, \mathcal{I}, \mathcal{V}) \models \phi'_i$ allora $(U_{\text{H}}(\phi'_i), \mathcal{I}', \mathcal{V}) \models \phi'_i$ per $i \in \{1, 2\}$. Se $(D, \mathcal{I}, \mathcal{V}) \models \phi$, cioè $(D, \mathcal{I}, \mathcal{V}) \not\models \phi'_1$ oppure $(D, \mathcal{I}, \mathcal{V}) \models \phi'_2$, allora $(U_{\text{H}}(\phi'_1), \mathcal{I}', \mathcal{V}) \not\models \phi'_1$ oppure $(U_{\text{H}}(\phi'_2), \mathcal{I}', \mathcal{V}) \models \phi'_2$ sfruttando l'ipotesi induttiva, cioè $(U_{\text{H}}(\phi), \mathcal{I}', \mathcal{V}) \models \phi$.
  - Sia ϕ della forma $(\phi'_1 \leftrightarrow \phi'_2)$ e supponiamo che se $(D, \mathcal{I}, \mathcal{V}) \models \phi'_i$ allora $(U_{\text{H}}(\phi'_i), \mathcal{I}', \mathcal{V}) \models \phi'_i$ per $i \in \{1, 2\}$. Se $(D, \mathcal{I}, \mathcal{V}) \models \phi$, cioè $(D, \mathcal{I}, \mathcal{V}) \models \phi'_1$ e $(D, \mathcal{I}, \mathcal{V}) \models \phi'_2$ oppure $(D, \mathcal{I}, \mathcal{V}) \not\models \phi'_1$ e $(D, \mathcal{I}, \mathcal{V}) \not\models \phi'_2$, allora $(U_{\text{H}}(\phi'_1), \mathcal{I}', \mathcal{V}) \models \phi'_1$ e $(U_{\text{H}}(\phi'_2), \mathcal{I}', \mathcal{V}) \models \phi'_2$ oppure $(U_{\text{H}}(\phi'_1), \mathcal{I}', \mathcal{V}) \not\models \phi'_1$ e $(U_{\text{H}}(\phi'_2), \mathcal{I}', \mathcal{V}) \not\models \phi'_2$ sfruttando l'ipotesi induttiva, cioè $(U_{\text{H}}(\phi), \mathcal{I}', \mathcal{V}) \models \phi$.
  - Sia ϕ della forma $(\forall x)(\phi')$ e, osservato che ϕ' potrebbe non essere chiusa, supponiamo che se $(D, \mathcal{I}, \mathcal{V}) \models \phi'[^u/_x]$ allora $(U_{\text{H}}(\phi'[^u/_x]), \mathcal{I}', \mathcal{V}) \models \phi'[^u/_x]$ per ogni $u \in U_{\text{H}}(\phi)$, dove $\phi'[^u/_x] \in FBF_{\text{pred,crs}}$. Se $(D, \mathcal{I}, \mathcal{V}) \models \phi$, cioè $(D, \mathcal{I}, \mathcal{V} \setminus \{(x, \mathcal{V}(x))\} \cup \{(x, d)\}) \models \phi'$ per ogni $d \in D$, allora in particolare $(D, \mathcal{I}, \mathcal{V} \setminus \{(x, \mathcal{V}(x))\} \cup \{(x, d)\}) \models \phi'$ per ogni $d \in D$ tale che $d = \mathcal{I}(u)$ per qualche $u \in U_{\text{H}}(\phi)$, cioè $(D, \mathcal{I}, \mathcal{V}) \models \phi'[^u/_x]$ per ogni $u \in U_{\text{H}}(\phi)$, e quindi $(U_{\text{H}}(\phi'[^u/_x]), \mathcal{I}', \mathcal{V}) \models \phi'[^u/_x]$ per ogni $u \in U_{\text{H}}(\phi)$ sfruttando l'ipotesi induttiva, cioè $(U_{\text{H}}(\phi), \mathcal{I}', \mathcal{V} \setminus \{(x, \mathcal{V}(x))\} \cup \{(x, u)\}) \models \phi'$ per ogni $u \in U_{\text{H}}(\phi)$, cioè $(U_{\text{H}}(\phi), \mathcal{I}', \mathcal{V}) \models \phi$.

- Il precedente teorema non vale per formule che contengono quantificatori esistenziali. Ad esempio, la formula ϕ data da $P(a) \wedge \exists x\, \neg P(x)$ è soddisfacibile, come si vede prendendo un qualsiasi ambiente $(D, \mathcal{I}, \mathcal{V})$ dove D ha almeno due valori d_1, d_2 tali che $\mathcal{I}(a) = d_1 \in \mathcal{I}(P)$ mentre $d_2 \notin \mathcal{I}(P)$. La formula ϕ non ha però alcun modello di Herbrand. Infatti $U_{\mathrm{H}}(\phi) = \{a\}$ e $B_{\mathrm{H}}(\phi) = \{P(a)\}$, quindi gli unici due possibili sottoinsiemi di $B_{\mathrm{H}}(\phi)$ sono $\emptyset$ e $\{P(a)\}$. L'interpretazione di Herbrand corrispondente al primo sottoinsieme rende $P(a)$ falsa ed $\exists x\, \neg P(x)$ vera, mentre l'interpretazione di Herbrand corrispondente al secondo sottoinsieme rende $P(a)$ vera ed $\exists x\, \neg P(x)$ falsa. Dunque nessuna delle due possibili interpretazioni di Herbrand soddisfa ϕ.

- Corollario: Sia $\phi \in \mathit{FBF}_{\mathrm{pred,crs}}$. Allora ϕ è soddisfacibile sse ha un modello di Herbrand.

- Dimostrazione: Se ϕ è soddisfacibile, cioè se esiste un ambiente $(D, \mathcal{I}, \mathcal{V})$ che soddisfa ϕ, allora ϕ ha un modello di Herbrand che è l'interpretazione di Herbrand di ϕ ottenuta da $(D, \mathcal{I}, \mathcal{V})$ tramite la costruzione mostrata nella dimostrazione del precedente teorema. Viceversa, se ϕ ha un modello di Herbrand, allora ϕ è banalmente soddisfacibile.

- Data una formula $\phi \in \mathit{FBF}_{\mathrm{pred,crs}}$ della forma $\forall x_1 \ldots \forall x_n\, \phi'$ dove ϕ' è la matrice, l'espansione di Herbrand di ϕ è definita ponendo $E_{\mathrm{H}}(\phi) = \{\phi'[{}^{u_1}/_{x_1}] \ldots [{}^{u_n}/_{x_n}] \mid u_1, \ldots, u_n \in U_{\mathrm{H}}(\phi)\}$. In altri termini, $E_{\mathrm{H}}(\phi)$ viene ottenuta da ϕ rimuovendo tutti i quantificatori universali e prendendo tutte le istanze ground della matrice. Poiché non contengono quantificatori né variabili, le formule appartenenti ad $E_{\mathrm{H}}(\phi)$ sono assimilabili a formule della logica proposizionale (però potrebbero essercene infinite).

- Esempio: Se ϕ è la formula $\forall x\, (P(a) \vee P(f(x)))$, allora $E_{\mathrm{H}}(\phi)$ comprende $P(a) \vee P(f(a))$, $P(a) \vee P(f(f(a)))$, $P(a) \vee P(f(f(f(a))))$ e così via, dove $P(a)$, $P(f(a))$, $P(f(f(a)))$, $P(f(f(f(a))))$, ecc. sono assimilabili a proposizioni.

- Teorema (di Gödel, Herbrand, Skolem): Sia $\phi \in \mathit{FBF}_{\mathrm{pred,crs}}$. Allora ϕ è soddisfacibile sse $E_{\mathrm{H}}(\phi)$ lo è, cioè sse esiste un ambiente che soddisfa tutte le formule di $E_{\mathrm{H}}(\phi)$.

- Dimostrazione: Sia ϕ della forma $\forall x_1 \ldots \forall x_n\, \phi'$ dove ϕ' è la matrice. Allora ϕ è soddisfacibile sse ϕ ha un modello di Herbrand $(U_{\mathrm{H}}(\phi), \mathcal{I}, \mathcal{V})$, cioè sse $(U_{\mathrm{H}}(\phi), \mathcal{I}, \mathcal{V} \setminus \{(x_i, \mathcal{V}(x_i)) \mid 1 \leq i \leq n\} \cup \{(x_i, u_i) \mid 1 \leq i \leq n\}) \models \phi'$ per ogni $u_1, \ldots, u_n \in U_{\mathrm{H}}(\phi)$, cioè sse $(U_{\mathrm{H}}(\phi), \mathcal{I}, \mathcal{V}) \models \phi'[{}^{u_1}/_{x_1}] \ldots [{}^{u_n}/_{x_n}]$ per ogni $u_1, \ldots, u_n \in U_{\mathrm{H}}(\phi)$, cioè sse $(U_{\mathrm{H}}(\phi), \mathcal{I}, \mathcal{V})$ soddisfa tutte le formule di $E_{\mathrm{H}}(\phi)$ (ne è un modello di Herbrand perché $U_{\mathrm{H}}(\phi'[{}^{u_1}/_{x_1}] \ldots [{}^{u_n}/_{x_n}]) = U_{\mathrm{H}}(\phi)$ per ogni $u_1, \ldots, u_n \in U_{\mathrm{H}}(\phi)$).

- Il problema della (in)soddisfacibilità per una formula ben formata della logica dei predicati è dunque riducibile al problema della (in)soddisfacibilità per un insieme di formule ben formate della logica proposizionale. L'aspetto critico è che questo insieme è infinito, tranne il caso in cui la formula originaria non contiene simboli di funzione. Il seguente teorema, che è alla base della maggior parte degli algoritmi di dimostrazione automatica di teoremi, mostra che il problema è ulteriormente riducibile ad un opportuno sottoinsieme finito di quell'insieme di formule ben formate della logica proposizionale.

- Teorema (di Herbrand): Sia $\phi \in \mathit{FBF}_{\mathrm{pred,crs}}$. Allora ϕ è insoddisfacibile sse esiste un sottoinsieme finito di $E_{\mathrm{H}}(\phi)$ tale che la congiunzione delle sue formule è insoddisfacibile.

- Dimostrazione: Segue dal teorema di Gödel, Herbrand, Skolem e dal teorema di compattezza (vedi Sez. 11.3).

- Sulla base del teorema di Herbrand possiamo costruire il seguente algoritmo che opera per refutazione al fine di stabilire la validità di una formula $\phi \in \mathit{FBF}_{\mathrm{pred}}$:
  1. Trasformare $\neg\phi$ in una formula $\gamma \in \mathit{FBF}_{\mathrm{pred,crs}}$ che è insoddisfacibile sse lo è $\neg\phi$ (vedi Sez. 11.1).
  2. Enumerare come $\gamma_0, \gamma_1, \gamma_2, \ldots$ le formule in $E_{\mathrm{H}}(\gamma)$.
  3. Assegnare 0 ad una variabile intera n.
  4. Finché $\bigwedge_{i=0}^{n} \gamma_i$ è soddisfacibile, incrementare n di 1.
  5. Segnalare che γ è insoddisfacibile, cioè che ϕ è valida.

- Poichè il problema affrontato è soltanto semi-decidibile come stabilito dal teorema di Church, Herbrand e Skolem (vedi Sez. 10.2), il precedente algoritmo termina sicuramente solo se ϕ è valida, altrimenti va avanti considerando approssimazioni di $E_{\mathrm{H}}(\gamma)$ costituite da insiemi sempre più grandi di formule senza mai terminare. Poiché le formule appartenenti ad $E_{\mathrm{H}}(\gamma)$ sono assimilabili a formule della logica proposizionale, la soddisfacibilità di ogni $\bigwedge_{i=0}^{n} \gamma_i$ può essere stabilita costruendone la tabella di verità.
- Esempio: Se ϕ è la formula $\exists x \exists y\,(\neg(P(a) \vee P(f(x))) \vee P(y))$, allora la formula γ usata dall'algoritmo è $\forall x \forall y\,((P(a) \vee P(f(x))) \wedge \neg P(y))$. Notato che $U_{\mathrm{H}}(\gamma)$ comprende a, $f(a)$, $f(f(a))$, $f(f(f(a)))$ e così via, abbiamo che $E_{\mathrm{H}}(\gamma)$ comprende $(P(a) \vee P(f(a))) \wedge \neg P(a)$, $(P(a) \vee P(f(a))) \wedge \neg P(f(a))$ e così via. Posto $p = P(a)$ e $q = P(f(a))$, l'algoritmo esamina come prima formula $\gamma_0 = (p \vee q) \wedge \neg p$ e determina che essa è soddisfacibile come si vede prendendo l'assegnamento di verità $A = \{q\}$. Poi l'algoritmo esamina la congiunzione di $\gamma_0 = (p \vee q) \wedge \neg p$ e $\gamma_1 = (p \vee q) \wedge \neg q$, la quale è equivalente a $(p \vee q) \wedge (\neg p \wedge \neg q)$ grazie all'idempotenza della congiunzione, cioè a $(p \vee q) \wedge \neg(p \vee q)$ grazie alle leggi di De Morgan, e quindi è insoddisfacibile. Pertanto l'algoritmo termina dopo due iterazioni segnalando che ϕ è valida. ■fltpl_13

11.3 Risoluzione di Robinson per la logica proposizionale

- L'algoritmo di refutazione basato sulla teoria di Herbrand deve calcolare ad ogni iterazione il valore di verità della congiunzione di un numero sempre crescente di formule ben formate della logica proposizionale, impiegando così ad ogni passo un tempo esponenziale rispetto al numero di proposizioni presenti in quelle formule. Un miglioramento delle prestazioni dell'algoritmo di refutazione può essere ottenuto se, invece di costruire la tabella di verità della congiunzione di quelle formule, si utilizza il metodo di risoluzione dopo aver trasformato quelle formule in clausole. Sebbene la complessità di ogni iterazione rimanga esponenziale nel caso pessimo, essa migliora significativamente nel caso medio.
- Il metodo di risoluzione venne introdotto da Robinson nel 1965 allo scopo di definire un sistema deduttivo che fosse più efficiente dei sistemi deduttivi alla Hilbert per dimostrare automaticamente teoremi. Questo metodo è particolarmente semplice da impiegare perché comprende soltanto una regola di inferenza. Esso procede per refutazione e si applica solo a insiemi di clausole.
- La regola di inferenza del metodo di risoluzione è la seguente:
$$\frac{\lambda_{1,1} \vee \ldots \vee \lambda_{1,m} \vee p \qquad \lambda_{2,1} \vee \ldots \vee \lambda_{2,n} \vee \neg p}{\lambda_{1,1} \vee \ldots \vee \lambda_{1,m} \vee \lambda_{2,1} \vee \ldots \vee \lambda_{2,n}}$$
dove $m, n \in \mathbb{N}$, ogni $\lambda_{i,j}$ è un letterale e la conclusione della regola è detta clausola risolvente (la clausola risolvente è una clausola di Horn qualora le premesse siano due clausole di Horn). Nel caso $m = n = 0$ la clausola risolvente è la clausola vuota, la quale viene denotata con $\square$.
- Ogni formula $\phi \in FBF_{\mathrm{prop}}$ può essere trasformata in forma normale congiuntiva e quindi può essere vista come un insieme di clausole in cui ogni clausola può essere vista come un insieme di letterali. La notazione insiemistica è pienamente adeguata per rappresentare forme normali congiuntive e clausole grazie alle proprietà commutative, associative e di idempotenza di congiunzione e disgiunzione.
- Esempio: La forma norma congiuntiva $(p \vee q \vee \neg r) \wedge (p \vee \neg q \vee r)$ può essere rappresentata come $\{\{p, q, \neg r\}, \{p, \neg q, r\}\}$, dove le due clausole hanno come risolventi $\{p, \neg r, r\}$ e $\{p, q, \neg q\}$.
- Poiché una formula in forma normale congiuntiva è soddisfacibile sse tutte le sue clausole sono soddisfacibili, l'insieme vuoto di clausole ($\emptyset$) è banalmente soddisfacibile. Poiché una clausola è soddisfacibile sse almeno uno dei suoi letterali è soddisfacibile, la clausola vuota ($\square$) è banalmente insoddisfacibile e quindi rappresenta una contraddizione. Ogni clausola che contiene sia una proposizione che la sua negazione è banalmente valida e quindi rappresenta una tautologia.
- Dato un insieme di clausole C e una clausola c, scriviamo $C \vdash_{\mathrm{R}} c$ per indicare che c è derivabile da C tramite risoluzione (cioè che esiste una sequenza non vuota di lunghezza finita di clausole tale che ogni clausola della sequenza o appartiene a C o è la risolvente di due clausole che la precedono nella sequenza, dove l'ultima clausola della sequenza è c – vedi Sez. 9.5). Diciamo che C è refutabile sse $C \vdash_{\mathrm{R}} \square$.

- La regola di inferenza del metodo di risoluzione preserva i modelli delle clausole originarie e il sistema deduttivo basato su tale regola risulta essere corretto e completo rispetto all'insoddisfacibilità.
- Teorema: Siano c_1, c_2, c delle clausole. Se c è una risolvente di c_1 e c_2, allora $c_1 \models c$ e $c_2 \models c$.
- Dimostrazione: Poiché c_1 e c_2 ammettono una clausola risolvente, esse sono entrambe non vuote e contengono almeno una proposizione p e la sua negazione $\neg p$, rispettivamente. Sia $c = (c_1 \setminus \{p\}) \cup (c_2 \setminus \{\neg p\})$. Per ogni assegnamento di verità $A \in 2^{Prop}$ tale che $A \models c_1$ e $A \models c_2$, ci sono due casi:
  - Se $p \in A$ allora da $A \models c_2$ e $A \not\models \neg p$ segue che $A \models (c_2 \setminus \{\neg p\})$ e quindi $A \models c$.
  - Se $p \notin A$ allora da $A \models c_1$ e $A \not\models p$ segue che $A \models (c_1 \setminus \{p\})$ e quindi $A \models c$.
- Dato un insieme di formule $\Phi \in 2^{FBF_{\text{prop}}}$, diciamo che esso è:
  - Soddisfacibile sse esiste $A \in 2^{Prop}$ tale che $A \models \phi$ per ogni $\phi \in \Phi$.
  - Finitamente soddisfacibile sse ogni sottoinsieme finito di Φ è soddisfacibile.
- Teorema (di compattezza): Un insieme di formule è soddisfacibile sse è finitamente soddisfacibile.
- Dimostrazione: Sia $\Phi \in 2^{FBF_{\text{prop}}}$. Se Φ è soddisfacibile allora ovviamente ogni sottoinsieme finito di Φ è soddisfacibile e quindi Φ è finitamente soddisfacibile.
  Supponiamo ora che Φ sia finitamente soddisfacibile. Dopo aver enumerato come $p_0, p_1, \ldots, p_k, \ldots$ le proposizioni presenti nelle formule di Φ (il cui insieme denotiamo con $Prop'$) e come $\gamma_0, \gamma_1, \ldots, \gamma_k, \ldots$ le formule ben formate che possono essere costruite con quelle proposizioni (il cui insieme denotiamo con $FBF_{Prop'}$ ed include Φ), definiamo l'insieme di formule $\Gamma = \bigcup_{n \in \mathbb{N}} \Gamma_n$ dove:
  $$\Gamma_n = \begin{cases} \Phi & \text{se } n = 0 \\ \Gamma_{n-1} \cup \{\gamma_{n-1}\} & \text{se } n > 0 \text{ e } \Gamma_{n-1} \cup \{\gamma_{n-1}\} \text{ è finitamente soddisfacibile} \\ \Gamma_{n-1} \cup \{\neg\gamma_{n-1}\} & \text{se } n > 0 \text{ e } \Gamma_{n-1} \cup \{\gamma_{n-1}\} \text{ non è finitamente soddisfacibile} \end{cases}$$
  Dimostriamo che ogni Γ_n è finitamente soddisfacibile procedendo per induzione su $n \in \mathbb{N}$:
  - Se $n = 0$ allora Γ_n coincide con Φ che è finitamente soddisfacibile per ipotesi iniziale.
  - Sia $n \geq 0$ e supponiamo che Γ_n sia finitamente soddisfacibile. Assumiamo che Γ_{n+1} non sia finitamente soddisfacibile, cioè che esistano $\Gamma'_n, \Gamma''_n \subseteq \Gamma_n$ tali che $\Gamma'_n \cup \{\gamma_n\}$ e $\Gamma''_n \cup \{\neg\gamma_n\}$ sono insoddisfacibili. Poiché $\Gamma'_n \cup \Gamma''_n \subseteq \Gamma_n$ e Γ_n è finitamente soddisfacibile per ipotesi induttiva, $\Gamma'_n \cup \Gamma''_n$ è soddisfacibile. Sia A un assegnamento di verità che soddisfa tutte le formule di $\Gamma'_n \cup \Gamma''_n$. Dal fatto che o $A \models \gamma_n$ o $A \models \neg\gamma_n$, segue che o A soddisfa tutte le formule di $\Gamma'_n \cup \{\gamma_n\}$ o A soddisfa tutte le formule di $\Gamma''_n \cup \{\neg\gamma_n\}$. Di conseguenza abbiamo raggiunto una contraddizione e quindi Γ_{n+1} è anch'esso finitamente soddisfacibile.

  L'insieme Γ è finitamente soddisfacibile. Infatti, sia $\{\gamma'_1, \ldots, \gamma'_h\}$ un sottoinsieme finito di Γ. Per ogni $j = 1, \ldots, h$ da $\gamma'_j \in \Gamma$ segue che $\gamma'_j \in \Gamma_{n_j}$ per qualche $n_j \in \mathbb{N}$. Pertanto $\{\gamma'_1, \ldots, \gamma'_h\}$ è un sottoinsieme finito di Γ_n dove $n = \max_{1 \leq j \leq h} n_j$ e, poiché Γ_n è finitamente soddisfacibile, $\{\gamma'_1, \ldots, \gamma'_h\}$ è soddisfacibile. L'insieme Γ è anche esaustivo, nel senso che o $\gamma_k \in \Gamma$ o $\neg\gamma_k \in \Gamma$ per ogni $k \in \mathbb{N}$. Infatti, se $\{\gamma_k, \neg\gamma_k\} \subseteq \Gamma$ per qualche $k \in \mathbb{N}$, allora Γ conterrebbe un sottoinsieme finito insoddisfacibile e quindi non potrebbe essere finitamente soddisfacibile.
  Dal fatto che Γ è finitamente soddisfacibile ed esaustivo segue che Γ è soddisfacibile. Infatti, osservato che o $p_k \in \Gamma$ o $\neg p_k \in \Gamma$ per ogni $k \in \mathbb{N}$ essendo Γ esaustivo, l'assegnamento di verità $A \in 2^{Prop'}$ definito ponendo $p_k \in A$ se $p_k \in \Gamma$ e $p_k \notin A$ se $\neg p_k \in \Gamma$ è tale che $A \models \gamma_k$ sse $\gamma_k \in \Gamma$ come si dimostra procedendo per induzione sulla struttura sintattica di $\gamma_k \in FBF_{Prop'}$:
  - Se $\gamma_k \in Prop'$ allora $A \models \gamma_k$ sse $\gamma_k \in \Gamma$ per definizione di A.
  - Sia γ_k della forma $(\neg\gamma')$ e supponiamo che $A \models \gamma'$ sse $\gamma' \in \Gamma$. Allora $A \models \gamma_k$ sse $A \not\models \gamma'$, cioè sse $\gamma' \notin \Gamma$ sfruttando l'ipotesi induttiva, cioè sse $\gamma_k \in \Gamma$ essendo Γ esaustivo.

- Sia γ_k della forma $(\gamma'_1 \vee \gamma'_2)$ e supponiamo che $A \models \gamma'_i$ sse $\gamma'_i \in \Gamma$ per $i \in \{1,2\}$. Allora $A \models \gamma_k$ sse $A \models \gamma'_1$ o $A \models \gamma'_2$, cioè sse $\gamma'_1 \in \Gamma$ o $\gamma'_2 \in \Gamma$ sfruttando l'ipotesi induttiva, cioè sse $\gamma_k \in \Gamma$. Infatti, se $\gamma_k \in \Gamma$ ma $\gamma'_1, \gamma'_2 \notin \Gamma$, allora $\neg\gamma'_1, \neg\gamma'_2 \in \Gamma$ essendo Γ esaustivo. Di conseguenza $\{(\gamma'_1 \vee \gamma'_2), \neg\gamma'_1, \neg\gamma'_2\}$ sarebbe un sottoinsieme finito insoddisfacibile di Γ, contraddicendo così il fatto che Γ è finitamente soddisfacibile. Analogamente, se $\gamma'_1 \in \Gamma$ (risp. $\gamma'_2 \in \Gamma$) ma $\gamma_k \notin \Gamma$, allora $\neg\gamma_k \in \Gamma$ essendo Γ esaustivo. Di conseguenza $\{(\neg\gamma'_1 \wedge \neg\gamma'_2), \gamma'_1\}$ (risp. $\{(\neg\gamma'_1 \wedge \neg\gamma'_2), \gamma'_2\}$) sarebbe un sottoinsieme finito insoddisfacibile di Γ, contraddicendo così il fatto che Γ è finitamente soddisfacibile.
- Sia γ_k della forma $(\gamma'_1 \wedge \gamma'_2)$ e supponiamo che $A \models \gamma'_i$ sse $\gamma'_i \in \Gamma$ per $i \in \{1,2\}$. Allora $A \models \gamma_k$ sse $A \models \gamma'_1$ e $A \models \gamma'_2$, cioè sse $\gamma'_1 \in \Gamma$ e $\gamma'_2 \in \Gamma$ sfruttando l'ipotesi induttiva, cioè sse $\gamma_k \in \Gamma$. Infatti, se $\gamma_k \in \Gamma$ ma $\gamma'_1 \notin \Gamma$ (risp. $\gamma'_2 \notin \Gamma$), allora $\neg\gamma'_1 \in \Gamma$ (risp. $\neg\gamma'_2 \in \Gamma$) essendo Γ esaustivo. Di conseguenza $\{(\gamma'_1 \wedge \gamma'_2), \neg\gamma'_1\}$ (risp. $\{(\gamma'_1 \wedge \gamma'_2), \neg\gamma'_2\}$) sarebbe un sottoinsieme finito insoddisfacibile di Γ, contraddicendo così il fatto che Γ è finitamente soddisfacibile. Analogamente, se $\gamma'_1 \in \Gamma$ e $\gamma'_2 \in \Gamma$ ma $\gamma_k \notin \Gamma$, allora $\neg\gamma_k \in \Gamma$ essendo Γ esaustivo. Di conseguenza $\{(\neg\gamma'_1 \vee \neg\gamma'_2), \gamma'_1, \gamma'_2\}$ sarebbe un sottoinsieme finito insoddisfacibile di Γ, contraddicendo così il fatto che Γ è finitamente soddisfacibile.
- Sia γ_k della forma $(\gamma'_1 \rightarrow \gamma'_2)$ e supponiamo che $A \models \gamma'_i$ sse $\gamma'_i \in \Gamma$ per $i \in \{1,2\}$. Allora $A \models \gamma_k$ sse $A \not\models \gamma'_1$ oppure $A \models \gamma'_2$, cioè sse $\gamma'_1 \notin \Gamma$ oppure $\gamma'_2 \in \Gamma$ sfruttando l'ipotesi induttiva, cioè sse $\gamma_k \in \Gamma$. Infatti, se $\gamma_k \in \Gamma$ ma $\gamma'_1 \in \Gamma$ e $\gamma'_2 \notin \Gamma$, allora $\neg\gamma'_2 \in \Gamma$ essendo Γ esaustivo. Di conseguenza $\{(\gamma'_1 \rightarrow \gamma'_2), \gamma'_1, \neg\gamma'_2\}$ sarebbe un sottoinsieme finito insoddisfacibile di Γ, contraddicendo così il fatto che Γ è finitamente soddisfacibile. Analogamente, se $\gamma'_1 \notin \Gamma$ (risp. $\gamma'_2 \in \Gamma$) ma $\gamma_k \notin \Gamma$, allora $\neg\gamma'_1 \in \Gamma$ e $\neg\gamma_k \in \Gamma$ essendo Γ esaustivo. Di conseguenza $\{\neg(\gamma'_1 \rightarrow \gamma'_2), \neg\gamma'_1\}$ (risp. $\{\neg(\gamma'_1 \rightarrow \gamma'_2), \gamma'_2\}$) sarebbe un sottoinsieme finito insoddisfacibile di Γ, contraddicendo così il fatto che Γ è finitamente soddisfacibile.
- Sia γ_k della forma $(\gamma'_1 \leftrightarrow \gamma'_2)$ e supponiamo che $A \models \gamma'_i$ sse $\gamma'_i \in \Gamma$ per $i \in \{1,2\}$. Allora $A \models \gamma_k$ sse $A \models \gamma'_1$ e $A \models \gamma'_2$ oppure $A \not\models \gamma'_1$ e $A \not\models \gamma'_2$, cioè sse $\gamma'_1, \gamma'_2 \in \Gamma$ oppure $\gamma'_1, \gamma'_2 \notin \Gamma$ sfruttando l'ipotesi induttiva, cioè sse $\gamma_k \in \Gamma$. Infatti, se $\gamma_k \in \Gamma$ ma $\gamma'_1 \in \Gamma$ e $\gamma'_2 \notin \Gamma$, allora $\neg\gamma'_2 \in \Gamma$ essendo Γ esaustivo. Di conseguenza $\{(\gamma'_1 \leftrightarrow \gamma'_2), \gamma'_1, \neg\gamma'_2\}$ sarebbe un sottoinsieme finito insoddisfacibile di Γ, contraddicendo così il fatto che Γ è finitamente soddisfacibile. Analogamente, se $\gamma'_1, \gamma'_2 \in \Gamma$ (risp. $\gamma'_1, \gamma'_2 \notin \Gamma$) ma $\gamma_k \notin \Gamma$, allora ($\neg\gamma'_1, \neg\gamma'_2 \in \Gamma$ e) $\neg\gamma_k \in \Gamma$ essendo Γ esaustivo. Di conseguenza $\{\neg(\gamma'_1 \leftrightarrow \gamma'_2), \gamma'_1, \gamma'_2\}$ (risp. $\{\neg(\gamma'_1 \leftrightarrow \gamma'_2), \neg\gamma'_1, \neg\gamma'_2\}$) sarebbe un sottoinsieme finito insoddisfacibile di Γ, contraddicendo così il fatto che Γ è finitamente soddisfacibile.

Poiché $\Phi \subseteq \Gamma$ e Γ è soddisfacibile, anche Φ è soddisfacibile.

- Corollario: Un insieme di formule è insoddisfacibile sse ammette un sottoinsieme finito tale che la congiunzione delle sue formule è insoddisfacibile.
- Teorema (di risoluzione): Un insieme di clausole è insoddisfacibile sse è refutabile.
- Dimostrazione: In virtù del corollario al teorema di compattezza, un insieme di clausole è insoddisfacibile sse ammette un sottoinsieme finito insoddisfacibile, quindi possiamo ragionare su un insieme finito di clausole C.
  Se C è refutabile allora C è insoddisfacibile. Infatti, se C fosse soddisfacibile allora esisterebbe un assegnamento di verità $A \in 2^{Prop}$ che soddisfa tutte le clausole di C. Poiché C è refutabile e il metodo di risoluzione preserva i modelli delle clausole originarie, A sarebbe un modello anche di $\Box$, ma ciò è assurdo perchè $\Box$ non ammette modelli.
  Se C è insoddisfacibile allora C è refutabile come si dimostra procedendo per induzione sul numero $n \in \mathbb{N}$ di letterali presenti nelle clausole di C:
  - Se $n = 0$ allora $C = \{\Box\}$ e quindi C è banalmente refutabile (il caso $C = \emptyset$ non è compatibile con l'ipotesi che C sia insoddisfacibile).
  - Sia $n \geq 1$ e supponiamo che ogni insieme finito di clausole con un numero di letterali minore di n sia refutabile qualora sia insoddisfacibile. Assumendo $\Box \notin C$ per evitare casi banali, dal fatto che C è insoddisfacibile ed $n > 0$ segue che C deve contenere almeno due clausole e ci deve essere almeno una proposizione p che compare in un letterale positivo di una clausola $c_1 \in C$ e in un letterale negativo di un'altra clausola $c_2 \in C$.

Indichiamo con $R_p(C)$ l'insieme di clausole ottenuto da C eliminando c_1 e c_2 ed aggiungendo la risolvente di c_1 e c_2 rispetto a p, cioè $c_1' \cup c_2'$ dove $c_1' = c_1 \setminus \{p\}$ e $c_2' = c_2 \setminus \{\neg p\}$. L'insieme $R_p(C)$ è insoddisfacibile. Infatti, se esistesse un assegnamento di verità $A \in 2^{Prop}$ che soddisfa tutte le clausole di $R_p(C)$, allora da A potremmo ricavare due assegnamenti di verità $A', A'' \in 2^{Prop}$ identici ad A su $Prop \setminus \{p\}$ ponendo $p \notin A'$ e $p \in A''$. Poiché C è insoddisfacibile, A' ed A'' non possono soddisfare tutte le clausole di C. In particolare, esisteranno $c' \in C$ tale che $A' \not\models c'$ e $c'' \in C$ tale che $A'' \not\models c''$. Poiché A' ed A'' sono ottenuti da A che soddisfa tutte le clausole di $R_p(C)$, le due clausole c' e c'' non possono appartenere ad $R_p(C)$ e quindi deve essere $c' = c_1$ e $c'' = c_2$. Poiché $c_1' \cup c_2' \in R_p(C)$, deve inoltre valere che A soddisfa $c_1' \cup c_2'$. Tuttavia, se A soddisfa c_1' allora A' soddisfa c_1, cioè c'. Analogamente, se A soddisfa c_2' allora A'' soddisfa c_2, cioè c''. Poiché abbiamo raggiunto una contraddizione, l'insieme $R_p(C)$ non può essere soddisfacibile.
Dal fatto che $R_p(C)$ è insoddisfacibile e contiene $n-2$ letterali, sfruttando l'ipotesi induttiva segue che $R_p(C)$ è refutabile e quindi tale è C in quanto $R_p(C)$ è ottenuto da C per risoluzione.

- Dato un insieme di clausole C, il metodo di risoluzione determina:
  - l'insieme $ris(C) = C \cup \{c \mid c \text{ risolvente di } c_1, c_2 \in C\}$ quando viene applicato una sola volta;
  - l'insieme $ris^*(C) = \bigcup_{n \in \mathbb{N}} ris^n(C)$ quando viene applicato un numero arbitrario di volte, dove:
$$ris^n(C) = \begin{cases} C & \text{se } n = 0 \\ ris(ris^{n-1}(C)) & \text{se } n > 0 \end{cases}$$

  Poiché $ris^*(C) \equiv C$, abbiamo che C è refutabile sse $\square \in ris^*(C)$.

- Sulla base del metodo di risoluzione possiamo costruire il seguente algoritmo che opera per refutazione al fine di stabilire la validità di una formula $\phi \in FBF_{\text{prop}}$:
  1. Trasformare $\neg\phi$ in una formula $\gamma \in FBF_{\text{prop}}$ in forma normale congiuntiva (vedi Sez. 11.1).
  2. Trasformare γ in un insieme di clausole C dove le clausole sono insiemi di letterali.
  3. Ripetere i seguenti passi:
     (a) assegnare C a C';
     (b) assegnare $ris(C)$ a C;

     finché non vale che $\square \in C$ oppure $C = C'$.
  4. Se $\square \in C$ segnalare che ϕ è valida, altrimenti segnalare che ϕ non è valida.

- Esempio: La formula $((p \to (q \to r)) \to ((p \to q) \to (p \to r)))$ è valida perché:
  - La formula è equivalente a $(\neg(\neg p \vee (\neg q \vee r)) \vee (\neg(\neg p \vee q) \vee (\neg p \vee r)))$.
  - La sua negazione è equivalente a $((\neg p \vee (\neg q \vee r)) \wedge ((\neg p \vee q) \wedge (p \wedge \neg r)))$.
  - La corrispondente forma normale congiuntiva è $(\neg p \vee \neg q \vee r) \wedge (\neg p \vee q) \wedge (p) \wedge (\neg r)$.
  - Il corrispondente insieme di clausole è $\{\{\neg p, \neg q, r\}, \{\neg p, q\}, \{p\}, \{\neg r\}\}$.
  - L'applicazione della risoluzione alle clausole $\{\neg p, q\}$ e $\{p\}$ dà luogo a $\{q\}$.
  - L'applicazione della risoluzione alle clausole $\{\neg p, \neg q, r\}$ e $\{p\}$ dà luogo a $\{\neg q, r\}$.
  - L'applicazione della risoluzione alle clausole $\{\neg q, r\}$ e $\{q\}$ dà luogo a $\{r\}$.
  - L'applicazione della risoluzione alle clausole $\{r\}$ e $\{\neg r\}$ dà luogo a $\square$. ■fltpl_14

11.4 Unificazione di formule di logica dei predicati

- L'algoritmo di refutazione basato sulla teoria di Herbrand è in generale notevolmente inefficiente perché per stabilire la validità di una formula deve di solito generare un gran numero di elementi dell'espansione di Herbrand della negazione di quella formula. Un miglioramento delle prestazioni dell'algoritmo di refutazione può essere ottenuto se, invece di generare di volta in volta le varie formule ground, si riconosce che molte di quelle formule sono uguali a meno di opportune sostituzioni che le unificano. Questo processo di unificazione di formule è utile anche per estendere adeguatamente il metodo di risoluzione alla logica dei predicati.

- Generalizziamo la nozione di sostituzione introdotta in Sez. 10.1 denotandola attraverso una sequenza finita eventualmente vuota di legami $[{}^{t_1}/_{x_1}, \ldots, {}^{t_n}/_{x_n}]$ dove:

  - $t_i \in \mathit{Ter}$ ed $x_i \in \mathit{Var}$ per ogni $i = 1, \ldots, n$.
  - $t_i \neq x_i$ per ogni $i = 1, \ldots, n$.
  - $x_i \neq x_j$ per ogni $i, j = 1, \ldots, n$ tali che $i \neq j$.

  Denotiamo con ε la sostituzione vuota.

- Sia Θ l'insieme di tutte le sostituzioni basate su *Ter* e *Var*. Data $\theta \in \Theta$, indichiamo con $t\theta$ il termine ottenuto da $t \in \mathit{Ter}$ applicandogli θ, con $\phi\theta$ la formula ottenuta da $\phi \in \mathit{FBF}_{\text{pred}}$ applicandole θ e con $\Phi\theta$ l'insieme di formule ottenuto da $\Phi \in 2^{\mathit{FBF}_{\text{pred}}}$ applicando θ a tutte le formule di Φ.

- Esempio: Se ϕ è la formula $P(x, f(y)) \wedge Q(z)$ e $\theta = [{}^{c}/_{x}, {}^{g(a)}/_{y}, {}^{z'}/_{z}]$, allora $\phi\theta = P(c, f(g(a))) \wedge Q(z')$.

- Date due sostituzioni $\theta_1 = [{}^{t_1}/_{x_1}, \ldots, {}^{t_n}/_{x_n}]$ e $\theta_2 = [{}^{u_1}/_{y_1}, \ldots, {}^{u_m}/_{y_m}]$, la composizione di θ_1 e θ_2, denotata da $\theta_1 \circ \theta_2$, è ottenuta da $[{}^{t_1\theta_2}/_{x_1}, \ldots, {}^{t_n\theta_2}/_{x_n}, {}^{u_1}/_{y_1}, \ldots, {}^{u_m}/_{y_m}]$ eliminando ogni legame ${}^{t_i\theta_2}/_{x_i}$ tale che $t_i\theta_2 = x_i$ così come ogni legame ${}^{u_j}/_{y_j}$ tale che $y_j \in \{x_1, \ldots, x_n\}$.

- La composizione di sostituzioni ha ε come elemento neutro ed è associativa ma non commutativa. Inoltre risulta $\phi(\theta_1 \circ \theta_2) = (\phi\theta_1)\theta_2$ per ogni $\phi \in \mathit{FBF}_{\text{pred}}$ e $\theta_1, \theta_2 \in \Theta$, quindi $[{}^{t_1}/_{x_1}, \ldots, {}^{t_n}/_{x_n}]$ può essere vista come $[{}^{t_1}/_{x_1}] \circ \ldots \circ [{}^{t_n}/_{x_n}]$.

- Esempio: Se $\theta_1 = [{}^{f(z')}/_{x}, {}^{b}/_{y}, {}^{y}/_{z}]$ e $\theta_2 = [{}^{c}/_{z'}, {}^{a}/_{y}, {}^{b}/_{z}]$, allora $\theta_1 \circ \theta_2 = [{}^{f(c)}/_{x}, {}^{b}/_{y}, {}^{a}/_{z}, {}^{c}/_{z'}]$ mentre $\theta_2 \circ \theta_1 = [{}^{c}/_{z'}, {}^{a}/_{y}, {}^{b}/_{z}, {}^{f(z')}/_{x}]$.

- Dati un insieme di formule $\Phi \in 2^{\mathit{FBF}_{\text{pred}}}$ e una sostituzione $\theta \in \Theta$, diciamo che θ è un unificatore di Φ sse $\phi_1\theta = \phi_2\theta$ per ogni $\phi_1, \phi_2 \in \Phi$. In questo caso diciamo che Φ è unificabile.

- È evidente che un insieme di formule sintatticamente diverse non può essere unificabile se in quelle formule non compaiono occorrenze libere di variabili, tuttavia gli algoritmi di refutazione lavorano sulla matrice della formula data e la matrice non contiene quantificatori.

- Dati un insieme unificabile di formule $\Phi \in 2^{\mathit{FBF}_{\text{pred}}}$ e un suo unificatore $\theta \in \Theta$, diciamo che θ è l'unificatore più generale (upg) di Φ sse per ogni unificatore θ' di Φ esiste una sostituzione θ'' tale che $\theta' = \theta \circ \theta''$. Intuitivamente, l'upg è l'unificatore col minor numero di legami. Quando esiste, l'upg è unico a meno di ridenominazione delle variabili.

- Esempio: L'insieme di formule $\Phi = \{P(x, a), P(y, b)\}$ non è unificabile perché a e b sono due costanti diverse e non esiste nessuna sostituzione che le trasforma nella stessa costante. L'insieme di formule $\Phi' = \{P(x, a), P(y, a)\}$ è invece unificabile e due possibili unificatori per esso sono $\theta_1 = [{}^{b}/_{x}, {}^{b}/_{y}]$ e $\theta_2 = [{}^{y}/_{x}]$. Osserviamo che θ_2 è più generale di θ_1 in quanto θ_1 può essere ottenuto da θ_2 componendo quest'ultimo con la sostituzione $[{}^{b}/_{y}]$. Risulta che $[{}^{y}/_{x}]$ e $[{}^{x}/_{y}]$ sono entrambi upg di Φ'.

- Nel 1965 Robinson sviluppò un algoritmo per calcolare l'upg, se esiste, di un insieme di formule ben formate. L'idea dell'algoritmo è di attraversare da sinistra a destra le formule fino a raggiungere simboli differenti e tentando a quel punto di unificare le sottoformule che iniziano con quei simboli. L'algoritmo risultante è non deterministico, nel senso che per lo stesso insieme di formule può generare upg che differiscono per i nomi delle variabili. Inoltre l'algoritmo è intuitivo ma inefficiente, in quanto la complessità nel caso pessimo è esponenziale rispetto alla lunghezza delle formule. Algoritmi di unificazione più efficienti sono stati sviluppati successivamente; ad esempio l'algoritmo di unificazione di Martelli e Montanari ha complessità quasi lineare.

- Dato un insieme finito $\Phi \in 2^{FBF_{\text{pred}}}$, l'algoritmo di unificazione di Robinson procede come segue:

  1. Presumere che Φ sia unificabile.
  2. Assegnare la sostituzione vuota ε a θ.
  3. Finché Φ è presumibilmente unificabile e $|\Phi\theta| > 1$, ripetere i seguenti passi:
     (a) Attraversare da sinistra a destra le formule di $\Phi\theta$ fino ad incontrare la posizione più a sinistra in cui le formule differiscono.
     (b) Se nessuno dei simboli che occupano quella posizione nelle varie formule di $\Phi\theta$ è una variabile, allora Φ non è unificabile, altrimenti:
        i. siano $x \in Var$ e $t \in Ter$ due di quei simboli con $t \neq x$ (scelta non deterministica);
        ii. se x compare in t allora Φ non è unificabile, altrimenti assegnare $\theta \circ [{}^{t}/_{x}]$ a θ.
  4. Se $|\Phi\theta| > 1$ segnalare che Φ non è unificabile, altrimenti segnalare che θ è l'upg.

- Il cuore dell'algoritmo (ii) applicato ai termini può essere rappresentato attraverso la seguente tabella:

	c_2	x_2	ξ_2
c_1	unificabili da ε sse $c_1 = c_2$	$[{}^{c_1}/_{x_2}]$	non unificabili
x_1	$[{}^{c_2}/_{x_1}]$	$[{}^{x_1}/_{x_2}]$	$[{}^{\xi_2}/_{x_1}]$ previo occur check
ξ_1	non unificabili	$[{}^{\xi_1}/_{x_2}]$ previo occur check	$*$

  dove:

  - $c_1, c_2 \in Cos$.
  - $x_1, x_2 \in Var$.
  - $\xi_1, \xi_2 \in Ter \setminus (Cos \cup Var)$.
  - Per occur check si intende il controllo consistente nel verificare che x_i non compaia in ξ_j, altrimenti non è possibile unificare. Questo controllo è necessario per garantire la terminazione dell'algoritmo; in sua assenza, l'algoritmo tenterebbe invano di unificare termini come ad esempio x ed $f(x)$ senza mai arrestarsi.
  - Nel caso $*$, ξ_1 e ξ_2 non sono unificabili se il funtore di ξ_1 differisce dal funtore di ξ_2, altrimenti bisogna procedere ricorsivamente sugli argomenti di ξ_1 e ξ_2 verificando che gli upg per le varie coppie di argomenti siano coerenti tra loro. Ad esempio $g(1,2)$ e $g(y,y)$ non sono unificabili, ma considerando i due argomenti separatamente si avrebbero i due legami incoerenti ${}^{1}/_{y}$ e ${}^{2}/_{y}$.

- Teorema (di Robinson): Ogni insieme unificabile di formule ha l'upg.

- Dimostrazione: Si tratta di verificare che l'algoritmo di unificazione di Robinson termina per ogni insieme finito di formule e, nel caso in cui l'insieme sia unificabile, calcola l'upg dell'insieme.
  Sia $\Phi \in 2^{FBF_{\text{pred}}}$ un insieme finito di formule. Poiché Φ contiene un numero finito di variabili distinte, l'algoritmo termina sempre. Infatti, in una generica iterazione, o si stabilisce che Φ non è unificabile, o il numero di variabili distinte viene ridotto di uno, quindi il numero di iterazioni non può superare il numero di variabili distinte che compaiono nelle formule di Φ.
  Se Φ è unificabile l'algoritmo termina con successo e θ è un unificatore di Φ. Questo unificatore è proprio l'upg di Φ perché, indicato con θ_i il valore di θ dopo i iterazioni dell'algoritmo, per ogni altro unificatore θ' di Φ esiste una sostituzione θ''_i tale che $\theta' = \theta_i \circ \theta''_i$ come si vede procedendo per induzione su $i \in \mathbb{N}$:

  - Se $i = 0$ allora posto $\theta''_i = \theta'$ risulta $\theta' = \theta_i \circ \theta''_i$ perché $\theta_i = \varepsilon$.
  - Sia $i \geq 0$ e supponiamo che esista θ''_i tale che $\theta' = \theta_i \circ \theta''_i$. Ci sono due casi:
    * Se $\Phi\theta_i$ contiene una sola formula, allora il risultato segue banalmente dal fatto che in questo caso l'algoritmo termina.
    * Se $\Phi\theta_i$ contiene più di una formula, allora l'algoritmo esegue un'ulteriore iterazione. Poiché Φ è unificabile, esisteranno $x \in \mathit{Var}$ e $t \in \mathit{Ter}$ tali che x non compare in t e quindi $\theta_{i+1} = \theta_i \circ [{}^{t}/_{x}]$. Sia θ''_{i+1} la sostituzione ottenuta da θ''_i eliminando $[{}^{t\theta''_i}/_{x}]$. Allora $\theta_{i+1} \circ \theta''_{i+1} = \theta_i \circ [{}^{t}/_{x}] \circ \theta''_{i+1} = \theta_i \circ \theta''_{i+1} \circ [{}^{t\theta''_{i+1}}/_{x}]$ perché x non viene rimpiazzata in θ''_{i+1}. Poiché $\theta_i \circ \theta''_{i+1} \circ [{}^{t\theta''_{i+1}}/_{x}] = \theta_i \circ \theta''_{i+1} \circ [{}^{t\theta''_i}/_{x}]$ in quanto x non compare in t, risulta $\theta_{i+1} \circ \theta''_{i+1} = \theta_i \circ \theta''_{i+1} \circ [{}^{t\theta''_i}/_{x}] = \theta_i \circ \theta''_i = \theta'$ sfruttando l'ipotesi induttiva.

- Esempi:

  - L'insieme di formule:
    $$\begin{aligned}\Phi \;=\; \{&P(a, f(x, c), y),\\ &P(z, f(g(z'), y), y),\\ &P(a, f(g(a), y), c)\}\end{aligned}$$
    è unificabile perché:
    * Partendo con $\theta = \varepsilon$, le tre formule di $\Phi\theta$ differiscono sui simboli a, z, a, i quali sono unificabili ponendo $\theta = [{}^{a}/_{z}]$ alla prima iterazione dell'algoritmo.
    * Le tre formule di $\Phi\theta$ differiscono sui simboli $x, g(z'), g(a)$, i quali sono unificabili ponendo $\theta = [{}^{a}/_{z}, {}^{g(z')}/_{x}]$ alla seconda iterazione dell'algoritmo e $\theta = [{}^{a}/_{z}, {}^{g(z')}/_{x}, {}^{a}/_{z'}]$ alla terza iterazione dell'algoritmo.
    * Le tre formule di $\Phi\theta$ differiscono sui simboli c, y, y, i quali sono unificabili ponendo $\theta = [{}^{a}/_{z}, {}^{g(z')}/_{x}, {}^{a}/_{z'}, {}^{c}/_{y}]$ alla quarta iterazione dell'algoritmo. Questa sostituzione è l'upg di Φ e riduce tutte le formule di Φ alla unica formula $P(a, f(g(a), c), c)$.
  - L'insieme di formule:
    $$\begin{aligned}\Phi \;=\; \{&P(x_1, x_2, \ldots, x_n),\\ &P(f(x_0, x_0), f(x_1, x_1), \ldots, f(x_{n-1}, x_{n-1}))\}\end{aligned}$$
    è unificabile ma l'algoritmo di unificazione di Robinson impiega un tempo esponenziale rispetto ad n per stabilirlo. Infatti, l'upg di Φ contiene legami quali ${}^{f(x_0,x_0)}/_{x_1}, {}^{f(f(x_0,x_0),f(x_0,x_0))}/_{x_2}, \ldots$ dove il termine che sostituisce x_i, $1 \leq i \leq n$, contiene 2^i occorrenze di variabili e quindi l'occur check che deve essere effettuato prima di applicare la sostituzione ad x_i impiega un tempo proporzionale a 2^i. ■ftpl_15

11.5 Risoluzione di Robinson per la logica dei predicati

- Il metodo di risoluzione può essere esteso alla logica dei predicati facendo uso di sostituzioni ed unificazioni. In effetti, la regola di inferenza del metodo di risoluzione per la logica proposizionale (vedi Sez. 11.3) è un caso particolare della seguente regola:

$$\frac{\{\pi_{1,1},\ldots,\pi_{1,h}\} \subseteq c_1\theta_1 \qquad \{\pi_{2,1},\ldots,\pi_{2,k}\} \subseteq c_2\theta_2 \qquad \{\pi_{1,1},\ldots,\pi_{1,h},\neg\pi_{2,1},\ldots,\neg\pi_{2,k}\} \text{ unificabile}}{((c_1\theta_1 \setminus \{\pi_{1,1},\ldots,\pi_{1,h}\}) \cup (c_2\theta_2 \setminus \{\pi_{2,1},\ldots,\pi_{2,k}\}))\,\theta}$$

  dove:

  - c_1, c_2 sono due clausole della logica dei predicati. Esse saranno state ottenute trasformando nella forma normale congiuntiva $\bigwedge_{i=1}^{n}(\bigvee_{j=1}^{m_i}\pi'_{i,j})$ la matrice di una formula $\gamma \in FBF_{\text{pred,crs}}$ che è insoddisfacibile sse lo è $\neg\phi$, dove $\phi \in FBF_{\text{pred}}$ è la formula di partenza.
  - θ_1, θ_2 sono due sostituzioni tali che $c_1\theta_1, c_2\theta_2$ non hanno variabili in comune. Esse consentono di unificare (a meno della negazione iniziale) letterali come $P(x)$ e $\neg P(f(x))$ che sono congiuntamente insoddisfacibili ma non possono essere unificati per via dell'occur check.
  - $h, k \in \mathbb{N}_{\geq 1}$. Diversamente dal caso della logica proposizionale in cui si risolvono due soli letterali alla volta, h e k non sono necessariamente entrambi uguali ad uno. Se si imponesse $h = k = 1$, l'insieme di clausole $\{\{P(x), P(y)\}, \{\neg P(x), \neg P(y)\}\}$, che è insoddisfacibile perché x ed y sono quantificate universalmente e quindi il caso $x = y$ non può essere escluso, non sarebbe refutabile col metodo di risoluzione in quanto come ulteriori clausole si otterrebbero soltanto $\{P(y), \neg P(y)\}$ e $\{P(x), \neg P(x)\}$ senza mai arrivare alla generazione di $\square$.
  - θ è l'upg di $\{\pi_{1,1},\ldots,\pi_{1,h},\neg\pi_{2,1},\ldots,\neg\pi_{2,k}\}$. I $\pi_{1,i}$ sono tutti positivi/negativi, i $\pi_{2,j}$ sono tutti negativi/positivi e la negazione va introdotta per consentire l'unificazione di letterali opposti.

- In logica proposizionale la regola si riduce a $\dfrac{\{p\} \subseteq c_1 \qquad \{\neg p\} \subseteq c_2}{(c_1 \setminus \{p\}) \cup (c_2 \setminus \{\neg p\})}$.

- Esempio: Le due clausole $\{P(z), \neg Q(y), P(f(x))\}$ e $\{\neg P(x), \neg R(x)\}$ possono essere trasformate rispettivamente in $\{P(z), \neg Q(y), P(f(x))\}$ e $\{\neg P(x'), \neg R(x')\}$ mediante le sostituzioni $\theta_1 = \varepsilon$ e $\theta_2 = [^{x'}/_{x}]$ così da evitare variabili condivise. Le due clausole risultanti hanno come risolvente $\{\neg Q(y), \neg R(f(x))\}$ essendo $\theta = [^{f(x)}/_{z}, ^{f(x)}/_{x'}]$ l'upg di $\{P(z), P(f(x)), P(x')\}$.

- Anche nella logica dei predicati la regola di inferenza del metodo di risoluzione preserva i modelli delle clausole originarie e il sistema deduttivo basato su tale regola risulta essere corretto e completo rispetto all'insoddisfacibilità. Questo può essere dimostrato stabilendo dapprima un collegamento tra la regola di risoluzione proposizionale e la regola di risoluzione predicativa.

- Teorema (lifting lemma): Siano c_1, c_2 due clausole della logica dei predicati e siano c'_1, c'_2 due loro istanze ground, rispettivamente. Per ogni risolvente c' di c'_1, c'_2 (secondo la regola di risoluzione proposizionale) esiste una risolvente c di c_1, c_2 (secondo la regola di risoluzione predicativa) tale che c' è un'istanza ground di c.

- Dimostrazione: Date due clausole della logica dei predicati c_1, c_2, siano θ_1, θ_2 due sostituzioni tali che $c_1\theta_1, c_2\theta_2$ non hanno variabili in comune. Date due istanze ground c'_1, c'_2 di c_1, c_2, rispettivamente, risulta che c'_1, c'_2 sono anche istanze ground di $c_1\theta_1, c_2\theta_2$, rispettivamente, e quindi esistono due sostituzioni θ'_1, θ'_2 tali che $c'_1 = (c_1\theta_1)\theta'_1$ e $c'_2 = (c_2\theta_2)\theta'_2$. Sia $\theta' = \theta'_1 \circ \theta'_2$. Poiché θ'_1, θ'_2 rimpiazzano variabili distinte, risulta $c'_1 = (c_1\theta_1)\theta'$ e $c'_2 = (c_2\theta_2)\theta'$.

Al fine di evitare casi banali, supponiamo che la regola di risoluzione proposizionale sia applicabile a c'_1, c'_2. Sia dunque c' una risolvente di c'_1, c'_2, diciamo $c' = (c'_1 \setminus \{p\}) \cup (c'_2 \setminus \{\neg p\})$ dove $p \in c'_1$ e $\neg p \in c'_2$. Poiché i letterali p e $\neg p$ derivano dall'applicazione di θ' ai letterali di $c_1\theta_1, c_2\theta_2$, rispettivamente, esisteranno $h \in \mathbb{N}_{\geq 1}$ letterali $\pi_{1,1}, \ldots, \pi_{1,h}$ in $c_1\theta_1$ tali che $p = \pi_{1,1}\theta' = \ldots = \pi_{1,h}\theta'$ e $k \in \mathbb{N}_{\geq 1}$ letterali $\pi_{2,1}, \ldots, \pi_{2,k}$ in $c_2\theta_2$ tali che $\neg p = \pi_{2,1}\theta' = \ldots = \pi_{2,k}\theta'$. Pertanto θ' è un unificatore di $\{\pi_{1,1}, \ldots, \pi_{1,h}, \neg\pi_{2,1}, \ldots, \neg\pi_{2,k}\}$ e quindi la regola di risoluzione predicativa è applicabile a $c_1\theta_1, c_2\theta_2$. Poichè $\{\pi_{1,1}, \ldots, \pi_{1,h}, \neg\pi_{2,1}, \ldots, \neg\pi_{2,k}\}$ è unificabile, in virtù del teorema di Robinson questo insieme ha l'upg che denotiamo con θ e quindi $\theta' = \theta \circ \theta''$. Di conseguenza c' è un'istanza ground della risolvente $c = ((c_1\theta_1 \setminus \{\pi_{1,1}, \ldots, \pi_{1,h}\}) \cup (c_2\theta_2 \setminus \{\pi_{2,1}, \ldots, \pi_{2,k}\}))\,\theta$ di c_1, c_2 perché $c' = (c'_1 \setminus \{p\}) \cup (c'_2 \setminus \{\neg p\}) = ((c_1\theta_1)\theta' \setminus \{p\}) \cup ((c_2\theta_2)\theta' \setminus \{\neg p\}) = ((c_1\theta_1 \setminus \{\pi_{1,1}, \ldots, \pi_{1,h}\}) \cup (c_2\theta_2 \setminus \{\pi_{2,1}, \ldots, \pi_{2,k}\}))\,\theta' = ((c_1\theta_1 \setminus \{\pi_{1,1}, \ldots, \pi_{1,h}\}) \cup (c_2\theta_2 \setminus \{\pi_{2,1}, \ldots, \pi_{2,k}\}))\,(\theta \circ \theta'') = c\theta''$.

- Teorema: Siano c_1, c_2, c delle clausole della logica dei predicati. Se c è una risolvente di c_1 e c_2, allora $c_1 \models c$ e $c_2 \models c$.

- Dimostrazione: Poiché c_1 e c_2 ammettono una clausola risolvente, date due sostituzioni θ_1, θ_2 tali che $c_1\theta_1, c_2\theta_2$ non hanno variabili in comune risulta che esistono $h \in \mathbb{N}_{\geq 1}$ letterali $\pi_{1,1}, \ldots, \pi_{1,h}$ in $c_1\theta_1$ e $k \in \mathbb{N}_{\geq 1}$ letterali $\pi_{2,1}, \ldots, \pi_{2,k}$ in $c_2\theta_2$ tali che $\{\pi_{1,1}, \ldots, \pi_{1,h}, \neg\pi_{2,1}, \ldots, \neg\pi_{2,k}\}$ è unificabile. Indicato con θ l'upg di questo insieme, siano $c = ((c_1\theta_1 \setminus \{\pi_{1,1}, \ldots, \pi_{1,h}\}) \cup (c_2\theta_2 \setminus \{\pi_{2,1}, \ldots, \pi_{2,k}\}))\,\theta$ e $\pi = \pi_{1,1}\theta = \ldots = \pi_{1,h}\theta = \neg\pi_{2,1}\theta = \ldots = \neg\pi_{2,k}\theta$. Per ogni ambiente $\mathcal{E} \in Amb$ tale che $\mathcal{E} \models c_1$ e $\mathcal{E} \models c_2$, ci sono due casi:

  - Se $\mathcal{E} \models \pi$ allora da $\mathcal{E} \models (c_2\theta_2)\theta$ e $\mathcal{E} \not\models \neg\pi$ segue che $\mathcal{E} \models (c_2\theta_2)\theta \setminus \{\pi_{2,1}, \ldots, \pi_{2,k}\}\theta$ e quindi $\mathcal{E} \models c$.
  - Se $\mathcal{E} \not\models \pi$ allora da $\mathcal{E} \models (c_1\theta_1)\theta$ segue che $\mathcal{E} \models (c_1\theta_1)\theta \setminus \{\pi_{1,1}, \ldots, \pi_{1,k}\}\theta$ e quindi $\mathcal{E} \models c$.

- Dato un insieme di formule $\Phi \in 2^{FBF_{\text{pred}}}$, diciamo che esso è:

  - Soddisfacibile sse esiste $\mathcal{E} \in Amb$ tale che $\mathcal{E} \models \phi$ per ogni $\phi \in \Phi$.
  - Finitamente soddisfacibile sse ogni sottoinsieme finito di Φ è soddisfacibile.

- Teorema (di compattezza): Un insieme di formule della logica dei predicati è soddisfacibile sse è finitamente soddisfacibile.

- Corollario: Un insieme di formule della logica dei predicati è insoddisfacibile sse ammette un sottoinsieme finito tale che la congiunzione delle sue formule è insoddisfacibile.

- Teorema (di risoluzione): Un insieme di clausole della logica dei predicati è insoddisfacibile sse è refutabile.

- Dimostrazione: In virtù del corollario al teorema di compattezza, un insieme di clausole è insoddisfacibile sse ammette un sottoinsieme finito insoddisfacibile, quindi possiamo ragionare su un insieme finito di clausole C.
  Se C è refutabile allora C è insoddisfacibile. Infatti, se C fosse soddisfacibile allora esisterebbe un ambiente $\mathcal{E} \in Amb$ che soddisfa tutte le clausole di C. Poiché C è refutabile e il metodo di risoluzione preserva i modelli delle clausole originarie, $\mathcal{E}$ sarebbe un modello anche di $\square$, ma ciò è assurdo perchè $\square$ non ammette modelli.
  Se C è insoddisfacibile allora in virtù del teorema di Herbrand esiste un insieme C' di istanze ground delle clausole di C che è insoddisfacibile. Dal teorema di risoluzione per la logica proposizionale segue che C' è refutabile. Dal lifting lemma segue che anche C è refutabile.

- Dato un insieme di clausole C della logica dei predicati, il metodo di risoluzione determina:
  - l'insieme $ris(C) = C \cup \{c \mid c \text{ risolvente di } c_1, c_2 \in C\}$ quando viene applicato una sola volta;
  - l'insieme $ris^*(C) = \bigcup_{n \in \mathbb{N}} ris^n(C)$ quando viene applicato un numero arbitrario di volte, dove:
$$ris^n(C) = \begin{cases} C & \text{se } n = 0 \\ ris(ris^{n-1}(C)) & \text{se } n > 0 \end{cases}$$

  Poiché $ris^*(C) \equiv C$, abbiamo che C è refutabile sse $\square \in ris^*(C)$.

- Sulla base del metodo di risoluzione possiamo costruire il seguente algoritmo che opera per refutazione al fine di stabilire la validità di una formula $\phi \in FBF_{\text{pred}}$:
  1. Trasformare $\neg\phi$ in una formula $\gamma \in FBF_{\text{pred,crs}}$ che è insoddisfacibile sse lo è $\neg\phi$ (vedi Sez. 11.1).
  2. Trasformare la matrice γ' di γ in una formula γ'' in forma normale congiuntiva (vedi Sez. 11.1).
  3. Trasformare γ'' in un insieme di clausole C dove le clausole sono insiemi di letterali.
  4. Ripetere i seguenti passi:
     (a) assegnare C a C';
     (b) assegnare $ris(C)$ a C;

     finché non vale che $\square \in C$ oppure $C = C'$.
  5. Se $\square \in C$ segnalare che ϕ è valida, altrimenti segnalare che ϕ non è valida.

- Diversamente dal caso della logica proposizionale, l'algoritmo mostrato sopra potrebbe non terminare in quanto il problema della validità è soltanto semi-decidibile nella logica dei predicati. Diversamente dall'algoritmo di refutazione basato sulla teoria di Herbrand, questo algoritmo di refutazione basato sul metodo di risoluzione e sul calcolo dell'upg evita di generare tutte le istanze ground della matrice della formula ottenuta da quella di partenza e quindi ha delle prestazioni migliori.

- Esempi:
  - L'insieme soddisfacibile di clausole $C = \{\{P(0)\}, \{\neg P(x), P(succ(x))\}\}$ ottenuto dalla formulazione in logica dei predicati del principio di induzione determina un insieme $ris^*(C)$ che è infinito in quanto contiene elementi della forma $P(0), P(succ(0)), P(succ(succ(0))), \ldots$ che sono in corrispondenza biunivoca coi numeri naturali.
  - Riconsideriamo il paradosso del barbiere e utilizziamo l'algoritmo di refutazione basato sul metodo di risoluzione e sul calcolo dell'upg per dimostrare che la formula $\forall x\,(Rade(b, x) \leftrightarrow (\neg Rade(x, x)))$ è insoddisfacibile:
    * Poiché siamo interessati all'insoddisfacibilità della formula, non dobbiamo negarla.
    * La formula appartiene già ad $FBF_{\text{pred,crs}}$.
    * La matrice della formula è $(Rade(b, x) \leftrightarrow (\neg Rade(x, x)))$.
    * La matrice è equivalente a $(((\neg Rade(b, x)) \vee (\neg Rade(x, x))) \wedge (Rade(x, x) \vee Rade(b, x)))$ che è in forma normale congiuntiva.
    * Il corrispondente insieme di clausole è $\{\{\neg Rade(b, x), \neg Rade(x, x)\}, \{Rade(x, x), Rade(b, x)\}\}$.
    * Poiché x è una variabile comune alle due clausole, applichiamo ad esse le sostituzioni $[^{x_1}/_x]$ e $[^{x_2}/_x]$, rispettivamente, ottenendo pertanto $\{\{\neg Rade(b, x_1), \neg Rade(x_1, x_1)\}, \{Rade(x_2, x_2), Rade(b, x_2)\}\}$.
    * L'insieme di letterali $\{\neg Rade(b, x_1), \neg Rade(x_1, x_1), \neg Rade(x_2, x_2), \neg Rade(b, x_2)\}$ risulta essere unificabile.
    * Il suo upg è $[^b/_{x_1}, ^b/_{x_2}]$.
    * L'applicazione della risoluzione alle due clausole dà luogo a $\square$. ■fltpl_16

11.6 Prolog: clausole di Horn e strategia di risoluzione SLD

- La programmazione dichiarativa è un paradigma di programmazione in cui la computazione viene espressa senza descriverne il flusso di controllo. Diversamente da un programma imperativo, un programma dichiarativo specifica *cosa* deve essere fatto, ma *non come* deve essere fatto. Questo comporta un cambio di mentalità nell'attività di programmazione, ben evidenziato nel 1978 da Backus (uno degli sviluppatori del Fortran agli inizi degli anni 1950) nel suo famoso articolo intitolato "Can Programming Be Liberated from the Von Neumann Style? A Functional Style and Its Algebra of Programs".

- I programmi dichiarativi *non* sono sequenze di istruzioni finalizzate a modificare il contenuto della memoria, ma sequenze di espressioni logico-matematiche in cui il valore delle variabili non cambia più una volta che è stato attribuito.

- Le esecuzioni dei programmi dichiarativi *non* sono sequenze di passi che hanno l'effetto di cambiare il contenuto della memoria, ma sono assimilabili a deduzioni in un sistema formale.

- Principali differenze tra programmazione dichiarativa e programmazione imperativa:

  - I programmi dichiarativi sono espressi ad un livello concettuale più alto dei programmi imperativi, così come i loro esecutori operano ad un livello concettuale più alto rispetto ai compilatori e agli interpreti convenzionali. Questo favorisce la verifica di correttezza dei programmi dichiarativi.
  - Non ci sono effetti collaterali quali la modifica dello stato della memoria o dei dispositivi di input/output. Anzi, il concetto di stato della computazione inteso come il contenuto della memoria non esiste più nella programmazione dichiarativa perché i dati sono immutabili.
  - La conseguente trasparenza referenziale fa sì che venga restituito il medesimo risultato ogni volta che un'espressione viene valutata sugli stessi argomenti. Il risultato può quindi essere calcolato una volta per tutte e poi riutilizzato ogni volta che serve.
  - Se non ci sono dipendenze di dati tra due espressioni, il loro ordine può essere invertito. Ciò dà al compilatore o all'interprete la possibilità di scegliere la strategia più opportuna dal punto di vista prestazionale per valutare le espressioni, come pure la possibilità di valutarle in parallelo.
  - L'iterazione viene sempre descritta attraverso la ricorsione.

- Il primo filone ad affermarsi nell'ambito della programmazione dichiarativa fu quello della programmazione funzionale, con linguaggi come Lisp (sviluppato alla fine degli anni 1950 da McCarthy), ML (sviluppato negli anni 1970 da Milner), FP, Scheme, Haskell, Erlang, Objective Caml ed F#. Quasi tutti questi linguaggi sono basati sul λ-calcolo di Church, il quale costituisce un sistema formale per studiare la definizione e l'applicazione di funzioni matematiche calcolabili.

- Il secondo filone ad affermarsi nell'ambito della programmazione dichiarativa fu quello della programmazione logica. Il principale linguaggio di questo paradigma è il Prolog (PROgramming in LOGic), sviluppato nel 1972 da Colmerauer e Roussel sulla base del metodo di risoluzione di Robinson e dell'interpretazione procedurale dell'implicazione data da Kowalski, che cominciò ad essere riconosciuto come un utile strumento di programmazione grazie al compilatore implementato nel 1977 da Warren. Analogamente al Lisp, anche il Prolog ebbe origine da ricerche nel campo dell'intelligenza artificiale.

- Benché l'algoritmo di refutazione basato sul metodo di risoluzione di Robinson sia notevolmente più efficiente dell'algoritmo di refutazione basato sulla teoria di Herbrand, l'applicazione ripetuta della regola di inferenza per risoluzione può generare un numero eccessivo di clausole irrilevanti o ridondanti. Per evitare ciò, nei linguaggi di programmazione logica come il Prolog si utilizzano delle strategie di risoluzione che scelgono in modo mirato le clausole da cui generare una risolvente. Queste strategie si dividono in complete ed incomplete a seconda che riescano sempre a derivare $\square$ da un insieme insoddisfacibile di clausole oppure no.

- Dato un insieme di clausole C, diciamo che una dimostrazione per risoluzione $c_1 \dots c_n$ a partire da C è ottenuta applicando:
  - La strategia di risoluzione lineare sse per ogni $i = 2, \dots, n$ la clausola c_i è la risolvente della clausola precedente c_{i-1} e di una clausola $c' \in C \cup \{c_1, \dots, c_{i-1}\}$.
  - La strategia di risoluzione di input sse per ogni $i = 2, \dots, n$ la clausola c_i è la risolvente di una clausola di input $c \in C$ e di una clausola $c' \in C \cup \{c_1, \dots, c_{i-1}\}$.
  - La strategia di risoluzione SLD sse per ogni $i = 2, \dots, n$ la clausola c_i è la risolvente della clausola precedente c_{i-1} e di una clausola di input $c \in C$ (combinazione efficiente di risoluzione lineare e risoluzione di input).
- Teorema: La strategia di risoluzione lineare è completa per tutti gli insiemi di clausole.
- Teorema: La strategie di risoluzione di input ed SLD sono complete per gli insiemi di clausole di Horn.
- Il linguaggio Prolog consente di definire delle relazioni e di compiere delle interrogazioni su di esse. Le clausole di Horn costituite da fatti e regole vengono usate in Prolog per rappresentare informazioni sotto forma di relazioni tra dati. Il metodo di risoluzione di Robinson basato sulla strategia SLD viene impiegato – insieme ad un algoritmo di unificazione efficiente – per risolvere problemi definiti attraverso clausole di Horn costituite da vincoli, i quali vengono interpretati come obiettivi da raggiungere istanziandone le parti eventualmente non specificate.
- La sintassi di un programma Prolog prevede una sequenza di clausole di Horn date da fatti e regole che sono terminate dal punto:
  - Ogni fatto presente nel programma rappresenta un'informazione ed è costituito da una formula atomica della logica dei predicati:
    ```
    P(t_1, ..., t_n).
    ```
  - Ogni regola presente nel programma rappresenta una relazione tra più informazioni:
    ```
    P(t_1, ..., t_n) :- Q_1(t_1_1, ..., t_1_m1), ..., Q_k(t_k_1, ..., t_k_mk).
    ```
    La parte sinistra della regola è una formula atomica della logica dei predicati detta testa, il simbolo `:-` rappresenta l'implicazione nel verso $\leftarrow$ e la parte destra della regola è una sequenza non vuota di formule atomiche della logica dei predicati detta corpo. Le virgole che separano le formule presenti all'interno del corpo della regola rappresentano delle congiunzioni.
- L'esecuzione di un programma Prolog viene attivata da un obiettivo. L'obiettivo è descritto come una regola senza testa in cui il simbolo `:-` è sostituito dal simbolo `?-`, cioè è una clausola di Horn data da un vincolo. Il raggiungimento dell'obiettivo viene perseguito applicando il metodo di risoluzione basato sulla strategia SLD ai fatti e alle regole del programma e all'obiettivo dato. In caso di refutazione, se l'obiettivo non è una clausola ground allora le sue variabili vengono istanziate sulla base delle unificazioni effettuate durante le applicazioni della regola di inferenza per risoluzione.
- In un contesto guidato dagli obiettivi come quello del Prolog, ogni regola di un programma si presta ad una duplice intepretazione:
  - Il significato dichiarativo della regola è che la formula di testa `P(t_1, ..., t_n)` è vera se le formule del corpo `Q_1(t_1_1, ..., t_1_m1), ..., Q_k(t_k_1, ..., t_k_mk)` sono vere.
  - Il significato procedurale della regola è che per raggiungere l'obiettivo `P(t_1, ..., t_n)` bisogna prima raggiungere gli obiettivi `Q_1(t_1_1, ..., t_1_m1), ..., Q_k(t_k_1, ..., t_k_mk)`.
- Dati un programma Prolog (insieme di clausole C) ed un obiettivo (clausola iniziale c_1), in virtù dell'interpretazione procedurale bisogna trovare per tutte le formule atomiche dell'obiettivo considerate da sinistra a destra dei fatti che siano unificabili con quelle formule o delle regole le cui teste siano unificabili con quelle formule. Nel caso la risoluzione avvenga con una regola, bisogna poi procedere nello stesso modo con le formule atomiche del corpo della regola considerate da sinistra a destra, seguendo l'ordine LIFO per coerenza con la strategia SLD. Per motivi di efficienza, conviene che le regole ricorsive siano ricorsive a destra, cioè che i predicati presenti nella testa di tali regole compaiano nelle formule atomiche più a destra del corpo delle regole.

- Poiché ogni formula atomica presente in un obiettivo potrebbe essere unificabile con più fatti e teste di regole di un programma, in Prolog la risoluzione della formula atomica più a sinistra di un obiettivo viene esaminata rispetto a tutte le clausole del programma considerate nell'ordine in cui sono scritte. Per motivi di efficienza, nei programmi conviene quindi elencare i fatti prima delle regole.

- L'esecuzione di un programma Prolog su un obiettivo può essere rappresentata graficamente tramite un albero e-o, così chiamato perché ogni nodo rappresenta un obiettivo (congiunzione) e ha tanti nodi figli quanti sono i fatti e le regole alternativi tra loro con i quali può avvenire la risoluzione (disgiunzione). Durante l'applicazione della risoluzione, l'albero viene costruito in profondità per via dell'ordine LIFO imposto dalla strategia SLD.

- La radice dell'albero e-o contiene l'obiettivo iniziale, mentre ogni foglia può contenere $\square$ oppure un obiettivo alla cui formula atomica più a sinistra non è applicabile la risoluzione. Ogni ramo dell'albero è etichettato con i legami creati per unificazione durante la corrispondente applicazione della regola di inferenza per risoluzione. Ogni cammino dalla radice ad una foglia contenente $\square$ è un cammino di successo, mentre tutti gli altri cammini sono cammini di fallimento o cammini infiniti (considerare le clausole di un programma Prolog sempre nello stesso ordine rende la strategia SLD incompleta).

- Esempio: Consideriamo il seguente programma Prolog:

```
genitore(giovanni, andrea).
genitore(andrea, paolo).
antenato(X, Y) :- genitore(X, Y).
antenato(X, Y) :- genitore(X, Z), antenato(Z, Y).
```

dove il predicato `genitore(X, Y)` (risp. `antenato(X, Y)`) indica che `X` è genitore (risp. antenato) di `Y`. Osserviamo che ogni clausola che contiene delle variabili ha una quantificazione universale implicita su tutte quelle variabili, quindi le due regole del programma dato vanno intese nel seguente modo:

$\forall X \forall Y \, (genitore(X, Y) \rightarrow antenato(X, Y))$
$\forall X \forall Y \forall Z \, (genitore(X, Z) \wedge antenato(Z, Y) \rightarrow antenato(X, Y))$

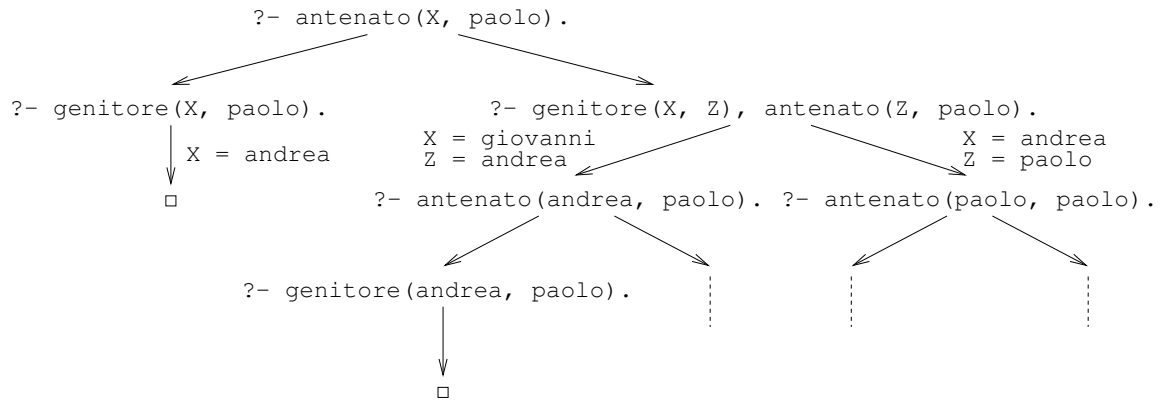
Consideriamo poi l'obiettivo:

```
?- antenato(X, paolo).
```

Osserviamo che ogni obiettivo che contiene delle variabili ha una quantificazione esistenziale implicita su tutte quelle variabili, quindi l'obiettivo dato va inteso nel seguente modo:

$\exists X \, (antenato(X, paolo))$

che dal punto di vista procedurale consiste nel trovare un antenato della persona specificata sulla base delle informazioni fornite dal programma. Poiché l'esecuzione di un programma Prolog procede per refutazione dell'obiettivo, l'obiettivo deve essere negato prima di iniziare ad applicare la regola di inferenza per risoluzione e quindi la quantificazione esistenziale che precede eventualmente l'obiettivo diventa universale.
L'albero e-o risultante è il seguente:



in cui si nota la presenza di due cammini di successo (soluzioni `X = andrea` e `X = giovanni`) e di diversi cammini di fallimento. ■fltpl_17

11.7 Prolog: termini e predicati

- I caratteri utilizzabili all'interno di un programma Prolog coincidono con quelli utilizzabili nella maggior parte degli altri linguaggi di programmazione. In particolare abbiamo le 26 lettere minuscole, le 26 lettere maiuscole, le 10 cifre decimali, il sottotratto, la barra verticale, i simboli di punteggiatura, le parentesi tonde e quadrate, gli operatori aritmetici e relazionali e i caratteri di spaziatura.
- I caratteri possono essere combinati per formare termini così classificati:
  - Atomi che si dividono in:
    - Costanti costituite da:
      - numeri interi e reali;
      - sequenze di lettere, cifre e sottotratti che iniziano con una minuscola;
      - sequenze racchiuse tra apici di lettere, cifre e sottotratti che iniziano con una maiuscola.
    - Variabili costituite da:
      - sequenze di lettere, cifre e sottotratti che iniziano con una maiuscola;
      - singoli sottotratti `_`, che rappresentano un qualsiasi valore.
  - Termini composti costituiti dalla concatenazione di:
    - Funtori che sono come le costanti non numeriche, cioè costituiti da:
      - sequenze di lettere, cifre e sottotratti che iniziano con una minuscola;
      - sequenze racchiuse tra apici di lettere, cifre e sottotratti che iniziano con una maiuscola.
    - Argomenti separati da virgole e racchiusi tra parentesi tonde, costituiti da:
      - atomi;
      - termini composti.
  - Dati strutturati che si dividono in:
    - Stringhe costituite da sequenze di caratteri racchiusi tra virgolette.
    - Liste costituite da sequenze di elementi separati da virgole e racchiusi tra parentesi quadre, le quali sono dotate di un operatore | per distinguere un certo numero di elementi iniziali dai restanti elementi.
- I simboli di predicato seguono le stesse convenzioni dei funtori.
- Esempi: `10` e `-13.5` sono costanti numeriche, `gatto` e `'Proprietario_di'` sono costanti simboliche o funtori, `X` e `Risultato_operazione` sono variabili, `"ciao!"` e `"Divisione illegale"` sono stringhe, `[]` e `[1, 2, 3]` sono liste.
- Esempi relativi alle liste:
  - Predicato per stabilire se una sequenza di caratteri è una lista:

    ```
    lista([]).
    lista([X | L]).                    /* lista([_ | _]). */
    ```

  - Predicato per stabilire se un elemento appartiene ad una lista:

    ```
    membro(X, [X | L]).                /* membro(X, [X | _]).                */
    membro(X, [Y | L]) :- membro(X, L).   /* membro(X, [_ | L]) :- membro(X, L). */
    ```

    Dati gli obiettivi:

    ```
    ?- membro(2, []).
    ?- membro(2, [0, 2, 4]).
    ?- membro(Z, [1, 3, 5]).
    ```

    abbiamo che il primo fallisce, il secondo ha successo ed il terzo dà luogo a tanti cammini di successo (ciascuno con la propria istanziazione di `Z`) quanti sono gli elementi presenti nella lista. Dunque, a seconda di come viene specificato l'input, questo predicato può essere usato sia per cercare un elemento in una lista, sia per attraversare tutti gli elementi di una lista.

  - Predicato per stabilire se una lista è un prefisso di un'altra lista:

```
prefisso([], L) :- lista(L).
prefisso([X | L1], [X | L2]) :- prefisso(L1, L2).
```

  - Predicato per stabilire se una lista è un suffisso di un'altra lista:

```
suffisso([], L) :- lista(L).
suffisso(L, L) :- lista(L).
suffisso(L1, [_ | L2]) :- suffisso(L1, L2).
```

  - Predicato per stabilire se una lista è una sottolista di un'altra lista:

```
sottolista(L1, L2) :- prefisso(L1, L2).
sottolista(L1, [_ | L2]) :- sottolista(L1, L2).
```

  - Predicato per aggiungere un elemento all'inizio di una lista:

```
inserisci_elem(X, L, [X | L]).
```

  - Predicato per aggiungere un elemento alla fine di una lista:

```
accoda_elem(X, [], [X]).
accoda_elem(X, [Y | L], [Y | LX]) :- accoda_elem(X, L, LX).
```

  - Predicato per concatenare due liste:

```
concatena(L, [], L) :- lista(L).
concatena(L1, [X | L2], L) :- accoda_elem(X, L1, L1X), concatena(L1X, L2, L).
```

  - Predicato per invertire l'ordine degli elementi di una lista:

```
inverti([], []).
inverti([X | L1], L2) :- inverti(L1, L1Inv), accoda_elem(X, L1Inv, L2).
```

- Al fine di agevolare l'attività di programmazione, il Prolog mette a disposizione dei predicati predefiniti. I seguenti simboli rappresentano i consueti operatori aritmetici e relazionali in notazione infissa su termini di tipo numerico e forzano la valutazione di espressioni aritmetiche in un contesto logico:
  - I simboli `+`, `-`, `*`, `/`, `mod` rappresentano rispettivamente le operazioni di addizione, sottrazione, moltiplicazione, divisione, resto della divisione su valori numerici (sono funtori, non predicati).
  - Il simbolo `is` rappresenta un predicato che ha successo sse il suo operando sinistro è unificabile con il valore numerico del suo operando destro (tutte le variabili eventualmente presenti nell'operando destro devono essere già state istanziate). Questo è l'unico predicato che consente di valutare un'espressione aritmetica in Prolog e di assegnarne il valore ad una variabile.
  - I simboli `=:=`, `=\=`, `>`, `>=`, `<`, `=<` rappresentano rispettivamente i predicati di uguale-a, diverso-da, maggiore-di, maggiore-uguale, minore-di, minore-uguale su valori numerici (tutte le variabili eventualmente presenti nei due operandi devono essere già state istanziate).
- I seguenti simboli costituiscono delle estensioni degli operatori di uguaglianza in notazione infissa a termini di tipo arbitrario (quindi non necessariamente numerico):
  - Il simbolo `=` rappresenta un predicato che ha successo sse i suoi due operandi sono unificabili e in tal caso produce il loro upg (non effettua l'occur check).
  - Il simbolo `\=` rappresenta un predicato che ha successo sse i suoi due operandi non sono unificabili.
  - Il simbolo `==` rappresenta un predicato che ha successo sse i suoi due operandi sono sintatticamente identici (delle variabili istanziate eventualmente presenti nei due operandi viene considerato il valore anziché il nome).
  - Il simbolo `\==` rappresenta un predicato che ha successo sse i suoi due operandi non sono sintatticamente identici.

- Esempi:

```
?- 3 is 1 + 2.          ?- 2 + 1 =:= 4 - 1.    ?- a < -30.      ?- f(a, Y) = f(Z, g(7)).
  yes                     yes                    domain error     yes Z=a, Y=g(7)
?- 2 + 1 is 4 - 1.      ?- 2.5 > 9.            ?- 2 = 2.        ?- X == 2.
  no                      no                     yes              no
?- X is 1 + 2.          ?- Z < 15.             ?- a = b.        ?- X = 2, X == 2.
  yes X=3                 instantiation error    no               yes X=2
?- 8 is Y * 2.          ?- Z is 4, Z < 15.     ?- X = 2.
  instantiation error     yes Z=4                yes X=2
```

- Esempi relativi alle funzioni numeriche:
  - Predicato per calcolare il fattoriale di un numero naturale:

```
fattoriale(0, 1).
fattoriale(N, F) :- N > 0, N1 is N - 1, fattoriale(N1, F1), F is N * F1.
```

  - Predicato per calcolare l'n-esimo numero di Fibonacci ($n \geq 1$):

```
fibonacci(1, 1).
fibonacci(2, 1).
fibonacci(N, F) :- N > 2, N1 is N - 1, N2 is N - 2,
                   fibonacci(N1, F1), fibonacci(N2, F2), F is F1 + F2.
```

■fltpl_18

- Ulteriori esempi relativi alle liste:
  - Il predicato che stabilisce se un elemento appartiene ad una lista può essere reso più efficiente ridefinendolo come segue:

```
membro(X, [X | _]).
membro(X, [Y | L]) :- X \== Y, membro(X, L).
```

    In questo modo le due clausole sono alternative tra loro e quindi al più una di esse può dare luogo ad un'esecuzione di successo. Per esempio, nel caso in cui l'obiettivo sia il seguente:

```
?- membro(1, [3, 1, 5, 1, 7].
```

    c'è un'unica esecuzione di successo che termina quando il secondo argomento diventa `[1, 5, 1, 7]`. La vecchia definizione del predicato avrebbe invece avuto due esecuzioni di successo.
  - Predicato per calcolare il numero di elementi presenti in una lista:

```
lunghezza([], 0).
lunghezza([_ | L], N) :- lunghezza(L, M), N is M + 1.
```

  - Predicato per fondere ordinatamente due liste ordinate di valori numerici:

```
fondi(L, [], L) :- lista(L).
fondi([], L, L) :- L \== [], lista(L).
fondi([X | L1], [Y | L2], [X | L]) :- X < Y, fondi(L1, [Y | L2], L).
fondi([X | L1], [Y | L2], [X, Y | L]) :- X =:= Y, fondi(L1, L2, L).
fondi([X | L1], [Y | L2], [Y | L]) :- X > Y, fondi([X | L1], L2, L).
```

- Esempi relativi agli algoritmi di ordinamento:
  - Mergesort:

```
mergesort([], []).
mergesort([X], [X]).
mergesort([X1, X2 | L], LOrd) :- dimezza([X1, X2 | L], L1, L2),
                                 mergesort(L1, L1Ord), mergesort(L2, L2Ord),
                                 fondi(L1Ord, L2Ord, LOrd).
dimezza([], [], []).
dimezza([X], [X], []).
dimezza([X1, X2 | L], [X1 | L1], [X2 | L2]) :- dimezza(L, L1, L2).
```

  - Quicksort:

```
quicksort([], []).
quicksort([P | L], LOrd) :- partiziona(L, P, Inf, Sup),
                            quicksort(Inf, InfOrd), quicksort(Sup, SupOrd),
                            concatena(InfOrd, [P | SupOrd], LOrd).
partiziona([], P, [], []).
partiziona([X | L], P, [X | Inf], Sup) :- X < P, partiziona(L, P, Inf, Sup).
partiziona([X | L], P, Inf, [X | Sup]) :- X >= P, partiziona(L, P, Inf, Sup).
```

- Il Prolog mette a disposizione dei predicati predefiniti per discriminare tra differenti tipi di termini:
  - `var(T)` ha successo sse `T` è una variabile non istanziata.
  - `nonvar(T)` ha successo sse `T` non è una variabile non istanziata.
  - `integer(T)` ha successo sse `T` è istanziato con una costante numerica intera.
  - `float(T)` ha successo sse `T` è istanziato con una costante numerica reale.
  - `atom(T)` ha successo sse `T` è istanziato con una costante non numerica.
  - `compound(T)` ha successo sse `T` è istanziato con un termine composto.
  - `functor(T, F, N)` ha successo sse `T` è un termine composto il cui funtore ha nome `F` ed è applicato ad `N` argomenti.
  - `arg(N, T, A)` ha successo sse `T` è un termine composto il cui `N`-esimo argomento è `A`.
- Ulteriori esempi relativi alle funzioni numeriche:
  - Predicato per calcolare la somma o la differenza a seconda del tipo degli operandi:

```
somma_diff(X, Y, Z) :- nonvar(X), nonvar(Y), Z is X + Y.
somma_diff(X, Y, Z) :- nonvar(X), nonvar(Z), Y is Z - X.
somma_diff(X, Y, Z) :- nonvar(Y), nonvar(Z), X is Z - Y.
```

  - Predicato per calcolare il prodotto o il quoziente a seconda del tipo degli operandi:

```
prod_quoz(X, Y, Z) :- nonvar(X), nonvar(Y), Z is X * Y.
prod_quoz(X, Y, Z) :- nonvar(X), nonvar(Z), X =\= 0, Y is Z / X.
prod_quoz(X, Y, Z) :- nonvar(X), nonvar(Z), X =:= 0, Z =:= 0.
prod_quoz(X, Y, Z) :- nonvar(Y), nonvar(Z), Y =\= 0, X is Z / Y.
prod_quoz(X, Y, Z) :- nonvar(Y), nonvar(Z), Y =:= 0, Z =:= 0.
```

  - Predicato per calcolare quoziente e resto della divisione tra due numeri naturali:

```
divisione(X, Y, Q, R) :- integer(X), integer(Y), X >= 0, Y > 0,
                         Q is X / Y, R is X mod Y.
```

- Esempi relativi a termini e loro unificazione:
  - Predicato per stabilire se un termine è ground:

```
ground(T) :- integer(T).
ground(T) :- float(T).
ground(T) :- atom(T).
ground(T) :- compound(T), functor(T, _, N), ground_tutti_arg(N, T).
ground_tutti_arg(0, _).
ground_tutti_arg(N, T) :- N > 0, arg(N, T, A), ground(A),
                          N1 is N - 1, ground_tutti_arg(N1, T).
```

  - Predicato per stabilire se un termine è un sottotermine di un altro termine:

```
sottoterm(T, T).
sottoterm(T1, T2) :- T1 \== T2, compound(T2), functor(T2, _, N), sottoarg(T1, N, T2).
sottoarg(T1, N, T2) :- arg(N, T2, A), sottoterm(T1, A).
sottoarg(T1, N, T2) :- N > 1, N1 is N - 1, sottoarg(T1, N1, T2).
```

  - Algoritmo di unificazione di Robinson (stesso effetto del predicato = a parte l'occur check):

```
unifica(T1, T2, []) :- integer(T1), integer(T2), T1 =:= T2.
unifica(T1, T2, []) :- float(T1), float(T2), T1 =:= T2.
unifica(T1, T2, []) :- atom(T1), atom(T2), T1 == T2.
unifica(T1, T2, [[T1, T2]]) :- var(T1), var(T2).
unifica(T1, T2, [[T2, T1]]) :- var(T1), integer(T2).
unifica(T1, T2, [[T2, T1]]) :- var(T1), float(T2).
unifica(T1, T2, [[T2, T1]]) :- var(T1), atom(T2).
unifica(T1, T2, [[T2, T1]]) :- var(T1), compound(T2), occur_check(T1, T2).
unifica(T1, T2, [[T1, T2]]) :- var(T2), integer(T1).
unifica(T1, T2, [[T1, T2]]) :- var(T2), float(T1).
unifica(T1, T2, [[T1, T2]]) :- var(T2), atom(T1).
unifica(T1, T2, [[T1, T2]]) :- var(T2), compound(T1), occur_check(T2, T1).
unifica(T1, T2, L) :- compound(T1), compound(T2),
                      functor(T1, F, N), functor(T2, F, N),
                      unifica_tutti_arg(N, T1, T2, L).
unifica_tutti_arg(0, _, _, []).
unifica_tutti_arg(N, T1, T2, L) :- N > 0, arg(N, T1, A1), arg(N, T2, A2),
                                   unifica(A1, A2, LA),
                                   N1 is N - 1, unifica_tutti_arg(N1, T1, T2, L1),
                                   coerenti(L1, LA), concatena(L1, LA, L).
coerenti(_, []).
coerenti(L1, [[T, X] | L2]) :- no_stessa_v_diff_t(X, T, L1), coerenti(L1, L2).
no_stessa_v_diff_t(_, _, []).
no_stessa_v_diff_t(X, T, [[_, Y] | L]) :- X \== Y, no_stessa_v_diff_t(X, T, L).
no_stessa_v_diff_t(X, T, [[T, X] | L]) :- no_stessa_v_diff_t(X, T, L).
occur_check(_, T) :- integer(T).
occur_check(_, T) :- float(T).
occur_check(_, T) :- atom(T).
occur_check(X, T) :- var(T), X \== T.
occur_check(X, T) :- compound(T), functor(T, _, N), occur_check_tutti_arg(X, N, T).
occur_check_tutti_arg(_, 0, _).
occur_check_tutti_arg(X, N, T) :- N > 0, arg(N, T, A), occur_check(X, A),
                                  N1 is N - 1, occur_check_tutti_arg(X, N1, T).
```

11.8 Prolog: input/output, taglio, negazione, miscellanea

- Il Prolog mette a disposizione dei predicati predefiniti per effettuare operazioni di input/output:
  - `read(T)` ha successo sse è possibile acquisire tramite tastiera una sequenza di caratteri che finisce con il punto (l'acquisizione termina premendo il tasto invio) e questa sequenza escluso il punto è unificabile con `T`.
  - `write(T)` ha successo stampando `T` sullo schermo.
  - `nl` ha successo stampando un'andata a capo sullo schermo.
- Esempi:
  - Predicato per stampare un valore e andare a capo sullo schermo:

```
write_nl(T) :- write(T), nl.
```

  - Predicato per risolvere il problema delle torri di Hanoi:

```
hanoi(1, P, A, _) :- stampa_mossa(P, A).
hanoi(N, P, A, I) :- N > 1, N1 is N - 1, hanoi(N1, P, I, A),
                     stampa_mossa(P, A), hanoi(N1, I, A, P).
stampa_mossa(P, A) :- write("sposta da "), write(P), write(" a "), write_nl(A).
```

- Dati un programma Prolog ed un obiettivo, la costruzione in profondità del corrispondente albero e-o viene sospesa quando si raggiunge la foglia di un cammino di successo. A quel punto l'interprete chiede all'utente se vuole terminare l'esecuzione oppure se vuole continuare l'esecuzione in cerca di altre soluzioni del problema. In quest'ultimo caso, l'esecuzione riprende risalendo dalla foglia verso la radice fino ad incontrare il primo nodo la cui formula atomica più a sinistra ammette un'unificazione alternativa a quelle che avevano condotto ai cammini precedentemente intrapresi.
- Poiché non è noto a priori se l'esecuzione di un programma Prolog rispetto ad un obiettivo avrà successo, la costruzione in profondità del corrispondente albero e-o equivale ad agire per tentativi e revoche (backtrack). Prima si tenta col cammino più a sinistra. Se questo è un cammino di fallimento oppure è un cammino di successo ma l'utente vuole trovare ulteriori soluzioni al problema, allora si torna indietro sui passi appena compiuti annullando gli eventuali legami creati per unificazione e poi si tenta col cammino successivo e così via per i rimanenti cammini. L'esecuzione fallisce sse tutti i cammini sono cammini di fallimento.
- Il Prolog mette a disposizione i seguenti predicati predefiniti di natura logica che hanno un impatto sulla costruzione dell'albero e-o:
  - `true` ha sempre successo e quindi fa avanzare la costruzione del ramo.
  - `fail` non ha mai successo e quindi blocca la costruzione del ramo.
  - `!`, detto taglio, ha successo una e una sola volta e ha l'effetto collaterale di bloccare alcuni rami. Il taglio ha lo scopo di ottenere prestazioni migliori e di evitare soluzioni ridondanti, ma va usato con attenzione perché la sua introduzione nelle clausole di un programma potrebbe alterare l'insieme delle soluzioni di un problema rappresentato da un certo obiettivo.
- Supponiamo che un nodo di un albero e-o contenga l'obiettivo `Q_1, ..., Q_h, !, Q_h+1, ..., Q_k`, che il taglio evidenziato nell'obiettivo sia stato introdotto per via dell'ultima risoluzione effettuata e che da questo nodo parta un cammino che porta al nodo contenente l'obiettivo `!, Q_h+1, ..., Q_k`. Giunti a quest'ultimo nodo, il taglio ha successo e vengono bloccati tutti gli eventuali cammini che iniziano dai fratelli destri dei nodi che si trovano lungo il cammino tra i due nodi che contengono i due obiettivi considerati. Poi si procede con l'obiettivo rimanente `Q_h+1, ..., Q_k`. Quando l'esecuzione torna indietro al nodo contenente l'obiettivo `!, Q_h+1, ..., Q_k` a causa di fallimento o della ricerca di ulteriori soluzioni al problema, l'esecuzione passa direttamente al nonno del nodo contenente l'obiettivo `Q_1, ..., Q_h, !, Q_h+1, ..., Q_k` in virtù del fatto che il taglio ha già avuto successo e che tutti gli eventuali cammini che iniziano dai fratelli destri dei nodi che si trovano lungo il cammino tra i due nodi che contengono i due obiettivi considerati sono stati bloccati.

- Esempi:
  - Il predicato che stabilisce se un elemento appartiene ad una lista può essere reso più efficiente ridefinendolo come segue:

    ```
    membro(X, [X | _]) :- !.
    membro(X, [_ | L]) :- membro(X, L).
    ```

    Tuttavia osserviamo che l'obiettivo:

    ```
    ?- membro(Z, [1, 3, 5]).
    ```

    dà luogo ad una sola esecuzione di successo (in cui `Z` viene istanziata ad `1`) a causa del taglio, quindi il predicato non può più essere usato per attraversare tutti gli elementi di una lista.
  - Predicato per implementare la negazione che ha successo sse l'argomento fallisce (sono importanti sia l'ordine tra `!` e `fail` nella prima regola che l'ordine tra le due regole):

    ```
    negazione(T) :- T, !, fail.
    negazione(T).
    ```

    Esso equivale a:

    ```
    negazione(T) :- T, !, fail; true.
    ```

    dove il punto e virgola rappresenta una disgiunzione, ha meno priorità della virgola ed è comodo per accorpare più regole aventi la stessa testa.
  - Predicato per implementare l'istruzione if-then-else (l'ordine tra le due regole non è importante):

    ```
    if_then_else(C, I1, _) :- C, !, I1.
    if_then_else(C, _, I2) :- negazione(C), !, I2.
    ```

- Dato un programma Prolog, le formule atomiche presenti nel corpo delle regole e negli obiettivi sono tutte positive perché questo è il formato delle clausole di Horn. A volte sarebbe tuttavia utile poter esprimere anche formule atomiche negative. A tale scopo il Prolog mette a disposizione il predicato `not` la cui interpretazione è conforme all'assunzione di mondo chiuso: è vero solo ciò che è stabilito dai fatti del programma o che può essere dedotto applicando le regole del programma, tutto il resto è falso.
- `not(T)` ha successo sse `T` fallisce, cioè il Prolog implementa la negazione come fallimento. La regola di negazione come fallimento è un'approssimazione dell'assunzione di mondo chiuso. La negazione così implementata non coincide con la negazione della logica matematica a causa della possibile presenza di cammini infiniti nell'albero e-o di `T`. La regola di negazione come fallimento è corretta e completa se `T` è ground. In alcune implementazioni del Prolog, `not(T)` è rimpiazzato da `\+(T)`.
- Esempi:
  - Dati il seguente programma Prolog:

    ```
    studente(franco).
    sposato(roberto).
    studente_celibe(X) :- studente(X), not(sposato(X)).
    ```

    e il seguente obiettivo:

    ```
    ?- studente_celibe(X).
    ```

    abbiamo successo con `X` istanziata a `franco`. Osserviamo che quando viene considerata la formula atomica `not(sposato(X))` durante l'esecuzione, la variabile `X` è già stata istanziata e quindi l'argomento del `not` è assimilabile ad un termine ground.
    Se invece la regola del programma fosse stata espressa nel seguente modo:

    ```
    studente_celibe(X) :- not(sposato(X)), studente(X).
    ```

    allora l'argomento del `not` non sarebbe stato assimilabile ad un termine ground. Poiché `X` sarebbe stata istanziata a `roberto`, avremmo erroneamente avuto un fallimento. Ciò mostra che la regola della negazione come fallimento non è sempre corretta e completa quando l'argomento del `not` non è ground.
  - L'occur check può essere riformulato tramite la singola clausola:

    ```
    occur_check(X, T) :- not(sottoterm(X, T)).
    ```

    a patto di invocarla con `T` ground.

- Il Prolog mette a disposizione dei predicati predefiniti di secondo ordine per raccogliere tutte le soluzioni di un problema rappresentato da un obiettivo (anziché calcolarle una alla volta):
  - `findall(T, O, L)` ha successo sse `L` è unificabile con la lista con eventuali ripetizioni di tutte le istanze del termine `T` per le quali l'obiettivo `O` ha successo.
  - `bagof(T, O, L)` ha successo sse `L` è unificabile con la lista non vuota con eventuali ripetizioni di tutte le istanze del termine `T` per le quali l'obiettivo `O` ha successo, dove le istanze sono organizzate in base alle variabili eventualmente presenti in `O`.
  - `setof(T, O, L)` ha successo sse `L` è unificabile con la lista non vuota senza ripetizioni di tutte le istanze del termine `T` per le quali l'obiettivo `O` ha successo, dove le istanze sono organizzate in base alle variabili eventualmente presenti in `O`.
- Esempi relativi agli alberi binari (rappresentati attraverso un funtore `ab` dove il primo argomento è un numero intero associato alla radice, il secondo argomento è il sottoalbero sinistro e il terzo argomento è il sottoalbero destro; l'atomo `nil` rappresenta l'albero vuoto):
  - Predicato per stabilire se una sequenza di caratteri è un albero binario:

```
albero_bin(nil).
albero_bin(ab(N, Sx, Dx)) :- integer(N), albero_bin(Sx), albero_bin(Dx).
```

  - Predicato per stabilire se un elemento appartiene ad un albero binario visitando quest'ultimo in ordine anticipato:

```
cerca_albero_bin_ant(N, ab(N, _, _)) :- !.
cerca_albero_bin_ant(N, ab(_, Sx, _)) :- cerca_albero_bin_ant(N, Sx).
cerca_albero_bin_ant(N, ab(_, _, Dx)) :- cerca_albero_bin_ant(N, Dx).
```

  - Predicato per stabilire se un elemento appartiene ad un albero binario visitando quest'ultimo in ordine posticipato:

```
cerca_albero_bin_post(N, ab(_, Sx, _)) :- cerca_albero_bin_post(N, Sx).
cerca_albero_bin_post(N, ab(_, _, Dx)) :- cerca_albero_bin_post(N, Dx).
cerca_albero_bin_post(N, ab(N, _, _)) :- !.
```

  - Predicato per stabilire se un elemento appartiene ad un albero binario visitando quest'ultimo in ordine simmetrico:

```
cerca_albero_bin_simm(N, ab(_, Sx, _)) :- cerca_albero_bin_simm(N, Sx).
cerca_albero_bin_simm(N, ab(N, _, _)) :- !.
cerca_albero_bin_simm(N, ab(_, _, Dx)) :- cerca_albero_bin_simm(N, Dx).
```

  - Predicato per stabilire se una sequenza di caratteri è un albero binario di ricerca:

```
albero_bin_ric(nil).
albero_bin_ric(ab(N, Sx, Dx)) :- albero_bin(ab(N, Sx, Dx)),
                                 minore_ug(Sx, N), maggiore_ug(Dx, N).
minore_ug(nil, _).
minore_ug(ab(M, Sx, Dx), N) :- M =< N, minore_ug(Sx, N), minore_ug(Dx, N).
maggiore_ug(nil, _).
maggiore_ug(ab(M, Sx, Dx), N) :- M >= N, maggiore_ug(Sx, N), maggiore_ug(Dx, N).
```

  - Predicato per stabilire se un elemento appartiene ad un albero binario di ricerca:

```
cerca_albero_bin_ric(N, ab(N, _, _)) :- !.
cerca_albero_bin_ric(N, ab(M, Sx, _)) :- N < M, cerca_albero_bin_ric(N, Sx).
cerca_albero_bin_ric(N, ab(M, _, Dx)) :- N > M, cerca_albero_bin_ric(N, Dx).
```

- Esempi relativi ai grafi diretti (rappresentati attraverso un funtore `gd` dove il primo argomento è una lista senza ripetizioni di atomi che denotano i vertici mentre il secondo argomento è una lista senza ripetizioni di termini basati sul funtore `arco`, i cui due argomenti sono rispettivamente il vertice da cui l'arco esce e il vertice in cui l'arco entra):

  - Predicato per stabilire se una sequenza di caratteri è un grafo diretto:

```
grafo_dir(gd(LV, LA)) :- controlla_v(LV), controlla_a(LA, LV).
controlla_v([]).
controlla_v([V | LV]) :- not(membro(V, LV)), controlla_v(LV).
controlla_a([], _).
controlla_a([arco(V1, V2) | LA], LV) :- membro(V1, LV), membro(V2, LV),
                                        not(membro(arco(V1, V2), LA)),
                                        controlla_a(LA, LV).
```

  - Predicato per stabilire se un elemento appartiene ad un grafo diretto visitando quest'ultimo in ampiezza a partire da un certo vertice iniziale:

```
cerca_grafo_dir_amp(V, gd(LV, LA), I) :- membro(I, LV), cerca_amp(V, [I], LA, []).
cerca_amp(V, [V | _], _, _) :- !.
cerca_amp(V, [V1 | LV], LA, LVVis) :- V \== V1, not(membro(V1, LVVis)),
                                      findall(V2, membro(arco(V1, V2), LA), LVAdiac),
                                      concatena(LV, LVAdiac, NuovaLV),
                                      cerca_amp(V, NuovaLV, LA, [V1 | LVVis]).
```

  - Predicato per stabilire se un elemento appartiene ad un grafo diretto visitando quest'ultimo in profondità a partire da un certo vertice iniziale:

```
cerca_grafo_dir_prof(V, gd(LV, LA), I) :- membro(I, LV), cerca_prof(V, [I], LA, []).
cerca_prof(V, [V | _], _, _) :- !.
cerca_prof(V, [V1 | LV], LA, LVVis) :- V \== V1, not(membro(V1, LVVis)),
                                       findall(V2, membro(arco(V1, V2), LA), LVAdiac),
                                       concatena(LVAdiac, LV, NuovaLV),
                                       cerca_prof(V, NuovaLV, LA, [V1 | LVVis]).
```

- Il Prolog mette a disposizione dei predicati predefiniti per accedere alle clausole, i quali sono resi possibili dal fatto che le clausole stesse sono viste come termini in cui simboli come `:-` e la virgola sono visti come predicati in notazione infissa (in Prolog non c'è distinzione tra istruzioni e dati):

  - `clause(H, B)` ha successo sse il programma contiene una clausola la cui testa è unificabile con `H` e il cui corpo è unificabile con `B` (il corpo è `true` nel caso di un fatto).
  - `call(G)` invoca `G` come obiettivo corrente e ha successo sse `G` ha successo.
  - `asserta(C)` aggiunge la clausola `C` all'inizio del programma.
  - `assertz(C)` aggiunge la clausola `C` alla fine del programma.
  - `retract(C)` ha successo sse il programma contiene una clausola unificabile con `C` e in tal caso elimina dal programma la prima clausola unificabile con `C`.
  - `retractall(H)` ha successo sse il programma contiene una clausola la cui testa è unificabile con `H` e in tal caso elimina dal programma tutte le clausole la cui testa è unificabile con `H`.

  Questi predicati sono utilizzabili al volo dentro gli obiettivi e le aggiunte o eliminazioni che essi determinano hanno luogo a tempo di esecuzione.

- Un metainterprete è un interprete scritto nello stesso linguaggio nel quale sono scritti i programmi da interpretare. Grazie ai predicati predefiniti per accedere alle clausole, è piuttosto semplice scrivere metainterpreti in Prolog:

  – Metainterprete per il Prolog puro:

  ```
  raggiungi(true).
  raggiungi((G1, G2)) :- raggiungi(G1), raggiungi(G2).
  raggiungi(G) :- clause(G, B), raggiungi(B).
  ```

  – Metainterprete per il Prolog puro esteso con taglio e negazione:

  ```
  raggiungi(true).
  raggiungi((G1, G2)) :- raggiungi(G1), raggiungi(G2).
  raggiungi(!) :- !.
  raggiungi(not(G)) :- not(raggiungi(G)).
  raggiungi(G) :- clause(G, B), raggiungi(B).
  ```

■ftpl_20

Capitolo 12

Attività di laboratorio in Linux

12.1 Cenni di storia di Linux

- Il sistema operativo Linux si ispira a Unix, il sistema operativo sviluppato a partire dai primi anni 1970 presso i Bell Lab della AT&T e successivamente diffusosi in ambito universitario. Nel 1983 Richard Stallman lanciò un progetto per creare GNU, un clone di Unix che fosse liberamente distribuibile e modificabile, così da sottrarre Unix ad aziende come Sun, SCO e IBM che lo stavano commercializzando. Nel 1991 il lavoro fu completato con l'implementazione del nucleo da parte di Linus Torvalds.

- Linux, nelle sue varie distribuzioni (come ad esempio Debian, Ubuntu, Mint e RedHat), può essere utilizzato su qualsiasi personal computer, sia di tipo desktop che di tipo laptop, tanto che diversi costruttori hanno ormai messo sul mercato computer con Linux preinstallato al posto di sistemi operativi proprietari. Grazie alla sua affidabilità, Linux è molto spesso impiegato su macchine server. Nel 2003 è stata sviluppata una versione modificata, nota come Android, che è adottata dal 2008 come sistema operativo da molti produttori di smart phone.

- Linux è l'esempio più importante di software libero ed open source. Altri esempi di questo genere sono il browser web Mozilla Firefox, i pacchetti software Open Office e Libre Office, il sistema di supporto alla didattica a distanza Moodle, il client di posta elettronica Alpine, gli editor di testo Gvim ed Emacs, il sistema di web conferencing Big Blue Button.

- Il software libero è soggetto ad una licenza d'uso ma, diversamente dal software proprietario, garantisce quattro libertà fondamentali definite negli anni 1980 da Richard Stallman, il fondatore della Free Software Foundation. Esse sono:

  - la libertà di eseguire il programma per qualsiasi scopo;
  - la libertà di studiare il programma e di modificarlo;
  - la libertà di distribuire copie del programma a chiunque ne abbia bisogno;
  - la libertà di migliorare il programma e di distribuirne pubblicamente i miglioramenti in modo tale che tutti ne possano beneficiare.

- È utile osservare come essere a sorgente aperto (cioè ispezionabile), che è un aspetto tecnico, sia un prerequisito per essere un software libero, che è un aspetto etico e sociale che favorisce la nascita di comunità mondiali di sviluppatori che mantengono un prodotto software. Va inoltre precisato che un software di questo genere non è necessariamente gratuito (freeware), anche se quasi sempre lo è.

- Comando per ottenere informazioni su un comando di Linux (manuale in linea):
  `man` ◁*comando*▷

12.2 Gestione dei file in Linux

- In Linux i file sono organizzati in directory secondo una struttura gerarchica ad albero, in cui la radice è denotata con "`/`", i nodi interni corrispondono alle directory e le foglie corrispondono ai file. Ogni file è conseguentemente individuato dal suo nome di percorso, il quale è ottenuto facendo precedere il nome del file dalla concatenazione dei nomi delle directory che si incontrano lungo l'unico percorso nell'albero che va dalla radice al file. Tutti questi nomi sono separati da "`/`" nel nome di percorso.
- Ogni directory ha un nome di percorso formato allo stesso modo. Tuttavia, sono disponibili le seguenti tre abbreviazioni per i nomi di percorso delle directory:
  - "`.`" denota la directory di lavoro, cioè la directory dove l'utente sta lavorando.
  - "`..`" denota la directory genitrice della directory di lavoro.
  - "`~`" denota la home directory dell'utente.
- Il nome di un file o di una directory è una sequenza di lettere, cifre decimali, sottotratti e trattini ed è consigliabile che non contenga spazi al suo interno. Le lettere minuscole sono considerate diverse dalle corrispondenti lettere maiuscole (case sensitivity). Di solito il nome di un file contiene anche un'estensione che individua il tipo del file, come ad esempio:
  - `.c` per un file sorgente del linguaggio C.
  - `.h` per un file di intestazione di una libreria del linguaggio C.
  - `.o` per un file oggetto di un linguaggio compilato come il C.
  - `.html` per un file sorgente del linguaggio interpretato HTML.
  - `.txt` per un file di testo senza formattazione.
  - `.doc` per un file contenente un documento in formato Word.
  - `.ps` per un file contenente un documento in formato PostScript.
  - `.pdf` per un file contenente un documento in formato PDF.
  - `.tar` per un file risultante dall'accorpamento di più file in uno solo.
  - `.gz` per un file risultante dalla compressione del contenuto di un file.
  - `.zip` per un file risultante dalla compressione del contenuto di uno o più file.
- Poiché i comandi di Linux corrispondono a file eseguibili, non si danno estensioni ai file eseguibili cosicché nemmeno i comandi hanno delle estensioni. Inoltre, se un file è nascosto (per esempio un file di configurazione), il suo nome inizia con "`.`" e non va confuso con un'estensione.
- Esempio di nome di percorso di un file sorgente C:
  `/home/users/bernardo/programmi/conversione_mi_km.c`
- Ogni file ha ad esso associato le seguenti informazioni:
  - Identificativo dell'utente proprietario del file (di solito è l'utente che ha creato il file).
  - Identificativo del gruppo di utenti proprietario del file (di solito è uno dei gruppi di cui l'utente che ha creato il file fa parte).
  - Dimensione del file espressa in Kbyte.
  - Data e ora in cui è avvenuta l'ultima modifica del file.
  - Diritti di accesso. Questi sono espressi attraverso tre triplette binarie che rappresentano i diritti di accesso dell'utente proprietario, del gruppo di utenti proprietario e del resto degli utenti, rispettivamente. In ciascuna tripletta il primo bit esprime il permesso ("`r`") o il divieto ("`-`") di lettura, il secondo bit il permesso ("`w`") o il divieto ("`-`") di scrittura e il terzo bit il permesso ("`x`") o il divieto ("`-`") di esecuzione.

- Le medesime informazioni sono associate ad ogni directory, con le seguenti differenze:
  - La dimensione della directory è il numero di Kbyte necessari per memorizzare le informazioni che descrivono il contenuto della directory (quindi non è la somma delle dimensioni dei suoi file).
  - Il diritto di esecuzione va inteso per la directory come diritto di accesso alla directory (quindi determina la possibilità per un utente di avere quella directory come directory di lavoro).
- Comandi relativi alle directory:
  - Visualizza il nome di percorso della directory in cui l'utente sta lavorando:
    `pwd`
    All'inizio della sessione di lavoro, la directory di lavoro coincide con la home directory dell'utente.
  - Visualizza il contenuto di una directory:
    `ls` ◁*directory*▷ (elenco dei nomi di file e sottodirectory)
    `ls -lF` ◁*directory*▷ (elenco completo di tutte le informazioni)
    `ls -alF` ◁*directory*▷ (elenco completo comprensivo dei file nascosti)
    dove *directory* può essere omessa se è la directory di lavoro. Se il contenuto della directory non sta tutto in una schermata, aggiungere il seguente filtro al precedente comando:
    `| more`
    al fine di visualizzare una schermata per volta.
  - Visualizza l'occupazione su disco di una directory:
    `du -h` ◁*directory*▷
  - Cambia la directory di lavoro:
    `cd` ◁*directory*▷
    dove *directory* può essere omessa se è la home directory.
  - Crea una directory:
    `mkdir` ◁*directory*▷
  - Copia una directory in un'altra directory:
    `cp -r` ◁$directory_1$▷ ◁$directory_2$▷
  - Copia il contenuto di una directory in un'altra directory:
    `cp -r` ◁$directory_1$▷`/*` ◁$directory_2$▷
    dove $directory_1$ può essere omessa se è la directory di lavoro.
  - Ridenomina una directory:
    `mv` ◁$directory_1$▷ ◁$directory_2$▷
  - Cancella una directory:
    `rmdir` ◁*directory*▷ (se vuota)
    `rm -r` ◁*directory*▷ (se non vuota)
  - Accorpa una directory in un singolo file:
    `tar -cvf` ◁*file*▷`.tar` ◁*directory*▷
  - Accorpa il contenuto di una directory in un singolo file:
    `tar -cvf` ◁*file*▷`.tar` ◁*directory*▷`/*`
    dove *directory* può essere omessa se è la directory di lavoro.
  - Estrai ciò che è stato precedentemente accorpato in un unico file:
    `tar -xvf` ◁*file*▷`.tar`
  - Accorpa e comprimi una directory in un singolo file:
    `zip -r` ◁*file*▷`.zip` ◁*directory*▷
  - Accorpa e comprimi il contenuto di una directory in un singolo file:
    `zip -r` ◁*file*▷`.zip` ◁*directory*▷`/*`
    dove *directory* può essere omessa se è la directory di lavoro.
  - Decomprimi ed estrai ciò che è stato precedentemente accorpato e compresso in un unico file:
    `unzip` ◁*file*▷`.zip`

- Comandi relativi ai file:
  - Visualizza il contenuto di un file di testo o sorgente:
    `cat ◁`*file*`▷`
    Se il contenuto del file non sta tutto in una schermata, aggiungere il seguente filtro al precedente comando:
    `| more`
    al fine di visualizzare una schermata per volta.
  - Visualizza un file contenente un documento in formato PostScript:
    `gv ◁`*file*`▷.ps &`
  - Visualizza un file contenente un documento in formato PDF:
    `acroread ◁`*file*`▷.pdf &`
  - Trasforma un file di testo o sorgente in formato PostScript:
    `enscript -B -o ◁`$file_2$`▷.ps ◁`$file_1$`▷`
    e poi in formato PDF:
    `ps2pdf ◁`$file_2$`▷.ps ◁`$file_2$`▷.pdf`
  - Stampa un file contenente un documento in formato PostScript o PDF:
    `lpr -P◁`*stampante*`▷ ◁`*file*`▷`
  - Copia un file in un altro file (eventualmente appartenente ad un'altra directory):
    `cp ◁`$file_1$`▷ ◁`$file_2$`▷`
  - Ridenomina un file (eventualmente spostandolo in un'altra directory):
    `mv ◁`$file_1$`▷ ◁`$file_2$`▷`
  - Cancella un file:
    `rm ◁`*file*`▷`
  - Comprimi il contenuto di un file:
    `gzip ◁`*file*`▷` oppure `zip ◁`*file*`▷.zip ◁`*file*`▷`
  - Decomprimi il contenuto di un file precedentemente compresso:
    `gunzip ◁`*file*`▷.gz` oppure `unzip ◁`*file*`▷.zip`

- Quando un file o una directory si trova su chiavetta USB, occorre procedere nel seguente modo:
  inserire la chiavetta USB nella porta USB
  `mount /pendrive`
  effettuare le operazioni desiderate attraverso la directory `/pendrive`
  `umount /pendrive`
  estrarre la chiavetta USB dalla porta USB

- I seguenti comandi servono per modificare la proprietà e i diritti di accesso relativi a file e directory:
  `chown ◁`*identificativo nuovo utente proprietario*`▷ ◁`*file o directory*`▷`
  `chgrp ◁`*identificativo nuovo gruppo proprietario*`▷ ◁`*file o directory*`▷`
  `chmod ◁`*nuovi diritti di accesso*`▷ ◁`*file o directory*`▷`
  dove i nuovi diritti di accesso vanno espressi attraverso tre cifre ottali corrispondenti alle tre triplette binarie.

- Il seguente comando è consigliabile per proteggere il contenuto della home directory di un utente in un sistema multiutente:
  `chmod 700 ~`
  dove la tripletta ottale `700` corrisponde alle tre triplette binarie `111 000 000` che stanno per `rwx -- --`, ovvero attribuisce tutti i diritti di accesso all'utente proprietario e nessun diritto di accesso agli altri utenti. ■fegpp_1

12.3 L'editor gvim

- Un file sorgente C, così come un file di testo, può essere creato in Linux attraverso il seguente comando:
  `gvim` ◁*file*▷
  Questo è un editor di testi particolarmente rapido, che evidenzia la sintassi in base all'estensione del file. Molti dei suoi comandi sono presenti anche nelle sue precedenti versioni non grafiche `vim` e `vi`, dove quest'ultimo è l'editor nativo di Unix e Linux ed è l'unico disponibile quando si deve lavorare su un dispositivo a livello sistemistico.
- Comandi di `gvim` per iniziare o terminare l'inserimento di caratteri in un file:
  - Entra in modalità inserimento testo nel punto in cui si trova il cursore oppure nel punto successivo:
    `i` oppure `a`
  - Entra in modalità inserimento testo all'inizio oppure alla fine della linea su cui si trova il cursore:
    `I` oppure `A`
  - Entra in modalità inserimento testo aprendo una nuova linea sotto oppure sopra la linea su cui si trova il cursore:
    `o` oppure `O`
  - Esci dalla modalità inserimento testo:
    `<esc>`
- Comandi di `gvim` per modificare rapidamente un file (fuori dalla modalità inserimento testo):
  - Sostituisci n caratteri consecutivi con n occorrenze di un nuovo carattere ($n = 1$ se omesso) a partire dal carattere su cui si trova il cursore:
    ◁*n*▷ `r` ◁*nuovo carattere*▷
  - Sostituisci n parole consecutive ($n = 1$ se omesso) con una sequenza di nuove parole a partire dalla parola su cui si trova il cursore:
    ◁*n*▷ `cw` ◁*nuove parole*▷ `<esc>`
  - Cancella n caratteri consecutivi ($n = 1$ se omesso) a partire dal carattere su cui si trova il cursore:
    ◁*n*▷ `x`
  - Cancella n parole consecutive ($n = 1$ se omesso) a partire dalla parola su cui si trova il cursore:
    ◁*n*▷ `dw`
  - Cancella n linee consecutive ($n = 1$ se omesso) a partire dalla linea su cui si trova il cursore:
    ◁*n*▷ `dd`
  - Copia n linee consecutive ($n = 1$ se omesso) a partire dalla linea su cui si trova il cursore:
    ◁*n*▷ `Y`
  - Incolla sotto oppure sopra la linea su cui si trova il cursore l'ultima sequenza di caratteri cancellati, l'ultima sequenza di parole cancellate o l'ultima sequenza di linee cancellate o copiate con `Y`:
    `p` oppure `P`
  - Ripeti l'ultimo comando tra quelli elencati sopra più `i`, `a`, `I`, `A`, `o`, `O`:
    `.`
  - Inserisci il contenuto di un altro file sotto la linea su cui si trova il cursore:
    `:r` ◁*file*▷ `<ret>`
  - Annulla l'ultimo comando tra quelli elencati sopra più `i`, `a`, `I`, `A`, `o`, `O`:
    `u`
- Comandi di `gvim` per salvare o scartare le ultime modifiche (fuori dalla modalità inserimento testo):
  - Salva le ultime modifiche:
    `:w <ret>`
  - Esci da `gvim`:
    `:q <ret>`
  - Salva le ultime modifiche ed esci da `gvim`:
    `:wq <ret>`
  - Esci da `gvim` senza salvare le ultime modifiche:
    `:q! <ret>`

- Comandi di `gvim` per muoversi all'interno di un file più celermente che con i tasti freccia (fuori dalla modalità inserimento testo):
  - Vai alla prima occorrenza di una stringa dopo l'attuale posizione del cursore:
    `/`◁*stringa*▷ `<ret>`
    e poi per cercare una alla volta tutte le successive occorrenze:
    `/ <ret>`
  - Vai alla prima linea del file:
    `1G`
  - Vai all'ultima linea del file:
    `G`
  - Vai alla k-esima linea del file:
    ◁*k*▷`G` oppure `:`◁*k*▷ `<ret>`
  - Spostati indietro di una parola rispetto all'attuale posizione del cursore:
    `b`
  - Spostati avanti di una parola rispetto all'attuale posizione del cursore:
    `e`
  - Spostati indietro di mezza schermata rispetto all'attuale posizione del cursore:
    `<ctrl>u`
  - Spostati avanti di mezza schermata rispetto all'attuale posizione del cursore:
    `<ctrl>d`
  - Spostati indietro di una schermata rispetto all'attuale posizione del cursore:
    `<ctrl>b`
  - Spostati avanti di una schermata rispetto all'attuale posizione del cursore:
    `<ctrl>f`
- Per impostare le opzioni di `gvim`, bisogna accedere ai file di configurazione `.vimrc` e `.gvimrc` nella propria home directory e scrivere rispettivamente in essi con `gvim` comandi del tipo:

```
set gfn=Courier\ 10\ Pitch\ 11          set guiheadroom=50
set lines=30                            colorscheme darkblue
set window=29
set columns=109
set textwidth=108
set ignorecase
set incsearch
set hlsearch
syntax on
```

  Comando di `gvim` per controllare il valore di un'opzione e il file in cui è impostata:
  `:verbose set <opzione>?`
- Alcuni suggerimenti relativi allo stile di programmazione quando si scrive un file sorgente C:
  - Usare commenti per documentare lo scopo del programma: breve descrizione, nomi degli autori e loro affiliazioni, numero e data di rilascio della versione corrente, modifiche apportate nelle versioni successive alla prima.
    Usare commenti per documentare lo scopo delle costanti simboliche, dei tipi definiti, delle funzioni, dei parametri formali, delle variabili locali e dei gruppi di istruzioni correlate.
    Riportare come commenti le considerazioni effettuate durante l'analisi del problema (input, output e loro relazioni) e i passi principali individuati durante la progettazione dell'algoritmo.
  - Lasciare almeno una riga vuota tra le inclusioni di librerie e le definizioni di costanti simboliche, tra queste ultime e le definizioni di tipi, tra queste ultime e le dichiarazioni di funzioni, tra queste ultime e le definizioni di funzioni, e tra due definizioni consecutive di funzioni.
    All'interno della definizione di una funzione, lasciare una riga vuota tra la sequenza di dichiarazioni di variabili locali e la sequenza di istruzioni.
    All'interno della sequenza di istruzioni di una funzione, lasciare una riga vuota tra due sottosequenze consecutive di istruzioni logicamente correlate.

  - Indentare il corpo di ciascuna funzione di almeno due caratteri rispetto a "`{`" e "`}`".
    Indentare opportunamente le istruzioni di controllo del flusso `if`, `switch`, `while`, `for` e `do-while`.
  - Disporre su più righe e allineare gli identificatori di variabili dichiarati dello stesso tipo.
    Disporre su più righe e allineare le dichiarazioni dei parametri formali delle funzioni.
    Disporre su più righe e allineare i parametri effettivi contenuti nelle invocazioni delle funzioni.
  - Un commento può estendersi su più righe e può comparire da solo o al termine di una direttiva, dichiarazione o istruzione. Se esteso su più righe, non deve compromettere l'indentazione.
    Una dichiarazione o istruzione può estendersi su più righe a patto di non andare a capo all'interno di un identificatore o una costante letterale. Se estesa su più righe, non deve compromettere l'indentazione.
  - Per gli identificatori introdotti dal programmatore, si rimanda alla Sez. 2.5. È inoltre consigliabile che essi siano tutti espressi nella stessa lingua.
  - Lasciare uno spazio vuoto prima e dopo ogni operatore binario.
    Lasciare uno spazio vuoto dopo `if`, `else`, `switch`, `case`, `while`, `for` e `do`.
    Non lasciare nessuno spazio vuoto tra l'identificatore di una funzione e "`(`".
    Non lasciare nessuno spazio vuoto dopo "`(`" e prima di "`)`".
    Non lasciare nessuno spazio vuoto prima di "`,`", "`;`", "`?`" e "`:`".

- Esercizio: Creare una directory chiamata `conversione_mi_km` e scrivere al suo interno con `gvim` un file chiamato `conversione_mi_km.c` per il programma di Sez. 2.2.

- Esercizio: Creare una directory chiamata `conversione_mi_km_file` e scrivere al suo interno con `gvim` un file chiamato `conversione_mi_km_file.c` per il programma di Sez. 2.8. ■fegpp_2

12.4 Il compilatore gcc

- Un programma C può essere reso eseguibile in Linux attraverso il compilatore `gcc`.

- Comando per compilare un programma C scritto su un singolo file sorgente:
  `gcc -ansi -Wall -O` ◁*file sorgente*▷`.c -o` ◁*file eseguibile*▷
  dove:

  - L'opzione `-ansi` impone al compilatore di controllare che il programma rispetti lo standard ANSI.
  - L'opzione `-Wall` impone al compilatore di riportare tutti i messaggi di warning (potenziali fonti di errore).
  - L'opzione `-O` impone al compilatore di ottimizzare il file eseguibile.
  - L'opzione `-o` permette di dare al file eseguibile un nome diverso da quello di default `a.out`. È opportuno dare al file eseguibile lo stesso nome, a meno dell'estensione `.c`, del file sorgente.

- Sequenza di comandi per compilare un programma C scritto su $n \geq 2$ file sorgenti:
  `gcc -ansi -Wall -O -c` ◁$file_1$▷`.c`
  `gcc -ansi -Wall -O -c` ◁$file_2$▷`.c`
  ⋮
  `gcc -ansi -Wall -O -c` ◁$file_n$▷`.c`
  `gcc -ansi -Wall -O` ◁$file_1$▷`.o` ◁$file_2$▷`.o ...` ◁$file_n$▷`.o -o` ◁*file eseguibile*▷
  dove:

  - L'opzione `-c` impone al compilatore di produrre un file oggetto (anziché un file eseguibile) avente lo stesso nome del file sorgente ed estensione `.o`.
  - L'ultimo comando crea un file eseguibile collegando i file oggetto precedentemente ottenuti.

- Se il programma include il file di intestazione della libreria standard `math.h`, potrebbe rendersi necessario aggiungere l'opzione `-lm` in fondo al comando `gcc`.
- Il file eseguibile viene prodotto solo se il compilatore non riscontra:
  - Errori lessicali: violazioni del lessico del linguaggio.
  - Errori sintattici: violazioni delle regole grammaticali del linguaggio.
  - Errori semantici: violazioni del sistema di tipi del linguaggio.
- Quando trova un errore, il compilatore emette un messaggio per fornire informazioni su dove si trova l'errore nel file sorgente e sul genere di errore incontrato. In base ai messaggi d'errore occorre modificare il programma e poi ricompilarlo. È buona norma modificare e ricompilare il programma anche in caso di segnalazioni di warning.
- Comando per lanciare in esecuzione il file eseguibile di un programma C:
  `./`◁*file eseguibile*▷
- Se si compiono preventivamente operazioni come le seguenti:
  `gvim ~/.cshrc`
  cambiare la linea contenente la definizione di `path` come segue:
  `set path = ($path .)`
  uscire da `gvim`
  `source ~/.cshrc`
  allora il comando per lanciare in esecuzione il file eseguibile di un programma C si semplifica come segue:
  ◁*file eseguibile*▷

12.5 L'utility di manutenzione `make`

- È buona norma raccogliere tutti i file sorgenti che compongono un programma – ad eccezione dei file di libreria standard – in una directory, eventualmente organizzata in sottodirectory al suo interno. Per la manutenzione del contenuto di tale directory si può ricorrere in Linux all'utility `make`, la quale esegue i comandi specificati in un file di testo chiamato `Makefile` da creare all'interno della directory stessa.
- Il `Makefile` è una sequenza di direttive, ciascuna della seguente forma:
  `#`◁*eventuale commento*▷
  ◁*obiettivo*▷`:` ◁*eventuale lista delle dipendenze*▷
  ◁*azione*▷
  dove:
  - L'obiettivo è il nome della direttiva e nella maggior parte dei casi coincide con il nome di un file che si vuole produrre. La sintassi del comando `make` è di conseguenza la seguente:
    `make` ◁*obiettivo*▷
    dove *obiettivo* può essere omesso se è il primo della lista all'interno del `Makefile`.
  - Le dipendenze rappresentano i file da cui dipende il conseguimento dell'obiettivo. Se anche uno solo di questi file non esiste e non è specificata all'interno del `Makefile` una direttiva che stabilisce come ottenerlo, allora l'obiettivo non può essere raggiunto e l'esecuzione di `make` termina segnalando un errore.
  - L'azione, che deve essere preceduta da un carattere di tabulazione, rappresenta una sequenza di comandi che l'utility `make` deve eseguire per raggiungere l'obiettivo a partire dai file da cui l'obiettivo dipende. Se l'obiettivo è un file esistente, l'azione viene eseguita solo se almeno uno dei file da cui l'obiettivo dipende ha una data di ultima modifica successiva alla data di ultima modifica del file obiettivo, così da evitare lavoro inutile.

- Nel caso della compilazione, l'obiettivo è il file eseguibile, il suo ottenimento dipende dalla disponibilità dei file sorgente e l'azione per costruirlo è il comando `gcc`.
- Esempi:
  - `Makefile` per la compilazione di un programma C scritto su un singolo file sorgente:

```
◁file eseguibile▷: ◁file sorgente▷.c Makefile
        gcc -ansi -Wall -O ◁file sorgente▷.c -o ◁file eseguibile▷
pulisci:
        rm -f ◁file sorgente▷.o
pulisci_tutto:
        rm -f ◁file eseguibile▷ ◁file sorgente▷.o
```

  - `Makefile` per la compilazione di un programma C scritto su $n \geq 2$ file sorgenti:

```
◁file eseguibile▷: ◁file_1▷.o ◁file_2▷.o ... ◁file_n▷.o Makefile
        gcc -ansi -Wall -O ◁file_1▷.o ◁file_2▷.o ... ◁file_n▷.o -o ◁file eseguibile▷
◁file_1▷.o: ◁file_1▷.c ◁file di intestazione di librerie non standard inclusi▷ Makefile
        gcc -ansi -Wall -O -c ◁file_1▷.c
◁file_2▷.o: ◁file_2▷.c ◁file di intestazione di librerie non standard inclusi▷ Makefile
        gcc -ansi -Wall -O -c ◁file_2▷.c
  ⋮
◁file_n▷.o: ◁file_n▷.c ◁file di intestazione di librerie non standard inclusi▷ Makefile
        gcc -ansi -Wall -O -c ◁file_n▷.c
pulisci:
        rm -f *.o
pulisci_tutto:
        rm -f ◁file eseguibile▷ *.o
```

    dove `*.o` significa tutti i file il cui nome termina con `.o`.
- Esercizio: Nella directory `conversione_mi_km` scrivere con `gvim` un `Makefile` per la compilazione con `gcc` di `conversione_mi_km.c`, eseguire il `Makefile` e lanciare in esecuzione il file eseguibile ottenuto al fine di testarne il corretto funzionamento.
- Esercizio: Nella directory `conversione_mi_km_file` scrivere con `gvim` un `Makefile` per la compilazione con `gcc` di `conversione_mi_km_file.c`, eseguire il `Makefile` e lanciare in esecuzione il file eseguibile ottenuto al fine di testarne il corretto funzionamento. Prima di ogni esecuzione, nella stessa directory occorre preventivamente scrivere con `gvim` un file chiamato `miglia.txt` da cui il programma acquisirà il valore da convertire. Al termine di ogni esecuzione, il risultato della conversione si troverà in un file chiamato `chilometri.txt` creato dal programma nella stessa directory. ■fegpp_3

12.6 Il debugger gdb

- È praticamente impossibile che un programma compilato con successo emetta i risultati attesi le prime volte che viene eseguito. In particolare, possono verificarsi i seguenti errori:
  - Errori a tempo di esecuzione: interruzione dell'esecuzione del programma in quanto il computer è stato indotto dal programma ad eseguire operazioni illegali che il sistema operativo ha rilevato (p.e. l'uso di una variabile di tipo puntatore che vale `NULL` per accedere ad un gruppo di dati determina la violazione di un'area di memoria riservata).
  - Errori di logica: ottenimento di risultati diversi da quelli attesi in quanto l'algoritmo su cui si basa il programma non è corretto rispetto al problema da risolvere.
  - Altri errori: ottenimento di risultati diversi da quelli attesi a causa della mancata comprensione di alcuni aspetti tecnici del linguaggio di programmazione utilizzato.

- Per tenere sotto controllo la presenza di questi errori occorre pianificare un'adeguata verifica a priori (tecniche formali) e un adeguato testing a posteriori (tecniche empiriche).
- Quando uno di questi errori si verifica, per comprenderne e rimuoverne le cause si deve ricorrere ad un debugger. Questo strumento permette di eseguire il programma passo passo, di ispezionare il valore delle variabili come pure il contenuto dello stack dei record di attivazione e di impostare dei punti di arresto (breakpoint) in corrispondenza di determinate parti del programma.
- Comando per lanciare in esecuzione il debugger `gdb` su un programma C:
  `gdb` ◁*file eseguibile*▷
- L'utilizzo di `gdb` richiede che il programma sia stato compilato specificando anche l'opzione `-g` ogni volta che `gcc` è stato usato. Tale opzione (potenzialmente incompatibile con `-O`) arricchisce il file eseguibile con le informazioni di cui `gdb` necessita per svolgere le funzioni precedentemente indicate.
- Questi sono alcuni dei comandi di più frequente utilizzo disponibili in `gdb`:
  - Avvia l'esecuzione del programma con l'assistenza di `gdb`:
    `run`
    L'esecuzione si arresta al primo breakpoint, al primo errore a tempo di esecuzione, oppure a seguito di normale terminazione.
  - Visualizza il contenuto dello stack dei record di attivazione in termini di chiamate di funzione in corso di esecuzione:
    `bt`
    Ciò è utile all'atto del verificarsi di un errore a tempo di esecuzione.
  - Visualizza il valore di un'espressione C:
    `print` ◁*espressione C*▷
    dove l'espressione non deve contenere identificatori non visibili nel punto in cui il comando viene dato. Ciò è particolarmente utile per conoscere il contenuto delle variabili durante l'esecuzione.
  - Esegui la prossima istruzione senza entrare nell'esecuzione passo passo delle funzioni da essa invocate:
    `next`
    oppure:
    `n`
  - Esegui la prossima istruzione entrando nell'esecuzione passo passo delle funzioni da essa invocate:
    `step`
    oppure:
    `s`
  - Continua l'esecuzione fino ad incontrare il prossimo breakpoint:
    `continue`
    oppure:
    `c`
  - Imposta un breakpoint all'inizio di una funzione:
    `break` ◁*funzione*▷
    oppure presso una linea di un file sorgente:
    `break` ◁*file sorgente:*▷ ◁*numero linea*▷
    dove l'indicazione del file sorgente può essere omessa se c'è un unico file sorgente. Ad ogni breakpoint impostato viene automaticamente associato un numero seriale univoco.
  - Imponi una condizione espressa in C al verificarsi della quale l'esecuzione deve arrestarsi presso un breakpoint precedentemente impostato:
    `condition` ◁*numero seriale breakpoint*▷ ◁*espressione C*▷
    dove l'espressione non deve contenere identificatori non visibili nel punto di arresto. Se l'espressione viene omessa si impone un arresto incondizionato presso il breakpoint, cancellando così l'eventuale condizione definita in precedenza per lo stesso breakpoint.

  - Elimina un breakpoint:
    `clear` ◁*funzione*▷
    oppure:
    `clear` ◁*file sorgente:*▷ ◁*numero linea*▷
    oppure:
    `delete` ◁*numero seriale breakpoint*▷
  - Ottieni informazioni sui comandi di `gdb`:
    `help`
    oppure:
    `apropos`
  - Esci da `gdb`:
    `quit`
    oppure:
    `q`

- Esercizio: In una nuova directory scrivere con `gvim` il seguente programma C:

```
#include <stdio.h>

unsigned long numero_successivo = 1;

void          stampa_numero(void);
unsigned long genera_numero(void);

int main(void)
{
  int i;

  for (i = 0;
       (i < 5);
       ++i)
    stampa_numero();
  return(0);
}

void stampa_numero(void)
{
  int modulo,
      numero;

  modulo = genera_numero() - 17515;
  numero = 12345 % modulo;
  printf("%d\n",
         numero);
}

unsigned long genera_numero(void)
{
  numero_successivo = numero_successivo * 1103515245 + 12345;
  return(numero_successivo / 65536 % 32768);
}
```

  e, dopo averlo compilato con `gcc` e lanciato in esecuzione, scoprire attraverso l'ausilio di `gdb` il motivo per cui l'esecuzione del programma si interrompe prematuramente.

- L'ambiente `gdb` supporta anche il reverse debugging, cioè la possibilità di eseguire il programma passo passo a ritroso. Per abilitare il reverse debugging, occorre dare i seguenti comandi dentro `gdb`:
  ```
  break _start
  run
  record
  continue
  ```
  e talvolta può essere necessario usare anche l'opzione `-static` durante la compilazione al fine di escludere le librerie dinamiche. Ecco alcuni comandi di reverse debugging disponibili in `gdb`:
  ```
  reverse-next
  reverse-step
  reverse-continue
  ```

- Esercizio: Nella directory `conversione_mi_km` lanciare in esecuzione il programma e verificarne il comportamento nel caso in cui come distanza in miglia si introduca un valore non numerico. Modificare poi `conversione_mi_km.c` con `gvim` per aggiungere la validazione stretta dell'input (vedi Sez. 4.4), rieffettuare la compilazione tramite il `Makefile` e rieseguire il programma per verificarne di nuovo il comportamento nello stesso caso di prima.

- Esercizio: Nella directory `conversione_mi_km_file` lanciare in esecuzione il programma e verificarne il comportamento nel caso in cui il file `miglia.txt` non sia presente nella directory e nel caso in cui tale file contenga come distanza in miglia un valore non numerico. Modificare poi `conversione_mi_km_file.c` con `gvim` per aggiungere il controllo di errata apertura di file in lettura e la validazione stretta dell'input da file (se il valore contenuto nel file di input è errato, scrivere un messaggio di errore nel file di output), rieffettuare la compilazione tramite il `Makefile` e rieseguire il programma per verificarne di nuovo il comportamento negli stessi due casi di prima. ■fegpp_4

12.7 Implementazione dei programmi C introdotti a lezione

- Esercizi: Per ciascuno dei seguenti esempi di programma/libreria introdotti durante le lezioni teoriche, creare una nuova directory in cui scrivere usando `gvim` i file sorgenti (completi di validazione stretta di tutti i valori acquisiti in ingresso – vedi Sez. 4.4) e il `Makefile` per la loro compilazione con `gcc`, eseguire il `Makefile` e lanciare in esecuzione il file eseguibile ottenuto al fine di testarne il corretto funzionamento (avvalendosi di `gdb` ogni volta che si verificano errori a tempo di esecuzione):

  1. Programma per la determinazione del valore di un insieme di monete (Sez. 3.11).
  2. Programma per il calcolo della bolletta dell'acqua (Sez. 4.3).
  3. Programma per il calcolo dei livelli di radiazione (Sez. 4.4).
  4. Programma per la determinazione del valore di un insieme di monete modificato facendo uso di array, stringhe e istruzioni di ripetizione al fine di evitare la ridondanza di codice presente nella versione originale del programma.
  5. Libreria per l'aritmetica con le frazioni (Sezz. 5.10 e 5.7).
  6. Libreria per le operazioni matematiche espresse in versione ricorsiva (Sez. 5.8).
  7. Programma per la statistica delle vendite (Sez. 6.8).
  8. Libreria per la gestione delle figure geometriche (Sez. 6.10).
  9. Libreria per la gestione delle liste ordinate (Sez. 6.11).

  Si rammenta che per ogni programma che fa uso di file è anche necessario scrivere con `gvim` i relativi file di input prima che il programma venga eseguito. Si ricorda inoltre che per ogni libreria è anche necessario scrivere con `gvim` un file sorgente che include il file di intestazione della libreria, definisce la funzione `main` e fa uso di tutte le funzioni esportate dalla libreria. ■fegpp_5_6_7_8

12.8 Il compilatore/interprete gprolog

- Un programma Prolog può essere creato con un editor come `gvim`, viene memorizzato in un file sorgente con estensione `.pl` o `.pro` e viene eseguito attraverso un interprete Prolog.
- Comando per lanciare in esecuzione l'interprete Prolog:
  `gprolog`
  Usare `<ctrl>d` per uscire.
- L'interprete Prolog consente all'utente di fare interrogazioni e di consultare, eseguire e correggere programmi Prolog mediante un ciclo leggi-esegui-scrivi in cui l'interprete:
  - Visualizza "`?-`" e aspetta di acquisire da tastiera un obiettivo, il quale deve finire con il punto (l'acquisizione termina premendo il tasto invio).
  - Tenta di raggiungere l'obiettivo sulla base dei:
    * Predicati predefiniti.
    * Predicati che sono stati preventivamente caricati nell'interprete utilizzando il seguente obiettivo basato sul predicato predefinito `consult` per consultare file sorgenti Prolog:
      `consult('◁`*file sorgente*`▷').`
  - Riporta l'esito dell'esecuzione:
    * Se l'obiettivo non può essere raggiunto, l'interprete risponde `no`.
    * Se l'obiettivo può essere raggiunto ed è ground, l'interprete risponde `yes` o `true`.
    * Se l'obiettivo può essere raggiunto e non è ground, l'interprete visualizza i legami creati tramite unificazione per le variabili non istanziate presenti nell'obiettivo.
  - Negli ultimi due casi, dopo aver trovato quella soluzione l'interprete visualizza "`?`" e aspetta di acquisire da tastiera una decisione:
    * Se l'utente preme il tasto invio, l'esecuzione termina.
    * Se l'utente preme "`;`", l'esecuzione continua in cerca di un'altra soluzione.
    * Se l'utente preme "`a`", l'esecuzione continua in cerca di tutte le altre soluzioni.
- Comando per compilare un programma Prolog:
  `gplc ◁`*file sorgente*`▷ -o ◁`*file eseguibile*`▷`
- Comando per lanciare in esecuzione il file eseguibile di un programma Prolog:
  `./◁`*file eseguibile*`▷`
- Lanciare in esecuzione il file eseguibile di un programma Prolog equivale a lanciare in esecuzione l'interprete Prolog e caricare i predicati definiti in quel programma.

12.9 Implementazione dei programmi Prolog introdotti a lezione

- Esercizi: Per ognuno dei seguenti esempi di predicato/programma introdotti durante le lezioni teoriche, creare una nuova directory in cui scrivere i file sorgenti usando `gvim` e caricarli in `gprolog` al fine di testarne il corretto funzionamento:
  1. Programma contenente i predicati relativi al calcolo del fattoriale e di Fibonacci (Sez. 11.7).
  2. Programma contenente i predicati relativi alle liste (Sez. 11.7).
  3. Programma contenente i predicati relativi agli algoritmi di ordinamento (Sez. 11.7).
  4. Programma contenente i predicati relativi all'algoritmo di unificazione di Robinson (Sez. 11.7).
  5. Programma contenente i predicati relativi agli alberi binari (Sez. 11.8).
  6. Programma contenente i predicati relativi ai grafi diretti (Sez. 11.8).
  7. Programma contenente i predicati relativi ai metainterpreti (Sez. 11.8). ■fegpl_1_2_3_4