2. Elementi fondamentali del linguaggio

Riscrittura di termini

⌘ *Mathematica* è un sistema basato sulla riscrittura di termini (term rewriting).

Data in input una espressione (che definiremo in modo più preciso nel seguito), l'operazione fondamentale eseguita dal Kernel è riconoscere quei termini che sa come sostituire con altri termini (possibilmente più semplici).

Vediamone un esempio:

```
a * a + D[a ^ 3, a]
```

$4\ a^2$

Questi sono i passi seguiti.
Nella espressione :

```
a * a + D[a ^ 3, a];
```

il Kernel riscrive a*a come a^2

```
a ^ 2 + D[a ^ 3, a];
```

Dopodiché riscrive D[a^3, a] come 3 a^2

```
a ^ 2 + 3 a ^ 2;
```

Infine riconosce che a^2+3 a^2 può essere riscritto come 4 a^2

```
4 a ^ 2;
```

Vediamolo con Trace

```
(*  //    Postfix  *)
a * a + D[a ^ 3, a] // Trace
```

$\{\{a\ a,\ a^2\},\ \{\partial_a a^3,\ 3\ a^2\},\ a^2 + 3\ a^2,\ 4\ a^2\}$

In altre parole, il Kernel mima il modo in cui una persona esegue della matematica (il Kernel lo fa in modo completamente algoritimico).

⌘ Le **espressioni** sono l'unico tipo di oggetto in *Mathematica*: esse vengono usate per rappresentare sia il codice che i dati.

Le espressioni hanno una struttura annidata: espressioni più grandi sono composte da espressioni più piccole, che a loro volta sono composte da espressioni via via più piccole, fino ad arrivare agli **atomi** (sotto-espressioni che non possono essere suddivise) del linguaggio.

Quando il Kernel esegue la riscrittura di termini, rimpiazza sempre una espressione con un'altra. Questa consistenza di rappresentazione e di operazione rappresenta la caratteristica più importante del linguaggio di programmazione in *Mathematica*.

2. Elementi fondamentali del linguaggio

- **Riscrittura di termini**

2.1 Espressioni: espressioni normali (2.1.1)

⌘ Ogni cosa in *Mathematica* è una espressione.

Esistono fondamentalmente **due** tipi di espressione: atomica e normale.
Gli **atomi** possono essere simboli, numeri o stringhe di caratteri (cfr. 2.1.2).

Le **espressioni normali** hanno la forma:

```
Head[part1, part2, ...]
```

in cui Head, part1, part2, ecc. sono ciascuna una espressione.

⌘ La espressione:

```
Sin[Log[2.5, 7]];
```

è una espressione normale:
- la sua Head è un atomo (il simbolo Sin);
- la sua **unica** parte è un'altra espressione normale (Log[2.5, 7]).

A sua volta:

```
Log[2.5, 7];
```

è una espressione normale:
- la sua Head è un atomo (il simbolo Log);
- la sue **due** parti sono il numero reale 2.5 ed il numero intero 7.

⌘ La sintassi delle espressioni è disegnata per essere simile al costrutto di *chiamata di funzione* in linguaggi quali C.

Risulta abbastanza immediato associare le Head simboliche (quali Sin o Log) a funzioni; faremo pertanto riferimento:
- alla Head di una espressione come ad una funzione;
- alle parti di una espressione come agli argomenti della chiamata alla funzione.

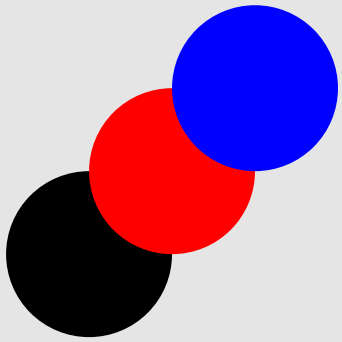
⌘ Non ogni espressione normale, tuttavia, può essere pensata come una chiamata di funzione.
Una espressione, infatti, può semplicemente rappresentare dati.
Ad esempio:

```
RGBColor[1, 0, 0]
```

è una direttiva grafica: essa dice al Kernel che una data primitiva grafica (cui RGBColor[1,0,0] è associata) deve essere resa in colore rosso.
Non vi è alcuna chiamata di funzione associata al simbolo RGBColor e l'espressione RGBColor[1,0,0] non può essere riscritta in alcun modo.

```
? RGBColor  (* Built-in Symbol *)
```

```
Graphics[{Disk[{-1, -1}],
   RGBColor[1, 0, 0], Disk[],
   RGBColor[0, 0, 1], Disk[{1, 1}]},
 ImageSize → Small]
```



⌘ Ogni espressione in *Mathematica* può essere costruita usando solo **tre** blocchi di costruzione sintattica: atomi, virgole, parentesi quadre [ ].

Riprendiamo l'esempio già visto, in cui usiamo forme speciali di input per le operazioni elementari somma e prodotto :

```
a * a + D[a ^ 3, a];
```

Possiamo ridare lo stesso input così (usando atomi, virgole e parentesi quadre):

```
Plus[ Power[a, 2], D[ Power[a, 3], a ]];
```

⌘Il parser di *Mathematica* converte input, quali a*a in Power[a, 2], a^3 in Power[a,3] e così via.

```
(* FullForm *)
a * a // FullForm
a^3 // FullForm
(* FullForm restituisce la forma interna di una espressione *)
(* NOTA :  a*a  è  Power[a,2] ,  non è  Times[a,a]  *)
```

```
Power[a, 2]
```

```
Power[a, 3]
```

```
(* Map *)
mylist = {x * y , a + a , a + b};
mylist // FullForm
(* Map  applica  FullForm  agli elementi  della  lista  mylist *)
Map[FullForm , mylist]
```

```
List[Times[x, y], Times[2, a], Plus[a, b]]
```

```
{Times[x, y], Times[2, a], Plus[a, b]}
```

⌘ Forme sintattiche quali * , ^ , + sono dette *forme speciali di input* (cfr. 2.3) e servono per snellire la scrittura del codice.

Espressioni aventi (come Head) Plus, Times, Power, ecc., hanno pure *forme speciali di output*.
È per questo che l'output viene stampato in notazione matematica standard:

```
{a * a, a^3}
```

Come detto, FullForm serve ad ottenere in output la forma interna di una espressione.

```
4 a^2 // FullForm
```

2. Elementi fondamentali del linguaggio

- **Riscrittura di termini**

2.1 Espressioni: espressioni normali (2.1.1)

2.1 Espressioni: atomi (2.1.2)

Un atomo, in *Mathematica,* è una espressione che non può essere suddivisa in espressioni più piccole.

⌘ Esistono fondamentalmente **tre** tipi di atomi: simbolo, numero, stringa di caratteri.

- **Simboli**

Un simbolo è una sequenza di lettere, cifre ed il carattere $ (tale sequenza **non** deve iniziare con una cifra).

Esempi di simboli:

```
{a,
 abc,
 a2,
 a2b,
 $a,
 a$}
```

```
{a, abc, a2, a2b, $a, a$}
```

⌘ I simboli **non** sono simili alle variabili di linguaggi di programmazione, come C.
Essi sono più potenti, dato che non è necessario che ad un simbolo sia stato assegnato alcun valore, al fine di poterlo usare in un calcolo.

Un simbolo *segnala* se stesso.
Un simbolo non è meramente un sostituto (proxy) per un dato.

```
(* Esempio di calcolo simbolico.
   Il risultato è matematicamente vero, per valori arbitrari *)
a + b - 2 a
```

```
-a + b
```

Tutti i **simboli definiti da sistema** iniziano con la maiuscola o con $

```
$MachinePrecision
$Version
```

```
15.9546
```

```
14.0.0 for Linux x86 (64-bit) (December 13, 2023)
```

```
$Version = 3
```

Set: Symbol $Version is Protected.

```
3
```

- **Numeri**
- **Stringhe di caratteri**

Una stringa di caratteri (o semplicemente una stringa) è una qualsiasi sequenza di caratteri, racchiusa tra una coppia di doppi apici.

```
"Hello world"
(* Questa è una stringa.
   N.B. Mathematica non stampa la coppia di apici, quando stampa la stringa *)
```

```
Hello world
```

Possiamo usare InputForm per vedere che l'output è effettivamente una stringa

```
"Hello world";
InputForm[%]
(* InputForm[expr] stampa una versione di
 expr adatta ad essere un input per Mathematica *)
```

```
"Hello world"
```

In una stringa, la sequenza di caratteri \" sta per il (singolo) doppio apice "
Di conseguenza, l'input seguente è pure una stringa:

```
"For example, \"Hello world\" is a string."
```

```
For example, "Hello world" is a string.
```

Ci sono molte Built-in per agire sulle stringhe di caratteri, quali quelle per determinare la loro lunghezza, concatenarle, farne uno shift da maiuscola a minuscola (e viceversa), individuare e rimpiazzare sottostringhe (cfr. EIWL 11 e/o 3.6 "Stringhe di caratteri") e cosi' via.

2. Elementi fondamentali del linguaggio

- **Riscrittura di termini**

2.1 Espressioni: espressioni normali (2.1.1)

2.1 Espressioni: atomi (2.1.2)

Un atomo, in *Mathematica,* è una espressione che non può essere suddivisa in espressioni più piccole.

⌘ Esistono fondamentalmente **tre** tipi di atomi: simbolo, numero, stringa di caratteri.

- **Simboli**
- **Numeri**

Esistono **quattro** tipi di numeri in *Mathematica*: interi, razionali, reali, complessi.

Integer: consiste di una sequenza di cifre decimali *dddddd*

Rational: ha la forma *intero1/intero2*

Real: ha la forma *ddd.ddd* ossia è un numero a virgola mobile

Complex: ha la forma *a+b I* in cui *a, b* possono essere di qualsiasi dei tre tipi precedenti.

```
(* Integer *)
1234567890
```

```
1234567890
```

```
(* Rational *)
2/3
```

$$\frac{2}{3}$$

```
(* Real *)
nr = 21 345.6789;
(* Di default , in  output vengono mostrate 6  cifre  significative *)
(* NumberForm può essere usato per specificare quante cifre mostrare in output *)
{nr,
  NumberForm[nr, DefaultPrintPrecision → 9],
  FullForm[nr],
  Precision[nr],
  ScientificForm[nr],
  ScientificForm[nr, 9]} // TableForm
(* Il back-tick nell'output di FullForm[nr] è un NumberMarks,
ossia un argomento opzionale che può essere usato per
 specificare quante cifre significative devono essere stampate *)
```

```
21 345.7
21345.6789
21345.6789`
MachinePrecision
2.13457 × 10^4
2.13456789 × 10^4
```

```
(* ScientificForm[nr]  suggerisce  come  riscrivere  nr  in  forma  esatta  nint *)
nint = 123 456 789  * 10^-4;
tt = {nint,
  123 456 789. 10^-4 ,
  123 456 789  10.^-4,
  123 456 789  10^-4.}
(* Il punto decimale  indica  al  Kernel che il numero è in MachinePrecision *)
Map[MachineNumberQ, tt]
```

$$\left\{\frac{123\,456\,789}{10\,000}, 12\,345.7, 12\,345.7, 12\,345.7\right\}$$

```
{False, True, True, True}
```

```
(* Rationalize *)
nr = 21345.6789;
Rationalize[nr]
Rationalize[nr, 10^-7]
(* Rationalize[x] restituisce p/q se Abs[p/q - x] < 1/(100 q)^2 *)
(* Se Rationalize[x] restituisce x,
significa che nessun numero razionale soddisfa la disequazione qui sopra *)
(* In questo caso, proviamo a specificare una tolleranza *)
```

```
21345.7
```

$$\frac{213\,456\,789}{10\,000}$$

```
(* N  numericizza *)
nint = 123456789 * 10^-4;
tt = {N[nint], N[nint, 20]}
Map[FullForm, tt]
(* N[nint,20] è un BigNumber, ossia un numero a precisione arbitraria:
  qui ha precisione 20 *)
Map[MachineNumberQ, tt]
```

```
{12345.7, 12345.678900000000000}
```

```
{12345.6789`, 12345.6789`20.}
```

```
{True, False}
```

```
(* Complex *)
(* Notare il terzo l'output di cc :
  MachinePrecision si propaga come un virus *)
cc = {2/3 + 4 I,
  2/3 + (45/10) I,
  2/3 + 4.5 I}
```

$$\left\{\frac{2}{3}+4\,i,\ \frac{2}{3}+\frac{9\,i}{2},\ 0.666667+4.5\,i\right\}$$

⌘ Ognuno dei tipi numerici può avere virtualmente un numero di cifre illimitato.

⌘ L'esempio che segue è in **aritmetica esatta**:
Mathematica suppone che l'input (intero) significhi che il calcolo deve essere fatto in modo esatto, per cui usa tante cifre quante sono necessarie per ottenere l'output esatto.

Paragoniamo gli esempi seguenti:

```
(* INTERO: Input esatto. Output esatto,
qui è un intero di 52 cifre *)
(*  β = base , η = esponente *)
{{β, η} = {5 , 73};
 out = β ^ η ,
 Map[Precision, {β, η, out}]}
```

```
{1 058 791 184 067 875 423 835 403 125 849 552 452 564 239 501 953 125, {∞, ∞, ∞}}
```

```
(* REALE:  Un input in precisione macchina → Output in precisione macchina *)
{{β, η} = {5. , 73};
 out2 = β ^ η,
 Map[Precision, {β, η, out2}]}
(* la Machine Precision  è  virale *)
{{β, η} = {5 , 73.};
 out3 = β ^ η,
 Map[Precision, {β, η, out3}]}
```

$$\{1.05879 \times 10^{51}, \{\text{MachinePrecision}, \infty, \text{MachinePrecision}\}\}$$

$$\{1.05879 \times 10^{51}, \{\infty, \text{MachinePrecision}, \text{MachinePrecision}\}\}$$

```
(* REALE (precisione arbitraria , Big Number):
 Qui, l' input a precisione più bassa ha (circa) 25 cifre decimali.
  Output ha (circa) 23 cifre decimali *)
η = 73;
(* β ha 24 zeri *)
β = 5.000000000000000000000000;
{out4 = β ^ η, Map[Precision, {β, out4}]}
```

$$\{1.05879118406787542383 54 \times 10^{51}, \{24.699, 22.8356\}\}$$

⌘ La presenza del punto decimale, nell'input, è interpretata come voler significare che l'input è approssi mato (non esatto) ed è noto solo fino al numero di cifre che sono state esplicitamente scritte (fornite in input, appunto).

⌘ *Mathematica*, di conseguenza, esegue il calcolo in precisione arbitraria, mantenendo traccia di quante cifre nella risposta (output) sono giustificate dal numero di cifre significative nell'input & nelle operazione che sono state svolte.

Nell'esempio appena visto sono state perse le 2 cifre meno significative (si è partiti dalla precisione di

circa 25 e si è arrivati ad una precisione di circa 23):

```
(* Precision fornisce il numero di
 cifre significative in un numero approssimato. *)
β = 5.0000000000000000000000;
η = 73;
out4 = β ^ η;
{Precision[β], Precision[out4]}
(* NB. Il risultato di Precision può non corrispondere
 esattamente al numero di cifre mostrato nella cella.
  Il motivo è che la precisione numerica viene
 calcolata in base binaria e poi convertita in decimale.  *)
```

```
{24.699, 22.8356}
```

⌘ Si può specificare la precisione di un numero approssimato, esplicitamente, usando la sintassi *numero`precision* oppure usando N[ ]

```
η = 73;
(* Input con 25 cifre decimali . Output, qui, ha (circa) 24 cifre decimali *)
β = 5`25 ;
{out5 = β ^ η, Map[Precision, {β, out5}]}

β = N[5, 25] ;
{out6 = β ^ η, Map[Precision, {β, out6}]}
```

```
{1.05879118406787542383540 × 10^51, {25., 23.1367}}
```


```
{1.05879118406787542383540 × 10^51, {25., 23.1367}}
```


⌘ I numeri approssimati, che vengono dati in input con un numero di cifre **non** superiore a quello messo a disposizione dall'hardware a virgola mobile del computer, vengono memorizzati in formato, nativo, in virgola mobile a doppia precisione.
Tutte le operazioni su tali numeri sono eseguite in hardware.
Tali numeri sono detti *a precisione di macchina* .

```
$MachinePrecision
```

```
15.9546
```

⌘ La variabile (read-only) di sistema $MachinePrecision specifica la precisione dei numeri a virgola mobile nativi, che può variare su differenti architetture.

⌘ La funzione MachineNumberQ[ ] può essere usata per determinare se un numero approssimato è un

numero macchina

```
MachineNumberQ[5.0000000000000000000000]
(* Questo input ha 24 zeri,
quindi ha troppe cifre per poter essere un numero macchina *)
```

```
False
```

```
MachineNumberQ[5.0]
(* Questo input ha, implicitamente,
un numero di cifre significative pari a $MachinePrecision *)
```

```
True
```

```
5.^73
(* Questo calcolo è svolto in hardware a virgola mobile *)
```

1.05879×10^{51}

⌘ Di default, *Mathematica* fa vedere (displays) solo le prime sei cifre di un numero a precisione macchina, a meno che non venga richiesto diversamente.

⌘ Per vedere tutte le cifre, possiamo usare FullForm[ ] .
FullForm[expr] stampa la forma interna di qualsiasi cosa presente nell'espressione expr

```
{5.^73, FullForm[5.^73]}
```

$\{1.05879 \times 10^{51}, 1.0587911840678756`*^51\}$

```
Sqrt[2]+a
(* Sqrt[2]  è  2^(1/2) *)
FullForm[Sqrt[2]+a]
```

$\sqrt{2} + a$

```
Plus[Power[2, Rational[1, 2]], a]
```

```
Sqrt[2.]+a
FullForm[Sqrt[2.]+a]
```

```
1.41421 + a
```

```
Plus[1.4142135623730951`, a]
```

```
plot = Plot[x^2, {x, 0, 1}];
largeOutput = FullForm[plot];
```

⌘ In alternativa, si può usare InputForm[ ].
InputForm[ ] fa vedere (displays) ciò che si deve scrivere in input per ottenere qualcosa di uguale alla espressione data

```
InputForm[5.^73]
```

```
1.0587911840678756*^51
```

```
5.^73 == 1.0587911840678756*^51
```

```
True
```

```
(* InputForm[ Sqrt[2]+a] *)
```

```
InputForm[ Sqrt[2.] + a]
```

```
1.4142135623730951 + a
```

```
Sqrt[2.] + a == 1.4142135623730951 + a
```

```
True
```

```
? InputForm
```

Nota.
La precisione di un numero macchina è sempre $MachinePrecision.
Al contrario, se la precisione di un numero arbitrario è uguale od inferiore a $MachinePrecision, tale numero viene comunque memorizzato internamente come numero a precisione arbitraria.
Conviene, pertanto, usare MachineNumberQ per essere sicuri.

```
Map[MachineNumberQ,
 {N[1/3, 5],
  N[1/3, 15],
  N[1/3, $MachinePrecision],
  N[1/3, MachinePrecision],
  N[1/3]}]
```

```
{False, False, False, True, True}
```

```
(* NB. $MachinePrecision è il valore numerico del simbolo MachinePrecision *)
{FullForm[$MachinePrecision],
 FullForm[MachinePrecision]}
{(* Equal *) $MachinePrecision == MachinePrecision,
 (* SameQ *) $MachinePrecision === MachinePrecision}
```

```
{15.954589770191003`, MachinePrecision}
```

```
{True, False}
```

```
(* NOTE  abbreviate su Equal, SameQ  *)
(* SameQ: numeri Real , diversi nell'ultimo bit, sono  considerati identici *)
(* Equal: numeri approssimati, a precisione macchina o maggiore,
sono considerati uguali se differiscono
 negli ultimi 7 bit i.e. ultime 2 cifre decimali *)
(* Equal:
 usa approssimazioni numeriche per stabilire la uguaglianza tra numeri esatti *)
```

```
(* NOTE.
   Equal può restituire , in output, l'input UnEvaluated .
   SameQ  restituisce sempre un booleano *)
x == y
x === y
```

```
x == y
```

```
False
```

Nota.

Un input che contiene un punto decimale è sempre considerato essere un numero approssimato, anche quando noi sappiamo che non lo è.

D'altra parte *Mathematica* non può fare ipotesi diverse, altrimenti si incorrerebbe in errori:

```
(* Il risultato di questo calcolo è zero approssimato *)
3/4 - 0.75
Precision[%]
```

```
0.
```

```
MachinePrecision
```

Se nell'esempio qui sopra avessimo inteso scrivere 0.75 (ossia "tre quarti") in modo esatto, allora avremmo dovuto dare in input 75/100 (che viene ridotto a 3/4) :

```
75 / 100
```

$$\frac{3}{4}$$

```
3 / 4 - 75 / 100
Precision[%]
```

```
0
```

```
∞
```

Per riassumere.
Ci sono due classi di numeri:
esatti, che includono interi, razionali, complessi con coefficienti esatti;
approssimati, fatti da numeri che contengono sempre la virgola mobile.

⌘ I numeri **approssimati** sono suddivisi in due sottoclassi:
a precisione di **macchina**;
a precisione **arbitraria**.

⌘ *Mathematica* segue una convenzione inusuale per maneggiare i numeri esatti.
Di default, *Mathematica* non esegue mai una (qualsiasi) operazione numerica che potrebbe convertire una espressione esatta in una approssimata.
Per esempio:

```
Sqrt[3]
```

$$\sqrt{3}$$

Mathematica non riscrive l'espressione esatta Sqrt[3] come un numero, perché per fare ciò dovrebbe inserire una approssimazione (dato che Sqrt[3] non ha una rappresentazione decimale finita).

⌘ D'altra parte, *Mathematic*a valuta invece:

```
Sqrt[3.]
```

```
1.73205
```

L'argomento **3.** (della funzione Sqrt[ ]) è considerato approssimato, perché contiene la virgola mobile.
Dato che il numero **3.** è già approssimato, *Mathematica* non ha alcun problema a calcolare la sua radice quadrata in modo approssimato.

⌘ Possiamo chiedere a *Mathematica* di valutare numericamente ogni espressione esatta, usando la funzione N (come già visto):

```
enne = N[Sqrt[3]]
(* Di default, il risultato è un numero macchina *)
Precision[enne]
{FullForm[enne], InputForm[enne]}
```

```
1.73205
```

```
MachinePrecision
```

```
{1.7320508075688772`, 1.7320508075688772}
```

⌘ Un secondo argomento (opzionale) alla funzione N specifica la precisione desiderata nel risultato (come già visto):

```
enne20 = N[Sqrt[3], 20]
Precision[enne20]
```

```
1.7320508075688772935
```

```
20.
```

⌘ In generale, se anche solo uno degli argomenti (passati ad una funzione built-in numerica) è approssimato, la funzione verrà valutata, convertendo tale argomento a numero approssimato, nella precisione più alta giustificabile.
Rivediamolo con un esempio:

```
(*  Qui, l'addendo  è un numero macchina *)
add = 2.5;
Precision[add]
(* Il risultato è un numero macchina *)
{sum = 1 + add,
 Precision[sum]}
```

```
MachinePrecision
```

```
{3.5, MachinePrecision}
```

```
(* Qui, l'addendo ha 31 cifre dopo la virgola  *)
addendo = 2.5000000000000000000000000000001;
Precision[addendo]
(* Il risultato è un numero in precisione arbitraria *)
{sum = 1 + addendo,
 Precision[sum]}
```

```
31.3979
```

```
{3.5000000000000000000000000000001, 31.5441}
```

```
(* NB. Cambiamo il primo addendo, per creare un caso di Cancellazione  Numerica *)
canc = (-25 / 10) + addendo;
{canc // Chop,
 Precision[canc]}
```

```
{0, 0.}
```

```
(* Un altro esempio di perdita di precisione nell'output *)
addendo2 = 1.0000000090000000000000000000001;
Precision[addendo2]
(* Abbiamo perso 8 cifre con una sola sottrazione *)
{subtract = 1 - addendo2, Precision[subtract]}
```

```
31.
```

```
{-9.00000000000000000000 × 10^-9, 22.9542}
```

⌘ Un buon riferimento è il tutorial Numbers

```
Hyperlink[Framed["Tutorial sui Numeri"],
 "https://reference.wolfram.com/language/tutorial/Numbers.html"]
```

Tutorial sui Numeri

- **Stringhe di caratteri**
- **Esercizio 1S pg. 25S**

2. Elementi fondamentali del linguaggio

- **Riscrittura di termini**

2.1 Espressioni: espressioni normali (2.1.1)

2.1 Espressioni: atomi (2.1.2)

2.2 Valutazione di espressioni

Il processo di valutazione (Evaluation) di base è semplice.
⌘ Il Kernel continua a riscrivere termini fino a che non rimane nulla che esso sappia riscrivere in una forma diversa.

⌘ Dato che la riscrittura di termini (term rewriting) rimpiazza una espressione con un'altra, qualsiasi cosa sia rimasta (quando tale processo termina) deve essere una espressione valida: questo implica che l'insieme di tutte le espressioni è *chiuso* rispetto alla valutazione.

⌘ Questo permette che ogni espressione sia annidata dentro una qualsiasi altra (anche se il risultato del fare ciò potrebbe non avere senso ;-).

⌘ In analogia alla chiamata di funzione in altri linguaggi, chiamiamo *valore di ritorno* (return value) di una data espressione il risultato della valutazione di tale espressione.

Trace[ ]
È possibile ottenere una descrizione *post mortem* della valutazione di una qualsiasi espressione, utilizzan dola come argomento di Trace[] i.e. impacchettando tale espressione dentro la head Trace.

```
Trace[Sin[Log[2.5, 7]]]
(* Come vedremo tra poco, FullForm[Sin[Log[2.5,7]]] e TreeForm[Sin[Log[2.5,7]]]
 restituiscono subito 0.851012661490406`,che è un numero a precisione macchina *)
```

```
{{Log[2.5, 7], 2.12368}, Sin[2.12368], 0.851013}
```

```
(* Ricordo che Log[a,b] viene riscritto come Log[b]/Log[a]*)
Log[a, b] == Log[E, b] / Log[E, a] == Log[b] / Log[a]
Trace[ Log[a, b]];
```

```
True
```

Nota. Per esteso, i passi della valutazione di Sin[Log[2.5,7]] sono :
(1a) Log[2.5, 7] viene riscritto come Log[7]/Log[2.5]

(1b) Log[7] viene valutato numericamente
(1c) Log[2.5] viene valutato numericamente
1. Il quoziente dei due risultati precedenti viene valutato numericamente.
2. Sin del quoziente precedente viene valutato numericamente
→ Il risultato è un atomo: è un numero reale che non può essere ulteriormente riscritto.
Il processo, pertanto, termina.

```
(* 1a *) Log[2.5, 7] == Log[7] / Log[2.5]
(* 1b *) {Log[7] , Log[7.]}
(* 1c *) Log[2.5]
(* 1 *) Log[7.] / Log[2.5]
(* 2 *) Sin[%]
```

```
True
```

```
{Log[7], 1.94591}
```

```
0.916291
```

```
2.12368
```

```
0.851013
```

Valutazione standard (e non-standard)

⌘ L'esempio appena visto mosta un punto importante dell processo di valutazione.
In generale, le parti di una espressione normale vengono valutate prima dell'intera espressione.
Questo modo di procedere è detto *valutazione standard* .

⌘ Gli argomenti di certe funzioni di *Mathematica,* al contrario, non vengono valutati prima che la funzione sia invocata (called).
Questo modo di procedere è detto *valutazione non-standard* .
Ne vedremo un esempio più avanti.

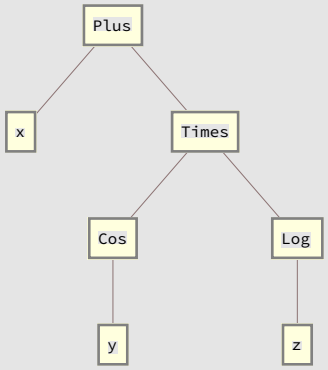
⌘ In termini informatici, una espressione è un albero (tree) e la sua valutazione viene eseguita "depth-first" i.e. vengono valutate per prime le parti (sotto-espressioni) che stanno a maggiore profondità (foglie) nell'albero rappresentante l'espressione stessa.

⌘ Le parentesi graffe indicano il livello di profondità (Depth) della sotto-espressione correntemente valutata.
L'annidamento delle parentesi graffe diventa maggiore via via che il processo di valutazione si addentra nella espressione, fino a raggiungerne le foglie.
Viceversa, l'annidamento delle parentesi graffe diventa minore via via che il processo di valutazione

risale nell'espressione, tornando indietro verso la sua radice (Head).

⌘ Possiamo usare TreeForm[ ] per stampare una espressione, esplicitamente, in forma di alberi (con le eventuali limitazioni imposte dall'output ASCII).
L'output della TreeForm[ ] può risultare non troppo leggibile, specie per espressioni molto grandi ed articolate.
In tale caso, è meglio usare FullForm[ ], studiandola attentamente.

```
TreeForm[x + Cos[y] Log[z], ImageSize → Small]
```

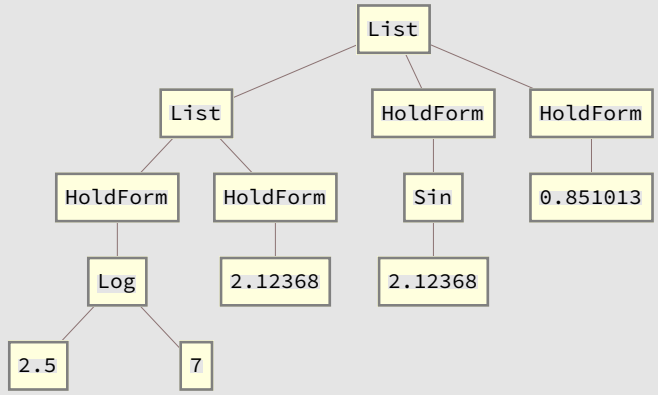


HoldForm[ ]

Se applichiamo TreeForm alla espressione **Trace**[Sin[Log[2.5,7]]] dell'esempio precedente, nell'albero (rappresentante tale espressione) appare HoldForm[ ].
HoldForm[expr] restituisce la stampa di una espressione, mantenendo tale espressione in formato non valutato.

```
trace = Trace[Sin[Log[2.5, 7]]]
trace // TreeForm
```

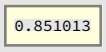
```
{{Log[2.5, 7], 2.12368}, Sin[2.12368], 0.851013}
```



⌘ Nel nostro esempio, HoldForm[ ] (usata da Trace[ ]) serve per stampare i risultati intermedi (messi in evidenza da Trace[ ]) come se fossero non valutati.

⌘ Se HoldForm[ ] **non** fosse presente nei risultati intermedi, otterremmo direttamente la TreeForm dell'atomo rappresentante il risultato della valutazione complessiva.
Questo accade proprio perche' (senza HoldForm) l'argomento di TreeForm[ ] viene valutato a numero reale **prima** che la TreeForm stessa venga valutata.
Lo stesso vale per FullForm.
Vediamolo:

```
(* HoldForm, usata da Trace, serve per stampare output intermedi di Trace.
  Senza HoldForm, otteniamo TreeForm dell'atomo rappresentante l'output finale,
che risulta dalla valutazione complessiva *)
espressione = Sin[Log[2.5, 7]];
TreeForm[espressione, ImageSize → Small]
FullForm[espressione]
```




```
0.851012661490406`
```

⌘ Per vedere la struttura interna di una espressione, la cui valutazione complessiva restituisce un numero (e.g. Sin[Log[2.5, 7]]), è necessario inibirne la valutazione.

Per fare ciò, possiamo impacchettare tale espressione dentro una head che impedisca ai suoi argomenti di essere valutati .
Un esempio di tale head è, appunto, HoldForm[ ] .
⌘ HoldForm[ ] inibisce la valutazione dei suoi argomenti. In questo modo, permette di vedere esplicitata la struttura di una espressione (cui HoldForm[ ] sia stata applicata).
⌘ HoldForm[ ] realizza, pertanto, una valutazione non-standard

```
? HoldForm
```

Symbol

HoldForm[*expr*] prints as the expression *expr*, with *expr* maintained in an unevaluated form.

⌘ Un buon riferimento è il tutorial **Evaluation**

```
Hyperlink[Framed["Tutorial su Evaluation"],
 "https://reference.wolfram.com/language/tutorial/Evaluation.html"]
```

Tutorial su Evaluation

Attributes[ ]
Per verificare se una Head simbolica inibisce (o meno) la valutazione delle sue parti, possiamo esaminarne le caratteristiche, con la built-in Attributes[ ]

```
Attributes[HoldForm]
(* HoldForm ha la caratteristica HoldAll i.e. per tutte le
 parti incluse nell'head HoldForm[] è inibita la valutazione *)
(* HoldForm ha la caratteristica Protected
 i.e. il suo nome non è ridefinibile da utente *)
```

```
{HoldAll, Protected}
```

Un punto di attenzione (ancora su Trace, HoldForm e attributo HoldAll)

HoldForm vs Hold

⌘ Come detto in precedenza, il procedimento di valutazione continua fino a che non rimane più nulla che possa essere riscritto in un'altra forma.
Se non ci fossero head come HoldForm, non ci sarebbe modo di ottenere il risultato di una valutazione parziale di una espressione (quale è, appunto, un componente di una Trace[ ]).

SetDelayed

Come detto all'inizio di questa sezione 2.2, in analogia alla chiamata di funzione in altri linguaggi, chiamiamo *valore di ritorno* (return value) di una data espressione il risultato della valutazione di tale espressione.
In altre parole, diciamo che ogni espressione restituisce un'altra espressione come suo valore.

⌘ Esistono dei casi, però, in cui l'affermazione precedente sembra essere falsa.
Un esempio è dato dall'operatore SetDelayed[ ] (cfr. 2.3.4), che pare non restituire alcun valore:

```
s2 = Sqrt[2] (* Set *)
s3 := Sqrt[3]
(*   :=  e' il simbolo sintattico di SetDelayed  *)
```

$\sqrt{2}$

Null

L'esempio qui sopra pare implicare che non esista alcun valore di ritorno della SetDelayed.

⌘ In effetti, SetDelayed restituisce il simbolo speciale Null, che di norma non appare nell'output.

```
? Null
```

Symbol

Null is a symbol used to indicate the absence of an
expression or a result. It is not displayed in ordinary output. When
Null appears as a complete output expression, no output is printed.

```
expr1; expr2; expr3;
(* la valutazione qui sopra restituisce Null, che non viene mostrato *)
(* Il punto-e-virgola inibisce l'output di Set *)
```

⌘ Null appare se è parte di una espressione più grande:

```
s2 = Sqrt[2];
1 + %
```

$1 + \sqrt{2}$

```
s3 := Sqrt[3]
(* Notiamo che non serve il punto-e-virgola per inibire l'output di SetDelayed *)
1 + %
```

```
1 + Null
```

```
Clear[a, b];

SeedRandom[3];
a = RandomReal[];
{a, a, a}

SeedRandom[3];
b := RandomReal[];
{b, b, b}
```

```
{0.478554, 0.478554, 0.478554}
```

```
{0.478554, 0.00869692, 0.347029}
```

⌘ DISGRESSIONE (su Null e sul punto-e-virgola)

```
(* Disgressione su Null e sul punto-e-virgola *)
f1[x_] := Module[
    {tmp = x},
    While[tmp > 2, tmp = Sqrt[tmp]];
    (* NOTIAMO il punto-e-virgola dopo While[] *)
    tmp
  ];
f1[100.]

f2[x_] := Module[{tmp = x},
    While[tmp > 2, tmp = Sqrt[tmp]]
     (* Nella Module,
     lo spazio tra While[] e statement tmp è interpretato come prodotto *)×
     tmp
    (* Pertanto, viene restituito Null
     (esito di While[]) moltiplicato per il valore salvato in tmp *)
  ];
f2[100.]
```

```
1.77828
```

```
1.77828 Null
```

⌘ Null appare se sopprimiamo esplicitamente un output (magari perche' è troppo grande, oppure perché non ci interessa vederlo):

```
Timing[ Total[Range[123 456]] ]
(* Se siamo interessati solo al tempo di esecuzione,
sopprimiamo il risultato del calcolo *)
Timing[ Total[Range[123 456]]; ]
```

```
{0.000118, 7 620 753 696}
```

```
{0.000091, Null}
```

⌘ SetDelayed[ ] è un esempio di funzione che opera producendo un *effetto collaterale* (side effect):
il risultato atteso dalla esecuzione della funzione non è il *return value*, ma è piuttosto un cambiamento apportato allo stato della sessione di *Mathematica* (oppure, in generale, al computer — ad esempio, la scrittura di dati in un file).

⌘ Nell'esempio visto, l'*effetto collaterale* è la creazione di una **regola** di riscrittura per il simbolo s3

```
s3 := Sqrt[3]; s3^2
```

```
3
```

⌘ DISGRESSIONE (su TypeOf e Return)

Attenzione alle dipendenze cicliche

⌘ Per completare questa sezione, dobbiamo analizzare un ultimo argomento.
Una assunzione implicita, nel processo di valutazione, è che il sistema sia disegnato in modo che l'insieme di tutte le espressioni sia parzialmente ordinato rispetto alla valutazione stessa.
In termini equivalenti, si suppone che (nel processo di valutazione) non esistano dipendenze cicliche.

⌘ Costruiamo un esempio in cui l'assunzione qui sopra venga violata:

```
(* Ricetta per un disastro! *)
yin := yang
(* il primo statement dice al Kernel che yin può essere riscritto come yang *)
yang := yin
(* il secondo statement dice al Kernel che yang può essere riscritto come yin *)
```

⌘ Per superare un caso come quello qui sopra, nel Kernel (per fortuna) è built-in un interruttore di circuito (circuit breaker), detto *iteration limit*

```
yin
```

••• $IterationLimit: Iteration limit of 4096 exceeded.

```
Hold[yin]
```

⌘ Dopo che la riscrittura yin/yang è avvenuta per 4096 (2^12) volte, il Kernel avvolge il risultato_corrente in una Hold[ ] (che inibisce ulteriori valutazioni) e restituisce, appunto, Hold[risultato_corrente].

NOTA. Il fatto che, in questo esempio, il risultato finale sia lo stesso dell'espressione originale (Hold[yin] se si valuta yin, Hold[yang] se si valuta yang) è una circostanza fortuita, dovuta alle definizioni usate.

⌘ Possiamo esaminare in dettaglio il processo ciclico, usando Trace[ ].

⌘ Message[ ]

La funzione Message[ ] causa l'apparizione del messaggio di errore (in rosso) sull'avere superato $IterationLimit .

⌘ La funzione Message[ ] viene invocata direttamente dal Kernel e non è parte della espressione originale (e neppure di qualsiasi delle sue forme intermedie).
Message[ ] può essere chiamata direttamente (dal programmatore, per associare messaggi di errore o warnings alle funzioni che lei/lui scrive).

```
? Message
```

Symbol

Message[*symbol*::*tag*] prints the message *symbol*::*tag* unless it has been switched off.

Message[*symbol*::*tag*, e_1, e_2, ...] prints a message, inserting the values of the e_i as needed.

▪ Esercizio 1 pg. 30

Non comprendere bene il meccanismo di valutazione (Evaluation process) può essere fonte di errori comuni.

Perchè, per esempio, *Mathematica* non restituisce un risultato in alta precisione dalla computazione numerica che segue?

```
tre = N[Sqrt[3.], 90]
```

```
1.73205
```

Usando FullForm, ci ricordiamo che Sqrt[3.] viene valutato immediatamente (a numero in precisione macchina).
Quindi la funzione N non può, successivamente, ampliare a 90 la sua precisione.

```
(* mach3 = Sqrt[3.];
tre=N[ mach3 , 90]; *)
tre // FullForm
tre // Precision
```

```
1.7320508075688772`
```

```
MachinePrecision
```

Un modo corretto per ottenere Sqrt[3] con 90 cifre di precisione è il seguente (cfr. 2.1.2):

```
(* exact3 = Sqrt[3];
tre=N[ exact3 , 90]; *)
tre90 = N[Sqrt[3], 90]
tre90 // Precision
```

```
1.732050807568877293527446341505872366942805253810380628055806979451933016908800003`.
 708114619
```

```
90.
```