

# Linguaggi di programmazione

I linguaggi di programmazione si possono dividere in varie categorie:

- **Linguaggi imperativi:** assemblers, Fortran, Pascal, C, Ada95, Bliss, PL/1, C++, Python, etc..
- **Linguaggi logici:** Prolog
- **Linguaggi funzionali:** FP, Lisp, Common Lisp, Scheme, ML, Ocaml, Haskell

Ognuna delle categorie citate può contenere dei linguaggi ad oggetti; il paradigma ad oggetti è quindi ortogonale alla classificazione data, anche se presume alcune caratteristiche tipiche dei linguaggi imperativi (memoria e stato di un'istanza).

## Paradigma imperativo

Le caratteristiche essenziali dei linguaggi imperativi sono strettamente legate alla **macchina di Von Neumann**, che è formata da due componenti fondamentali: la **memoria** (passiva) e il **processore** (attiva). L'attività del processore è eseguire calcoli, e assegnare valori a celle di memoria, il che implica che il concetto di variabili è un'astrazione di una cella di memoria fisica.

I linguaggi imperativi adottano uno **stile prescrittivo**: un programma scritto in un linguaggio imperativo prescrive le operazioni che il processore deve eseguire per modificare lo stato del sistema, e poi esegue le istruzioni nell'ordine in cui appaiono nel programma (con un'eccezione: le strutture di controllo). Sono nati di più per manipolazione numerica che simbolica.

La struttura del programma consiste in una parte di **dichiarazione** in cui si dichiarano tutte le variabili del programma e il loro tipo, e una parte che descrive l'algoritmo risolutivo utilizzato, mediante **istruzioni del linguaggio**. Le istruzioni si dividono in *lettura e scrittura, assegnamento e controllo*.

## C/C++

Il C è uno dei linguaggi fondamentali per la programmazione: è il linguaggio base per l'implementazione di Unix, e quindi sia di Linux, Windows o macOS. È un linguaggio di livello **relativamente basso**, mentre la sua "counterpart" C++ è leggermente di più alto livello, ed effettua alcune modifiche per rendere C più semplice.

Di fatto, C++ è un C esteso, quindi i tipi di dati fondamentali si comportano essenzialmente allo stesso modo.

Un esempio di programma può essere questo:

<pre>/* Versione C */ #include &lt;stdio.h&gt;  int main() {     printf("Hello world!\n");     return 0; }</pre>	<pre>// Versione C++ #include &lt;iostream&gt;  int main() {     std::cout &lt;&lt; "Hello world!";     &lt;&lt; std::endl;     return 0; }</pre>
--	---

Guardiamo nel dettaglio ciascuna versione:

- **Versione C**
  - **/\* Versione C \*/**: commento – permette anche l'uso di commenti multilina.
  - **#include <stdio.h>**: questa riga include l'header file *stdio.h*, che è necessario per utilizzare la funzione di input/output standard

- **Int main()** : inizia la definizione della funzione principale. Ogni programma deve avere una funzione main, che è il punto di ingresso del programma. Deve essere **int** perché deve ritornare un codice di sistema.
- **printf("Hello world!\n");**: utilizza la funzione printf per stampare il messaggio "Hello world!" sulla console. Il \n è un carattere di nuova riga che inserisce una riga vuota dopo la stampa del messaggio
- **Return 0;** : indica la fine della funzione main restituendo il valore 0, che di solito indica che il programma è stato eseguito con successo.

- **Versione C++**

- **//Versione C++:** commento – permette anche l'uso di commenti multi linea
- **#include <iostream>:** dichiarazione dell' header file è inclusa, che è necessaria per l' output in C++
- **Std::cout:** inizia la stampa della stringa nello standard output
- **Std::endl** è utilizzato per inserire un carattere di nuova riga e svuotare il buffer dell'output
- **Return 0;** : indica che il programma è stato eseguito correttamente

## Compilazione ed esecuzione

Per compilare ed eseguire un programma C/C++ abbiamo bisogno di un **compilatore esterno**, come gcc/clang. In sistemi moderni, possiamo semplicemente cliccare Run nell'editor per poter eseguirlo. Supponendo di aver salvato il programma in un file chiamato **hello.c**, possiamo compilare il programma invocando il compilatore:

**prompt\$ cc hello.c**

Se non ci sono errori, l'eseguibile sarà convenzionalmente chiamato su Unix **a.out**, mentre su Windows gli verrà dato un nome di un file exe. Questo, a sua volta, potrà essere richiamato direttamente dal terminale:

**prompt\$ a.out**  
Hello World!

Possiamo richiedere anche un nome specifico per l'eseguibile usando l'opzione **-o**.

## I nomi

In C, i vari elementi del linguaggio (variabili, tipi, classi, funzioni, metodi) sono denotati da **nomi**, o identificatori; sono stringhe di lettere, numeri ed il carattere **\_**. Devono per forza iniziare con una lettera o l'underscore.

## Modificatori di dichiarazione

Per dichiarare variabili, possiamo anche usare dei **modificatori**. Questi modificatori sono detti *extern* e *static*, il quale svolgono un ruolo cruciale nella gestione della visibilità e dello spazio di memoria. Le dichiarazioni **esterne** sono spesso utilizzate nei file di interfaccia, indicando che la definizione dell'oggetto dichiarato si troverà in un altro punto del file, o in un file esterno.

Il modificatore **static**, invece, viene spesso utilizzato per fissare dichiarazioni o definizioni nello spazio di memoria globale del file in cui appaiono, **rendendole non visibili all'esterno di quel file**.

Osserviamo un esempio pratico:

```
#ifndef _QUPCP_H
#define _QUPCP_H

extern const int qupcp_N;
extern bool qupcp_find(int, int);
extern void qupcp_unite(int, int);
#endif
```

In un file header, definiamo gli **extern** come le interfacce dei metodi che vogliamo implementare. Quindi, ogni volta che implementiamo un header di questo tipo, dovremmo andare a definire metodi con questi tipi di ritorno e questi parametri.

Invece, con il modificatore **static**:

```
static int set_id_of[qupcp_N];  
static int find_root(int x);
```

Il modificatore **static**, in queste dichiarazioni, significa che *set\_id\_of* e *find\_root* sono fissati nello spazio di memoria globale di questo file specifico, e il loro valore viene mantenuto tra le chiamate di funzione.

### Costanti

Le versioni più recenti di C e C++ offrono la possibilità di modificare le dichiarazioni di vari oggetti, utilizzando l'attributo **const**; consente di dichiarare costanti in modo più flessibile. Usare **const** permette i seguenti vantaggi:

- **Tipizzazione**: mantengono il loro tipo di dati, fornendo un livello di tipizzazione più robusto rispetto alle macro definite con `#define`.
- **Ambito di visibilità limitato**: sono valide solo nel blocco o nel file in cui sono dichiarate
- **Debugging più agevole**: poiché sono oggetti veri e propri, è possibile usare il debugger per esaminare e monitorare il valore delle costanti durante l'esecuzione del programma
- **Controllo dell'accesso in lettura/scrittura**: è possibile usarle per indicare se un dato può essere modificato o meno. Per esempio, dichiarando un puntatore *const*, si impedisce la modifica dei dati a cui punta.
- **Sintassi più chiara**
- **Scope e spazio dei nomi**: rispettano lo scope e lo spazio dei nomi del linguaggio, migliorando la gestione e l'organizzazione del codice.

Le costanti definite con **const** devono essere inizializzate, a meno che non siano dichiarate *extern*. Una volta dichiarate, ovviamente, tentativi di modificarle genereranno errori.

### Costanti e puntatori

Le complicazioni aumentano, poiché ci sono due oggetti da considerare: il puntatore e l'oggetto puntato. La dichiarazione di Const deve essere quindi in grado di distinguere tra i due.

Vediamo un esempio:

```
char* p;  
char s[] = "Ford Prefect";  
  
const char* pc = s;  
pc[3] = 'X';  
pc = p;  
  
char *const cp = s;  
cp[3] = 'Y';  
cp = p;  
  
const char *const cpc = s;  
cpc[3] = 'Z';  
cpc = p;
```

**pc** è un puntatore a carattere costante. Il contenuto puntato da **pc** non può essere modificato attraverso **pc**, ma è consentito cambiare il puntatore stesso.

**Errore**: tentativo di modificare un carattere costante

**cp** è un puntatore costante a carattere. Il contenuto puntato da **cp** può essere modificato, ma il puntatore stesso non può essere riassegnato a un altro indirizzo

**cpc** è un puntatore costante a carattere costante, e non si può modificare né il contenuto puntato né il puntatore stesso.

### Tipi fondamentali

Di seguito, abbiamo i tipi fondamentali che sono presenti sia in C che in C++:

- **Int**: interi da -231 a 231-1, assumendo un hardware con interi di 32 bit.
- **Char**: caratteri, rappresentati come numeri interi da -128 a 127 (ovvero byte da 8 bit).
- **Float**: numeri floating point
- **Bool**: Boolean
- **Puntatori e riferimenti**

Int e char possono essere anche dichiarati *unsigned*, per spostare l'intervallo tra 0 e 232 per int, e 0 e 28 per char.

Il C ha due tipi aggregati:

- **Arrays**, assimilabili a puntatori. Si scrive *nome[dimensione]*; se si lascia la dimensione vuota, vuol dire che l' array non ha una dimensione specifica.
- **Strutture**: aggregazioni di campi di tipo diverso.

Altri tipi importanti sono:

- **Void**, tipo particolare che denota la mancanza di informazione
- **Enum [nome] { <c1>, <c2>, ..., <cn> }**, tipo di enumerazione di costanti.
- **Classe**, come base della programmazione object-oriented.

### Variabili

Un programma C deve manipolare valori che vanno associati a nomi, ovvero a variabili e/o funzione; tutti questi nomi vengono introdotti in àà un programma per mezzo di dichiarazioni, che prendono di solito la forma:

```
<tipo> <modificatori> <nome> <modificatori> [= <inizializzazione>];
```

L'inizializzazione di una variabile è pressochè identica a C#. Inserendo un \* dopo il tipo di una variabile, indichiamo che esso è un **puntatore**.

Solo all'interno di C++, possiamo fare una **referenza** usando &.

```
char* ps    float** m; int(*pf)(int*, float*) ;           int& rx = x;
```

1. Puntatore ad una locazione di memoria che contiene un carattere
2. Puntatore ad una locazione di memoria che contiene un puntatore ad una locazione di memoria contenente un float.
3. Puntatore ad una funzione che accetta un puntatore ad un intero ed un puntatore ad un float e che ritorna un intero
4. Referenza all'intero x

### Array

Gli array sono una componente importante del C/C++, ma il loro comportamento è molto sofisticato e, allo stesso tempo, di basso di livello. Sono da considerarsi come un **blocco di memoria fisica**. Quindi, una dichiarazione come **unsigned char s[80];**, di fatto riserva un blocco di 80 ottetti nella RAM del calcolatore.

Gli array non hanno attributi associati, a parte il **tipo** – non si può estrarre la dimensione dell' array dal contenuto di una variabile. L'inizializzazione dell' array può avvenire specificando la dimensione di esso, o anche no:

```
int un_due_tre[3] = {1, 2, 3};  float an_array[] = {3.0, 42.0};
```

Ovviamente, è importante non impostare una dimensione di un array, e poi metterci dentro più elementi di quanto sia la dimensione.

### Stringhe

Le stringhe sono un particolare tipo di **array** che terminano con un carattere nullo \0. Di fatto:

```
char stringa[] = "Dylan Dog";  
Equivale a char stringa{} = {'D', 'y', 'l', 'a', 'n', ' ', 'D', 'o', 'g', '\0'};
```

La terminazione di essa con un \0 è una convenzione, dove in C equivale all'intero 0, mentre in C++ equivale al valore **false**.

## Strutture

Le strutture sono **aggregazioni di tipi diversi**, ovvero possono avere all'interno diversi tipi di variabili, che poi saranno inizializzati.

La keyword con cui parte è sempre **struct**, seguita dal nome, e tra parentesi graffe i tipi di variabili che vogliamo che la struct abbia, come se fosse un metodo alla fine. Per creare una nuova variabile appartenente alla struct, **possiamo crearla direttamente anche dopo la struct**.

```
struct polar_complex { float magnitude; float angle; } c = { 42.0, 3.14 };
```

Come negli oggetti, i campi di una struttura si estraggono con la **notazione punteggiata**, mentre in C++ è possibile usare dei costruttori per inizializzare le strutture.

## Puntatori

Il concetto di **puntatore** è fondamentale per qualunque tipo di programmazione; infatti, ogni sistema ha, a qualche livello, un concetto equivalente. Dato un qualunque tipo T, il tipo associato T\* è il tipo **puntatore a T**. Ovvero, *una variabile di tipo T\* contiene l'indirizzo in memoria di un oggetto di tipo T*.

Un'altra operazione fondamentale è la **de-referenziazione**, ovvero l'operazione di accedere al valore memorizzato in una variabile puntatore (quindi accedendo al valore memorizzato all'indirizzo indicato dal puntatore).

Abbiamo alcune caratteristiche importanti:

- Il **nome** di un array può essere usato come un puntatore al suo primo elemento
- Possiamo inoltre eseguire **operazioni aritmetiche** sui puntatori, ovvero, dato un puntatore p, l'operazione p+2 è valida, e punterà a due puntatori oltre il primo. Questo può essere utile, per esempio, per verificare la lunghezza di una stringa, dato che sono semplicemente un array di char.

Es. Int x = 42;

```
int* p = &x //p contiene indirizzo x,  
          ottenuto tramite &  
int* q = p;  
int y = *q;
```

```
char s[] = "Catarella";  
char* p = 0;  
int l = 0;  
For (p = s; *p; p++) l++;
```

## Blocchi

In C, il **blocco** è un insieme di operazioni racchiuso tra graffe { e }, dove ogni blocco introduce un ambito, ovvero uno **scope**. Con i blocchi si usano i soliti operatori.

## Operatori condizionali/di iterazione

Introduciamo i seguenti operatori, che sono gli stessi rispetto a quelli di Java e C#:

- If (<espressione>) <blocco o statement> else <blocco/statement/else if (<espressione>)
- While (<espressione>) <blocco o statement>
- Do <blocco o statement> while (<espressione>)
- For (<inizio>; <espressione>; <espressione di aumento>) <blocco o statement>
- Switch (<espressione>) { case <valore letterale 1>: <statement>\* break;
- Return <espressione>;
- Goto <label>;
- <label> : <statement>

In C++ aggiungiamo il **try-catch**.

## Espressioni ed operatori

In C/C++, abbiamo una sequenza di espressioni, le quali sono costruite a partire da operatori.

- **Accesso ai membri**
  - o `.` accesso ai membri di una struttura o di un oggetto
  - o `—>` accesso ai membri di un oggetto attraverso un puntatore
  - o `[]` accesso agli elementi di un array o un vettore
  - o `&` ottiene l'indirizzo di memoria di una variabile
- **Chiamata di una funzione**
  - o `()` chiama una funzione
- **Dimensionamento**
  - o `Sizeof` restituisce la dimensione in byte di un tipo o di un oggetto
- **Aritmetici**
  - o `+, -, *, /, %` addizione, sottrazione, moltiplicazione, divisione, modulo
- **Incrementi e decrementi**
  - o `++, --` incremento e decremento di variabili
- **Shift**
  - o `<<, >>` operazioni di shift a sinistra e a destra
- **Logici e Booleani**
  - o `~` NOT bitwise
  - o `!` NOT logico
  - o `<, <=, >, >=` confronti
  - o `==, !=` uguaglianza e disuguaglianza
  - o `&, ^, |` AND, XOR, OR bitwise
  - o `&&, ||` AND, OR logici
- **Condizionali**
  - o `<espressione> ? <vero> : <falso>` operatore ternario
- **Assegnamento**
  - o `=` assegna valore operando destro a operando sinistro
  - o `*=` moltiplica operando sinistro per destro e assegna risultato a sinistro
  - o `/=` divide operando sinistro per operando destro e assegna risultato a sinistro
  - o `%=` resto divisione operando sinistro per destro e assegna risultato a sinistro
  - o `+=` aggiunge operando destro a sinistro e assegna risultato a sinistro
  - o `-=` sottrae operando destro da operando sinistro e assegna risultato a sinistro
  - o `<<=, >>=` sposta bit operando sinistro in direzione specificata da operando destro
  - o `&=` AND bit a bit tra sinistro e destro e assegna a sinistra
  - o `|=` OR bit a bit tra sinistro e destro e assegna a sinistra
  - o `^=` XOR bit a bit tra sinistro e destro e assegna a sinistra

In C++ introduciamo anche:

- **typeid**: restituisce un oggetto `std::type_info` che contiene informazioni sul tipo di un'espressione o di un tipo.
- **dynamic\_cast**: casting dinamico per convertire puntatori o riferimenti a oggetti di una classe di base in puntatori o riferimenti a oggetti di una classe derivata durante l'esecuzione.
- **static\_cast**: casting statico per eseguire conversioni di tipo a compile-time tra tipi correlati.
- **reinterpret\_cast**: reinterpreta tipo di puntatore o di un riferimento. Converte bit a bit tra i tipi, senza controllo di sicurezza.
- **const\_cast**: rimuove la qualifica di `const` o `volatile` da un puntatore o da un riferimento.
- **throw**: genera un'eccezione.

## Funzioni

Un programma è costituito da un insieme di funzioni; una di queste, **main**, ha il ruolo particolare di rappresentare il punto d'inizio di un programma, e ha come tipo di ritorno **int** perché deve ritornare al

sistema un codice di sistema, che cambia se l'esecuzione ha avuto successo o un errore. Viene definita in questo modo:

```
<return type> <function name> ( <arg>* ) {  
    <declaration>*  
    <statement>*  
}
```

Le **variabili locali** vengono dichiarate all'interno di una funzione, e di solito devono essere inizializzate.

### Parametri

I **parametri** vengono passati ad una funzione "per valore", con l'eccezione degli array. I puntatori ci permettono di simulare ciò che in altri linguaggi viene chiamato "passaggio di parametri per riferimento":.

```
int x = 3; int y = 42;  
void swap(int* a, int* b){  
    int temp = *a; *a = *b; *b = temp;  
    swap(&x, &y);  
}
```

Possiamo anche usare direttamente il & all'interno di parametri, piuttosto che usare i puntatori con \*. Se usiamo questo, allora poi non dovremmo usare né & né \* nella chiamata.

Se noi passiamo un parametro di tipo array, C lo **converte automaticamente** un'espressione di tipo T[] ad un **puntatore al primo elemento**. Ovviamente possiamo passare anche struct.

### Compilazione

Per compilare il programma, si invoca il compilatore **gcc hello.c**. Il codice sorgente di applicazioni e librerie è distribuito su un insieme di files e directories. I programmi C si basano sulla distinzione tra **header files** (estensione .h o .hpp) e **file di implementazione** (.c, .cc, ecc...). Questa distinzione fa leva sul **pre-processore C/C++**.

Chiamando il compilatore dalla linea di comando fa sì che venga chiamato sia il preprocessore che il linker; dati opzioni alla linea di comando.

### Preprocessore

Il preprocessore è un programma che trasforma testo, ma non è necessario che il testo sia un programma. Opera sulla base di direttive, le quali più utili sono di tre tipi: inclusione di testo, definizione di macro, condizionali. Queste direttive funzionano assieme.

- **Inclusione:** #include "file.h" #include <file.h>
  - o "" : directory corrente, <>: directory di sistema
- **Definizione di macro:** #define PI 3.14L #define max(x,y)  
( (x)<(y) ? (y) : (x) )
- **Condizionali:** #ifdef PI ... #else ... #endif
  - o Verifica se la macro PI è stata definita in precedenza nel codice. Se è stata definita, il codice verrà compreso nella compilazione. Se non lo è, allora ignorerà il blocco
  - o **IMPORTANTE** nei file header, perché permette di evitare la duplicazione di funzioni ecc....

Usare il preprocessore era l'unico modo di introdurre costanti simboliche in C, e buona parte degli header files di sistema contengono parecchie definizioni di costanti che vengono processate dal preprocessore.

### Modularizzazione

Ogni programma dovrebbe essere **modularizzato** in maniera appropriata; questa operazione corrisponde all'operazione di **compilazione separata**. Un compilatore di solito agisce su un solo file, e il risultato è un file

oggetto contenente dei riferimenti irrisolti a codice non direttamente disponibile. Il linker ha il compito di risolvere questi riferimenti e, nel caso, segnalare degli errori qualora qualche riferimento rimanga irrisolto.

## Librerie

Una **libreria** è un file in un particolare formato che può essere manipolato dal linker, sia staticamente che dinamicamente. Una libreria dinamica ha l'estensione **.so** in Unix, e **.dll** in Windows. È essenzialmente una collezione di file oggetto con un indice associato che permette al linker di andare a caricare il codice corrispondente a un dato *entry point*, ovvero ad una funzione.

Di seguito un esempio.

## UNION-FIND

Consideriamo una libreria QUICK-UNION-PCP, che implementa l'algoritmo UNION\_FIND. Potremmo mettere tutto il codice in un solo file con la funzione **main**, che usa le funzioni *find* e *unite*.

Invece, quello che dobbiamo fare è **scorporare** l'interfaccia della libreria QUICK-UNION-PCP, e metterla in un file **QUPCP.h**:

```
#ifndef _QUPCP_H
#define _QUPCP_H

extern const int N;    //inizializzazione non
desiderata
extern bool find(int, int);
extern void unite(int, int);
extern void init_quick_find_pcp();
#endif
```

#ifndef - #define : serve per  
controllare se all'interno di un  
programma in cui carichi l' header  
sia già definito un metodo

Se è già definito: non ridefinirlo

Il parametro N **non è visibile all'esterno dell'implementazione**. L'unico elemento non parametrico è il tipo (int) della rappresentazione usata per codificare gli elementi. Visto che all'interno di QUPCP.h abbiamo soltanto, essenzialmente, un'interfaccia, allora poi all'interno di QUPCP.cc potremmo **cambiare completamente l'implementazione** di essa.

Le "costanti", per convenzione, devono essere definite con un \_ prima, per evitare di avere nomi banali.

## Namespaces (C++)

Ritornando all'esempio di prima, talvolta è utile avere delle variabili con lo stesso nome. Questo però potrebbe generare conflitti all'interno del proprio programma. Per questo, è opportuno introdurre il concetto di **namespace**, in C++. Tutti i nomi delle variabili all'interno dello namespace, quindi, non avranno conflitto di nomi con altre librerie o altri file, perché i **nomi apparteranno solo a un dato namespace**.

```
#ifndef _QUPCP_H
#define _QUPCP_H

namespace QUPCP {
    extern const int N; // Check this.
    bool find(int, int);
    void unite(int, int);
    void init_quick_find_pcp();
}

#endif
```

## Memoria dinamica

Finora, abbiamo visto solo dichiarazioni e definizioni di oggetti C/C++ che vengono allocati sullo stack di sistema; questo è detto **automatic memory**. Il C/C++ ci obbligano a gestire esplicitamente la memoria dinamica (**heap**).

Non esiste la nozione di **garbage collection**. Viene allocata e de-allocata usando la coppia di funzioni **malloc** e **free**.

```
int *p = (int*) malloc(10 * sizeof  
(int));  
free(p);
```

Allocà un puntatore a 10 interi nello  
heap con malloc

La funzione **malloc** restituisce un puntatore di tipo **void\*** ad una zona di memoria ; è necessario quindi inserire tutte le necessarie conversioni di tipo per evitare problemi con il compilatore. Le dimensioni del blocco di memoria ritornato dipendono dal parametro passato alla funzione; se non vi è memoria disponibile, malloc **restituisce un puntatore nullo**.

Per manipolare il **free store**, usiamo gli operatori *new* e *delete*. Possiamo quindi anche evitare di usare **malloc** nel frammento precedente, e invece scrivere:

```
int *p = new int[10];  
delete [] p;
```

## Stream

C e C++ offrono librerie sofisticate per gestire input, output e files, focalizzandosi sugli **stream**. In C++, in particolare, questi sono istanze di classi di libreria, mentre in C sono spesso associati a files.

### Gli stream in C

Nella libreria C **stdio.h**, troviamo tre stream fondamentali:

- **stdin** standard input
- **stdout** standard output
- **stderr** standard error

Altri stream sono creati e associati a strutture di tipo FILE.

## Output

Le funzioni di output più semplici sono le seguenti:

- **int fputc(int c, FILE\* ostream)** scrive il carattere c su ostream
- **int fputs(const char\* s, FILE\* ostream)** scrive la stringa s su ostream
- **int fprintf(FILE\* ostream, const char\* format, ...)**
  - scrive la stringa format su ostream dopo averla interpretata sulla base degli argomenti forniti
  - Interpreta la stringa format sostituendo alle direttive % delle stringhe che dipendono dal parametro corrispondente.
  - Può contenere anche dei **token**:
    - \n: new line
    - \t: tabulazione
    - %d: intero int
    - %f: floating point float
    - %s: stringa char\*
- **printf("%d", int) == fprintf(stdout, "%d", int)**

## Input in C

In C, le funzioni principali dedicate alla gestione dell'input sono le seguenti:

- `int fgetc(FILE* ostream)` legge un carattere da istream e lo restituisce
- `char* fgets(char* s, int n, FILE* istream)`
  - o legge al più n caratteri da istream nella stringa s e la restituisce; se c'è un newline, o se istream è alla fine, l'operazione di lettura si ferma e un puntatore nullo viene restituito.
- `int fscanf(FILE* istream, const char* format, ...)`
  - o legge l'input da istream sotto il controllo del contenuto della stringa format; i parametri passati devono essere dei puntatori ad aree di memoria dove è possibile depositare il valore letto. È in grado di eseguire il parse delle stringhe.
- `scanf("%d", &x) == fscanf(stdin, "%d", &x)`

Vediamo un esempio:

```
int read_int_from_stream(FILE* in) {  
    int x = 0;  
    fscanf(in, "%d", &x);  
    return x;  
}  
  
int an_int = read_int_from_stream(stdin);
```

## File I/O in C

Le funzioni di input ed output agiscono su stream che possono essere associati a files. Per associare uno stream ad un file, si usa **fopen**; per rompere questa associazione, si usa invece la funzione **fclose**. La funzione **remove**, invece distrugge un file.

- `FILE* fopen(const char* file name, const char* mode)`
  - o Il parametro filename è il nome completo del file da aprire
  - o Il parametro mode controlla come viene aperto: *r* in sola lettura, *w* azzerà o crea un file in scrittura, *a* esegue un append
  - o Ritorna un puntatore ad uno stream FILE, o il puntatore nullo se viene segnalato qualche errore
- `int fclose(FILE* stream)`
  - o segnala al file system che il file associato a stream non verrà più usato dal programma
  - o Il valore ritornato è **0** se la chiamata va a buon termine, altrimenti **EOF** in caso contrario.

## Gli stream in C++

Gli stream sono **istanze di classi**, come: `std::cin`, `std::cout`, e `std::cerr`, rispettivamente input, output ed errore. Altri stream sono creati e associati a files usando le classi `ifstream` (input file stream) e `ofstream` (output file stream).

## Output

In C++, si usa l'operatore `<<` per scrivere qualcosa su un output stream. L'operatore è associativo a destra, e prende due parametri di input: un output stream, e un valore di un qualche tipo riconosciuto.

La stringa **endl** è di fatto un puntatore ad una funzione, che l'operatore `<<` riconosce e richiama; il suo effetto è di scrivere un carattere newline sullo stream, e di assicurarsi che esso sia svuotato.

Abbiamo anche a disposizione la specializzazione dell'operatore `<<` e `>>`, che ha come prerequisito la definizione delle nozioni di operator e di overloading in C++.

Un esempio:

```
((cout << "Il numero ") << 42) << endl;
```

## Input

In C++, si usa l'operatore `>>` per leggere qualcosa da un input stream. L'operatore è associativo a destra, e prende due parametri in input: un input stream, e un puntatore (o una referenza) ad un oggetto di qualche tipo riconosciuto.

Vediamo un esempio:

```
istream& operator>>(istream& s, complex& a) {  
    double re = 0, im = 0;  
    char c = 0;  
  
    s >> c;  
    if (c == '(') {  
        s >> re >> c;  
        if (c == ',') s >> im >> c;  
        if (c != ')') s.clear(ios_base::badbit);  
    } else {  
        s.putback(c);  
        s >> re;  
    }  
    if (s) a = complex(re, im);  
    return s;  
}
```

Funzione sovraccaricata operator>>. Variabili locali: dichiarate `re` (parte reale), `im` (parte immaginaria), `char c` (carattere temporaneo)

Lettura del primo carattere dallo stream di input s e assegnato a c

Se il carattere letto è una parentesi aperta.. Viene letto il valore della parte reale e il successivo carattere, Se il carattere letto è una virgola, leggi la parte immaginaria e un carattere Se il carattere dopo non è una parentesi chiusa, viene impostato lo stato di errore Se il carattere iniziale non era una parentesi aperta, o se la parentesi non è stata trovata, il carattere viene rimesso nello stream di input Viene letto il valore della parte reale nel caso in cui non sia stata trovata una parentesi aperta

Se tutto è andato bene, il numero complesso a viene inizializzato con i valori letti Restituzione dello stream di input modificato

## File I/O in C++

In C++, l'input e l'output su files si basa sulla costruzione di istanze delle classi **ifstream** e **ofstream**, entrambi definite in `<iostream.h>`. Tali istanze possono essere manipolate tramite gli operatori `>>` e `<<`.

Vediamo un esempio:

```
ifstream bar("bar.txt");  
int qd;  
  
bar >> qd; bar.close(); cout << qd;  
  
ofstream bar("bar.txt", "a");  
bar << '\n' << "Vogons" << endl;  
bar.close();
```

Supponiamo che il file `bar.txt` contenga la stringa **42**; la stringa **42**, quindi, apparirà sullo standard output.

Con la seconda chiamata invece, il file conterrà anche la stringa **Vogons**, perché attraverso il modo `a`, viene effettuato l'append; essa è preceduta da un newline.

## La direttiva `typedef`

La direttiva **typedef** in C serve a creare alias per tipi di dati esistenti; ciò fornisce un modo per dichiarare nuovi nomi per tipi di dati esistenti, rendendo il codice più leggibile e fornendo un livello di astrazione. Abbiamo, per esempio, due categorie di esempi:

- **Alias per array**

```
typedef char buffer[1024]; buffer x;  
buffer diventa un alias per l'array di caratteri di dimensione 1024
```

- **Alias per puntatore a struttura**

```
struct _person { char name; int age; char coll[80]; };  
typedef struct _person* Person;  
Person p = (Person) malloc(sizeof(struct _person));  
Person diventa un alias per un puntatore a struct_person.
```

## Paradigma funzionale e logico

Sono linguaggi ad altissimo livello, generati per **manipolazione simbolica non numerica**. Il programma è visto come una collezione di funzioni o di relazioni che operano su dati astratti, senza modificare lo stato del sistema. Questo stile di programmazione è detto **dichiarativo**, perché si focalizza su cosa calcolare piuttosto che come calcolarlo.

Questi linguaggi offrono diversi vantaggi, come la facilità di verifica, l'ottimizzazione e la parallelizzazione del codice, ma anche alcune sfide, come la gestione delle risorse, l'interazione con l'utente e la compatibilità con altri paradigmi.

### Paradigma logico

È un modo di programmare che si basa sulla logica matematica. Il programmatore scrive delle frasi che descrivono il problema da risolvere, usando un linguaggio speciale chiamato linguaggio logico. Queste frasi sono formate da **fatti e regole** che il sistema usa per dedurre le risposte alle domande che gli vengono poste.

Tutto quello che fa il computer in questo caso è "rispondere a una domanda" posta dallo sviluppatore, in base a regole e fatti specificati anch'essi dallo sviluppatore.

### Prolog

Uno di questi tipi di linguaggi è il **linguaggio prolog**, costituito da un paio di strutture:

- **Asserzioni incondizionate** (fatti)
- **Asserzioni condizionate** (regole): si opera sempre l'implicazione
  - o **A: conclusione (conseguente)**
  - o **Il resto: premesse (antecedenti)**
- **Interrogazione** (query): l'interrogazione vera e proprio del computer; restituisce **true** o **false** in base alla richiesta e ai fatti.
  - o L'interrogazione viene fatta direttamente nel terminale, e non nel file sorgente.

collega(X,Y) :-  
lavora(X,Z),  
lavora(Y,Z),  
diverso(X,Y). } *regole*

lavora(ugo,ibm).  
lavora(gino,samsung).  
lavora(enrica,ibm).  
lavora(salvo,samsung). } *fatti*

:- collega(X,Y). } *query*

È un linguaggio di programmazione di **stile dichiarativo**: l'ambiente si fa carico di generare le possibili permutazioni della lista L secondo un processo di deduzione matematica.

### Paradigma funzionale

Il paradigma funzionale è un modo di programmare basato sul concetto di **funzione**. Una funzione è una relazione matematica che associa ad ogni elemento di un insieme, detto **dominio**, un solo elemento di un altro insieme, detto **codominio**.

Una volta definita la funzione, una funzione può essere applicata ad un elemento del dominio, chiamato **argomento** per ottenere l'elemento del codominio corrispondente, chiamato **valore**, attraverso un processo che si chiama **valutazione**.

### LISP (LISt Processing)

In LISP, le liste sono la struttura dati fondamentale, e il programma stesso è rappresentato da una lista, in modo da poter essere manipolato come un dato (**omoiconicità**).

Esempio:

(**defun** member (item list) → definisce funzione member con parametri item e list  
(cond ((null list) nil) → se la lista è vuota restituisce nil  
((equal item (first list)) T) → se l'elemento è uguale al primo elemento della lista restituisce vero  
(T (member item (rest list)))) → altrimenti richiama la funzione member con il resto della lista  
↳ questa condizione deve essere sempre valutata, indipendentemente dal valore delle condizioni precedenti  
(member 42 (list 12 34 42)) → item 42, list 12 34 42

## Logica formale

Un insieme di regole di inferenza costituisce la base di un calcolo logico. Diversi insiemi di regole danno vita a diversi calcoli logici.

Lo scopo di un calcolo logico è di manipolare delle formule logiche in modo completamente **sintattico**, al fine di stabilire una **connessione** tra un insieme di formule di partenza, ed un insieme di conclusioni.

### Dimostrazione

Per dimostrare una frase **F** usando delle affermazioni note **S**, dobbiamo seguire dei passi che ci portano da S a F. Questi passi formano una prova **D**, dove F segue da S, cioè che F è vera se sono vere le affermazioni in S.

Ogni passo della prova D è una frase P che può essere:

- Una delle affermazioni note in S
- Una frase che abbiamo già ottenuto in un passo precedente
- Una frase che deriva da altre frasi usando una regola d'inferenza

$$D \models S \vdash F$$

$$\begin{aligned} D &= \langle P_1, P_2, \dots, P_n \rangle \\ \text{dove} \\ P_n &= F \\ P_i &\in S \cup \{P_j \mid j < i\} \end{aligned}$$

## Logica proposizionale

La logica proposizionale si occupa delle **conclusioni che possiamo trarre da un insieme di proposizioni**. Una logica proposizionale è sintatticamente definita da un insieme P di proposizioni, ed all'insieme P è associata una **funzione di verità** V, che associa un valore di verità ad ogni elemento di P. Questa funzione è il ponte di connessione tra la sintassi e la semantica di un linguaggio logico.

Il valore di verità di una formula composta dipende dalla funzione di verità V e dalle sue componenti; la definizione di V viene quindi estesa sul dominio FBF

Le proposizioni possono essere combinate usando una serie di **connettivi logici**: ^ (congiunzione), ∨ (disgiunzione), ¬ (negazione), => (implicazione).

Chiamiamo **FBF** (formule ben formate) l'insieme di tutte le formule formate dagli elementi di P, e dalle loro combinazioni. Le formule atomiche in P e le loro negazioni vengono anche chiamati **letterali**.

$$\begin{aligned} V(\neg s) &= \text{non } V(s) \\ V(a \wedge b) &= V(a) \wedge V(b) \\ V(a \vee b) &= V(a) \vee V(b) \\ V(p \Rightarrow q) &= (\text{non } V(p)) \vee V(q) \end{aligned}$$

## Tavole di verità

Un modo per calcolare il valore di verità di una proposizione composta è quello di utilizzare la tavola di verità; questo costituisce la parte semantica di un insieme di proposizioni. Di seguito è rappresentata quella standard per tutti i connettivi logici disponibili:

P	Q	$P \wedge Q$	$P \vee Q$	$\neg P$	$P \Rightarrow Q$
V	V	V	V	F	V

V	F	F	V		F
F	V	F	V	V	V
F	F	F	F	V	V

## Calcoli logici

Un calcolo logico dice come generare nuove formule, ovvero espressioni sintattiche, a partire da un insieme di partenza **A** degli assiomi, basandoci su alcune assunzioni:

$$A \subseteq WFF$$

$$S \subseteq WFF$$

$$A = S$$

### Regole di inferenza

Il calcolo proposizionale è basato su una serie di regole di inferenza ben testato, che ci permettono di ottenere delle nuove formule a partire da un insieme di assiomi. La forma generale è:

$$\frac{F_1, F_2, \dots, F_k}{R}$$

$F_i$ : formula vera in FBF  
R: formula generata da inserire in FBF

Le regole di inferenza (congiunzione, modus ponens, disgiunzione, etc.) fanno parte del **calcolo naturale**; questi formalizzano i modi di derivare delle conclusioni a partire da un insieme di premesse. Permettono quindi di derivare direttamente una formula ben formata mediante una sequenza di passi ben codificati. Il *modus ponens* assieme al *terzo escluso* possono essere usati procedendo per assurdo alla dimostrazione di una data formula, chiamata **principio di risoluzione**.

### Modus ponens e tollens

$$\frac{p \Rightarrow q, \quad p}{q}$$

$P \Rightarrow q$ : Se piove, allora la strada è bagnata  
P: piove  
Q: (allora) la strada è bagnata

$$\frac{p \Rightarrow q, \quad \neg q}{\neg p}$$

$P \Rightarrow q$ : Se piove, allora la strada è bagnata  
 $\neg q$ : la strada non è bagnata  
 $\neg p$ : (allora) non piove

Modus ponens: il risultato e quello che stai verificando è positivo  
Modus tollens: è lo stesso del ponens, ma negativo.

La regola sintattica del **modus ponens** ci permette di aggiungere le conclusioni di un'implicazione al nostro insieme di formule ben formate vere da FBF.

La regola sintattica del **modus tollens** ci permette di aggiungere la permessa negata di una regola al nostro insieme di formule ben formate vere.

### Eliminazione ed introduzione di 'E'

#### Eliminazione

$$\frac{p_1 \wedge p_2 \wedge \dots \wedge p_n}{p_1}$$

Piove e la strada è bagnata  
(Segue che) piove

#### Introduzione

$$\frac{p_1, p_2, \dots, p_n}{p_1 \wedge p_2 \wedge \dots \wedge p_n}$$

La regola sintattica dell'**eliminazione** della congiunzione ci permette di aggiungere all'insieme FBF i singoli componenti di una congiunzione.

Introduzione di 'O'

$$\frac{p}{p \vee q} \quad \begin{array}{c} \text{Piove} \\ \text{Piove o c'è la vita su Marte} \end{array}$$

La regola sintattica dell'introduzione della **disgiunzione** ci permette di aggiungere i singoli componenti di una formula complessa; questa regola è detta anche "**di addizione**".

Altre regole

$$\begin{array}{cccc} \frac{P \setminus\setminus \neg p}{\text{vero}} & \frac{\neg p}{p} & \frac{p \setminus\setminus \text{vero}}{p} & \frac{P \wedge \neg p}{q} \\ \text{Terzo escluso} & \text{Eliminazione } \emptyset & \text{Eliminazione } \vee & \text{Contraddizione} \end{array}$$

- **Terzo escluso:** data una proposizione  $p$ , o  $p$  è vera o  $p$  è falsa, non ci sono altre possibilità
- **Eliminazione  $\emptyset$ :** permette di eliminare il connettivo  $\emptyset$  (che significa "non"). Es.  $\neg(\neg p) = p$
- **Eliminazione  $\vee$ :** permette di eliminare il connettivo  $\vee$ .
- **Contraddizione:** una formula è falsa se si può dimostrare che implica sia una proposizione che la sua negazione

**Principio di risoluzione**

È una regola di inferenza generalizzata semplice, facile da utilizzare e implementare. Opera su FBF trasformate in **forma normale congiunta**, ognuno dei congiunti di queste formule viene detto **clausola**.

Viene estesa la nozione di rimozione dell'implicazione sulla base del principio di contraddizione.

$$\frac{\frac{p \setminus\setminus \neg r, s \setminus\setminus r}{p \setminus\setminus s}}{\text{Clausola risolvente}} \quad \frac{\neg r, r}{\wedge} \quad \text{Clausola vuota}$$

La generazione della clausola vuota corrisponde all'aver dimostrato che il mio insieme di FBF contiene una contraddizione. Se ho derivato

$$\frac{\neg p, q_1 \vee q_2 \vee \dots \vee q_k \vee p}{q_1 \vee q_2 \vee \dots \vee q_k} \quad \frac{p, q_1 \vee q_2 \vee \dots \vee q_k \vee \neg p}{q_1 \vee q_2 \vee \dots \vee q_k} \quad \frac{A_1 \wedge A \vee B}{B}$$

*Unit resolution*

$r$  e  $\neg r$ , allora posso dedurre qualunque cosa, quindi anche la clausola vuota.

Quando una delle due clausole da risolvere è un **letterale**, anche negato, come nel caso di  $p$ , allora si parla di **risoluzione unitaria**.

Es. (Da) <Non piove>, <piove o c'è il sole> (segue che) <c'è il sole>

### Dimostrazioni per assurdo

È un metodo logico che serve a dimostrare che una frase è vera mostrando che la sua negazione è falsa.

Supponiamo di avere a disposizione un insieme di formule **FBF** vere, data una certa interpretazione  $V$ . Supponiamo di voler dimostrare che una certa proposizione  $p$  è vera. Procediamo con la **reductio ad absurdum**:



- Assumiamo che  $\neg p$  sia vera
- Se, combinandola con le proposizioni in FBF ottengo una contraddizione, allora concludo che  $p$  deve essere vera

Esempio: proviamo che  $p$  è vera

- $FBF = \{p\}$
- Assumiamo  $\neg p$
- $FBF \cup \{\neg p\}$  genera una contraddizione
- $P$  deve essere vera

Esempio: proviamo che  $q$  è derivabile

- $FBF = \{p \Rightarrow q, p, \neg w, e, r\}$
- Assumiamo  $\neg q$
- $FBF \cup \{\neg q\}$  genera una contraddizione
- $P \Rightarrow q \approx \neg p \vee q$  combinato con  $p$  produce  $p$ 
  - Abbiamo  $q \wedge \neg q$ , contraddizione
  - Se applico a  $q$  e a  $\neg q$  il principio di risoluzione ottengo la clausola vuota

### Assiomi

Gli assiomi sono delle proposizioni o dei principi che si assumono come **veri senza bisogno di dimostrarli**.

Devono essere tra loro indipendenti, non contraddittori e in numero finito. Questi sono detti anche sempre veri, indipendentemente dalla loro interpretazione; sono quindi dette **tautologie**.

- **A1:**  $A \Rightarrow (B \Rightarrow A)$
- **A2:**  $(A \Rightarrow (B \Rightarrow C)) \Rightarrow ((A \Rightarrow B) \Rightarrow (A \Rightarrow C))$
- **A3:**  $(\neg B \Rightarrow \neg A) \Rightarrow ((\neg B \Rightarrow A) \Rightarrow B)$

Alcune tautologie sono codificabili come regole di inferenza e viceversa.

- **A4:**  $\neg(A \wedge \neg A)$ : principio di non contraddizione
- **A5:**  $A \vee \neg A$ : principio del terzo escluso

### Costruire delle prove

Quando vogliamo dimostrare qualcosa usando la logica delle proposizioni, dobbiamo seguire dei passi precisi:

1. **Tradurre il problema nel linguaggio della logica delle proposizioni.** Dobbiamo scegliere bene quali sono le proposizioni che rappresentano le informazioni date, e quelle che vogliamo dimostrare. Devono essere chiare, semplici e non ambigue.
2. **Individuare i teoremi che vogliamo dimostrare.** Devono essere coerenti, verificabili e utili per il problema.
3. **Dimostrare i teoremi usando delle regole di inferenza.** Sono delle procedure che ci permettono di ottenere nuove proposizioni a partire da quelle già note.

Esempio dell'unicorno

Se l'unicornio è mitico, allora è immortale, ma se non è mitico, allora è mortale. Se è mortale o immortale, allora è cornuto. L'unicornio è magico se è cornuto.

**Domande:**

- A) L'unicornio è mitico?
- B) L'unicornio è magico?
- C) L'unicornio è cornuto?

### Passo 1

Se l'**(unicornio è mitico)**<sup>UM</sup>, allora l'**(unicornio è immortale)**<sup>UI</sup>, ma se non (è mitico)<sup>¬UM</sup> allora (è mortale)<sup>¬UI</sup>. Se l'**(unicornio è mortale)**<sup>¬UI</sup> o l'**(unicornio è immortale)**<sup>UI</sup>, allora (unicornio è cornuto)<sup>UC</sup>. L'**(unicornio è magico)**<sup>UMag</sup> se l'**(unicornio è cornuto)**<sup>UC</sup>

### Trascrizione

$$S = \{ UM \Rightarrow UI, \quad \neg UM \Rightarrow \neg UI, \quad \neg UI \vee UI \Rightarrow UC, \quad UC \Rightarrow Umag \}$$

**Proposizioni:**

UM = unicornio è mitico

UI = unicornio è mortale

Umag = unicornio è magico

UC = unicornio è cornuto

### Domanda 1

S | - UM ?

- P1: UM  $\Rightarrow$  UI
- P2:  $\neg$  UM  $\Rightarrow$   $\neg$  UI
- P3: UM  $\Leftrightarrow$  UI

### Domanda 2

S | - UC ?

- P1:  $\neg$  UI  $\vee$  UI  $\Rightarrow$  UC (S)
- P2:  $\neg$  UI  $\vee$  UI (A5)
- P3: UC (P1, P2, MP)

### Domanda 3

S | - UMag ?

- P1:  $\neg$  UI  $\vee$  UI  $\Rightarrow$  UC (S)
- P2:  $\neg$  UI  $\vee$  UI (A5)
- P3: UC (P1, P2, MP)
- P4: UC  $\Rightarrow$  Umag (S)
- P5: Umag (P3, P4, MP)

### Sintassi tipografica e semantica

Esiste una differenza tra sintassi e semantica. Un argomento a sostegno di questa differenza consiste nel considerare le seguenti stringhe:

2A

42

42

XLI

33

101010

Queste sono la rappresentazione in linguaggi diversi, dello stesso oggetto; possiamo quindi dire che:

- Un calcolo logico fornisce una manipolazione **sintattica** (simbolica). L'operatore di **derivazione** | - è un operatore sintattico
- La **semantica** di un insieme di formule dipende dalla funzione di valutazione V. Abbiamo la **conseguenza logica** |=
- **Completezza e Validità**: S | - f se e solo se S |= f, dove S è un insieme di formule iniziale, ed f è una FBF; il tutto è in dipendenza da una particolare funzione di verità V.

### Tautologie e modelli

Quando abbiamo delle formule logiche, possiamo assegnare dei valori di verità ai loro letterali, cioè alle proposizioni atomiche che le compongono; questo significa **interpretare una formula**. A seconda dell'interpretazione, le formule possono essere vere o false.

Se abbiamo un insieme di formule S, possiamo cercare un'interpretazione che le renda tutte vere; questo si chiamerà **modello di S**. Un modello ci dice in che modo le formule in S sono compatibili tra loro.

Ci sono delle formule che sono vere in qualsiasi interpretazione, cioè in qualsiasi modello; queste formule si chiamano **tautologie**. Esse sono delle verità logiche, che non dipendono dal significato dei letterali.

## Logica del Primo Ordine

Spoiler: non è la logica del Primo Ordine di Kylo Ren.

La logica proposizionale è molto utile, poiché ha caratteristiche computazionali chiare e una semantica altrettanto chiara. Però, non ci permette di fare asserzioni circa insiemi di elementi in maniera concisa.

Per risolvere questi problemi, la Logica del Primo Ordine introduce le:

- **V (variabili)**: insieme di simboli di variabili
- **C (costanti)**: insieme di simboli di costante
- **R (relazioni)**: insieme di simboli di relazione o predici di varia arità
- **F (funzioni)**: insieme di simboli di funzione di varia arità
- **Quantificatore**
- Connettivi logici e simboli di quantificazione (per ogni, esistenziale)

### Sintassi

La costruzione di un linguaggio logico del primo ordine è **ricorsiva**. I termini più semplici sono i predici **r C<sub>0</sub> x C<sub>1</sub> x ... x C<sub>k</sub>**, ovvero relazioni cartesiane su C scritte come  $r(c_1, c_2, \dots, c_k)$ .

Le funzioni sono definite con il seguente dominio e codominio  $f : C_0 \times C_1 \times \dots \times C_m \rightarrow C$ , e si scrive come  $f(c_1, c_2, \dots, c_m)$ .

Le formule ben formate di un linguaggio logico del primo ordine sono costruite ricorsivamente con:

- Un **termine**  $t_i$  può essere un elemento di C, di V, oppure un'applicazione di una funzione  $f(t_1, t_2, \dots, t_s)$
- Un termine costituito da un predico  $r(t_1, t_2, \dots, t_k)$  dove  $t_i$  è un termine appartiene ad FBF
- Diversi elementi di FBF connessi dai connettivi logici standard appartengono ad FBF; la combinazione di termini è  $t(t_1, t_2, \dots, t_r)$ .
- Le formule *per ogni*  $x . t(t_1, t_2, \dots, x, \dots, t_r)$  e *esiste*  $x . t(t_1, t_2, \dots, x, \dots, t_r)$  appartengono ad FBF.

### Regole

Introduciamo la **regola di eliminazione del quantificatore universale**:

$$\frac{x.T(\dots, x, \dots), c \mid C}{T(\dots, c, \dots)}$$

Date le regole per l'eliminazione del quantificatore universale, dobbiamo esibire delle regole per la manipolazione del quantificatore esistenziale:

$$\frac{T(\dots, c, \dots), c \mid C}{\$x.T(\dots, x, \dots)} \quad \begin{array}{l} \text{Introduzione \$} \\ \$x.\emptyset T(\dots, x, \dots) \circ \emptyset''x.T(\dots, x, \dots) \\ ``x.\emptyset T(\dots, x, \dots) \circ \emptyset\$x.T(\dots, x, \dots) \end{array}$$

## Programmazione logica

La programmazione logica non è soltanto rappresentata dal Prolog: essa costituisce un settore molto ricco, che cerca di utilizzare la logica matematica come base dei linguaggi di programmazione. Ha come obiettivi una semplicità del formalismo, deve essere un linguaggio livello, e deve avere una semantica chiara.

Il programma è scritto come un **insieme di formule**; ha un grande potere espressivo, e la computazione di un programma è in realtà la costruzione di una dimostrazione di una affermazione.

Come base formale usa le **clausole di Horn** (calcolo dei predicati del primo ordine ma con limitazione nel tipo di formule, sono clausole che hanno al più un solo letterale positivo), e il **meccanismo di risoluzione** (utilizzo di particolari tecniche per la dimostrazione di teoremi).

### Formule ben formate e “a clausole”

Ogni formula ben formata di un linguaggio logico del primo ordine può essere riscritta in **forma normale a clausole**; vi sono due tipi:

- **Forma normale congiunta (CNF)**: la formula è una congiunzione di disgiunzioni di predicati o di negazioni di predicati (letterali positivi e letterali negativi)
  - Un insieme di formule CNF è riscrivibile come un insieme (congiunzione) di implicazioni)
- **Forma normale disgiunta (DNF)**: la formula è una disgiunzione di congiunzioni di predicati o di negazioni di predicati (letterali positivi e letterali negativi)

### Linguaggio dichiarativo

Non contiene istruzioni, ma soltanto **fatti** (asserzioni vere nel contesto che stiamo descrivendo) e **regole** (ci danno gli strumenti per dedurre nuovi fatti da quelli esistenti). Un programma prolog ci dà informazioni su un sistema, ed è chiamato **base di conoscenza**; quindi, un programma prolog si **interroga**. Il programma poi risponderà sì o no.

La sintassi è la seguente:

a	% fatto/asserzione
c :- b1, b2, ..., bn.	% regola
libro(kowalski, prolog).	% fatto
ama(Gigi, X) :- ama(X, vino).	% regola
:- q1, q2, ..., qm termini	% goal
?- q1, q2, ..., qm	% query

Ogni regola o fatto o funzione deve terminare con un “”; ogni variabile deve iniziare con una maiuscola; i commenti si inseriscono dopo un “%”, o tra “/\*” e “\*/”.

### Termini

Le espressioni del Prolog sono chiamate **termini**: possono essere **atomi** (numeri, ecc...), **variabili** (iniziano con la lettera maiuscola) e **termini composti** (simbolo di funtore + uno o più argomenti).

### Atomi

Un atomo è una sequenza di caratteri alfanumerici che inizia con un carattere **minuscolo**, che inizia con il **\_**, qualsiasi cosa racchiusa tra apici, un numero, ecc....

### Variabili

Una variabile è una sequenza alfanumerica che inizia con un carattere **maiuscolo**, o con il carattere **\_**. Se sono composte soltanto da **\_** sono dette **anonime**.

Le variabili vengono **istanziate**, ovvero legate a un valore, con il procedere del programma, ovvero nella dimostrazione del teorema.

### Termini composti

Una composizione di termini consiste in un **funtore** (un simbolo di funzione o di predicato definito come atomo), e una sequenza di termini racchiusi tra parentesi tonde e separati da virgole (argomenti del funtore).

## Fatti/predicati

Un fatto consiste nell'avere un **nome di predicato**, che deve iniziare con una lettera minuscola, e zero o più argomenti.

## Regole

Le regole vengono usate quando si vuole esprimere che un certo fatto dipenda da un insieme di altri fatti; effettua realisticamente l'operazione dell'operatore “if”.

Una regola consiste di:

- **Testa**
  - o Corrisponde al **conseguente** di un'implicazione logica
- **:**-
- **Head**
  - o Corrisponde all'**antecedente** di un'implicazione logica
- Corrispondono alle **clausole di Horn**, ovvero hanno un solo predicato come conseguente.

Una relazione può essere definita da più regole (clausole) aventi lo stesso predicato come conclusione; le regole sono implicitamente connesse dall'operatore di congiunzione.

Può essere inoltre anche definita **ricorsivamente**; in questo caso, la definizione richiede almeno due proposizioni: una è ricorsiva che corrisponde al caso generale, e l'altra esprime il caso particolare più semplice.

antenato(X, Y) :- genitore(X, Y). → ricorsione  
Antenato(X, Y) :- genitore(Z, Y), antenato(X, Z). → caso base

## Interrogazioni

Una volta che le regole ed i fatti sono scritti e caricati nell'interprete, eseguire un programma Prolog significa **interrogare l'interprete**.

?- libro(kowalski, prolog)

Dato una query, e dati i fatti e le regole, il Prolog risponde “true” o “false”.

Le interrogazioni possono contenere variabili interpretate come **variabili esistenziali**; sono istanziate quando il Prolog prova a rispondere alla domanda. Tutte le variabili istanziate vengono mostrate nella risposta.

Es. ?- libro(kowalski, LINGUAGGIO).  
true LINGUAGGIO = prolog  
?- libro(AUTORE, prolog).  
true AUTORE = kowalski  
?- libro(AUTORE, LINGUAGGIO).  
True AUTORE = kowalski, LINGUAGGIO = prolog

## Unificazione

L'operazione di istanziazione di variabili durante la prova di un predicato è il risultato dell'**unificazione**. Dati due termini, l'unificazione crea un insieme di sostituzioni delle variabili, che permette di rendere “uguali” i due termini.

L'insieme di soluzioni è chiamato **most general unifier**, indicato con MGU; una sostituzione è indicata come una sequenza di coppie *variabile/Valore*.

## Diverse rappresentazioni

Per introdurre questo concetto, consideriamo un esempio: vogliamo descrivere un insieme di fatti riguardanti i corsi offerti dal dipartimento. Abbiamo tre possibilità:

1. Tutte le informazioni sono concentrate in una relazione con 6 campi  
`corso(linguaggi, lunedì, '9:30', 'U4', 3, Antoniotti).`  
 a. A partire da questa definizione possiamo poi costruire altri predici  
`aula(Corso, Edificio, Aula) :- corso(Corso, _, _, Edificio, Aula, _).`  
`docente(Corso, Docente) :- corso(Corso, _, _, _, _, Docente).`
2. Tutte le informazioni sono concentrate in una relazione con 4 campi; le informazioni sono concentrate in termini funzionali che rappresentano informazioni raggruppate logicamente  
`corso(linguaggi, orario(lunedì, '9:30'), aula('U4', 3), antoniotti).`  
 a. A partire da questa definizione possiamo poi costruire altri predici  
`aula(Corso, Edificio, Aula) :- corso(Corso, _, aula(Edificio, Aula), _).`  
`docente(Corso, Docente) :- corso(Corso, _, _, Docente).`  
`Aula(Corso, Luogo) :- corso(Corso, _, Luogo, _).`
3. I predici che abbiamo definito dalle relazioni con 6 o 4 campi possono essere rimodificate con predici binari  
`giorno(linguaggi, martedì).`  
`orari(linguaggi, '9:30').`  
`edificio(linguaggi, 'U4').`  
`aula(linguaggi, 3).`  
`docente(linguaggi, antoniotti).`  
 a. Le relazioni a 6 o 4 argomenti possono essere ricostruite a partire da queste relazioni binarie

## Liste

In Prolog, la definizione di una lista avviene racchiudendo i suoi elementi, che possono essere **termini o variabili logiche** tra '[' e ']'. Questo approccio consente di rappresentare una vasta gamma di dati in una struttura ordinata.

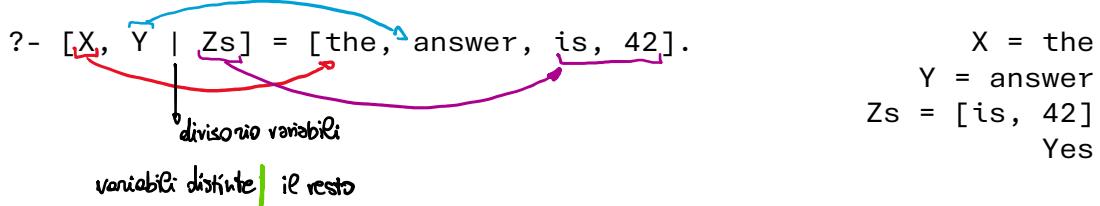
Sono strutture dati versatili; infatti, gli elementi possono essere qualsiasi cosa, inclusi altri elenchi. La lista vuota, indicata da [], è una parte integrante di questa rappresentazione dati, e ha un significato speciale.

`coda : [] [a]`  
`coda : [] [[a]]`  
`coda : [c] [[a, b], c]`

● : testa  
● : coda

Ogni lista non vuota può essere divisa in due parti:

- La **testa** è il primo elemento della lista
  - Se vengono inseriti più elementi distinti dalla virgola prima dell'operatore |, allora ogni variabile divisa da una virgola rappresenterà un elemento diverso.



- La **coda** rappresenta tutto il resto ed è sempre una lista a sua volta.

Per distinguere tra questi due componenti, Prolog utilizza l'operatore |.

## L'operatore =..

L'operatore =.. viene usato per scomporre un termine composto in una lista o per costruire un termine composto da una lista di elementi. Funziona così:

- Per scomporre un termine composto in una lista:
  - Term =.. List scomponete il termine Term in una lista List.
    - Es. `father(john, tom)` → `[father, john, tom]`.

- Per costruire un termine composto da una lista:
  - o Term =.. [Functor | Args] crea un termine composto con il simbolo del funtore Functor e gli argomenti forniti nella lista Args
    - Es. `[father, john, tom] —> father(john, tom).`

## Consult

Per quanto riguarda l'ambiente di lavoro, la base di conoscenza è **nascosta** ed accessibile solo tramite comandi specifici o GUI. Per inizializzare o caricare fatti e regole nell'ambiente, il comando principale da scrivere è **consult**, che richiede almeno un termine che indichi un file come argomento. Questo file deve per forza contenere un insieme di fatti e regole.

Lo sviluppatore può inserire fatti e regole direttamente nella console anche, utilizzando il termine speciale **user**. Per finire di immettere regole o fatti, usare CTRL+D.

Se è necessario ricaricare un file nell'ambiente Prolog con nuove definizioni, si può utilizzare direttamente il predicato **make()**. Questo ha l'effetto di rimuovere completamente i predicati dal database interno e quindi di reinserirli con le nuove definizioni.

## Strutture ad albero

Prolog lavora su strutture ad albero, poiché i programmi di Prolog stesso sono strutture che possono manipolare; usa la ricorsione per tutto, non ha una nozione semplice di assegnamento. Un programma Prolog è un insieme di **clausole di Horn** che rappresentano **fatti** dell'oggetto che si vuole usare e le relazioni che esistono tra di loro; **regole** di oggetti e relazioni (if.. then), e **queries**.

## Clausole

Iniziamo con l'**implicazione**: sappiamo che  $\neg B \vee A$  corrisponde a  $B \rightarrow A$ , scritto come  $A \rightarrow B$ .

Data una clausola  $A_1 \vee A_2 \vee \dots \vee A_n \vee \neg B_1 \vee \neg B_2 \vee \dots \vee \neg B_m$ , possiamo usare la regola di De Morgan:

$$(A_1 \vee A_2 \vee \dots \vee A_n) \vee \neg(B_1 \wedge B_2 \wedge \dots \wedge B_m) \ggg (A_1 \vee A_2 \vee \dots \vee A_n) \leftarrow (B_1 \wedge B_2 \wedge \dots \wedge B_m)$$

Una **clausola di Horn** ha al massimo un letterale possibile; in particolare, possiamo classificare le clausole di Horn come segue:

- |  |   |
|--|---|
| <ul style="list-style-type: none"> <li>- <b>Fatti</b>: <math>A \leftarrow</math></li> <li>- <b>Regole</b>: <math>A \leftarrow B_1 \wedge B_2 \wedge \dots \wedge B_m</math></li> <li>- <b>Queries (goals)</b>: <math>\leftarrow B_1 \wedge B_2 \wedge \dots \wedge B_m</math></li> <li>- <b>Contraddizioni</b>: <math>\leftarrow</math></li> </ul> | <ul style="list-style-type: none"> <li><b>Fatti</b>: <math>A</math>.</li> <li><b>Regole</b>: <math>A :- B_1, B_2, \dots, B_m</math>.</li> <li><b>Queries (goals)</b>: <math>\leftarrow B_1, B_2, \dots, B_m</math>.</li> <li><b>Contraddizioni</b>: fail</li> </ul> |
|--|---|

## Un programma logico

Introduciamo un programma logico che manipola la **rappresentazione unaria dei numeri naturali**, e che definisce **qual è la somma tra due numeri naturali**:

$$\text{sum}(\emptyset, X, X). \quad \text{sum}(s(X), Y, s(s(Y))) :- \text{sum}(X, Y, Z).$$

Prendiamo **s(n)** come il successore di un numero N; quindi: **0, s(0), s(s(0)), s(s(s(0)))** corrisponde a **0, 1, 2, 3**.

Richiediamo una query al programma in questo modo:

<i>Esiste x.sum(s(0), 0, X)</i>	$\{N / s(0)\}$
<i>Esiste w.sum(s(s(0)), s(0), W)</i>	$\{W / s(s(s(0)))\}$

## Sostituzioni

Una sostituzione ci dice quale **valore**, il che può essere anche un'altra variabile, possiamo mettere al posto di una variabile in un fatto o in una regola.

Una sostituzione può essere considerata una **funzione**, che è applicabile a un termine ( $T$  è un set di termini):  $\delta:T \rightarrow T$

$$S = \{X_1/v_1, X_2/v_2, \dots, X_k/v_k\}$$

$\downarrow$   
puoi anche riordinare  
le variabili

Es. Data la sostituzione  $\delta=\{X/42, Y/foo(s(0))\}$ , abbiamo  $S(\text{bar}(X, Y)) = \text{bar}(42, \text{foo}(s(0)))$ .

## Esecuzione del programma

Una computazione è il tentativo di dimostrare, utilizzando la regola di risoluzione che una formula logica segue logicamente da un programma; si sta cercando di dimostrare, quindi, che una certa affermazione è una conseguenza logica di un programma.

Come risultato secondario, la computazione deve produrre una **sostituzione che sia coerente con la derivazione logica dell'obiettivo**; verrà trovata quindi una sostituzione che rende vero l'obiettivo in modo logico.

Se si ha un programma  $P$  e una query del tipo  $\text{:}-p(t_1, t_2, \dots, t_m)$ , in cui  $X_1, X_2, \dots, X_n$  sono le variabili presenti in  $t_1, t_2, \dots, t_n$ , il significato della query è:

Esistono  $x_1, x_2, \dots, x_n$  tali che  $p(t_1, t_2, \dots, t_m)$  è **vero**

L'obiettivo della computazione è di trovare una sostituzione  $s$ , che è una mappatura delle variabili  $X_1, X_2, \dots, X_n$  ai termini  $s_1, s_2, \dots, s_n$ , in modo che  $P$  diventi vero quando valutato con  $P : - s[p(t_1, t_2, \dots, t_m)]$ .

In un insieme di clausole di Horn, si può ottenere la clausola vuota solo se una di esse non ha testa, e questo si verifica soltanto quando abbiamo una sola query  $G_0$  da dimostrare. L'obiettivo è quindi di dimostrare che la clausola vuota può essere ottenuta da  $P \cup \{G_0\}$ , attraverso il metodo della risoluzione all'assurdo, usando il principio di risoluzione.

Questo genera un problema: se, ad ogni passo, provassimo tutte le possibili risoluzioni e aggiungessimo tutte le clausole dedotte all'insieme iniziale, avremmo una **crescita esponenziale delle clausole**, rendendo il processo insostenibile. Per ovviare questo problema, dobbiamo essere selettivi nelle risoluzioni da effettuare e nelle clausole da aggiungere, per evitare il problema della crescita esponenziale.

## Linear Input Resolution (SLD)

Il sistema Prolog stabilisce la verità di una query eseguendo una sequenza di passi di risoluzione; l'ordine generale in cui questi passi vengono eseguiti rende i dimostratori di teoremi basati sulla risoluzione più o meno efficienti. La risoluzione viene sempre applicata **tra l'ultimo obiettivo dimostrato e una clausola del programma**, ma mai tra due clausole del programma o tra una clausola del programma e un obiettivo precedentemente derivato. Questa forma specifica di risoluzione è chiamata **SLD-Resolution** (Selection Function for Linear and Definite sentences Resolution, dove le frasi lineari sono clausole di Horn).

Partendo dal Goal  $G_i$  e dalla regola  $A_r$ :

$$G_i = A_{i,1}, A_{i,2}, A_{i,3}, \dots, A_{i,m}.$$

$$A_r : - B_{r,1}, B_{r,2}, \dots, B_{r,k}.$$

Se esiste un unificatore  $\delta$  tale che  $\delta[A_r] = \delta[A_{i,1}]$ , otteniamo un nuovo goal  $G_{i+1}$ :

$$G_{i+1} = : - B'_{r,1}, B'_{r,2}, \dots, B'_{r,k}, A'_{i,2}, A'_{i,3}, \dots, A'_{i,m}.$$

C'è uno step di risoluzione che il sistema Prolog esegue, dove i termini  $A' \dots$  and  $B' \dots$  sono i risultati  $\delta[A \dots] = A' \dots$  e  $\delta[B \dots] = B' \dots$ . La scelta di unificare il primo su goal di  $G_i$  è arbitrary: scegliere  $A_{i,m}$  o  $A_{i,c}$  con un random  $c$  appartenente a  $[1, m]$  sarebbe comunque legittimo.

Quindi, il passo di risoluzione in Prolog coinvolge l'unificazione di termini tra il **goal corrente** ( $G_i$ ) e la regola ( $A_r$ ), che genera un nuovo goal ( $G_{i+1}$ ) con i risultati dell'unificazione. La scelta di quale termine unificare del goal è arbitraria.

Se invece avessimo questo goal e questa regola:

$$G_i = A_{i,1}, A_{i,2}, A_{i,3}, \dots, A_{i,m} . \quad A_r.$$

Se esistesse un unificatore  $\delta$  tale che  $\delta[A_r] = \delta[A_{i,1}]$ , otteniamo un nuovo goal

$$G_{i+1} = :- A'_{i,2}, A'_{i,3}, \dots, A'_{i,m}.$$

In poche parole, unifichiamo il goal e la regola.

Nel processo di generazione di un goal, molte variabili dei letterali che sono all'interno di questo processo sono costruite: questa costruzione vuol dire **rinominare le variabili**. Una variante per una clausola  $C$  è una clausola  $C'$  ottenuta da  $C$  **rinominando** la sua variabile.

Questa operazione viene effettuata per evitare conflitti tra variabili in contesti diversi e preservare la semanticità originale; se noi non rinominassimo le variabili, allora durante il processo si farebbe sempre riferimento alle stesse variabili di partenza, andando a modificare il loro valore; se quindi avessimo bisogno poi dei valori originali dopo, non li avremmo più.

*Esempio:  $p(X) :- q(X, g(Z)).$  equivalente a  $p(X1) :- q(X1, g(FooFrobboz)).$*

Il risultato finale può essere un:

- **Successo** se la clausola vuota è prodotta: questo vale se, per un numero finito  $n$ ,  $G_n$  è la clausola vuota  $G_n = :-$
- **Fallimento finito** se, per un numero finito  $n$ ,  $G_n$  non è uguale a  $:-$ , e non è possibile trovare un nuovo risolvente tra  $G_n$  e una clausola di un programma.
- **Fallimento infinito** se continuamo a generare nuovi goal  $G_i$ , sempre diversi dalla clausola vuota.

### *Strategie di selezione*

Ci possono essere più di una clausola che può essere usata per risolvere il goal corrente, e quindi anche diverse strategie di ricerca possono essere usate:

- **Depth first**: una clausola è scelta ed è mantenuta in memoria esplorando i loro sottogoal in ordine, fino a quando non trova una **clausola vuota**, o non è possibile procedere (**finite failure**); in questo caso, le scelte fatte nel procedimento vengono riconsiderate
- **Breadth first**: Ogni possibile alternativa è considerata in parallelo.
- **Prolog**: depth first resolution strategy with backtracking: è efficiente con la memoria (non deve mantenere tutte le risoluzioni in memoria, ma soltanto una e poi sostituirla), ma non è completa per le clausole di Horn (può causare cicli infiniti o loop infiniti)

### *Derivation trees*

Dato un programma logico  $P$ , un goal  $G_0$  e una regola di calcolo  $R$ , un albero SLD per  $P \cup \{G_0\}$  via  $R$  è definito sulla base di questo processo:

- **Ogni nodo dell'albero è un goal** (possibilmente vuoto)
- **Il goal  $G_0$  è la radice dell'albero**
- Dato il nodo  $:- A_1, \dots, A_{m-1}, A_m, A_{m+1}, \dots, A_k$ , se  $A_m$  è il nodo selezionato attraverso la regola  $R$ , il suo nodo parente ha un nodo successore per ogni clausola del tipo  $C_i = A_i :- B_{i,1}, \dots, B_{i,q}$  e  $C_k = A_k$ . di  $P$ , cosicché  $A_i$  e  $A_m$  (o  $A_k$  e  $A_m$ ) sono unificabili attraverso una sostituzione generale  $\delta$ .
- Il nodo **successore** viene applicata la label con la su goal clause:
  - $:- \delta[A_1, \dots, A_{m-1}, B_{i,1}, \dots, B_{i,q}, A_{m+1}, \dots, A_k]$
  - $:- \delta[A_1, \dots, A_{m-1}, A_{m+1}, \dots, A_k]$

Mentre il branch dal nodo parente al successore è nominato con la sostituzione  $\delta$  e con la clausola selezionata  $C_i$  (o  $C_k$ )

La regola R può essere variabile: può essere una **Left-most rule**, **right-most rule**, o un “**choice of the best goal**”, ovviamente se diamo una definizione appropriata di cosa vuol dire “best”. L’albero implicito SLD che il sistema prolog genera ordina i successori dall’alto verso il basso in un ordine tipografico del programma, dei fatti e delle regole nel programma P.

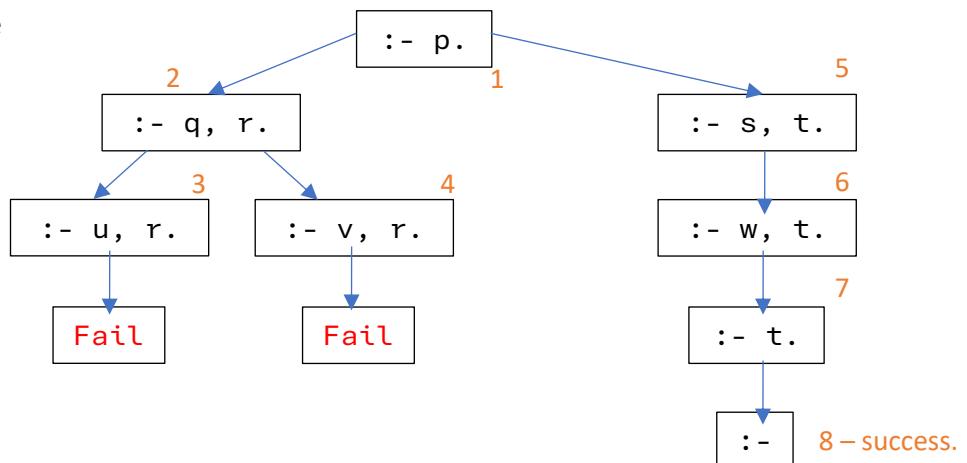
Un processo di costruzione di un albero SLD funziona in questo modo:

1. **Rappresentazione delle derivazioni:** ogni branch dell’albero SLD rappresenta una derivazione SLD. Questo significa che ogni branch terminante con il nodo vuoto rappresenta una derivazione SLD di successo, indicando che il goal iniziale  $G_0$  è stato dimostrato correttamente.
  2. **Influenza della regola R sulla struttura dell’albero:** la scelta della regola di calcolo R influenza la struttura dell’albero SLD sia in termini di ampiezza che di profondità. La regola R determina come i nodi vengono selezionati, generati e etichettati durante la costruzione dell’albero.
  3. **Soundness e completeness:** la regola R scelta non influisce sulla correttezza (soundness), o completezza. Indipendentemente da essa, il numero di percorsi di successo è lo stesso per tutti gli alberi SLD che possono essere costruiti per il programma  $P \cup \{G_0\}$
  4. **Effetto della regola R sui percorsi di fallimento:** l’effetto principale della scelta della regola R riguarda il numero di percorsi di fallimento; può influire quindi sul numero di tali percorsi, b
- possano essere finiti o infiniti.

Esempio di derivation tree

(CL1)  $p :- q, r.$   
(CL2)  $p :- s, t.$   
(CL3)  $q :- u.$   
(CL4)  $q :- v.$   
(CL5)  $s :- w.$   
(CL6)  $t.$   
(CL7)  $w.$

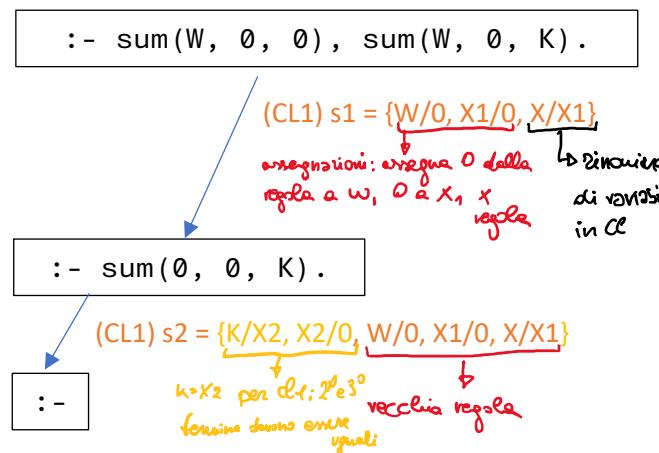
Goal:  $:- p.$



Esempio di derivation tree left

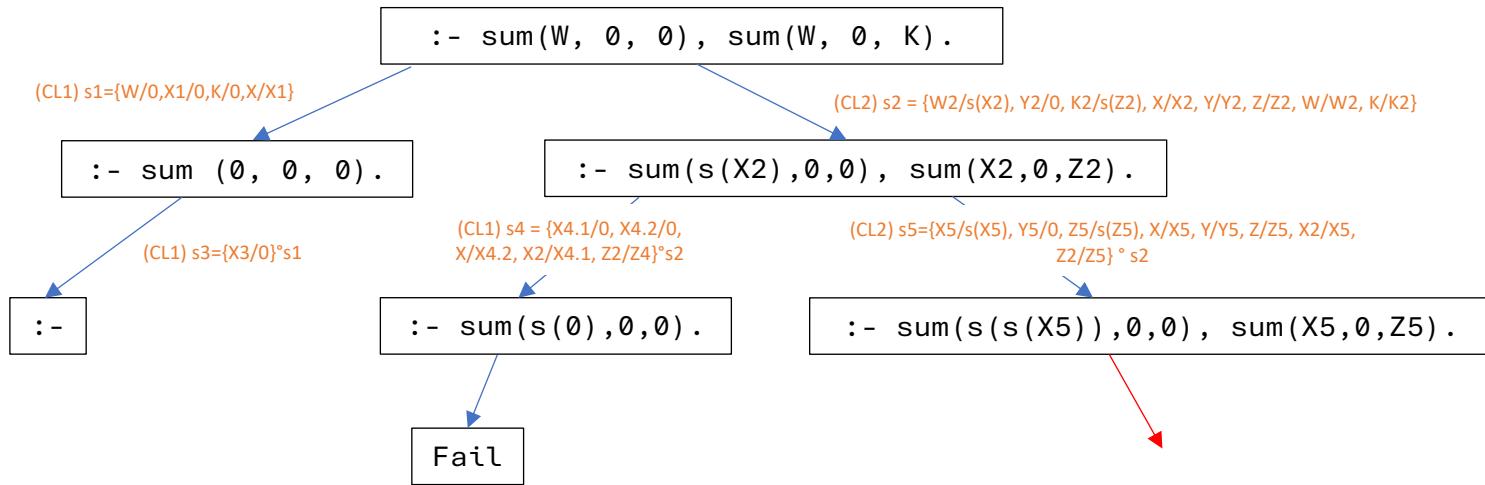
(CL1)  $\text{sum}(0, X, X).$   
(CL2)  $\text{sum}(\text{s}(X), Y, \text{s}(Z)) :- \text{sum}(X, Y, Z).$   
 $G_0 = :- \text{sum}(W, 0, 0), \text{sum}(W, 0, K).$

Per l’example solo il primo è usato, ma bisognerebbe reificare entrambi



Esempio di derivation tree right

Usa le stesse regole di prima.



## Modello di esecuzione di Prolog

Dato una regola  $p :- q, r$ , possiamo avere due interpretazioni:

- **Interpretazione dichiarativa:**  $p$  è vero se sono veri  $q$  e  $r$
- **Interpretazione procedurale:** il problema  $p$  può essere scomposto nei sotto problemi  $q$  e  $r$ .
  - o Quindi, un goal può essere visto come una chiamata ad una procedura, mentre una regola può essere vista come la definizione di una procedura in cui la testa è l'intestazione, mentre la parte destra è il corpo.

Nel suo modello di esecuzione, abbiamo i seguenti componenti:

- **Esecutore:** questa è la parte che gestisce l'esecuzione del programma
- **Programma:** rappresenta il codice Prolog che definisce le regole e i fatti
- **Indicatore di successo/fallimento:** serve a tenere traccia del successo o del fallimento delle operazioni
- **Istanza delle variabili:** questo è il contesto in cui vengono stanziate le variabili
- **Lista di goal:** i goal rappresentano chiamate alle procedure o alle regole

Le clausole nel database di un programma Prolog vengono considerate **da sinistra verso destra e dall'alto al basso**. Grazie all'operazione di backtracking, se un sottogoal fallisce, allora il dimostratore Prolog sceglie un'alternativa la quale possa essere valida, scandendo dall'alto verso il basso la lista delle clausole.

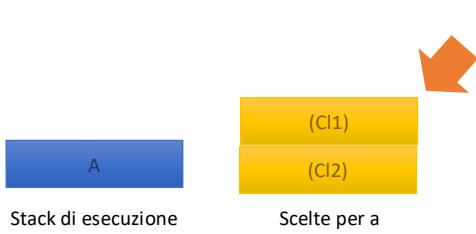
Consideriamo un esempio:

(CL1) a :- p, b.  
 (CL2) a :- p, c.  
 (CL3) p.

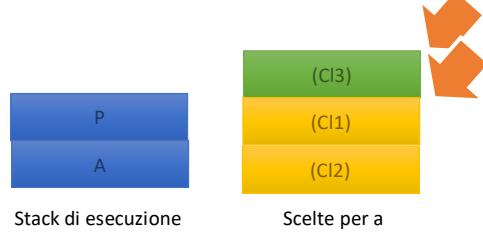
Consideriamo la valutazione della query  $?- a$ . Lo stato interno di Prolog è diviso in due pile:

- **Pila di esecuzione:** contiene i record di attivazione delle procedure, mantenendo traccia delle sostituzioni per l'unificazione delle regole
- **Pila di backtracking:** contiene i punti di scelta aperti durante la dimostrazione, in modo che il backtracking possa essere gestito efficacemente

1

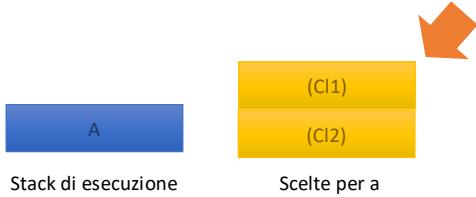


2



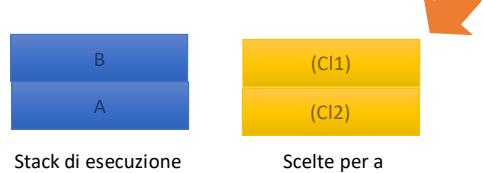
3

*La valutazione di p ha successo*



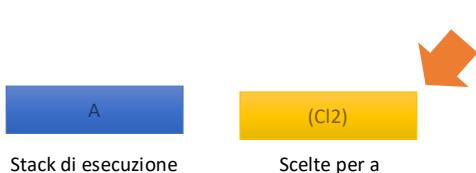
3

*Si inserisce b in cima allo stack, ma la valutazione di b fallisce - backtracking*

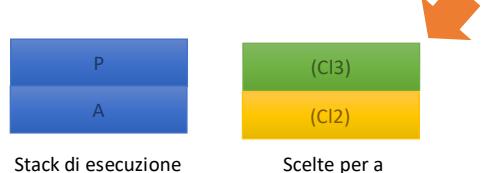


4

*Per proseguire con la valutazione di ?-a., si passa a considerare la seconda clausola.*

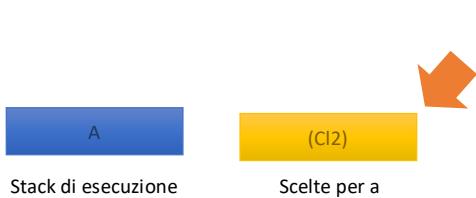


5



6

*La valutazione di p ha successo*



6

*Si inserisce c in cima allo stack; la valutazione di c fallisce. Quindi, viene attivato il backtracking. Ma visto che non ci sono più clausole, anche a fallisce e quindi lo stack di esecuzione si vuota.*



## Il predicato cut '!'

Talvolta, è importante interrompere il backtracking mentre si esegue un programma in Prolog; per questo, viene messo a disposizione l'operatore **cut**, rappresentato dall'operatore **'!'**. Non ha un'interpretazione logica, ma soltanto procedurale. Prendiamo un esempio:

$$C = a :- b_1, b_2, \dots, b_k, !, b_{k+1}, \dots, b_n.$$

Il cut, in questo predicato, produce questo effetto:

- Se il goal corrente G unifica con a, e  $b_1, \dots, b_k$  hanno successo, allora il dimostratore si impegna inderogabilmente alla scelta di C per dimostrare G.

- In poche parole: quando un cut è raggiunto durante l'esecuzione di un programma Prolog, qualsiasi alternativa precedente ad esso che potrebbe soddisfare il goal viene scelto in maniera definitiva, mentre ignora il resto
- $C = a :- b_1, b_2, \dots, b_k, !, b_{k+1}, \dots, b_n.$
- Se un qualche  $b_j$  con  $j > k$  fallisse, il backtracking si ferma a cut
  - Le altre scelte per i  $b_i$  con  $i \leq k$  sono di conseguenza rimosse dall'albero di derivazioni
  - Il backtracking non continuerà oltre il cut se uno dei goal successivi a questo cut dovesse fallire. Il backtracking può ancora avvenire per i goal che si trovano prima del cut.
- Quando il backtracking raggiunge cut, allora il cut fallisce e la ricerca procede dall'ultimo punto di scelta prima che G scegliesse C.
  - Se il backtracking raggiunge il cut, Prolog scarta tutte le scelte fatte dopo il cut, e ritorna al punto in cui il cut è stato raggiunto, quindi non verrà più tentato il backtracking per le alternative successive a quel punto.

Il cut può essere di due tipi:

- **Green cuts**: utili per esprimere determinismo, e quindi per rendere più efficiente il programma
  - Un programma prolog si dice **deterministico** quando una sola delle clausole serve per provare un dato goal
- **Red cuts**: usati per soli scopi di efficienza. Hanno come caratteristica principale quella di omettere alcune condizioni esplicite in un programma e, soprattutto, quella di modificare la semantica del programma equivalente senza cuts. Sono tendenzialmente indesiderabili.

## Predicati meta-logici

I predicati meta-logici ci permettono di manipolare variabili e termini in modi che vanno oltre la tipica logica dei predicati. Questi si concentrano sull'analisi delle variabili e dei termini stessi, consentendo un controllo più fine sul flusso di esecuzione e l'assegnazione di valori nelle variabili.

Quando poniamo una query al sistema Prolog, ci aspettiamo una risposta che sia un'istanza individuale derivabile dalla base di conoscenza. Il meccanismo di **backtracking** ci consente di estrarre tutte le istanze che possono essere derivate, una alla volta, ma qualche volta dovremmo poter estrarre tutte le istanze che soddisfano una query. Questa richiesta è da fare non al primo, ma al **secondo ordine (e oltre)**, poiché richiede tutti gli elementi di un insieme che soddisfano una proprietà parametrica.

## Var e nonvar

Due predicati che ci consentono di esaminare lo stato delle variabili sono:

- **var (X)**: restituisce vero se X è una variabile logica a cui non è ancora stato assegnato un valore, ed è utile per determinare se una variabile è non istanziata e quindi può essere utilizzata come destinazione
- **nonvar (X)**: restituisce vero se X non è una variabile logica, cioè è già stato stanziatato con un valore. Serve per verificare se una variabile contiene un valore specifico o è vuota.

## Atomic e compound

Abbiamo due predicati che ci permettono di gestire questo tipo di variabili:

- **atomic (X)**: questo predicato verifica se un termine è atomico, ovvero è un termine che rappresenta un **valore singolo** e non può essere suddiviso ulteriormente in parti più piccole.
  - Es.  $\text{atomic}(42) = \text{true}$ ,  $\text{atomic}([1,2,3]) = \text{false}$
- **compound (X)**: questo predicato verifica se un termine è composto, ovvero se è costituito da parti più piccole o argomenti. Questi sono usati per rappresentare strutture dati complesse, come funzioni e predicati.
  - Es.  $\text{compound}(\text{padre}(\text{John}, \text{Alice})) = \text{true}$ ,  $\text{compound}(42) = \text{false}$

## Predicati sugli insiemi

Abbiamo una serie di predicati sugli insiemi che Prolog fornisce:

- `findall(Template, Goal, Set)`
  - è vero se Set contiene tutte le istanze del Template che soddisfano Goal
  - Le istanze del Template sono ottenute tramite il backtracking
  - Es. `findall(C, father(X,C), Kids) -> Kids = [abraham, nanchor, haran, Isaac, lot, milcah, yiscah]`
- `bagof(Template, Goal, Bag)`
  - è vero se Bag contiene tutte le istanze del Template che soddisfano Goal
  - Le alternative sono costruite tramite il backtracking solo se ci sono variabili libere in Goal che non compaiono in Template
  - È possibile dichiarare quali variabili non dovrebbero essere considerate per il backtracking grazie alla sintassi `Var^G` come Goal. In questo caso, Var è da considerare come una **variabile esistenziale**.
    - Le variabili di esistenza sono variabili locali alla clausola bagof, e il loro scopo è limitato all'interno di quella clausola; questo significa che non vengono considerate nel backtracking quando si raccolgono soluzioni.
  - Es. `bagof(C, father(X, C), Kids). -> X = terach, KIDS = [abraham, haran, nanchor];`
- `setof(Template, Goal, Set)`
  - si comporta come bagof, ma Set **non contiene soluzioni duplicate**.

## Predicato call

Il predicato **call** è definito come `call(G) :- G`. Questo predicato serve a valutare una query o un goal dinamicamente; quindi, permette di eseguire un goal la cui struttura è definita a tempo di esecuzione. Quindi serve, ad esempio, per eseguire prediciati basati su dati o condizioni variabili.

*Es. ?- X=5, call(write(X)). -> call eseguirà dinamicamente il goal write(X), che stampa 5.*

### Esempio: il predicato apply

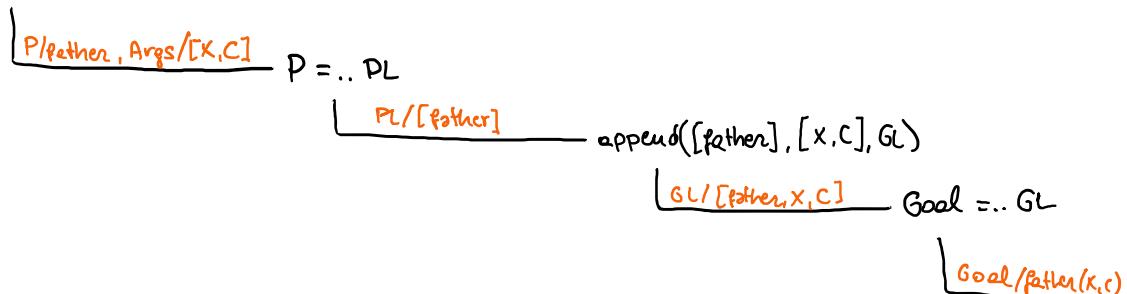
Grazie al predicato **call** e alle meta-variabili, possiamo definire un predicato **apply** che valuta una query composta da un funtore e una lista di argomenti:

```
my_apply(P, Args) :- P =.. PL, append(PL, Args, GL), Goal =.. GL, call(Goal).
```

<code>P =.. PL</code> Scomponi il termine P in una lista di elementi con l'uso di <code>=..</code> . Questo è utile se P è un termine composto, e lo trasforma in una lista PL.	<code>append(PL, Args, GL)</code> Concatena la lista PL con la lista Args. Il risultato è una nuova lista GL che contiene sia il simbolo del predicato che gli argomenti.	<code>Goal =.. GL</code> Usa l'operatore <code>=..</code> per costruire un nuovo termine Goal a partire dalla lista GL; quindi, crea un goal Prolog che corrisponde a P con gli argomenti in Args	<code>call(Goal)</code> Chiama il goal
--	--	--	---

Esempio:

?- my\_apply(father, [X, C]).



Questo predicato costruisce dinamicamente una query usando il funtore e gli argomenti specificati, e la esegue tramite "call".

### Gestire un database

Un programma Prolog è essenzialmente un database contenente fatti e regole. Prolog fornisce alcuni prediciati che possono essere usati per manipolare direttamente il database; attenzione però ad usarli, perché **modificano dinamicamente lo stato del programma**. Alcuni dei predicatori sono i seguenti:

- **listing**: consente di ottenere una lista dei fatti e delle **regole attualmente presenti nel database**.
  - o Con un database vuoto, il risultato sarà **true**
- **assert**, **asserta**, **assertz**: consente di aggiungere **nuovi fatti o regole al database**.
  - o La loro importanza non è tanto il loro successo, poiché assert ha sempre successo, ma l'effetto collaterale sulla modifica del database
  - o asserta aggiunge asserzioni all'inizio del database, mentre assertz li aggiunge alla fine.
- **retract**: permette di **rimuovere** fatti o regole dal database.
  - o Possiamo usare una variabile all'interno di retract: **retract(predicate(X))** rimuoverà tutte le occorrenze di predicate dal database, e **restituirà i valori di X associati**
  - o Per rimuovere ogni fatto all'interno del database quando non ci serve, possiamo usare **retract(predicate(\_))**, **fail**, con all'interno dei parametri di predicate tutti gli elementi che abbiamo come anonimi
    - Es. **Sum/3 → retract(sum(\_, \_, \_)), fail**
    - Lo scopo del "fail" è di forzare il backtracking. Prolog rimuove il primo fatto e fallisce, quindi effettua il backtracking e rimuove il secondo, e così via.

Prendiamo questo come esempio:

```
?- assert(happy(maya)).  
?- assert(happy(vincent)).  
?- assert(happy(marcellus)).  
?- assert(happy(butch)).  
?- assert(happy(vincent)).  
?- assert((naive(X) :- happy(X))).  
?- retract(happy(marcellus)).  
?- retract(happy(vincent)).
```

```
?- listing.  
happy(mia).  
happy(vincent).  
happy(marcellus).  
happy(butch).  
happy(vincent).  
naive(A) :- happy(A).  
true
```

### Memorizzazione

Avere la possibilità di manipolare la base di conoscenza di Prolog è un aspetto estremamente utile. Questa funzionalità può essere sfruttata, ad esempio, per **memorizzare risultati intermedi di varie computazioni**, al fine di evitare future e costose ripetizioni delle query. Il fatto appena asserito viene semplicemente e direttamente confrontato.

Proviamo un esempio:

```
addition_table(A) :- member(B,A), member(C,A), D is B+C, assert(sum(B,C,D)), fail.
```

Se adesso facciamo una query con: `?- addition_table([0,1,2,3,4,5,6,7,8,9]).`, il risultato sarà **false**. La cosa interessante però sono gli effetti sul database, e lo possiamo vedere attraverso listing:

```
?- listing(sum).  
sum(0,0,0).  
sum(0,1,1).  
sum(0,2,2).
```

```

...
sum(9,9,18) .
True

```

## Input ed output

I principali predicati primitivi per gestire l'I/O sono essenzialmente:

- **Read**: consente di leggere dati da uno stream di input. Prolog si aspetta un input dall'utente o da uno stream, e quindi analizza ciò che è stato fornito per costruire un termine prolog.
  - o **Read e read/1**: usato per leggere una singola riga dallo standard input, mentre read/1 consente di specificare da quale stream leggere i dati. Si può usare read/1 per esempio per leggere da file.
- **Write**: consente di scrivere dati su uno stream di output. In generale, viene utilizzato per scrivere dati su uno stream di output. Abbiamo la variante **writeln** anche, che funziona come su Java.
  - o **Write e write/2**: write è utilizzato per scrivere su user\_output, mentre write/2 consente di specificare il termine e il flusso di output.
- **Write\_term**: offre un maggiore controllo sulla formattazione dell'output. Può essere usato quindi, per esempio, per scrivere un termine con diverse opzioni di indentazione.
- **Open**: il predicato open viene utilizzato per aprire uno stream di input o output con sintassi `open(File Name, Mode, Stream)`
  - o **Filename**: atomo che rappresenta il nome del file
  - o **Mode**: atomo che specifica la modalità di apertura, che può essere "read", "write" o "append"
  - o **Stream**: variabile non istanziata che verrà unificata con l'identificazione dello stream aperto
- **Close**: il predicato close viene utilizzato per chiudere uno stream precedentemente aperto, con sintassi `close(Stream)`.
  - o Dopo aver finito di lavorare con uno stream, è buona pratica chiuderlo per liberare le risorse e assicurarsi che tutto venga salvato.

## Gli interpreti

Prolog è particolarmente adatto per essere usato come strumento per la costruzione di interpreti per linguaggi specializzati, chiamati anche **Domain Specific Languages**. Esempi tipici includono interpreti per automi, sistemi per la deduzione automatica, sistemi per la manipolazione dell'elaborazione del linguaggio naturale. Ci concentreremo sugli automi.

## Gli automi

### Linguaggi regolari

Come possiamo costruire un interprete per riconoscere in modo non deterministico linguaggi regolari?

Guardiamo un esempio. Innanzitutto definiamo:

```

accept([I | Is], S) :-
    delta(S, I, N),
    accept(Is, N).
accept([], Q) :- final(Q).

Definisce "accept" che prende una lista di input e uno stato corrente S
"Delta" rappresenta il passaggio da uno stato a un altro: stato S, Input I, nuovo stato N
Il sistema procede a valutare il resto dell'input (meno I) dal nuovo stato N
Base: verifica se lo stato corrente è uno finale e se input è una lista vuota.

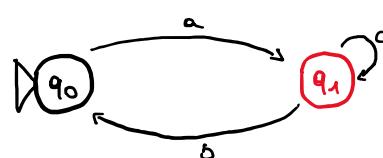
```

Adesso possiamo passare a definire i **fatti**, ovvero i "dati del problema": quale carattere c'è bisogno per passare da uno stato all'altro, e quali sono gli stati iniziali e finali.

```

initial(q0).
final(q1).
delta(q0, a, q1).
delta(q1, b, q0).
delta(q1, c, q1).

```



Adesso possiamo finire definendo invece l'ultimo predicato, ovvero quello che veramente farà partire l'operazione: il predicato **recognise** che decide se, dato una certa lista di elementi dell'alfabeto, questa lista è valida o no:

`recognize(Input) :- initial(S), accept(Input, S).`

Possiamo fare un paragone tra quello che succede in prolog, e quello che succederebbe in un automa:

### Prolog

```

recognize([a,b,a,c,c,b,a]).  

    v  

✓ initial(q0). accept([a,b,a,c,c,b,a], q0).  

    v  

accept([a,b,a,c,c,b,a], q0).  

    v  

accept([a | a,b,a,c,c, ,a ,q0]).  

    v  

✓ delta(q0, a, N), accept a,b,a,c,c,b,a, N . »N=q1  

    v  

accept([a1b,a,c,c,b,a], q1) => N=q1  

    v  

accept([b1a,c,c,b,a], q1) => N=q0  

    v  

accept([a1c,c,b,a], q0) => n = q1  

    v  

accept([c1c,b,a], q1) => n = q1  

    v  

accept([c1b,a], q1) => n = q1  

    v  

accept([b1a], q1) => n = q0  

    v  

accept([a], q0) => n = q1 => accept([], q1), final(q1).

```

### Automa

$$\begin{aligned}
&\hat{\delta}(q_0, abaccba) = \hat{\delta}(\delta(q_0, a), bacca) = \hat{\delta}(q_1, bacca) : \\
&= \hat{\delta}(\delta(q_1, b), accba) - \hat{\delta}(q_0, accba) - \hat{\delta}(\delta(q_0, a), ccba) = \\
&= \hat{\delta}(q_1, ccba) = \hat{\delta}(\delta(q_1, c), cba) = \hat{\delta}(q_1, cba) = \hat{\delta}(\delta(q_1, c), ba) = \\
&= \hat{\delta}(q_1, ba) = \hat{\delta}(\delta(q_1, b), a) = \hat{\delta}(q_0, a) = \hat{\delta}(\delta(q_0, a), \epsilon) = \\
&= \hat{\delta}(q_1, \epsilon) = q_1 \quad \text{ACCETTATO}
\end{aligned}$$

VERO

### Linguaggi liberi dal contesto

Per costruire invece un interprete per riconoscere in modo non deterministico dei linguaggi liberi dal contesto, avremmo bisogno di introdurre una **pila**, per vari motivi:

- **Gestione dello stato:** è necessario tenere traccia dello stato corrente o delle variabili di stato
- **Backtracking:** può essere utile per tenere traccia dei punti di scelta e delle alternative durante il processo di valutazione
- **Grammatiche ricorsive:** quando si interpretano linguaggi con produzioni ricorsive, una pila può essere utilizzata per gestire la ricorsione e il parsing
- **Gestione di strutture dati complesse:** può essere utilizzata per mantenere una traccia delle strutture dati correnti

Introduciamo quindi, con un esempio, un nuovo interprete.

```
% accept(Input, Stato, Pila).  
accept([I | Is], Q, S) :-  
    delta(Q, I, S, Q1, S1),  
    accept(Is, Q1, S1).  
accept([], Q, []) :- final(Q).
```

Accept prende un input, stato corrente e lo stato corrente della pila  
Delta prende Q, I, S e restituisce i **nuovi stati Q1 e S1**  
Richiama ricorsivamente Accept con il resto della stringa e i nuovi stati  
Base: se Is è vuota, verifica se Q è uno stato finale

Definiamo la knowledge base:

```
initial(q0). nuovi  
final(q1). oggiungi
```

```

delta(q0, a, P, q0, [a | P]) .
delta(q0, b, P, q0, [b | P]) .
delta(q0, c, P, q0, [c | P]) .
delta(q0, r, P, q1, P) .
delta(q1, c, [c | P], q1, P) .
delta(q1, b, [b | P], q1, P) .
delta(q1, a, [a | P], q1, P) .

```

Transizione da q0 a q0 quando legge **a**; aggiunge a alla pila.  
 Transizione da q0 a q0 quando legge **b**; aggiunge b alla pila.  
 Transizione da q0 a q0 quando legge **c**; aggiunge c alla pila.  
 Transizione da q0 a q1 quando legge **r**; non modifica la pila.  
 Transizione da q1 a q1 quando legge **c**; rimuove c dalla pila.  
 Transizione da q1 a q1 quando legge **b**; rimuove b dalla pila.  
 Transizione da q1 a q1 quando legge **a**; rimuove a dalla pila.

Come si può notare dal codice, si inseriscono nella pila i simboli di cui ne abbiamo bisogno, ma alla fine, **si rimuovono tutti dalla pila** (e "al contrario", perché la pila rimuove da in cima), cosicchè si può verificare il caso base. Questa non è altro che una **convenzione** da mantenere in modo che sia un po' standardizzata.

Infine, introduciamo il predicato **recognise**, che quello che chiameremo noi, dato un certo input:

```
recognize(Input) :- initial(S), accept(Input, S, []).
```

Nella prossima pagina **esempio**.

*Es. `recognize([a,b,a,c,r,c,a,b,a]). = true`*

```

recognize([a,b,a,c,r,c,a,b,a]) :- initial(S),
accept(Input, S, []).

S = q0, Input = a,b,a,c,r,c,a,b,a

accept([a | b,a,c,r,c,a,b,a], q0, []) :- delta(q0, a, [], Q1, S1), accept([b,a,c,r,c,a,b,a], Q1, S1).
Q1 = q0, S1 = [a | []]

accept([b | a,c,r,c,a,b,a], q0, [a]) :- delta(q0,a,[a],Q1,S1), accept([a,c,r,c,a,b,a],Q1,S1).
Q1 = q0, S1 = [b | [a] ]

accept([a | c,r,c,a,b,a], q0, [b,a]) :- delta(q0,a,[b,a],Q1,S1), accept([c,r,c,a,b,a],Q1,S1).
Q1 = q0, S1 = [a | [b,a] ]

accept([c | r,c,a,b,a], q0, [a,b,a]) :- delta(q0,c,[a,b,a],Q1,S1), accept([r,c,a,b,a],Q1,S1).
Q1 = q0, S1 = [c | [a,b,a] ]

accept([r | c,a,b,a], q0, [c,a,b,a]) :- delta(q0,r,[c,a,b,a],Q1,S1), accept([c,a,b,a],Q1,S1).
Q1 = q1, S1 = [c,a,b,a]

accept([c | a,b,a], q1, [c,a,b,a]) :- delta(q1,c,[c,a,b,a],Q1,S1), accept([c,a,b,a],Q1,S1).
Q1 = q1, S1 = [a,b,a]

accept([a | b,a], q1, [a,b,a]) :- delta(q1,a,[a,b,a],Q1,S1), accept([a,b,a],Q1,S1).
Q1 = q1, S1 = [b,a]

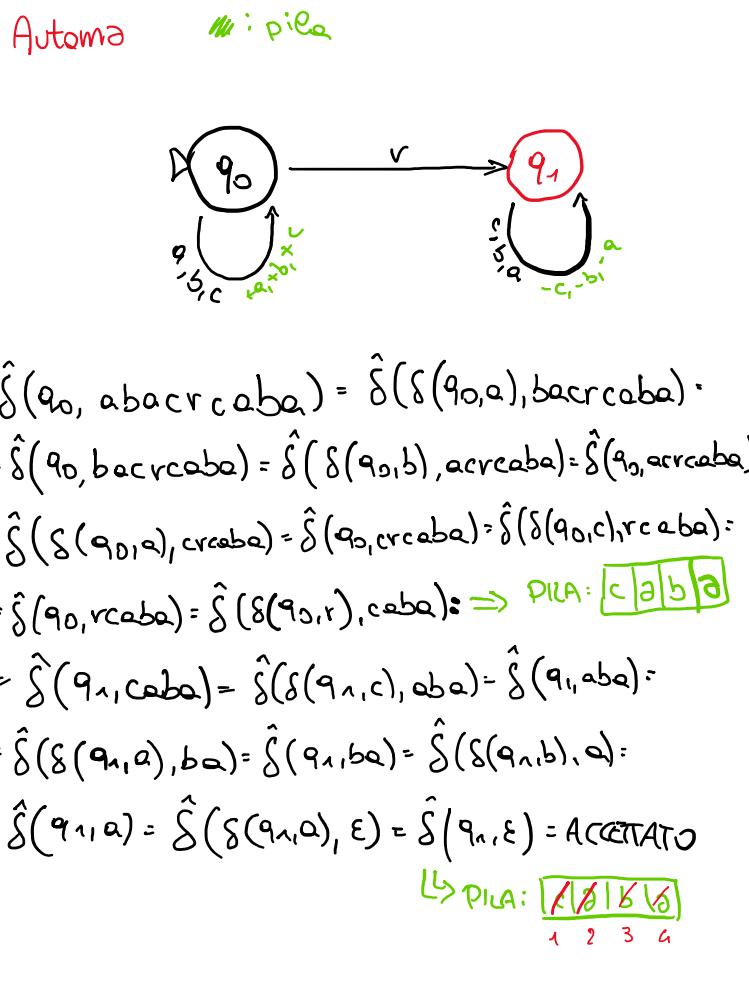
accept([b | a], q1, [b,a]) :- delta(q1,b,[b,a],Q1,S1), accept([b,a],Q1,S1).
Q1 = q1, S1 = [a]

accept([a], q1, [a]) :- delta(q1,a,[a],Q1,S1),
accept([a],Q1,S1).

Q1 = q1, S1 = []

accept([], q1, []) :- final(q1).
True.

```



## Programmazione funzionale

L'essenza dei linguaggi funzionali risiede nell'equazione **programmi = funzioni matematiche**, dove un programma è costituito dalla **combinazione di varie funzioni**, sia primitive che più complesse ottenute attraverso la composizione.

Programmare in un linguaggio funzionale implica la manipolazione di funzioni, denotate nella notazione normale come  $f(x_1, x_2, \dots, x_N)$ . Ogni funzione rappresenta un valore ottenuto tramite una mappa da un insieme di argomenti.

Gli oggetti e le strutture di controllo sono organizzati logicamente in espressioni e modi di combinarle. Si pensa in termini di *espressioni*, *composizione*, e *costruzione di astrazioni* per trattare gruppi di espressioni come unità separate. La definizione di funzione in questo contesto è una **regola che associa elementi di un insieme (dominio) a elementi di un altro insieme (codominio)**. La trasparenza referenziale propria della matematica viene mantenuta.

Un esempio è **quadrato(x) = x\*x**, dove x è un numero, detto **argomento**, che indica un generico elemento del dominio. È una variabile matematica, *senza una precisa locazione in memoria*, quindi non ha senso pensare di modificarla.

### Composizione di funzioni

Nei linguaggi funzionali, espressioni più complesse vengono costruite mediante composizione. Facciamo sempre con un esempio.

Se F è definita come composizione di G e H, scriveremo allora  $F = G \circ H$ , ed *applicare F equivale ad applicare G al risultato dell'applicazione di H*.

```
alla_quarta = quadrato ° quadrato
alla_quarta(5) = (quadrato ° quadrato)(5) = quadrato(quadrato(5))
```

### Ricorsione ed operatori speciali

Le espressioni matematiche, in un linguaggio funzionale, sono invece composte da **composizione di funzioni** spesso organizzate **ricorsivamente**, e controllate da operatori speciali, che possono essere per esempio if-then-else. Nella programmazione funzionale pura, non è possibile produrre effetti collaterali.

Esiste la **ricorsione in coda**, che vuol dire che l'ultima operazione che viene eseguita all'interno di una funzione è la chiamata ricorsiva; questo permette al compilatore di ottimizzare le chiamate ricorsive usando un'operazione di **jump**, evitando di creare aggiuntivi stack di attivazione.

Un esempio di operazione che può essere implementata con questo tipo di ricorsione è il **fattoriale**, a patto che si salvi il risultato di ogni chiamata come parametro; un esempio di operazione che invece non si può fare così è **Fibonacci**. Questo perché l'ultima operazione è la somma di due chiamate ricorsive, quindi non si può usare la ricorsione in coda.

### LISP

LISP non è propriamente un linguaggio, ma una famiglia di linguaggi; l'acronimo sta per **LIST Processing**, proprio perché qualsiasi cosa che si va ad interagire con esso viene trattata come se fosse una lista.

Le versioni minimali di LISP ammettono solo: funzioni primitive su **liste**, un operatore speciale per creare funzioni (**lambda**), un operatore condizionale (**cond**), ed un piccolo insieme di predicati ed operatori speciali che vedremo di seguito. La cons crea un'area di memoria creando una cella cons.

## Interprete e compilatori

Nei linguaggi imperativi, di solito la costruzione dei programmi avviene via il processo di “compilazione”. In LISP, invece, di solito si interagisce con un **ambiente** (il quale spesso contiene il *compilatore*), la cui interfaccia principale è un **interprete**.

## Espressioni

Una volta fatto partire l’interprete LISP, si può procedere ad una esplorazione delle espressioni di base del linguaggio.

Le **espressioni autovalutanti** sono quelle espressioni che hanno lo stesso valore della loro rappresentazione sintattica. Ad esempio, i numeri, le stringhe, i booleani e le lambda expression sono queste.

Ogni espressione **denota un valore**; le espressioni più semplici sono numeri e stringhe. Vediamo due esempi:

```
prompt> 42  
42
```

```
prompt> "Sapete che cos'è '42'?"  
"Sapete che cos'è '42'?"
```

In LISP, una **s-exp, o symbolic expression**, è un modo per rappresentare una lista annidata di dati. Può essere un atomo, un numero, una stringa o un simbolo, oppure una coppia di due s-exp, delimitata da parentesi tonde e separate da un punto.

## Operazioni

In LISP, gli operatori **devono essere prefissi** agli “argomenti” di un’operazione: quindi, per esempio:

$$40 + 2 = +(40,2) = (+\ 40\ 2)$$

Nel secondo caso, è come se stessimo definendo un’operazione + con due numeri come argomenti. Nel terzo caso, è come l’operazione di somma sarebbe scritta su LISP.

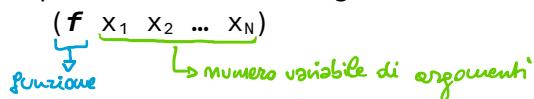
Un paio di particolarità:

$$(+\ ) = 0 \quad (*) = 1 \quad (-\ 42) = -42 \quad (/ \ 42) = 1/42$$

## Funzioni

La sintassi per le “chiamate”, “valutazioni” o **applicazione** di funzioni è molto semplice, ma relativamente diversa dalla notazione tradizionale. Ogni espressione in LISP ha la seguente forma:

$$(f \ x_1 \ x_2 \ ... \ x_N)$$



Le parentesi iniziale e finale sono obbligatorie, e lo spazio è anch’esso: deve essere almeno uno, e sono necessari per **separare tra di loro la funzione e gli argomenti**.

Non vi sono ambiguità possibili nell’interpretazione poiché **la funzione è sempre il primo elemento dell’espressione**. Al fine di rendere più leggibili le espressioni, di solito le si allinea in modo da avere gli argomenti di una chiamata allineati verticalmente.

Sono presenti diversi tipi di dati e operazioni, tra cui:

- **Numeri**: interi, virgola mobile, razionali e complessi
- **Booleani**: **T** (vero) e **NIL** (falso)
- **Stringhe**: sequenze di caratteri encapsulati tra “ ”
- **Operazioni su booleani**: null, and, or, not
- **Funzioni su numeri**:

- **+** somma
- **-** differenza
- **/** divisione
- **\*** moltiplicazione
- **mod** resto divisione intera
- **sin, cos, sqrt, tan, atan** funzioni trigonometriche
- **plusp** verifica se un numero è positivo
- **>** maggiore di
- **<** minore o uguale a
- **Zerop** verifica se un numero è zero
- **Etc..**

Abbiamo anche un ordine di valutazione: la valutazione **procede da sinistra verso destra**, all'interno degli operatori, a partire da  $x_1$  fino a  $x_N$  producendo i valori  $v_1, \dots, v_N$ .

Successivamente, la funzione **f** viene valutata successivamente e viene applicata ai valori  $v_1, \dots, v_N$ . Questa regola inerentemente ricorsiva.

Alcuni operatori speciali, quali *if*, *cond*, *defun*, *defparameter*, *etc* valutano gli argomenti in modo diverso.

### *Definizione di una funzione*

Per definire una funzione usiamo il predicato **defun**:

```
(defun name (argument) (actual expression))
```

### *Funzioni lambda*

Esiste un tipo speciale di funzioni chiamate **funzioni lambda**: sono funzioni anonime definite al volo senza assegnarle un nome specifico. Può essere utilizzata immediatamente o assegnata a una variabile.

```
(lambda (argument) (actual expression))
```

### *Funzioni condizionali*

Le funzioni condizionali usano il predicato **cond**, vengono usate per eseguire una serie di test condizionali e restituire il valore dell'espressione associata alla prima condizione che risulta vera.

```
(cond
```

```
  (test1 result1)
  (test2 result2)
```

```
  ...
  (T defaultResult))
```

*se messo altro  
caso c'è vero*

Una volta che LISP trova una di queste condizioni come vere, l'esecuzione si fermerà.

### *Operatori booleani*

LISP include gli operatori **and**, **or** e **not**, che funzionano come sempre. Possiedono anche dei casi base, i quali sono **T** per l'**and**, e **NIL** per l'**or**.

```
(and c1 c2... cK)      (or d1 d2 ... dM)      (not e)
```

### *Funzioni ricorsive*

Le funzioni di LISP possono essere richiamate ricorsivamente all'interno di altre funzioni. Di solito, sono utilizzate per implementare dei loop, il che non è possibile in nessun altro modo in LISP. Come funzionamento, sono molto simili a quelle che si trovano di solito in un linguaggio di tipo imperativo.

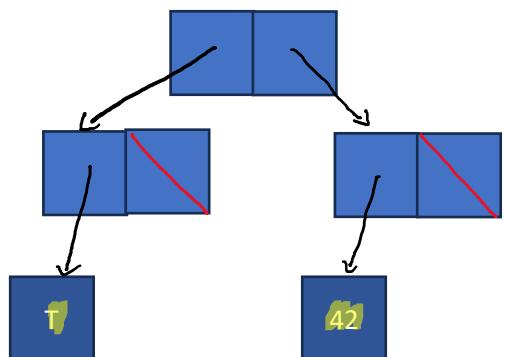
## Celle cons

L'operatore **cons** in LISP viene utilizzato per costruire nuove liste aggiungendo un elemento all'inizio di una lista esistente. Ha soltanto due argomenti:  
`(cons element list)`

Una cella cons è la struttura base che rappresenta una coppia di elementi, comunemente utilizzata per costruire liste. Ogni cella ha due parti: **car** (che contiene il primo elemento) e **cdr** (che contiene il secondo elemento/il resto/un'altra cella cons nel caso di liste).

Nel caso di liste, appunto, il cdr è in realtà un puntatore alla cella subito dopo.

`(cons (cons T NIL) (cons 42 NIL))`



## Liste

Le liste in LISP si creano usando il predicato **list**, e poi elencando tutti gli elementi che vogliamo inserire all'interno della lista:

`(list <elements>)`

Usiamo, come prima, la notazione **car** e **cdr** per estrarre i primi elementi di una lista. Successivamente, possiamo anche usare **first**, **second**, **third...** fino a **tenth** per poter accedere ai primi 10 elementi di una lista, e usare **rest** per accedere a tutto il resto.

Una lista non deve essere mischiata con la notazione `(<elements>)`: questa è una rappresentazione tipografica di una struttura dati in memoria senza un operatore in una posizione canonica, non una lista.

La differenza tra una lista e una cella cons è che le celle cons sono coppie di due oggetti, mentre le liste sono sequenze di oggetti.

Le liste sono implementate a partire dalle celle cons, seguendo la convenzione che una lista vuota è rappresentata dal simbolo nil e una lista non vuota è una cella cons il cui car è il primo elemento della lista e il cui cdr è il resto della lista.

`(cons 1 2) => (1. 2) (list 1 2) => (1 2)  
(cons 1 (cons 2 nil)) => (1 2)`

## Strutture dati

Per creare nuovi tipi di dati con campi specifici, usiamo il predicato **defstruct**, che è comparabile a una struct in C. Creare un tipo con defstruct ci provvede ad avere anche certi predicatori built-in utili per lavorare su questa tipologia di struttura dati.

`(defstruct name`

<code>Property1</code>	Nuovo oggetto da struct: <code>make-structname name :property value</code>
<code>Property2</code>	Verificare la struct di un oggetto: <code>structname-name</code>
<code>Property...</code>	Accedere a una delle proprietà: <code>structname-name property</code>

## Atomi

Gli atomi in LISP possono essere per esempio numeri, simboli e stringhe. Si può associare un simbolo a un valore utilizzando il predicato **setq**:

`(setq variable value)`

Possiamo usare predicati quali *numberp*, *symbolp* e *stringp* per verificare il tipo di un atomo.

### Quote

L'operatore quote, o abbreviato molte volte con un apostrofo, viene usato per prevenire la valutazione di un'espressione, **trattandola come dati letterali**. È particolarmente utile quando si vogliono rappresentare liste, simboli o interi come dati, senza valutarli.

$$(\text{quote} \text{ } <\text{expression}>) = (\text{' } <\text{expression}>)$$

Numeri e stringhe sono **autovalutanti**, ovvero non richiedono il predicato *quote* per essere trattati come dati letterali.

### Valutazione

Per valutare se due oggetti sono uguali, abbiamo tre predicati principali in LISP.

### Eval

Questa funzione è utilizzata per valutare un'espressione LISP. Prende un argomento, che è l'espressione da valutare, e restituisce il risultato della valutazione. Alcuni casi speciali:

#### 1. Atomo

- a. **Se è un numero:** restituisce valore
- b. **Se è una stringa:** restituisce la stringa
- c. **Se è un simbolo:** estrae il valore e restituisce
  - i. **Se il simbolo non è legato a un valore:** segnala un errore

#### 2. Cons-cell

- a. **Se il primo elemento è un operatore speciale:** valuta la lista in modo speciale
- b. **Se il primo elemento è un simbolo che denota una funzione nell'ambiente corrente:** applica la funzione nella lista
- c. **Se il primo elemento è un'espressione lambda, applica l'espressione alla lista che raccoglie i valori ottenuti valutando le espressioni.**

### Eql

È un predicato di uguaglianza rigida. **Restituisce T se due oggetti sono identici in termini di contenuto e tipo**, altrimenti restituisce NIL.

### Equal

È un predicato di uguaglianza più flessibile; restituisce T se **due oggetti sono strutturalmente equivalenti** (i loro contenuti sono uguali), anche se non sono identici in termini di tipo, o puntano alla stessa locazione di memoria.

### Map

LISP possiede due funzioni che consentono di **applicare una funzione a ciascun elemento di una lista**, creando una nuova lista contenente i risultati delle applicazioni.

1. **Map:** accetta una funzione e uno o più elementi come argomenti. Restituisce una nuova sequenza risultante dall'applicazione della funzione ai corrispondenti elementi degli elenchi, e può operare su più di una sequenza alla volta
  - a. `(map 'list #'(1 2 3) '(4 5 6)) -> restituisce (5 7 9)`
2. **Mapcar:** accetta una funzione e una lista come argomenti. Restituisce una nuova lista contenente i risultati dell'applicazione della funzione a ciascun elemento della lista di input.
  - a. `(mapcar #'square '(1 2 3 4)) -> restituisce (1 4 9 16)`

Let

Il predicato **let** permette di dichiarare e di inizializzare variabili locali all'interno di un blocco di codice. Limita, quindi, la visibilità delle variabili al solo blocco in cui sono definite.

## Trasparenza referenziale

Il principio fondamentale della trasparenza referenziale implica che **il significato di un intero può essere dedotto dal significato delle sue parti costituenti**.

Consideriamo l'espressione matematica  $f(x) + g(x)$ ; è possibile sostituire la funzione  $f$  con un'altra funzione  $h$ , a patto che entrambe generino gli stessi valori. Questa caratteristica differenzia i linguaggi funzionali puri dai linguaggi imperativi tradizionali, dove non è sempre garantita la sostituibilità delle espressioni, e dove possono sorgere ambiguità come:

$$f(x) + g(x) \neq g(x) + f(x)$$

$$f(x) + f(x) \neq 2*f(x)$$

## Julia

Julia è essenzialmente un linguaggio funzionale, basato su espressioni, come LISP. La sintassi è **free form**, con gli invii come spazi significanti. Si opera attraverso un **interprete a linea di comando**.

## Array

Una delle principali strutture dati sono gli array multidimensionali, che permettono di memorizzare e manipolare collezioni di valori indicizzati.

Possono avere una o più dimensioni, a seconda del numero di indici necessari per accedere ai loro dimensionali.

Un array unidimensionale è anche chiamato **vettore**, ed è equivalente ad una matrice colonna; un array bidimensionale è chiamato anche **matrice**, ed è una tabella di valori disposti in righe e colonne.

Gli array hanno un'interfaccia ricca e ben progettata, che permette di personalizzare il comportamento degli array in base al tipo degli elementi e della dimensionalità.

### Come creare un array

Esistono vari modi per creare array:

- **Sintassi delle parentesi quadre**

```
v = [1, 2, 3] -> 3-element Vector{Int64}:
w = [1 2 3] -> 1x3 Matrix{Int64}:
m = [[1 2]; [3 4]] -> 2x2 Matrix{Int64}:
```

- **Costruttore array** – elementi degli array inizializzati a valori casuali

```
x = Array{Float64,1} (undef, 5) -> vettore di 5 reali a 64 bit
y = Array{String,2} (undef, 2, 3) -> matrice 2x3 di stringhe
```

- **Funzioni zeros, ones, trues, falses** si creano array pre-riempiti con valori specifici

```
z = zeros(Int64, 3) -> 3-element Vector{Int64}:
t = trues(2, 2) -> 2x2 BitMatrix:
```

## Tipi composti

I tipi composti sono definiti dall'utente a partire da altri tipi esistenti. Per introdurre un tipo composto, si usa la parola chiave **struct**, che crea anche una funzione per costruire i valori di quel tipo.

Ha dei campi, che sono delle variabili che memorizzano i valori delle proprietà di quel tipo. Ogni campo ha un nome e un tipo, che si possono specificare dopo `::` nella definizione del tipo composto. Per esempio:

```
struct Person
    name :: String
    age
end
```

```
struct structName
    propertyName :: type
end
```

Per poi creare un valore di tipo Person, si usa la **funzione Person**, che prende come argomenti i valori dei campi in ordine.

```
p = Person("Ciao", 42)
```

```
variable =
structName(<initialisation>)
```

Per accedere ai valori dei campi di una persona, si usa la **notazione del punto**, seguita dal nome del campo. Per sapere il tipo di un valore, si usa la funzione incorporata **typeof**, che restituisce il nome del tipo.

I tipi composti possono essere usati negli array, come un qualsiasi altro tipo. Per esempio, si può creare un array di persone, con le parentesi quadrate.

```
ps = [Person("Ciao", 42), Person("Ancora", 42)]
2-element Vector{Person}:
```

Per ottenere tutti i valori dei campi di tutti gli elementi di un array, si possono usare le funzioni **map** o **filter**, oppure le *compreensioni di liste*, che sono delle espressioni che generano degli array a partire da altri array.

```
names = map(p -> p.name, ps) # ->: lambda
[p.name for p in ps]
```

## Strutture di controllo

Oltre alle strutture classiche come *if/elseif/else, while, for, try/catch, error, throw*, Julia ha anche delle strutture di controllo più avanzate, che si basano sull'uso di operatori speciali in combinazione con le funzioni.

Tra le strutture di controllo avanzate abbiamo:

### *Do-syntax*

È una sintassi speciale che permette di passare una funzione anonima come argomento a un'altra funzione, senza doverla definire esplicitamente. È utile per **estendere il linguaggio**, creando delle funzioni che possono eseguire delle operazioni prima o dopo aver chiamato la funzione anonima. Semplifica l'uso di valori temporanei.

Es.

```
open("outfile", "w") do ofd
    write(ofd, data)
end
```

### *Composizione e piping*

Sono due modi complementari per **concatenare più funzioni**, passando il risultato di una funzione come argomento alla successiva. La composizione è l'operazione usuale di applicare una funzione al risultato di un'altra funzione, che in Julia si può scrivere con `∘`.

Il piping è un concetto che permette di scrivere una *sequenza di funzioni in modo più leggibile* con l'operatore `|>`.

Es.

```
(sqrt ∘ +)(3, 6) = sqrt(+ (3, 6))
1:10 |> sum |> sqrt = sqrt(sum(1:10)); 1:10 is a range
```

### Broadcasting

È un meccanismo che permette di **applicare una funzione a tutti gli elementi di un array**, o di più array, in modo automatico e ottimizzato. È equivalente alla vettorizzazione, che è una tecnica per scrivere codice più semplice e veloce, evitando i cicli espliciti.

Questo si può attivare aggiungendo un **punto** dopo il nome della funzione, o dopo l'operatore. Il broadcasting si occupa di allineare le dimensioni degli array, e di estendere gli array di dimensioni inferiori.

`xs = [1.0, 2.0, 3.0]`

`sin.(xs) = broadcast(sin, xs) = applica sin a tutto xs`

### Funzioni generiche e metodi

Una funzione è una regola che associa un valore, o più valori a una tupla di argomenti. La stessa funzione può essere applicata a tuple di argomenti diversi, e restituire valori appropriati. La funzione è quindi **generica**, cioè non dipende da un tipo specifico di argomenti.

La scelta di quale funzione chiamare in base al tipo di argomenti viene fatta al momento dell'esecuzione – si chiama **dispatch dinamico**.

I diversi codici che vengono invocati, a seconda degli argomenti, si chiamano **metodi**; è una definizione di una funzione per un certo tipo o una certa combinazione di argomenti.

Es.

```
foobar(x :: Int, y :: Int) = x * y      #definizione  
foobar(3, 4) = 12      #chiamata
```

### Packages

Julia ha una versione di un gestore di pacchetti è sofisticata e ben integrata. Gli elementi del gestore di pacchetti sono i moduli – per isolare un'implementazione di una libreria e renderla accessibile attraverso una API ben definita – e i registri.

I pacchetti sono essenzialmente moduli che vivono in una cartella, e hanno dei metadati associati a loro, registrati nei file *Project.toml* e *Manifest.toml*. I registri, inclusi quello principale e standard, sono essenzialmente scambi di metadati dei pacchetti.

Es.

```
#definizione del modulo  
module Treaps  
  
#dichiarazione da cosa dipende questo modulo  
using Random  
  
#lista export dei nomi che saranno visibili a pacchetti che usano treaps  
export Treap, is_treap, insert, insert!, search, remove  
  
end
```

### Haskell

Haskell è un linguaggio di programmazione funzionale, che si basa sul concetto di funzione matematica; anch'esso non usa variabili, assegnazioni, cicli o istruzioni.

È un **linguaggio puro**, che significa che non ha effetti collaterali, come la modifica dello stato del programma, o l'interazione con l'ambiente esterno.

Altra caratteristica distintiva è la sua valutazione **pigra**, ovvero che le espressioni sono valutate solo quando sono necessarie, e non sono appena definite. Questo permette di scrivere codice più astratto e generico, e di manipolare oggetti infiniti, come liste o alberi. La valutazione pigra è derivata da alcune operazioni fondamentali del lambda-calcolo.

È **fortemente tipato**, che significa che ogni espressione ha un tipo ben definito, e il compilatore controlla che i tipi siano coerenti. Haskell usa anche l'inferenza di tipo, che significa che il programmatore non deve dichiarare esplicitamente i tipi delle espressioni.

È **compilato**, cioè il codice sergente viene tradotto in codice eseguibile da un compilatore, che ne controlla anche la correttezza sintattica e semantica. Il compilatore più usato è il **Glasgow Haskell Compiler (GHC)**, che offre anche un interprete interattivo GHCi.

Ecco degli esempi:

```
foobar x y = x + y + 42           mycond (2 < 3) (foobar 123) (foobar 42)
snafu z = foobar "Compilation Error" z      123
                                                123
```

Altri esempi, con anche l'esecuzione di esse:

```
mycond test a b = if test then a else b      mycond (2 > 3) (foobar 123) (foobar 42)
                                                42
                                                42

foobar z = do
    print z
    return z
```

## Scrivere codice

Per scrivere un programma in Haskell, si deve definire un insieme di tipi e di funzioni in un file, e indicare una funzione speciale chiamata *main* come punto di ingresso del programma.

La funzione *main* è quella che viene eseguita quando si avvia il programma. Per esempio:

```
- Programma Hello World
module Main where
    main = putStrLn "Hello World!"
```

Per usarlo nel compilatore, si può soltanto chiamare **main** nel terminale, e funziona. Si può usare anche GHCi come una calcolatrice per interpretare espressioni matematiche.

## Liste

Haskell ha anche il concetto di lista, che è una sequenza di elementi dello stesso tipo, separati da virgolette, e racchiusi tra parentesi quadre. Le liste possono essere costruite usando l'operatore :, che aggiunge un elemento in testa a una lista, e la lista vuota [].

```
[1, 2, 3]          42 : [123, -2] → [42, 123, -2] "foo" : [] → ["foo"]
```

Le liste in Haskell **devono avere elementi dello stesso tipo**, altrimenti il compilatore segnala un errore di tipo.

Haskell offre anche molte operazioni sulle liste, che permettono di manipolarle in vari modi. Per esempio, si può usare la funzione **zip**, che prende due liste e restituisce una lista di coppie, formate dagli elementi corrispondenti delle due liste.

Si può usare anche la funzione **map**, che prende una funzione e una lista, e restituisce una lista ottenuta applicando la funzione a ogni lista.

```
*Main > zip [1, 2, 3] ["foo", "bar", "foobar"]
[(1, "foo"), (2, "bar"), (3, "foobar")]

*Main> map (\ x -> x+1) [1, 2, -1, -2]
[2, 3, 0, -1] - \ x -> x + 1 è un'espressione lambda
```

Esiste anche il concetto di **tupla**, che è una collezione di elementi di tipi diversi, separati da virgolette, e racchiusi tra parentesi tonde. Le tuple hanno una dimensione fissa, e possono essere usate per rappresentare elementi di un prodotto cartesiano.

```
*Main> (1, "foo", bar)
(1, "foo", bar)
```

Possiamo anche per esempio definire una funzione **appendi**, per unire due funzioni assieme, e può essere scritta usando la parola chiave **let**, che introduce una definizione locale in un'espressione.

```
appendi [] ys = ys
appendi (x : xs) ys = let zs = appendi xs ys in x : zs
```

## Funzioni

Per definire una funzione in Haskell, si usa il simbolo `=` e si scrive un'equazione, che lega il nome della funzione e i suoi parametri al corpo della funzione, che è un'espressione. Lo stile delle equazioni si basa sul **pattern matching**, cioè sulla possibilità di distinguere diversi casi in base alla forma degli argomenti. Per esempio, questo calcola il fattoriale di un numero naturale, con due funzioni:

```
fact 0 = 1 - se l'argomento è zero, il risultato è uno
fact n | n > 0 = n * fact (n - 1)
```

L'operatore `|` è una **guardia**, che è una condizione booleana che deve essere vera per applicare l'equazione.

## Tipi

Ogni espressione ha un tipo, che indica il genere di valore che rappresenta; può essere dedotto automaticamente dal tipo, usando l'inferenza di tipo, oppure può essere esplicitamente dichiarato, usando le **annotazioni di tipo**. Per conoscere il tipo di un'espressione si può usare il comando `:type`.

```
*Examples> :type 42
42 :: Num a => a
```

Anche se il compilatore può inferire il tipo di un'espressione, è buona pratica dichiarare il tipo di un oggetto in Haskell, perché rende il codice più chiaro e sicuro. Per dichiarare il tipo di un oggetto, si usa il simbolo `::` e si scrive il tipo dopo il nome dell'oggetto. Per esempio:

```
fact :: Integer -> Integer
fact n = if n == 0 then 1
        else n * fact (n-1)
```

## *Tipi definiti*

Haskell permette di definire nuovi tipi, usando la parola chiave **data**. I tipi e i costruttori definiti dall'utente devono iniziare con una lettera maiuscola, per distinguerli dalle variabili e dalle funzioni. Per esempio:

```
data Month = January | February | March
```

Questo è un tipo algebrico, che significa che i suoi valori sono formati da una combinazione di costruttori; i costruttori sono dei valori che possono essere usati come pattern per il pattern matching, o come funzioni per costruire valori del tipo.

```
the_cruelest_month = April
```

I tipi definiti dall'utente possono avere anche dei parametri, che sono delle variabili di tipo che indicano il tipo degli elementi che compongono il valore. Per esempio:

```
data Planet = Planet { name :: String,  
                      mass :: Double,  
                      radius :: Double }  
deriving Show
```

Questo tipo è un tipo record, che significa che i suoi valori sono formati da un insieme di campi, che hanno un nome e un tipo. I campi possono essere usati come pattern per il pattern matching, o come funzioni per accedere ai valori dei campi.

### *Classi*

Ha anche il concetto di classe di tipo, che è un insieme di vincoli sui tipi, che indicano quali operazioni possono essere applicate ai valori di quel tipo. Per definirla, si usa **class** seguita dal nome della classe e da una variabile di tipo, che rappresenta il tipo generico che deve soddisfare i vincoli della classe.

Poi si elencano le funzioni che devono essere implementate per i tipi che appartengono alla classe, usando il simbolo **=>** per indicare la dipendenza dalla classe.

```
class Show a where  
    show :: a -> String
```

Per far sì che un tipo appartenga a una classe di tipo, si deve definire un'istanza della classe per quel tipo usando **instance**, seguita dalla classe, tipo, e implementazioni delle funzioni richieste dalla classe, e la keyword **deriving** in fondo.

### Liste infinite

Per definire una lista infinita in Haskell, si può usare una funzione ricorsiva, che genera gli elementi della lista uno dopo l'altro usando **:**, che aggiunge un elemento in testa a una lista, e la lista vuota **[]**.

```
integers = [0 .. ]
```

Usa il simbolo **..** per indicare una sequenza infinita di numeri, senza specificare il limite superiore; indicando un numero dopo, si genera una lista specificando il limite superiore. Qualche funzione importante sono **let** (definizione locale in un'espressione), **(!!)** (restituisce l'elemento di una lista in una certa posizione  $- x !! n$ ), **take** (restituisce i primi elementi di una lista), **drop** (restituisce gli elementi di una lista dopo averne scartati alcuni).

## Esercizi laboratorio Prolog

### Treedict

Treedict è una libreria che implementa l'interfaccia dizionario.

Abbiamo:

```
_istreedict(treedict(_Nome, Root)) :- is_node(Root).  
- Questo predicato verifica se un oggetto treedict è valido. Richiede che Root sia un nodo valido  
dell'albero, ed è verificato da is_node.  
is_node(nil).  
- Definisce che nil è un nodo valido. È quindi un predicato di base che stabilisce che una foglia  
dell'albero è rappresentata da nil.  
is_node(node(_Key, _Value, _Left, _Right)).  
- Definisce che qualsiasi oggetto con una struttura node contenente un _Key, _Value, _Left e  
_Right è un nodo valido nell'albero.
```

```
node_is_leaf(node(_, _, nil, nil)).
```

- Verifica se un nodo node è una foglia dell'albero. Richiede che il nodo abbia un \_Left e un \_Right impostati su nil. Se queste sono soddisfatte, allora è una foglia.

```
treedict_insert(K, V, treedict(N, Root), treedict(N, NewRoot)) :- node_insert(K, V, Root, NewRoot).
```

- Predicato principale per l'inserimento di una coppia chiave-valore in un treedict. Richiede una struttura treedict con una radice Root. Il risultato è un nuovo treedict con la radice aggiornata

```
node_insert(K, V, nil, node(K, V, nil, nil)).
```

- Se l'albero è vuoto, verrà creato un nuovo nodo con la chiave K e il valore V
- Simboleggiato da nil, la quale è la radice nulla.

```
node_insert(K, V, node(NK, NV, NL, NR), node(NK, NV, NL, NewNR)) :- K @> NK,  
node_insert(K, V, NR, NewNR).
```

- Se K è maggiore di NK (ovvero la chiave nel nodo corrente), il predicato cerca di inserire la coppia chiave-valore nel sotto albero destro. Se K è maggiore, la chiave è inserita a destra.

```
node_insert(K, V, node(K, V, nil, nil), nil).
```

- Se K è uguale alla chiave nel nodo corrente, viene restituito nil, poiché la coppia chiave-valore esiste già nell'albero
- Poiché le chiavi sono uguali, il predicato restituisce nil perché non è necessario inserire nulla, poiché la chiave esiste già.

```
node_insert(K, V, node(NK, NV, NL, NR), node(NK, NV, NewNL, NR)) :- K @< NK,  
node_insert(K, V, NewNL, NL).
```

- Se K è minore di NK, il predicato cerca di inserire la coppia chiave-valore nel sotto albero sinistro. Se K è minore, la chiave sarà inserita a sinistra

```
node_insert(K, V, NL, NewNL), node_insert(K, V, node(NK, NV, NL, NR), node(K, V, NL, NR)) :- K =@= NK.
```

- Verifica se K è uguale a NK. In questo caso, entrambi le chiavi sono uguali, quindi il predicato chiama node\_insert sia a sinistra (NewNL) che a destra (NR) del nodo corrente.
  - o NL, NewNL: tenta di inserire la coppia chiave-valore nella sotto struttura sinistra del nodo corrente
  - o node\_insert(K, V, node(NK, NV, NL, NR), node(K, V, NL, NR)):  
rappresenta il tentativo di inserire la coppia chiave-valore come nuovo nodo figlio nel nodo corrente, sostituendo il nodo corrente. Questo se il tentativo di inserire il valore a sinistra non riesce.
- Quindi, crea un nuovo nodo con la chiave duplicata, ma questa volta con il valore V, e lo inserisce come figlio nel nodo corrente. Aggiorna il nodo corrente con un nuovo V
  - o Questo nuovo nodo ha la stessa chiave K e il nuovo valore V, mentre i figli sinistri e destri rimangono gli stessi.

```
treedict_search(K, treedict(_, Root), V) :- node_search(K, Root, node(K, V, _, _)).
```

- Definisce la ricerca di una chiave all'interno di un dizionario rappresentato come un albero binario
- Riceve tre argomenti: la chiave da cercare (K), il dizionario (treedict(\_, Root)), restituisce il valore associato alla chiave (V).

```
node_search(K, node(NK, V, L, R), node(NK, V, L, R)) :- K =@= NK, !.
```

- Se la chiave cercata (K) è uguale alla chiave nel nodo corrente (NK). La ricerca ha successo, e quindi l'intero nodo viene restituito come risultato.

```
node_search(K, node(NK, _, _, R), Node) :- K @> NK, !, node_search(K, R, Node).
```

- La chiave da cercare (K) è maggiore della chiave nel nodo corrente (NK).
- La ricerca continua nel sottoalbero destro, se vero. Il cut viene usato per evitare backtracking una volta che abbiamo trovato la corrispondenza desiderata.
  - o Quindi, se K @> NK vale, allora non deve più cercare nel nodo destro.
  - o Se un nodo ha una chiave più piccola di quella cercata e abbiamo deciso di esplorare il sottoalbero destro, non dobbiamo considerare ulteriori nodi a sinistra.

```
node_search(K, node(NK, _, L, _), Node) :- K @< NK, !, node_search(K, L, Node).
```

- La chiave cercata (K) è inferiore alla chiave nel nodo corrente (NK).

- Se K è minore di NK, il predicato K @< NK è vero, e la ricerca procede nel sottoalbero sinistro. Il cut viene utilizzato per evitare il backtracking.
  - o Se un nodo ha una chiave più grande di quella cercata e abbiamo deciso di esplorare il sottoalbero sinistro, non dobbiamo considerare ulteriori nodi a destra.

## Compress

Questo predicato è una definizione per la compressione di liste consecutive di elementi duplicati.

- `compress([], [])` . : base del caso, afferma che la compressione di una lista vuota è una lista vuota.
- `compress([X], [X]) :- !.` . : la compressione di una lista con un solo elemento è la stessa lista.
- `compress([X,X|Xs], Zs) :- compress([X|Xs], Zs), !.` . : se trovi due elementi consecutivi uguali, comprimi la lista eliminando uno di essi e continua a comprimere il resto della lista.
- `compress([X,Y|Xs], [X|Zs]) :- X \= Y, compress([Y|Xs], Zs).` . : se trovi due elementi consecutivi diversi, mantieni il primo nell' output e continua a comprimere il resto della lista.

## CountElement

Conta il numero di occorrenze in un elemento specifico A in una lista L1 e restituire il risultato in C.

- `countElement(L1, A, C) :- countd(L1, A, C)` . : forma di interfaccia, richiama il predicato interno per effettuare il conteggio effettivo.
- `countd([], _, 0)` . : la conta di un elemento in una lista vuota è zero.
- `countd([A|T], A, C) :- countd(T, A, C1), C is C1+1.` . : afferma che se l'elemento corrente è uguale a A, conta ricorsivamente il resto della lista T e incrementa il risultato C1 di 1.
- `countd([X|T], A, C) :- X \= A, countd(T, A, C).` . : se l'elemento corrente X è diverso da A, ignora l'elemento corrente e conta il resto della lista T senza incrementare il risultato.

## CountElement per liste

Conta gli elementi, ma tiene conto anche delle occorrenze di A all'interno di sotto liste di L1.

- `countElement2(L1, A, C) :- countd(L1, A, C)` . : interfaccia, richiama il predicato interno per effettuare il conteggio effettivo.
- `countd([], _, 0)` . : la conta di un elemento in una lista vuota è zero.
- `countd([A|T], A, C) :- countd(T, A, C1), C is C1+1.` . : se l'elemento corrente è uguale ad A, conta ricorsivamente il resto della lista T e incrementa il risultato C1 di 1.
- `countd([X|T], A, C) :- \+ is_list(X), X \= A, countd(T, A, C).` . : se l'elemento corrente X non è una lista e non è uguale ad A, ignora l'elemento corrente e conta il resto della lista T senza incrementare il risultato.
- `countd([X|T], A, C) :- is_list(X), countd(X, A, C1), countd(T, A, C2), C is C1+C2.` . : se l'elemento corrente X è una lista, conta ricorsivamente le occorrenze di A nella sottolista X e conta anche il resto della lista T, quindi somma i risultati C1 e C2.

## Subarray

Verifica se la seconda lista è un sottoinsieme della prima lista.

- `subarray(_, []).` . : qualsiasi lista è un sottoinsieme di una lista vuota.
- `subarray([A|T], [A|T2]) :- subarray2(T, T2, A).` . : se l'elemento corrente A è lo stesso nella testa di entrambe le liste, allora richiama subarray2 per gestire il resto delle liste.
- `subarray([A|T], [X|T2]) :- subarray(T, [X|T2]), A \= X.` . : se gli elementi correnti, A e X, sono diversi, allora ignora l'elemento corrente nella prima lista e procedi con la verifica.

- Il predicato subarray2/3 è utilizzato per gestire la parte in cui la testa della prima lista è uguale all'elemento corrente B.
  - o `subarray2( _, [ ], _ )` . : qualsiasi lista è un sottoinsieme di una lista vuota.
  - o `subarray2([A|T], [A|T2], B) :- subarray(T, T2), A \= B.` : se l'elemento corrente A è diverso da B, allora ignora l'elemento corrente nella prima lista e procedi con la verifica.
  - o `subarray2([A|T], [A|T2], B) :- subarray(T, T2), A = B.` : se l'elemento corrente A è uguale a B, allora ignora l'elemento corrente nella prima lista e procedi con la verifica.
  - o `subarray2([A|T], [X|T2], B) :- subarray2(T, [B,X|T2], B), X \= A.` : se l'elemento corrente A è diverso da B, e l'elemento corrente nella seconda lista X è diverso da quello nella prima lista, allora richiama subarray2 per gestire il resto delle liste.

## Subsequence

Verifica se la seconda lista è una sottosequenza della prima lista. Una sottosequenza è una sequenza di elementi che mantiene l'ordine relativo, ma non è necessario che siano consecutivi.

- `subsequence( _, [ ] )` . : qualsiasi lista è una sottosequenza di una lista vuota.
- `subsequence([A|T], [A|T2]) :- subsequence(T, T2).` : se l'elemento corrente A è lo stesso nella testa di entrambe le liste, allora ignora l'elemento corrente nella prima lista, e procedi con la verifica del resto delle liste.
- `subsequence([A|T], [X|T2]) :- subsequence(T, [X|T2]), A \= X.` : se gli elementi correnti A e X sono diversi, allora ignora l'elemento corrente nella prima lista e procedi con la verifica del resto delle liste.

## Flatten

“Appiattisce” una lista, ovvero trasforma una lista composta da liste nidificate in una lista piatta che contiene tutti gli elementi delle liste interne, mantenendo l'ordine originale.

- `flatten([ ], [ ])` . : la lista vuota appiattita è ancora una lista vuota.
- `flatten([X|T], Flat) :- flatten(H, FlatH), flatten(T, FlatT), append(FlatH, FlatT, Flat).` : per una lista con testa X e coda T, l'appiattimento è ottenuto appiattendo sia la testa H che la coda T, unendo i risultati ottenuti FlatH e FlatT tramite append per ottenere la lista appiattita Flat.
- `flatten(H, [H]) :- \+ is_list(H).` : se l'elemento H non è una lista, allora la lista appiattita è composta solo da questo elemento stesso.

## Double

Raddoppia gli elementi di una lista.

- `double([ ], [ ])` . : una lista vuota rimane vuota quando raddoppiata.
- `double([X|X1], [X,X|X2]) :- double(X1, X2).` : per una lista X e coda X1, il risultato di raddoppiare questa lista è ottenuto raddoppiando la testa e ricorsivamente raddoppiando la coda.
- `double([_|X1], [_,_|X2]) :- !, X1 \= X2.` : gestisce il caso in cui il primo elemento della lista (\_|X1) è diverso dal secondo elemento della lista (\_|\_X2).

## Pari

Crea una nuova lista che contiene gli elementi di indice pari di una lista di input.

- `pari([ ], [ ])` . : una lista vuota ha una rappresentazione vuota con il predicato pari.
- `pari([_,Y|Xs], [Y|Ys]) :- pari(Xs, Ys).` : per una lista con almeno due elementi (\_|Y|Xs), la nuova lista Ys contiene il secondo elemento Y della lista di input, seguito dalla rappresentazione di indice pari del resto della lista di input Xs.

## Sostituisce

Sostituisce tutte le occorrenze di un elemento specifico X con un altro elemento Y in una lista di input.

- `sostituisce(_, _, [], [])` : la sostituzione di elementi in una lista vuota restituisce una lista vuota.
- `sostituisce(X, Y, [X|Xs], [Y|Out]) :- sostituisce(X, Y, Xs, Out)` : se l'elemento corrente della lista è uguale a X, allora lo sostituisce con Y nella nuova lista e procedi a ricorsivamente sostituire il resto della lista.
- `sostituisce(X, Y, [Z|Xs], [Z|Out]) :- sostituisce(X, Y, Xs, Out)` : se l'elemento corrente della lista Z non è uguale a X, allora mantieni lo stesso elemento nella nuova lista e procedi a ricorsivamente sostituire il resto della lista Xs.

$$\begin{array}{c}
 \frac{F_1, F_2, \dots, F_k}{R} \quad \text{FBF} \\
 \text{formula generata da inserire in FBF}
 \\[10pt]
 \begin{array}{c|c|c|c}
 \text{modus ponens} & \text{modus tollens} & \text{clausola risolvente} & \text{Unit resolution} \\
 \frac{P \Rightarrow q, P}{q} & \frac{P \Rightarrow q, \neg q}{P} & \frac{P \vee \neg r, S \vee r}{P \vee S} & \frac{a, \neg a \vee b}{b} \\
 \hline
 \end{array}
 \\[10pt]
 \text{Clausole di Horn} \quad \text{Tautologia} \\
 \left[ \frac{\exists \vee \neg b \vee c \vee \dots}{b \wedge c \wedge \dots \Rightarrow a} \right], \quad \frac{A \vee \neg A}{\text{vero}}
 \end{array}$$

## Esercizi laboratorio C

### Treedict

Treedict è una libreria che implementa l'interfaccia dizionario all'interno di treedict.h, mentre in treedict.c impostiamo l'impostazione dei metodi.

#### Treedict.h

```
#ifndef _TREEDICT_H
#define _TREEDICT_H
```

Con queste direttive, definiamo il fatto che, se TREEDICT\_H è già stato definito all'interno del file, non è necessario ridefinirlo.

```
struct node; struct treedict;
```

Definiamo il fatto che devono esistere le strutture di un nodo e di un treedict.

```
typedef struct node* node_ref;
typedef struct treedict* treedict_ref;
```

Crea un alias node\_ref e treedict\_ref; quindi, ogni volta che useremo qualcosa con uno di questi nomi, in realtà creeremo dei puntatori a una struttura del tipo relativo.

```
extern treedict_ref treedict_new(char* id);
extern treedict_ref treedict_insert(treedict_ref, int, void*);
extern void* treedict_search(treedict_ref, int);
extern void* treedict_print(treedict_ref);
```

```
extern treedict_ref treedict_delete(treedict_ref, int);
```

Definiamo quali metodi esterni dovranno essere definiti; ovviamente, nel file .c ogni file dovrà essere definito così come è, ovvero rispettando tutti i parametri e tipo di ritorno che noi abbiamo definito nel file h.

```
extern node_ref node_insert(node_ref, int, void*);  
extern node_ref node_search(node_ref, int);  
extern void node_print(node_ref, int);
```

Definiamo quali metodi esterni dovranno essere definiti, stavolta per il nodo invece.

```
#endif
```

Finiamo la procedura define, definendo che ciò che sta all'interno di define e endif dovrà essere incluso come blocco.

Treedict.c

```
// librerie standard di C  
#include <stdio.h>  
#include <stdlib.h>
```

```
// libreria di Visual Studio per Win32 - rimuovi se non sei su Visual  
Studio
```

```
#include "stdafx.h"
```

```
// header file per treedict
```

```
#include "treedict.h"
```

Questo pezzo include tutti gli header necessari per il programma in C: include *stdio* (standard I/O), *stdlib* (standard library), *stdafx* (libreria per win32), e inoltre **treedict.h**; questo è per far includere l'intestazione dei nostri metodi.

```
// struttura dell'albero
```

```
struct treedict {
```

```
    char * id; // id dell'albero  
    node_ref root; // radice
```

```
};
```

```
// struttura del nodo
```

```
struct node {
```

```
    node_ref root; // radice  
    int key; // chiave  
    void * val; // valore  
    node_ref left; // nodo a sinistra  
    node_ref right; // nodo a destra
```

```
};
```

Grazie al costrutto **struct**, definiamo come vogliamo che le nostre strutture siano, e quindi effettivamente è come se declamassimo un nuovo tipo.

```
// definisco alcune delle funzioni dei nodi sopra in modo che possano  
essere usate dai metodi sottostanti
```

```

//crea un nuovo nodo
node_ref node_new(int key, void* val, node_ref left, node_ref right){

    //riferimento a un nuovo nodo
    node_ref new_nd = (node_ref) malloc(sizeof(struct node));

    //se il riferimento al nodo appena creato è null, allora non posso
    allocare memoria
    if (new_nd == NULL){
        fprintf(stderr, "treedict: node_new: cannot allocate memory.\n");
        exit(-2);
    }

    // "istanzio" le proprietà di struct node
    new_nd -> key = key;
    new_nd -> val = val;
    new_nd -> left = left;
    new_nd -> right = right;
    return new_nd;

}

```

Funzione per creare un nuovo nodo: se il nodo è nullo, vuol dire che non ho potuto allocare la memoria per crearlo; altrimenti, creane uno assegnando i valori dati in parametro al nuovo nodo.

```

//funzione per inserire un nodo
node_ref node_insert(node_ref n, int k, void * v){

    //se il nodo di riferimento è nullo, vuol dire che devo fare un nuovo
    albero praticamente
    if (n == NULL) {
        return node_new(k, v, NULL, NULL);
    }

    //se la key data è maggiore del nodo a cui si punta, allora si
    inserisce il nodo nel sottoalbero destro
    //altrimenti se la key data è minore del nodo a cui si punta, allora si
    inserisce il nodo nel sottoalbero sinistro
    //altrimenti ritorno un nuovo nodo che è identico a quello dato come
    riferimento
    if (n->key < k)
        return node_new(n->key, n->val, n->left, node_insert(n->right, k,
v));
    else if (n -> key > k)
        return node_new(n->key, n->val, node_insert(n->left, k, v), n-
>right);
    else if (n -> key == k)
        return node_new(n->key, n->val, n->left, n->right);

    //return "di sicurezza" perchè non dovrebbe mai arrivare qua
    return NULL;
}

```

Per inserire un nodo all'interno di un albero dato un riferimento a un nodo, usando le proprietà left e right. Se la chiave data è maggiore del nodo a cui si punta, allora lavoro sull'albero destro; se invece è più piccolo, lavoro sulla parte destra; altrimenti, ritorno un nuovo nodo che è identico a quello dato come riferimento. Probabilmente ritorno un nuovo nodo, così non ci possono essere problemi con la memoria.

```
//Crea un nuovo albero
treedict_ref treedict_new(char * name){

    //Allocà un nuovo spazio di memoria per un treedict
    treedict_ref new_td = (treedict_ref) malloc(sizeof(struct treedict));

    //Se non può creare lo spazio di memoria:
    if (new_td == NULL) {
        fprintf(stderr, "treedict: treedict_new: cannot allocate
memory.\n");
        exit(-1);
    }

    // "istanzia" i valori all'interno della struct
    new_td -> id = name;
    new_td -> root = NULL;

    return new_td;
}
```

Metodo per stanziare i nuovi parametri di un treedict: creo un albero senza una root, e assegno il nome dato come node del treedict.

```
//inserisce un nodo all'interno di un albero con un dato key e valore
treedict_ref treedict_insert(treedict_ref td, int key, void* val){

    //crea un nuovo albero "di appoggio"
    treedict_ref new_td = treedict_new(td -> id);

    //aggiungo all'inizio come root un nodo con node_insert
    //per verificare key etc., guarda "node_insert"
    new_td -> root = node_insert(td -> root, key, val);

    return new_td;
}
```

Per capire dove deve essere inserito il nodo, faccio analizzare l'intero albero dall'inizio al metodo node\_insert; in base alla chiave del nodo dato e in base alla chiave del nodo in cui ci siamo al momento, il nodo viene “automaticamente” aggiunto o a sinistra o a destra.

```
//cerca, data una chiave, un nodo all'interno dell'albero
void* treedict_search(treedict_ref td, int key){

    //riferimento al nodo da cercare - inizio dalla radice
    node_ref n_found = node_search(td->root, key);

    //se quello che ho trovato non è null e ha la stessa chiave, allora è
    //quello che ho trovato
    if (n_found != NULL)
```

```

        return n_found->val;
    else
        return NULL;
}

```

Faccio partire la ricerca del nodo direttamente dalla radice, e sposto il controllo di verificare la chiave etc. a node\_search. Se, alla fine, trovo un nodo, allora ritorno quello; altrimenti, vuol dire che non ho trovato niente, e ritorno false.

```

//stampa l'albero partendo dalla radice
void* treedict_print(treedict_ref td) {

    if (td == NULL) return NULL;

    fprintf(stdout, "Dict (Tree) %s.\n", td -> id);
    node_print(td->root, 0);
    return td -> id;
}

```

Stampo l'albero a partire dalla radice, aggiungendoci il fatto che è un treedict, e il nome dell'albero.

```
//distruggere un albero
```

```
treedict_ref treedict_delete(treedict_ref td, int key){

    //creazione di un nuovo albero appoggio
    treedict_ref new_td = treedict_new(td -> id);

    //se l'albero è vuoto, non c'è nulla da eliminare
    if (td == NULL || td->root == NULL){
        fprintf(stderr, "treedict: treedict_delete: tree is empty or
null.\n");
        return td;
    }

    //elimino il nodo con la chiave specificata
    new_td->root = node_delete(td->root, key);
}
```

```

    return new_td;
}
```

Se l'albero è nullo, o non possiede una radice (quindi vuol dire che è stato stanziatato un albero, ma è senza alcun nodo), allora è impossibile eliminare qualcosa. Altrimenti, creo un nuovo treedict, che è lo stesso tree di partenza (con lo stesso id) di quello iniziale, ma con ll'operazione di rimozione effettuata.

```
//metodo per cercare un nodo dato un root e una chiave
node_ref node_search(node_ref n, int key){
```

```

    //se la reference è null, allora ritorniamo null
    if (n == NULL) return NULL;
```

```

    //se la root ha la stessa key di quello che stiamo cercando: root è il
    //nodo che stiamo cercando
    if (n -> key == key) {
        return n;
        //altrimenti se la key data è minore rispetto alla key della root,
        //allora dobbiamo vedere il sottoalbero sinistro
    } else if (key < n -> key) {
```

```

        return node_search(n -> left, key);
    //altrimenti se la key data è maggiore rispetto alla key della
    root, allora dobbiamo vedere il sottoalbero destro
    } else if (key > n->key){
        return node_search(n -> right, key);
    }

    //return "di sicurezza)
    return NULL;
}

```

Metodo per la ricerca di un nodo: se la key del nodo da cercare è la key del nodo visto al momento, allora ritorno quello; altrimenti, se la key data è minore rispetto a quella visitata, allora andiamo nel sottoalbero sinistro; se la key data è maggiore rispetto a quella visitata, allora andiamo nel sottoalbero destro.

```

//stampa un nodo
void node_print(node_ref n, int level){
    if (n != NULL){
        printf("%*c", level, ' ');
        printf(" %d ==> '%s'\n", n->key, n->val);
        node_print(n->left, level+4);
        node_print(n->right, level+4);
    }
}

```

Dato un livello di stampa (es. Per partire dalla root, metto il livello zero), stampo ogni nodo. *Int level* serve inoltre per l'indentazione: più si va giù in un livello, più l'indentazione viene aumentata per mostrare che non sono sullo stesso livello.

```

//cancello un nodo
node_ref node_delete(node_ref n, int key) {

    //definisco che dovrò trovare il successore
    node_ref successor = NULL;

    if (n == NULL) { return NULL; } //se il nodo è null, non c'è nulla da
    eliminare
    else if (key < n->key){
        //se la chiave da cercare è più piccola, allora è nel sottoalbero
        sinistro
        n->left = node_delete(n->left, key);
    } else if (key > n->key){
        //se la chiave da cercare è più grande, allora è nel sottoalbero
        destro
        n->right = node_delete(n->right, key);
    } else {
        //sostituisco il nodo con il successore più piccolo nel
        sottoalbero destro/sinistro
        if (n->right == NULL) return n->left;
        else if (n->left == NULL) return n->right;

        //quando si elimina un nodo con due sottoalberi non vuoti,
        occorre trovare un successore
        successor = node_find_successor(n->right);
        n->key = successor->key;
    }
}

```

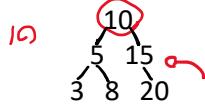
```

n->val = successor->val;
n->right = node_delete(n->right, successor->key);
}
return n;
}

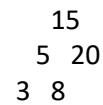
node_ref node_find_successor(node_ref n){
/*funzione ausiliaria per trovare il successore più piccolo di un nodo
se nodo ha sia parte destra che sinistra
garantisce di mantenere l'ordine*/
while (n->left != NULL) {
    n = n->left;
}
return n;
}

```

Funzione per cancellare un nodo. C'è bisogno di una funzione **find\_successor** è necessario per quando si vuole eliminare un nodo che ha due figli non vuoti. Cerchiamo, quindi, il nodo più piccolo nel sottoalbero sinistro, partendo però dal sottoalbero destro del nodo (quindi, andiamo a cercare il più piccolo tra quelli che sono più grandi di un nodo). Vediamo un esempio:

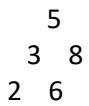


Per eliminare il 10, devo trovare un successore  
per mantenere l'ordine BST.



All'interno di **node\_delete**, dobbiamo **per forza** aggiornare i riferimenti a **n->left** e **n->right**; questo perché, in questo modo, aggiorniamo la struttura dell'albero con i collegamenti ai nuovi sottoalberi sinistri e destri, una volta eliminato un nodo.

L'ultimo else serve per **mantenere l'ordine BST** anch'esso: immaginiamo di voler eliminare sempre un nodo che ha un figlio sinistro e destro. Uno dei due figli viene spostato dove è più opportuno, mentre, grazie a questo metodo, noi possiamo mantenere l'ordine. Vediamo un esempio:



Se vogliamo eliminare il 3, il 2 verrà spostato in alto,  
mentre bisognerà spostare giustamente il 6

