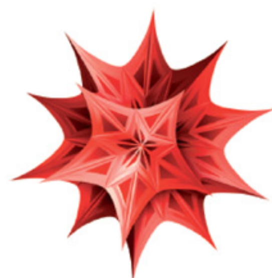


Wolfram *Mathematica*

Il software di riferimento per la Didattica, la Ricerca e lo Sviluppo

---

Lezione 9

L'approccio funzionale e il pattern matching

Crescenzio Gallo – Università di Foggia

crescenzio.gallo@unifg.it

Note:

- Il materiale visualizzato durante questo seminario è disponibile per il download all'indirizzo http://www.crescenziogallo.it/unifg/seminario-mathematica-2014/
- Per una migliore visione ingrandire lo schermo mediante il pulsante in alto a destra "Schermo intero"

---

11 – 25 Marzo 2014

Agenda

Introduzione

- *Mathematica* è un linguaggio
- *Mathematica* è un linguaggio funzionale e simbolico

Le basi della logica funzionale

- *L'approccio funzionale*
- *Il pattern e il pattern matching*
- *Le funzioni pure*
- *Ragionare per funzioni*
- Gli script one-line

Conclusioni

◄ | ►

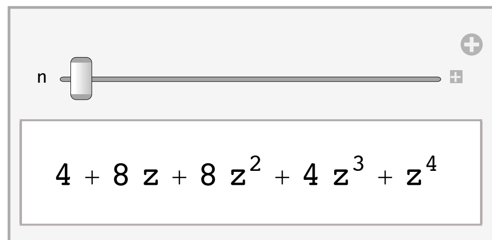
Introduzione: *Mathematica* è un linguaggio

Poichè siamo abituati ad usare *Mathematica* attraverso il front-end, capita spesso di pensare di non considerare la vera natura di *Mathematica* ossia quella di linguaggio di programmazione.
Già nel fare un semplice calcolo e renderlo parametrico e farlo computare al variare del parametro in effetti abbiamo sfruttato la capacità di programmazione del linguaggio *Mathematica*:

```
Expand[((z + 1)^2 + 1)^2]
```

$4 + 8 z + 8 z^2 + 4 z^3 + z^4$

```
Manipulate[Expand[((z + 1)^2 + 1)^n], {n, 2, 10, 1}]
```



Introduzione: *Mathematica* è un linguaggio funzionale e simbolico

Il principio di base di *Mathematica* è "Everything is an expression".

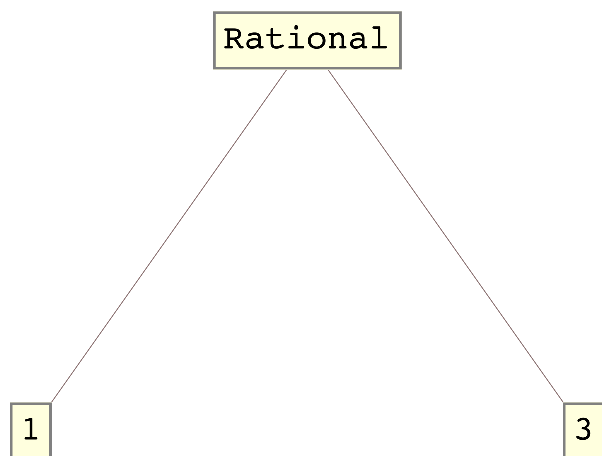
Ad eccezione delle entità atomiche (numeri, stringhe, simboli), tutto viene convertito internamente in espressioni o nidificazioni di espressioni.

La manipolazione delle espressioni permette a *Mathematica* di eseguire calcoli simbolici, di modificare interattivamente grafici o strutture dati, di animare o consentire all'utente di manipolare oggetti attraverso semplici interfacce.
Vediamo come viene rappresentato internamente un numero razionale:

```
FullForm[1/3]
```

```
Rational[1, 3]
```

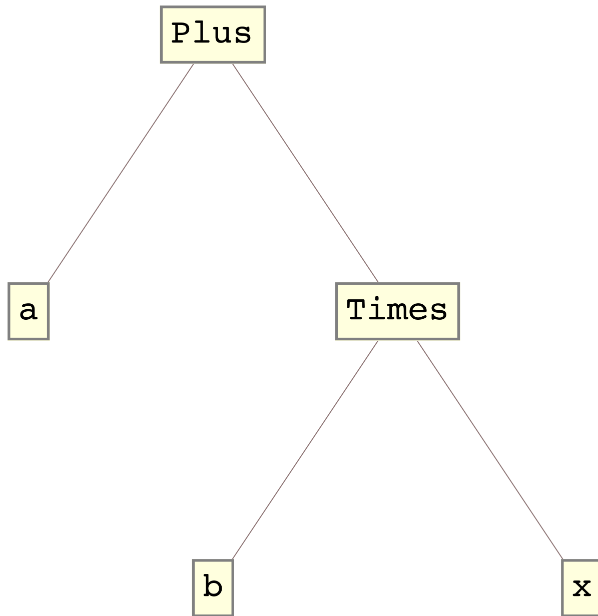
```
TreeForm[1/3]
```



Un binomio:

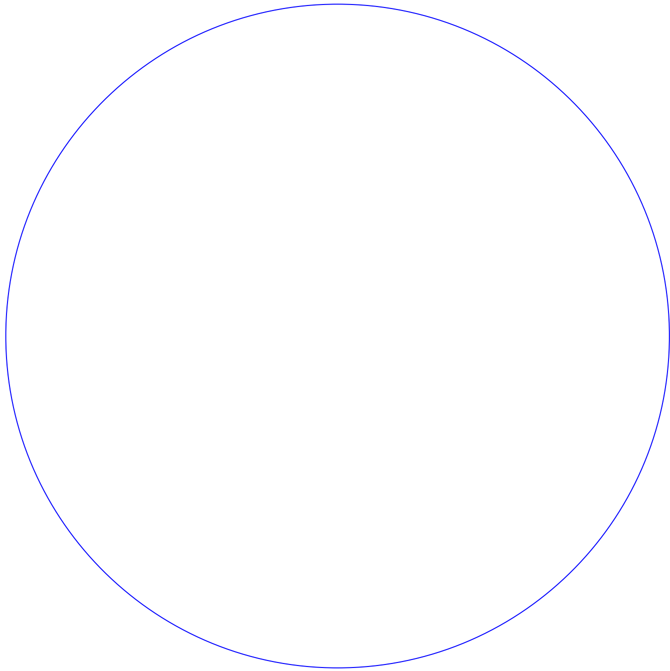
```
FullForm[a + b x]
Plus[a, Times[b, x]]
```

```
TreeForm[a + b x]
```



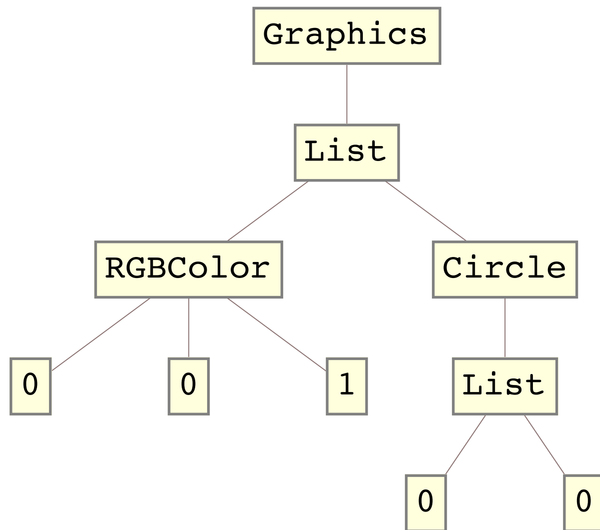
Un grafico:

```
gr = Graphics[{Blue, Circle[{0, 0}]}]
```



```
FullForm[gr]
Graphics[List[RGBColor[0, 0, 1], Circle[List[0, 0]]]]
```

```
TreeForm[gr]
```



Introduzione: *Mathematica* è un linguaggio funzionale e simbolico

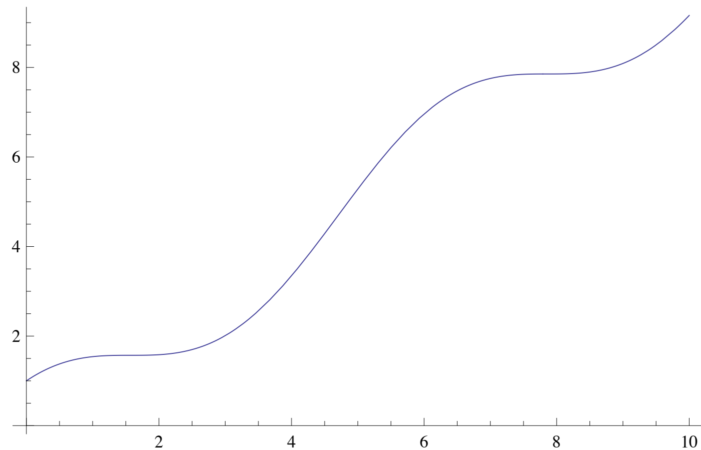
In *Mathematica* ogni oggetto è un'espressione che può essere gestita tramite le funzioni: implicazioni pratiche.

Qualsiasi espressione può essere inclusa in qualsiasi altra espressione.

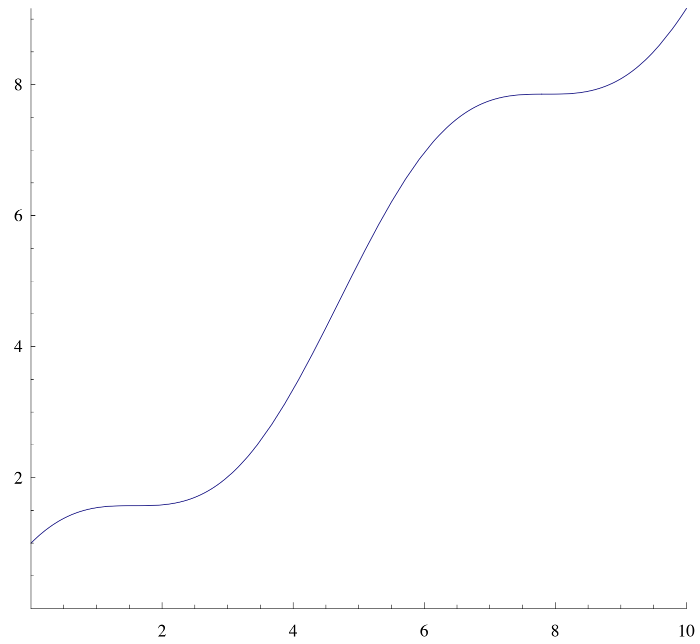
Esempio 1

Manipolare l'espressione di un grafico dopo averlo realizzato, ad esempio per ribaltare l'asse delle x rispetto a quello delle y:

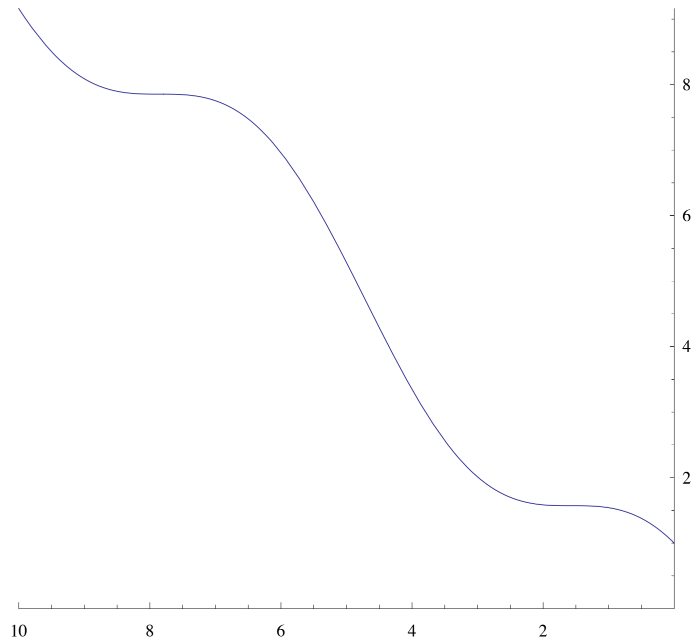
```
Plot[Cos[x] + x, {x, 0, 10}]
```




```
s = FullGraphics[Plot[Cos[x] + x, {x, 0, 10}]]
```



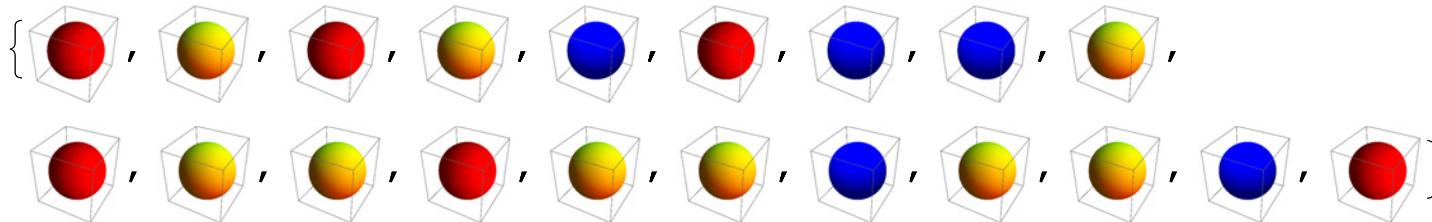

```
ReplaceAll[s, {x_Real, y_Real} → {-x, y}]
```



Esempio 2

Possiamo eseguire operazioni su liste di oggetti di qualsiasi natura. Ad esempio, contare le ricorrenze di ciascun elemento grafico in una lista di 20 elementi:

```
spheres = RandomChoice[{ , , }, 20]
```



```
Tally[spheres]
```

{{ , 6}, { , 9}, { , 5}}

Il concetto di *espressione* diviene fondamentale in relazione alla logica funzionale del linguaggio *Mathematica* perchè - per poter sfruttare al meglio le funzioni da applicare sui dati - bisogna considerare in maniera corretta i dati, ossia vederli sempre come espressioni ancor prima di essere liste, array, matrici, polinomi o qualsiasi altra cosa.

◂ | ▸

Le basi della logica funzionale: l'approccio *funzionale*

Vediamo un semplice esempio di come un algoritmo può essere riscritto secondo diversi stili di programmazione.

Esempio 3: la distanza tra un punto nel piano e l'origine degli assi.

Dato un array di coppie che rappresentano punti nel piano, dobbiamo calcolare l'array delle distanze di ciascun punto dall'origine degli assi:

```
punti = RandomInteger[{-5, 5}, {10, 2}]
{{-3, 5}, {-4, 2}, {1, 5}, {5, -3}, {2, -1}, {-4, 5}, {0, -4}, {1, -4}, {-4, -1}, {3, 4}}
```

Nell'approccio procedurale definisco un ciclo che prende ciascun elemento della lista e ne calcola la distanza dall'origine degli assi:

```
DistanzaP[lista_] :=
 Module[{risultato},
  risultato = Table[Null, {Length[lista]}];
  Do[
   risultato[[i]] = Sqrt[lista[[i, 1]]^2 + lista[[i, 2]]^2],
   {i, 1, Length[lista]}
  ];
  Return[risultato]]
```

```
DistanzaP[punti]
```

$\{\sqrt{34}, 2\sqrt{5}, \sqrt{26}, \sqrt{34}, \sqrt{5}, \sqrt{41}, 4, \sqrt{17}, \sqrt{17}, 5\}$

Nell'approccio funzionale applico una funzione all'intero data set, senza dover gestire io manualmente lo scorrimento degli elementi nel data set stesso:

```
DistanzaF[lista_] := Map[Norm, lista]
```

```
DistanzaP[punti]
```

$\{\sqrt{34}, 2\sqrt{5}, \sqrt{26}, \sqrt{34}, \sqrt{5}, \sqrt{41}, 4, \sqrt{17}, \sqrt{17}, 5\}$

Vediamo la differenza di tempo

```
punti = RandomInteger[{-5, 5}, {10 000, 2}];
```

```
AbsoluteTiming[a = DistanzaP[punti];]
```

```
{0.456635, Null}
```

```
AbsoluteTiming[b = DistanzaF[punti];]
```

```
{0.096230, Null}
```

Confrontiamo anche che i risultati siano uguali:

```
a == b
```

```
True
```

Si potrebbe osservare che nella funzione **DistanzaF** si è usata la funzione **Norm** al posto del calcolo della radice. In effetti, nell'approccio procedurale in genere si è portati più a costruire la soluzione passo per passo a partire dai dati disponibli e con le formule della teoria; mentre, con una forma mentis più rivolta alle funzioni ed all'approccio funzionale, si ha una maggiore propensione a cercare sempre una funzione che implementa la formula necessaria e poi applicare tale funzione al data set in questione.

Comunque, anche eliminando la semplicità della funzione **Norm**, si ha un vantaggio di tempo:

```
DistanzaF1[lista_] := Map[√(Apply[Plus, #^2]) &, lista]
```

Si noti come in una logica funzionale si è usato il valore dell'attributo **Listable** della funzione **Power** e l'**Apply** per costruire la somma degli elementi di un array, cose che nella logica procedurale non esistono e dunque non si sono usati:

```
AbsoluteTiming[c = DistanzaF1[punti];]
{0.097401, Null}
```

```
a === b === c
True
```

In ogni caso, anche operando al contrario (ossia usando **Norm** dentro l'algoritmo procedurale) non si ottiene alcun miglioramento:

```
DistanzaP1[lista_] :=
 Module[{risultato},
  risultato = Table[Null, {Length[lista]}];
  Do[
   risultato[[i]] = Norm[lista[[i]]],
   {i, 1, Length[lista]}
  ];
  Return[risultato]]
```

```
AbsoluteTiming[d = DistanzaP1[punti];]
{0.492821, Null}
```

```
d === c
True
```

Vediamo un altro possible approccio, anche questo molto vicino alla natura di *Mathematica*: ossia l'approccio "Rule-Based", che utilizza le regole di sostituzione al posto delle funzioni per operare sui dati:

```
DistanzaRB[lista_] := lista /. {x_, y_} :> Norm[{x, y}]
```

```
AbsoluteTiming[e = DistanzaRB[punti];]
```

```
{0.104299, Null}
```

```
e === c
```

```
True
```

```
?ReplaceAll
```

expr /. *rules* applies a rule or list of rules in an attempt to transform each subpart of an expression *expr*. ≫

```
punti[[1 ;; 20]]
```

```
{{3, -5}, {-1, 1}, {-1, -1}, {3, 5}, {2, 2}, {-2, 3}, {4, -4}, {-4, -4}, {-3, -3}, {5, 1},
 {-5, -2}, {-3, 4}, {-2, -1}, {-4, 5}, {-2, 1}, {0, -1}, {-1, -1}, {-3, 2}, {4, 3}, {-5, 0}}
```

```
punti[[1 ;; 20]] /. {x_, y_} :> Norm[{x, y}]
```

$\{\sqrt{34}, \sqrt{2}, \sqrt{2}, \sqrt{34}, 2\sqrt{2}, \sqrt{13}, 4\sqrt{2}, 4\sqrt{2}, 3\sqrt{2}, \sqrt{26}, \sqrt{29}, 5, \sqrt{5}, \sqrt{41}, \sqrt{5}, 1, \sqrt{2}, \sqrt{13}, 5, 5\}$

Le basi della logica funzionale: il *pattern* e il *pattern matching*

Se devo applicare una funzione ad un intero elenco di oggetti posso facilmente procedere anche senza ricorre ai pattern, nel senso che indistintamente applico la funzione all'intera lista di oggetti. In realtà, in molti casi non ci troviamo in situazioni così semplici; per cui, dato un insieme di valori, dobbiamo filtrare quelli a cui siamo interessati perchè solo a quelli dobbiamo applicare una certa funzione. In questo caso interviene il concetto di *pattern* e la tecnica del *pattern matching*.

Esempio 4: Miglioriamo la funzione Distanza

Nell'esempio di sopra, a prescindere da quale versione consideriamo, ci sono diversi punti deboli della soluzione costruita. Vediamo qualche esempio.

```
DistanzaF[{{2, -3}, {4, -1}, {2, 1}, {9, 3, 2}}]
```

$\left\{\sqrt{13}, \sqrt{17}, \sqrt{5}, \sqrt{94}\right\}$

```
DistanzaRB[{{2, -3}, {4, -1}, {2, 1}, {9, 3, 2}}]
```

$\left\{\sqrt{13}, \sqrt{17}, \sqrt{5}, \{9, 3, 2\}\right\}$

```
DistanzaF1[{{2, -3}, {4, -1}, {2, 1}, {9, 3, 2}}]
```

$\left\{\sqrt{13}, \sqrt{17}, \sqrt{5}, \sqrt{94}\right\}$

```
DistanzaP[{{2, -3}, {4, -1}, {2, 1}, {9, 3, 2}}]
```

$\left\{\sqrt{13}, \sqrt{17}, \sqrt{5}, 3\sqrt{10}\right\}$

Vediamo che solo alcuni hanno risposto correttamente, ossia **DistanzaF** e **DistanzaF1** perchè sono basati su funzioni che non dipendono dalla struttura della lista, se a due o tre dimensioni.
Se vogliamo rendere queste funzioni più corrette e farci restituire un valore diverso a seconda dei diversi casi, possiamo

ad esempio introdurre dei pattern per la variabile di input *lista* in modo che, a seconda dei casi, possiamo eseguire un calcolo diverso. Scegliamo una sola delle funzioni sopra usate:

```
Distanza[lista_] := Map[Norm, lista]
```

Se vogliamo farla funzionare solo nel caso di elementi di tipo coppie, possiamo usare il seguente filtro:

```
Distanza[lista : {{_, _} ..}] := Map[Norm, lista]

Distanza[{{2, -3}, {4, -1}, {2, 1}, {9, 3}}]
```

$\{\sqrt{13}, \sqrt{17}, \sqrt{5}, 3\sqrt{10}\}$

```
Distanza[{{2, -3}, {4, -1}, {2, 1}, {9, 3, 2}}]
Distanza[{{2, -3}, {4, -1}, {2, 1}, {9, 3, 2}}]
```

Bene abbiamo disabilitato la funzione nel caso di input misto (coppie/triple). Se vogliamo, possiamo usare una caratteristica delle funzioni di *Mathematica*, ossia l'*overloading* (cioè la possibilità di definire una funzione con lo stesso nome ma con differenti pattern di ingresso):

```
?? Distanza
```

```
Global`Distanza

Distanza[lista : {{_, _} ..}] := Norm /@ lista
```

```
Distanza[lista___] := "Input non consentito"

Distanza[{{2, -3}, {4, -1}, {2, 1}, {9, 3, 2}}]
Input non consentito

Distanza[{{2, -3}, {4, -1}, {2, 1}}]
```

$\{\sqrt{13}, \sqrt{17}, \sqrt{5}\}$

```
Distanza[pippo]
```

```
Input non consentito
```

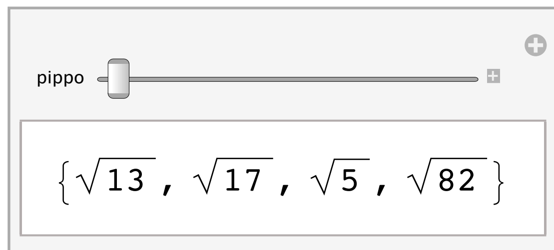
Potremmo volere ancora di più; guardiamo il seguente caso:

```
Distanza[{{2, -3}, {4, -1}, {2, 1}, {9, pippo}}]
```

$\left\{\sqrt{13}, \sqrt{17}, \sqrt{5}, \sqrt{81 + \text{Abs}[\text{pippo}]^2}\right\}$

che potrebbe essere in alcuni casi voluto; per esempio, posso in seguito specificare quanto vale *pippo*:

```
Manipulate[%, {pippo, 1, 5}]
```



In realtà, per l'esempio in questione potrebbe essere un effetto indesiderato. Allora possiamo rafforzare il *pattern matching* restringendo ulteriormente la definizione del pattern ammesso in ingresso. Per fare questo, azzero il contenuto della funzione Distanza perchè devo riscrivere la sua definizione principale:

```
Clear[Distanza]
```

```
Distanza[lista : {{_?NumericQ, _?NumericQ} ..}] := Map[Norm, lista]
```

```
Distanza[lista___] := "Input non consentito"
```

```
Distanza[{{2, -3}, {4, -1}, {2, 1}, {9, pippo}}]
```

```
Input non consentito
```

Dunque, ora abbiamo una funzione che opera solo su coppie di numeri.

Le basi della logica funzionale: le funzioni pure

Quando si comincia a scrivere codice funzionale si utilizzano sempre più i *pattern*, non solo per creare filtri da applicare agli argomenti in input ma anche per selezionare sottoinsiemi di dati su cui applicare una funzione (ad esempio tramite le funzioni **Map** e simili). Nella maggior parte di questi casi non serve definire una funzione che funga da filtro, ma basta scrivere il filtro stesso tramite una funzione "pura". La principale differenza tra una funzione "normale" ed una funzione "pura" è che quest'ultima non è salvata in memoria.

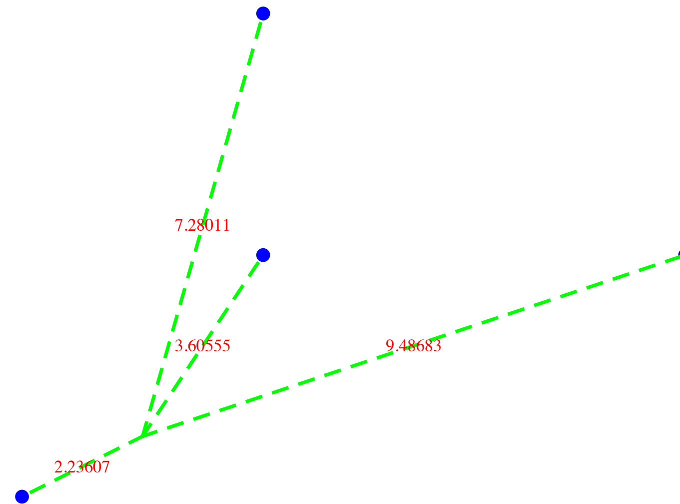
Riprendiamo l'esempio della distanza di una serie di punti dall'origine degli assi e aggiungiamo anche il grafico del risultato.

```
Quit[]
```

```
Distanza[lista : {{_?NumericQ, _?NumericQ} ..}] := Map[Norm, lista]
```

```
Distanza[lista___] := "Input non consentito"
```

Aggiungiamo il grafico che rappresenta le linee da ciascun punto verso l'origine degli assi; nel mezzo della linea deve comparire una label con il valore della distanza, qualcosa del tipo:



La funzione Distanza restituisce i valori numerici per le etichette:

```
punti = {{2, 3}, {-2, -1}, {2, 7}, {9, 3}}
{{2, 3}, {-2, -1}, {2, 7}, {9, 3}}
```

```
Distanza[punti]
```

$\{\sqrt{13}, \sqrt{5}, \sqrt{53}, 3\sqrt{10}\}$

Dobbiamo ora aggiungere tutta la parte di grafica. Seguiamo prima un approccio senza le funzioni pure.

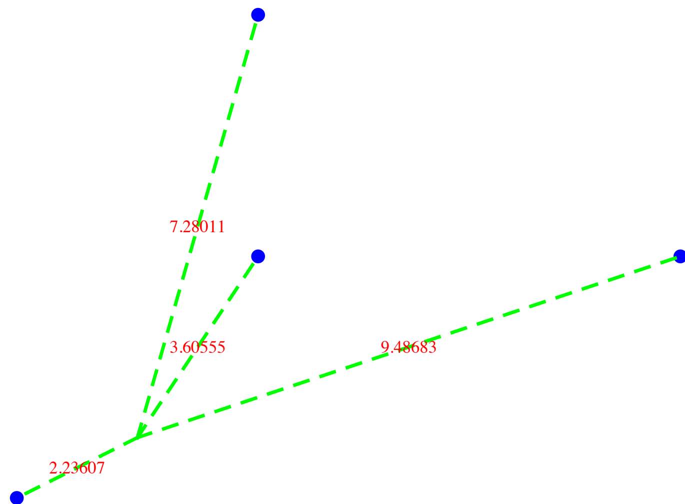
Dai punti e le distanze dobbiamo ricavare:
a) il punto blu di grandezza 0.02;
b) l'etichetta con il valore della distanza del punto dall'origine, posizionata nel punto pari alla metà del segmento distanza;
c) la linea dall'origine degli assi al punto assegnato.

Definiamo tre funzioni ausiliari esterne:

```
Punto[coppia : {_?NumericQ, _?NumericQ}] := {PointSize[0.02], Blue, Point[coppia]}
Linea[coppia : {_?NumericQ, _?NumericQ}] :=
 {Thickness[0.005], Dashing[0.02], Green, Line[{{0, 0}, coppia}]}
Etichetta[coppia : {_?NumericQ, _?NumericQ}, distanza_?NumericQ] :=
 {Red, Text[ToString[distanza], coppia / 2]}
```

Ora definiamo la funzione complessiva in termini delle precedenti:

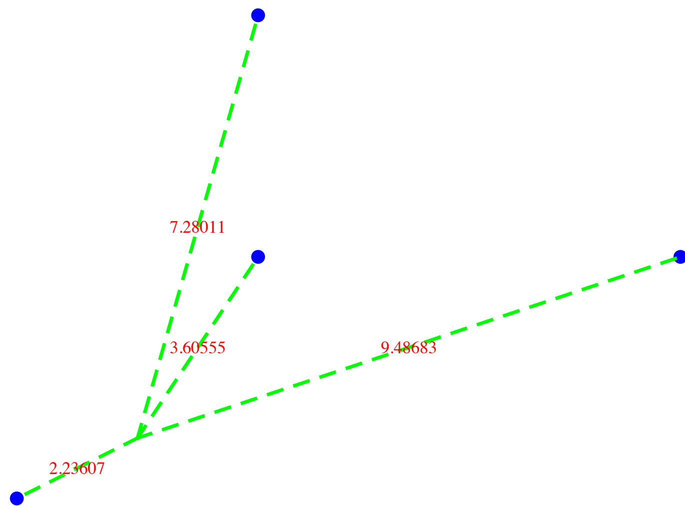
```
Distanza1[lista : {{_?NumericQ, _?NumericQ} ..}] := Graphics[
  Table[{Punto[lista[[i]]], Linea[lista[[i]]],
    Etichetta[lista[[i]], N[Norm[lista[[i]]]]]}, {i, Length[lista]}]]
Distanza1[punti]
```



Ma possiamo constatare che al di fuori del corpo Distanza1, le tre funzioni ausiliare non servono e pertanto risulta uno spreco di memoria averle definite esplicitamente. Possiamo creare tre funzioni pure che “vivono e muoiono” soltanto nel tempo necessario al loro compito.

```
Distanza[lista : {{_?NumericQ, _?NumericQ} ..}] :=
 Graphics[MapThread[{{Blue, PointSize[0.02], Point[#1] },
     {Thickness[0.005], Dashing[0.02], Green, Line[{{0, 0}, #1}]},
     {Red, Text[ToString[#2], #1 / 2]}} &, {punti, N[Map[Norm, lista]]}]]

Distanza[punti]
```



Le basi della logica funzionale: ragionare per funzioni

In questo esempio mostriamo come si può procedere passo passo ad una soluzione funzionale completa tramite il ragionamento sul problema e sulla sua soluzione generalizzata.

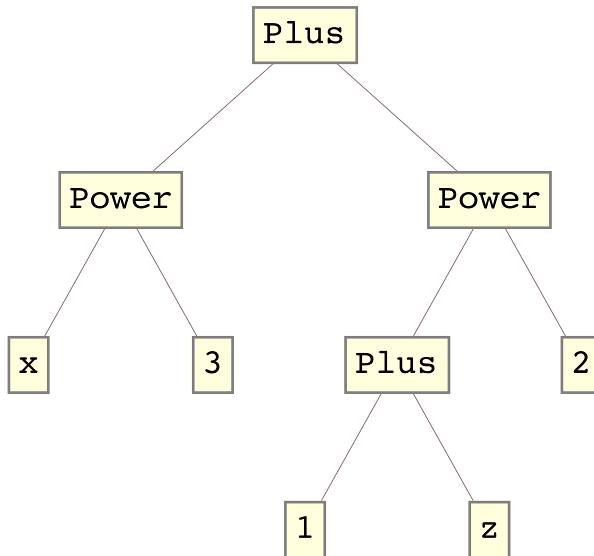
Il problema consiste nel dover sostituire i valori simbolici di una data espressione con una funzione di tali simboli

```
exp = x^3 + (1 + z)^2
```

$x^3 + (1 + z)^2$

Bisogna sostituire x e z con i valori sin(x) e sin(z).
Primo modo di procedere: vedendo l'espressione, individuo la posizione dei simboli e li sostituisco

```
TreeForm[exp]
```




```
MapAt[Sin, exp, {{1, 1}, {2, 1, 2}}]
```

$\text{Sin}[x]^3 + (1 + \text{Sin}[z])^2$

Ma se non posso vedere fisicamente l'espressione? Allora mi *calcolo* la posizione dei due simboli

```
Position[exp, x]
```

{{1, 1}}

```
Position[exp, z]
```

{{2, 1, 2}}

```
MapAt[Sin, exp, Join[Position[exp, x], Position[exp, z]]]
```

$\text{Sin}[x]^3 + (1 + \text{Sin}[z])^2$

E se l'espressione contiene un altro simbolo e non *z*?

Dal suo help scopro che **Position** richiede un pattern al secondo posto:

```
? Position
```

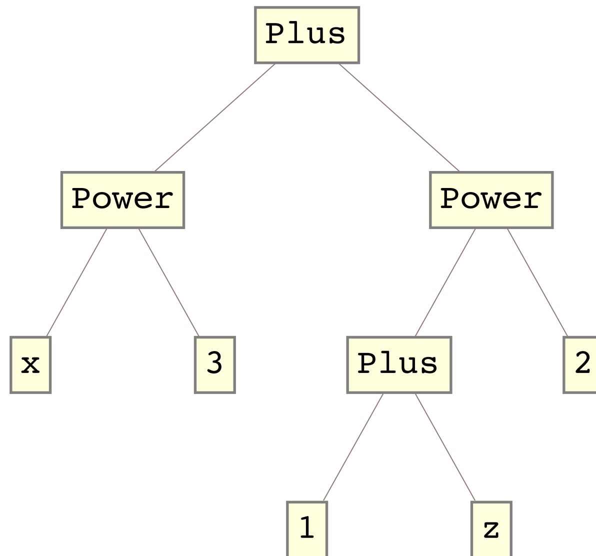
Position[*expr*, *pattern*] gives a list of the positions at which objects matching *pattern* appear in *expr*.
Position[*expr*, *pattern*, *levelspec*] finds only objects that appear on levels specified by *levelspec*.
Position[*expr*, *pattern*, *levelspec*, *n*] gives the positions of the first *n* objects found. ≫

Dunque basta che scrivere il pattern che definisce i simboli:

```
Position[exp, _Symbol]
{{0}, {1, 0}, {1, 1}, {2, 0}, {2, 1, 0}, {2, 1, 2}}
```

Ci sono troppi risultati ... perchè? Sempre dall'help scopro che **Position** considera anche il livello zero di ciascuna espressione o sotto espressione. Dunque, se chiedo la posizione di un simbolo mi darà anche la posizione dei simboli che formano le funzioni di *exp*:

```
TreeForm[exp]
```



Allora inibisco il controllo sui simboli delle *head* delle espressioni e sotto escpressioni (livello zero):

```
Position[exp, _Symbol, Heads → False]
```

```
{{1, 1}, {2, 1, 2}}
```

```
MapAt[Sin, exp, Position[exp, _Symbol, Heads → False]]
```

$\text{Sin}[x]^3 + (1 + \text{Sin}[z])^2$

E se su una qualsiasi espressione voglio applicare una qualsiasi funzione al posto dei simboli e non necessariamente "sin"?
Posso creare una funzione e generalizzare sia l'espressione sia la funzione da applicare:

```
CambiaVariabili[funzione_, espressione_] :=
 MapAt[funzione, espressione, Position[espressione, _Symbol, Heads → False]]
```

```
CambiaVariabili[Log, 2 x + x w^3 + 2 x^r + y + t + w]
```

$\text{Log}[t] + \text{Log}[w] + 2\,\text{Log}[x] + \text{Log}[w]^3\,\text{Log}[x] + 2\,\text{Log}[x]^{\text{Log}[r]} + \text{Log}[y]$

Un'altra soluzione si ha con **Replace**:

```
CambiaVariabili1[funzione_, espressione_] :=
 Replace[espressione, x_Symbol :> funzione[x], Infinity]
```

```
CambiaVariabili1[Log, 2 x + x w^3 + 2 x^r + y + t + w]
```

$\text{Log}[t] + \text{Log}[w] + 2\,\text{Log}[x] + \text{Log}[w]^3\,\text{Log}[x] + 2\,\text{Log}[x]^{\text{Log}[r]} + \text{Log}[y]$

◄ | ►

Le basi della logica funzionale: ragionare per funzioni

In questo secondo esempio mostriamo come, ragionando istintivamente per “procedure” (ossia immaginando sempre che la soluzione sia costruita elemento per elemento), ci si può abituare ad una visione ristretta sul modo di trovare una soluzione.

Il problema consiste nel dover generare una lista di lunghezza definita con interi casuali compresi nell’intervallo {-10, 10} ad esclusione dello zero.

Ragionando immediatamente per procedure, penso di costruire una lista di valori random e poi eliminare gli eventuali zeri sostituendoli con altri random non nulli.
Una prima soluzione potrebbe essere:

```
NonNulli[n_] :=
 Module[{a},
  a = RandomInteger[{-10, 10}, {n}];
  While[MemberQ[a, 0], a = ReplacePart[a, RandomInteger[{-10, 10}], Position[a, 0]]];
  Return[a]]
```

```
NonNulli[300]
{1, -4, -4, -10, 4, -5, 9, 7, 4, 9, 9, 3, -3, -6, 5, -4, 9, 2, 2, -9, 3, 7, 6, 8, 9, -4, 9, 9,
 1, 5, -4, 4, 3, 3, 2, 4, 6, -1, 1, 8, -9, 9, 8, -3, 8, -4, 9, 3, 7, 7, -8, 9, -8, -5, 1,
 -1, -2, -6, -4, 2, -10, 7, 6, 9, 6, 1, -5, -10, -9, 2, -5, 9, 6, -5, -9, 7, -4, -10, -5,
 -8, -1, -3, 7, -3, 5, -3, -2, -1, -1, 2, 3, 4, -2, 10, 5, 9, -2, 9, 9, -3, 1, -7, -10,
 -6, 8, 4, 6, 3, -1, -10, -3, -2, -1, -2, -5, 9, 2, 2, 1, -4, -1, 6, -6, 3, -3, -1, -3,
 -1, -8, -7, -4, 3, -1, 4, -7, -10, -8, 5, -9, -9, -3, 9, 2, -9, -2, 7, -1, -1, 7, 9, 5,
 -3, 7, -10, 6, 2, 2, 5, -9, -3, 9, 7, 6, 4, -4, -5, 9, -3, 1, 9, 10, 4, 3, 10, 1, 8, -7,
 4, -9, -5, 2, 5, 1, -6, 4, -4, -4, -7, 2, 4, 7, 1, 9, 7, 7, -2, 3, -10, -3, -1, -2, -8,
 -10, -3, -6, 9, 1, 8, -1, -5, 7, -8, 7, -3, -2, 5, 9, 8, 4, 1, 9, 7, -4, 8, -8, 5, -1,
 -10, 4, 3, 10, 4, -1, 2, 3, 7, -9, -3, -7, -10, -7, 6, 2, -8, 9, -1, 9, -3, 7, 6, 3, -10,
 8, -8, 10, 2, 6, 7, -5, -10, -4, -2, -3, -9, -6, -2, -10, -10, 3, -3, -1, -7, 8, 8, 9,
 -3, -3, 9, -2, 3, -3, 2, -8, 7, 5, 9, 1, 7, -6, 1, 1, -2, 4, 10, 4, -5, -6, -5, -1, -2}

MemberQ[NonNulli[3 000 000], 0]
False

AbsoluteTiming[MemberQ[NonNulli[3 000 000], 0]]
{0.759927, False}
```

Proviamo un approccio diverso

1)

```
NonNulli1[n_] := Table[(-1)^Random[Integer] RandomInteger[{1, 10}], {n}]

AbsoluteTiming[MemberQ[NonNulli1[3 000 000], 0]]
{0.551765, False}
```

2)

```
NonNulli2[n_] := Table[RandomChoice[{-1, 1}] * RandomInteger[{1, 10}], {n}]
```

```
AbsoluteTiming[MemberQ[NonNulli2[3 000 000], 0]]
```

```
{1.203304, False}
```

3)

```
NonNulli3[n_] :=
 RandomChoice[{-10, -9, -8, -7, -6, -5, -4, -3, -2, -1, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10}, n]
```

```
AbsoluteTiming[MemberQ[NonNulli3[3 000 000], 0]]
```

```
{0.177276, False}
```

Come si evince da questo ultimo esempio, ogni qualvolta si riesce ad evitare un ciclo (es. **Table**) il tempo di calcolo si riduce notevolmente :-)

◀ | ▶

Le basi della logica funzionale: gli script one-line

Negli esempi della slide precedente uno dei vantaggi offerti dall'approccio funzionale è che in realtà non si ha sempre bisogno di usare variabili intermedie per memorizzare i risultati parziali. Sfruttando il concetto secondo cui le funzioni si applicano ai dati ed il risultato fornisce ancora dei dati su cui applicare altre funzioni, si possono creare righe di codice molto sintetico e perciò molto veloce.
Nel paradigma funzionale teoricamente tutti gli algoritmi si possono scrivere come script one-line, ossia tutto il codice si può sintetizzare in un'unica istruzione. Questo è possibile grazie al fatto che secondo tale paradigma in realtà non sono necessarie variabili di appoggio e non sono necessari cicli. Sebbene questo punto di vista sia condivisibile in principio, nella realtà questa situazione non capita spesso e soprattutto non è semplice da ottenere se non in casi molto semplici. Ad ogni modo è bene abituarsi alla logica one-line almeno come modello di esempio, così si impara a rendere il proprio codice sempre più sintetico e vicino alla logica funzionale.
Vediamo un piccolo trucco per convertire più righe di codice in una sola riga.

Come scrivere codice one-line:

```
InstallJava[]
```

```
LinkObject['/Applications/Mathematica
   9-0-1-0.app/SystemFiles/Links/JLink/JLink.app/Contents/MacOS/JavaApplicationStub'
   -init "/tmp/m000010347551", 756, 6]
```

```
data = Import["http://www.bls.gov/web/laus/laumstrk.htm", {"Data"}];
(* importa la pagina html *)

data2 = Cases[data, {_Integer, _String, _Real}, Infinity];
(* Estraiamo i soli dati{rank, STATE, rate} *)

results = data2[[All, {2, 3}]]; (* eliminiamo il rank *)
```

```
SortBy[results, First]; (* ordiniamo alfabeticamente *)
Grid[results, Alignment → Left, Frame → All, Background → LightYellow, BaseStyle → {"Menu"}]
```

NORTH DAKOTA	2.7
NEBRASKA	3.6
SOUTH DAKOTA	3.6
UTAH	4.
IOWA	4.2
VERMONT	4.2
WYOMING	4.4
HAWAII	4.7
MINNESOTA	4.7
KANSAS	4.9
NEW HAMPSHIRE	5.2
VIRGINIA	5.2
LOUISIANA	5.4
MONTANA	5.4
OKLAHOMA	5.4
IDAHO	5.6
MISSOURI	6.
TEXAS	6.
WEST VIRGINIA	6.
ALABAMA	6.1
MARYLAND	6.1

COLORADO	6.2
DELAWARE	6.2
FLORIDA	6.3
WISCONSIN	6.3
ALASKA	6.4
MAINE	6.4
NEW MEXICO	6.6
SOUTH CAROLINA	6.6
WASHINGTON	6.7
INDIANA	6.8
PENNSYLVANIA	6.8
NORTH CAROLINA	6.9
NEW YORK	7.
MASSACHUSETTS	7.1
OHIO	7.1
OREGON	7.1
NEW JERSEY	7.2
ARKANSAS	7.4
CONNECTICUT	7.4
GEORGIA	7.4
ARIZONA	7.6
DISTRICT OF COLUMBIA	7.6
TENNESSEE	7.7

MISSISSIPPI	7.8
KENTUCKY	7.9
CALIFORNIA	8.3
MICHIGAN	8.3
ILLINOIS	8.9
NEVADA	9.
RHODE ISLAND	9.3

Mettiamo tutto insieme in una sola riga:

```
Grid[SortBy[Cases[data, {_Integer, _String, _Real}, Infinity][[All, {2, 3}]], First],
 Alignment → Left, Frame → All, Background → LightYellow, BaseStyle → {"Menu"}]
```

ALABAMA	6.1
ALASKA	6.4
ARIZONA	7.6
ARKANSAS	7.4
CALIFORNIA	8.3
COLORADO	6.2
CONNECTICUT	7.4
DELAWARE	6.2
DISTRICT OF COLUMBIA	7.6
FLORIDA	6.3
GEORGIA	7.4
HAWAII	4.7
IDAHO	5.6

ILLINOIS	8.9
INDIANA	6.8
IOWA	4.2
KANSAS	4.9
KENTUCKY	7.9
LOUISIANA	5.4
MAINE	6.4
MARYLAND	6.1
MASSACHUSETTS	7.1
MICHIGAN	8.3
MINNESOTA	4.7
MISSISSIPPI	7.8
MISSOURI	6.
MONTANA	5.4
NEBRASKA	3.6
NEVADA	9.
NEW HAMPSHIRE	5.2
NEW JERSEY	7.2
NEW MEXICO	6.6
NEW YORK	7.
NORTH CAROLINA	6.9
NORTH DAKOTA	2.7
OHIO	7.1

OKLAHOMA	5.4
OREGON	7.1
PENNSYLVANIA	6.8
RHODE ISLAND	9.3
SOUTH CAROLINA	6.6
SOUTH DAKOTA	3.6
TENNESSEE	7.7
TEXAS	6.
UTAH	4.
VERMONT	4.2
VIRGINIA	5.2
WASHINGTON	6.7
WEST VIRGINIA	6.
WISCONSIN	6.3
WYOMING	4.4

Conclusioni

Nonostante sembra che la logica funzionale abbia una curva di apprendimento più ripida (almeno a detta di molti utenti), una volta raggiunta una certa predisposizione al ragionamento per funzioni si riesce a scrivere codice *Mathematica* con una notevole sintesi, eleganza stilistica e leggibilità del codice che sono impagabili tanto per piccoli algoritmi quanto per applicazioni complesse.

Comunque, si tenga conto che *Mathematica* - pur essendo un linguaggio funzionale - mette a disposizione set di funzioni dedicati ai vari approcci di programmazione:

Procedural Programming

Rule-Based Programming
Graph Programming
CUDA Programming

Dunque un'altra delle sua caratteristiche è che si presenta come un linguaggio **pluriparadigmatico** per soddisfare diverse esigenze anche dal punto di vista dello stile di programmazione.

◄ | ►