Mathematica Avanzata

Introduzione

Quello che abbiamo visto finora, probabilmente servirà ed avanzerà per tutto quello che vi verrà in mente. Tuttavia, ci sono alcuni aspetti avanzati che è sempre bene conoscere, sia perchè a volte potranno snellirvi il compito e rendere più semplice la risoluzione di problemi avanzati, sia perchè vorrete fare i fricchettoni davanti a chi chiederà aiuto a dei guru di *Mathematica* come voi... vedremo come vengono effetivamente fatti i conti con il programma, come lavorare con funzioni pure e generiche, usare variabili locali nei programmi e così via.

Ho detto che il capitolo più lungo era quello della grafica? Mi sbagliavo, sarà questo... E stavolta non ci sono belle immagini a dimunuire la quantità di testo. Adesso si fa sul serio!!!

Cominciamo, quindi, perchè di carne al fuoco ce ne sarà davvero tanta...

Espressioni

Cosa sono

Consideriamo i seguenti comandi:

`Head[`*expr*`]`	restituisce l'head dell'espressione
`FullForm[`*expr*`]`	visualizza l'espressione nella forma piena di *Mathematica*

Vediamo cosa fanno queste funzioni. Consideriamo, per esempio, il seguente comando:

```
In[1]:= Head[{1, 2, 3}]
Out[1]= List
```

Quello che ha fatto, apparentemente, è riconoscere il suo argomento come una lista, ma effettivamente non è così:

```
In[2]:= Head[a - b * c]
Out[2]= Plus
```

A questo punto, le cose si fanno un pochetto più confuse. Cosa significa Plus, considerando fra l'altro che non compare il segno di addizione nell'espressione?

Il fatto è che *Mathematica* tratta tutto quanto viene scritto come combinazione di funzioni: quando scriviamo qualcosa come $a + b$, in realtà il programma interpreta quanto scritto come la funzione allegata a + da applicare con argomento *a*, *b*. Il comando FullForm permette di vedere meglio ciò:

```
In[3]:= FullForm[{1, 2, 3}]

Out[3]//FullForm= List[1, 2, 3]
```

Da qua, possiamo vedere che quando noi scriviamo la lista, l'interprete del kerner di *Mathematica* considera le parentesi graffe come la funzione List, avente come argomenti proprio gli elementi che compongono la lista. Vedete come la funzione List può essere chiamata con un numero arbitrario di argomenti, ma anche questo lo vedremo più avanti. Naturalmente, se effettuiamo operazioni multiple, le rispettive funzioni saranno annidate nel modo opportuno:

```
In[4]:= FullForm[Sqrt[a + b - c] / f[x, y]]

Out[4]//FullForm= Times[Power[Plus[a, b, Times[-1, c]], Rational[1, 2]],
                   Power[f[x, y], -1]]
```

Come potete vedere, l'espressione viene interpretata da Matematica come funzioni. Da questo deriva la potenza del programma in effetti. Sebbebe il funzionamento sia complicato, il concetto che sta alla base è relativamente semplice: considera l'albero delle funzioni, e poi esegue le operazioni effettuando le opportune sostituzioni nell'albero. Le regole di sostituzione variano a seconda del comando o della funzione, e, da un albero, ne risulta un altro: di conseguenza, da un input simbolico ne esce un altro simbolico. Semplice e potente. D'altronde, la potenza del programma sta nella sofisticazione delle regole di sostituzione. Nell'esempio di sopra, potete anche vedere che la forma piena tratta nella stessa maniera le funzioni definite e quelle non definite, come la f. Il fatto che non venga calcolata viene dal fatto che, non essendo definita, non esistono regole per la sua manipolazione, e *Mathematica* la lascia come tale.

```
In[5]:= Head[%]

Out[5]= Times
```

A questo punto, si comprende meglio anche il signidicato di Head. Restituisce, in pratica, il nome della funzione principale dell'espressione, quella che contiene tutte le altre. In questo caso, è appunto Times, cioè la funzione che moltiplica fra loro i suoi argomenti. Possiamo anche avere una rappresentazione più esplicita, mediante il seguente comando:

`TreeForm[`*expr*`]`	mostra l'espressione con notazione ad albero testuale

Considerando sempre l'espressione di prima, in questo caso abbiamo:

```
In[6]:= TreeForm[Sqrt[a + b - c] / f[x, y]]
```

```
Out[6]//TreeForm=
Times[
      Power[
            Plus[a, b,
                 Times[-1, c]]
            , Rational[1, 2]]
,
 Power[
       f[x, y]
       , -1]]
```

Il comando mostra una rappresentazione in caratteri ASCII dell'albero di parsing dell'espressione. Questo permette di avere una rappresentazione più chiara della nidificazione delle funzioni.

Inoltre, possiamo vedere come, nell'interpretazione della forma normale in cui scriviamo, in notazione FullForm, *Mathematica* sia in grado di riconoscere correttamente le precedenze, applicando Times prima di Sqrt, per esempio. Questo permette di eeguire correttamente le espressioni, ed il suo parsing ci evita di dover scrivere esplicitamente queste condizioni standard per la matematica.

Quando definiamo una funzione, possiamo anche definire il modo in cui deve essere scritta: Una volta dato il nome alla funzione, possiamo scriverla nei seguenti modi:

$f[x, y]$	notazione standard $f[x, y]$
$f @ x$	notazione prefissa $f[x]$
$x // f$	notazione postfissa $f[x]$
$x \sim f \sim y$	notazione infissa $f[x, y]$

Possiamo quindi vedere come, una volta definita una funzione, possiamo scriverla in uno dei modi seguenti. Da questo si vede come possiamo scrivere in maniera differente le funzioni, che *Mathematica* interpreta sempre in notazione Standard. Effettivamente, abbiamo già visto i diversi modi di scrivere una funzione:

```
In[7]:= {{1, 2}, {3, 4}} // MatrixForm
```

```
Out[7]//MatrixForm=
1  2
3  4
```

corrisponde a:

```
In[8]:= MatrixForm[{{1, 2}, {3, 4}}]
```

```
Out[8]//MatrixForm=
1  2
3  4
```

```
In[9]:= MatrixForm@{{1, 2}, {3, 4}}
```

$$\text{Out[9]//MatrixForm=} \begin{pmatrix} 1 & 2 \\ 3 & 4 \end{pmatrix}$$

Questo modo di trattare le espressioni in *Mathematica* ha un importante aspetto, cioè quello di poter trattare separatente le sue parti. Consideriamo, per esempio:

```
In[10]:= a = {1, 2, 3, 4, 5}
Out[10]= {1, 2, 3, 4, 5}
```

Sappiamo come fare per estrarre un elemento dalla lista:

```
In[11]:= a[[4]]
Out[11]= 4
```

Tuttavia, sappiamo ora anche come vengono trattate le liste in *Mathematica*:

```
In[12]:= FullForm[a]
Out[12]//FullForm= List[1, 2, 3, 4, 5]
```

A questo punto, possiamo dire che le doppie parentesi quadre, che corrispondono al comando Part in notazione piena, funzionano effettivamente per qualsiasi espressione nella forma che abbiamo visto, cioè con una funzione ed il numero di argomenti. Il comando estrare e visualizza l'n-simo argomento della funzione.

Ipotizziamo di avere la seguente somma:

```
In[13]:= somma = h + t + q + w + r + p
Out[13]= h + p + q + r + t + w

In[14]:= FullForm[somma]
Out[14]//FullForm= Plus[h, p, q, r, t, w]
```

Possiamo quindi pensare di usare lo stesso comando, per poter estrarre un generico argomento della funzione:

```
In[15]:= somma[[3]]
Out[15]= q
```

E questo è quello che accade. Vedete anche che, nonostante la somma goda della proprietà commutativa, la funzione abbia comunque gli argomenti ordinati per come li abbiamo scritti. Questo permette di poter effettuare con facilità queste operazioni anche nelle espressioni: la proprietà commutativa viene applicata nella definizione della somma. Non è detto che in un qualsiasi funzione f gli argomenti godano di proprietà commutativa, e quindi è giusto mantenerli ordinati.

Nella stessa maniera, possiamo anche considerare di estrarre parti annidate da più funzioni: supponiamo di avere un'espressione del tipo:

```
In[16]:= es = t + u^(x^2 + v x + i)

Out[16]= t + u^(i+v x+x^2)
```

```
In[17]:= TreeForm[es]

Out[17]//TreeForm= Plus[t, |                                    ]
                        Power[u, |                            ]
                                 Plus[i, |          , |         ]
                                         Times[v, x]  Power[x, 2]
```

Possiamo vedere che, se vogliamo estrarre l'esponente, basta indicare con un opportuno indice quello che vogliamo. Vediamo che, nel primo livello, l'esponente si trova nella posizione 2: nel secondo livello, alla posizione 2. Per cui, per estrarlo, possiamo scrivere:

```
In[18]:= es[[2, 2]]

Out[18]= i + v x + x^2
```

E abbiamo ottenuto l'esponente. Analogamente, potremmo anche sostituirlo. Per esempio, pensiamo di aver sbagliato, e che in realtà sia una cubica, e ci siamo scordati di scrivere $w\,x^3$: a questo punto, basta fare:

```
In[19]:= es[[2, 2]] += w x^3

Out[19]= i + v x + x^2 + w x^3
```

```
In[20]:= es

Out[20]= t + u^(i+v x+x^2+w x^3)
```

Al posto di andare a modificare l'espressione con il mouse, abbiamo introdotto la modifica direttamente dal notebook con un opportuno comando. Notate anche come, per brevità, abbia usato +=, comando assolutamente analogo a quanto compare in C. D'altronde, se avete perso tempo a programmare in *Mathematica* dovreste essere a conoscenza di questo simbolo.

Possiamo usare anche le regole per modificare la struttura delle espressioni. Per esempio, se volessimo trasformare tutte le somme in moltiplicazioni, basterebbe fare:

```
In[21]:= es /. Plus → Times
Out[21]= t u^(i v w x^6)
```

Cambiando il nome delle funzioni opportune:

```
In[22]:= FullForm[%]
Out[22]//FullForm= Times[t, Power[u, Times[i, v, w, Power[x, 6]]]]
```

Qua si può vedere meglio come abbiamo effettuato le sostituzioni delle funzioni Plus.

Possiamo anche applicare le altre funzioni tipiche delle liste:

```
In[23]:= Position[es, x]
Out[23]= {{2, 2, 2, 2}, {2, 2, 3, 1}, {2, 2, 4, 2, 1}}
```

Indica le posizioni, nella nidificazione, dove compare l'elemento x. Compare tre volte, ed infatti abbiamo tre liste di indici.

A questo punto, si tratta solo di trovare il problema che può essere risolto efficacemente in questa maniera. Tuttavia, a me è capitato raramente di dover usare queste notazioni, e soprattutto quando si trattava di correggere alcuni particolari errori. Probabilmente sono ancora troppo ingnorante per usarle al mio livello...

Valutazione delle espressioni

Come abbiamo potuto vedere, ci sono un sacco di funzioni in *Mathematica*, molte più di quanto ve ne servirebbero... Tutte, però, godono di una proprietà importante, e cioè quella di restituire sempre il risultato standard. Per esempio, vengono tolte automaticamente le parentesi dalle somme:

```
In[24]:= (a + b + (c + t + a))
Out[24]= {2 + b + c + t, 4 + b + c + t, 6 + b + c + t, 8 + b + c + t, 10 + b + c + t}
```

Questo funziona, in genere, per tutte le funzioni associative, e comunque per funzioni che godono di determinate proprietà. Naturalmente (e lo vedremo fra qualche pagina), anche noi siamo in grado di creare funzioni che sfruttino le medesime proprietà, come quella commutativa.

Per valutare le espressioni, *Mathematica* prima valuta e capisce cosa rappresenta l'head principale, e poi va a valutare, ricorsivamente, i singoli elementi dell'espressione. Per esempio, se scriviamo qualcosa come:

```
In[25]:= seno = Sin;

In[26]:= Simplify[Cos[x]^2 + seno[x]^2]

Out[26]= 1
```

Vediamo come abbia riconosciuto che seno rappresenta Sin, e poi abbia valutato tutto quanto. Per vedere le sottoespressioni utilizzate nella valutazione, possiamo utilzzare il seguente comando:

`Trace[`*espr*`]`	restituisce la lista delle sottoespressioni generate durante il calcolo dell'espressione

```
In[27]:= Trace[Simplify[Cos[x]^2 + seno[x]^2]]

Out[27]= {{{{{seno, Sin}, Sin[x]}, Sin[x]^2}, Cos[x]^2 + Sin[x]^2},
          Simplify[Cos[x]^2 + Sin[x]^2], 1}
```

Come possiamo vedere, prima passa da seno a Sin, applicandolo poi alla corrispondente funzione, e poi ne calcola il quadrato. Poi fa la somma fra i due addendi, ed applica la semplificazione. Esattamente la ricorsività che ci aspettavamo dalla valutazione standard.

Possiamo, volendo, anche cambiare la valutazione standard esplicitamente, associando delle specifiche regole a delle specifiche funzioni: per esempio, in questa espressione:

```
In[28]:= f[g[x]]

Out[28]= f[g[x]]
```

Vediamo che, dalla natura ricorsiva, prima viene applicata la g, con le sue regole e definizioni, e poi la f. Se creiamo delle regole per questo particolare pattern, possiamo associarle in modo distinto alle due funzioni:

```
In[29]:= f /: f[g[x_]] := regolaf[x]

In[30]:= g /: f[g[x_]] := regolag[x]

  — General::spell1 : Possible spelling error: new symbol
       name "regolag" is similar to existing symbol "regolaf". More…
```

Adesso, per lo stesso pattern dell'espressione, abbiamo due differenti regole. Dato che, però, la *g* viene valutata prima nell'ordine delle espressioni, otterrò:

```
In[31]:= f[g[z]]
Out[31]= regolag[z]
```

Viene applicata la regola associata alla g. Cancelliamola, e vediamo che succede:

```
In[32]:= Clear[g]; f[g[z]]
Out[32]= regolaf[z]
```

In questa maniera, prima viene valutata la *g* ma, dato che adesso non ha più nessuna definizione e nessuna regola, la sua valutazione lascia l'espressione inalterata. Dopo viene valutata la *f* che, riconoscendo il particolare tipo di espressione, applica la sua regola.

`f/:espr:=def`	applica la definizione specificata quando la f compare nell'espressione specificata

Lavorando con queste regole, siamo in grado di modificare a piacimento, e nel modo che ci serve, il modo in cui vengono calcolate determinate espressioni, dando il risultato nella forma che ci serve.

Un altro modo per modificare il calcolo delle espressioni consiste nello specificare quali parti debbano essere calcolate e quali no:

`Hold[`*expr*`]`	lascia *expr* in forma non valutata
`HoldComplete[`*expr*`]`	lascia *expr* non valutata ed evita che venga modificata da ulteriori calcoli a livelli superiori
`HoldForm[`*expr*`]`	mantiene *expr* non valutata, e scrive l'espressione senza l'head `HoldForm`
`ReleaseHold[`*expr*`]`	rimuove `Hold` e `HoldForm` da *expr*
`Extract[`*expr*`, `*index*`, Hold]`	prende una parte di *expr*, modificandola con `Hold` per prevenire la valutazione
`ReplacePart[`*expr*`, Hold[`*value*`], `*index*`, 1]`	sostituisce una parte di *expr*, estraendo *value* senza valutarlo

Consideriamo, per esempio, la seguente espressione:

```
In[33]:= 3 + 4 + 5 + Sin[4 * 7]
Out[33]= 12 + Sin[28]
```

Vogliamo, adesso, valutarla, ma stavolta non vogliamo che venga calcolato l'argomento del seno:

```
In[34]:= 3 + 4 + 5 + Sin[Hold[4 * 7]]
Out[34]= 12 + Sin[Hold[4 7]]
```

Vediamo come non venga valutato, Tuttavia, se ci interessa un output più pulito, conviene utilizzare HoldForm:

```
In[35]:= 3 + 4 + 5 + Sin[HoldForm[4 * 7]]
Out[35]= 12 + Sin[4 7]
```

In questa maniera, nascondiamo l'head, che è utile quando magari dobbiamo stampare la formula (per esempio quando la esportiamo in LaTeX oppure in Publicon).

L'ordine della valutazione è importante speciamente quando andiamo a usare espressioni in confroni e specialmente nei pattern (che vedremo fra poco). Supponiamo, per esempio, di vole creare una regola per modificare una funzione quando compare un determinato argomento. Per esempio, vorremmo fare questo:

```
In[36]:= regola = f[x_ + 3 * 5] → Exp[x];
  – General::spell : Possible spelling error: new symbol
      name "regola" is similar to existing symbols {regolaf, regolag}. More…
```

Vediamo quando la regola viene applicata:

```
In[37]:= f[4] /. regola
Out[37]= f[4]

In[38]:= f[w + 15] /. regola
Out[38]= ℯ^w
```

Come possiamo vedere, in quest'ultimo caso *Mathematica* ha considerato equivalenti gli argomenti delle funzioni, anche se nel primo caso abbiamo scritto una moltiplicazione, e nel secondo invece un numero.

```
In[39]:= f[i + 12 + 3] /. regola
Out[39]= ℯ^i
```

Vediamo che anche in questo caso abbiamo ottenuto la sostituzione, perchè la somma contenuta nell'argomento, se valutata, coincide sempre con quella valutata della regola. Quindi gli argomenti coincidono perchè, analogamente all'argomento della funzione, anche il contenuto della regola viene prima valutato. Possiamo lasciarlo in forma non valutata, se lo specifichiamo:

```
In[40]:= regola2 = HoldForm[f[x_ + 3 * 5]] → Exp[x];
```

In questo caso, evitiamo che venga valutata, impedendo di eseguire la moltiplicazione. Di conseguenza gli argomenti delle funzioni, pur essendo comunque equivalenti, non combacieranno più nel modo in cui sono scritte, e la regola non viene applicata:

```
In[41]:= f[w + 15] /. regola2
Out[41]= f[15 + w]
```

Invece, se andiamo a creare un argomento strutturalmente identico a quello ottenuto prima, la sostituzione avviene comunque, perchè in questo caso abbiamo veramente due argomenti che sono identici:

```
In[42]:= f[w + 3 5] /. regola2
Out[42]= f[15 + w]
```

Come vediamo, neanche in questo caso combaciano. Perchè? Sebbene abbiamo imposto la non valutaione della regola, la funzione viene comunque valutata prima di applicare la regola. Dobbiamo quindi lasciare senza valutazione anche l'argomento:

```
In[43]:= HoldForm[f[w + 3 5]] /. regola2
Out[43]= ℮^w
```

In questo caso si vede che strutturalmente gli argomenti coincidono, oltre che algebricamente, per cui la sostituzione viene regolarmente effettuata. Notiate, quindi, come sia necessaria una certa attenzione quando dobbiamo effettuare operazioni che coinvolgano l'ordine ed il momento in cui le espressioni vengono valutate, perchè ciò potrebbe portare ad un risultato diverso da quello che volevamo.

Possiamo anche lasciare inalterata solamente la parte destra della regola di sostituzione:

lhs -> *rhs*	valuta entrambi *lhs* e *rhs*
lhs :> *rhs*	lascia senza valutazione *rhs*

Supponiamo di avere quest'espressione:

```
In[44]:= Hold[x] /. x → 3 + 7
Out[44]= Hold[10]
```

Come potete vedere, tutto va per il verso giusto:

```
In[45]:= Hold[x] /. x :→ 3 + 7
Out[45]= Hold[3 + 7]
```

Come potete vedere, nel primo caso prima viene valutata la regola, poi viene applicata: in questa maniera Hold ha già come argomento 10. Nel secondo caso, invece, viene sostituita la regola senza averla valutata, quindi non andrò a sostituire con 10, ma con 3 + 7. Di conseguenza, Hold avrà come argomento 3 + 7, che resterà non valutato, come abbiamo appena visto.

Ripeto, quindi: quando volete eseguire e creare regole complicate, state sempre attenti all'ordine di valutazione delle espressioni, mettendo magari le cose nel modo che preferite utilizzando nei punti giusti Hold assieme alle sue varianti.

Capito?

- **Pattern**

Cosa sono

I pattern sono un metodo avanzato per l'utilizzo delle espressioni e, fra l'altro, sono uno dei motivi principali della potenza di calcolo simbolico offerta da *Mathematica*. Permettono, ua volta capito bene il concetto, di definire funzioni che creano operazioni algebriche simboliche di qualsiasivoglia complessità. Consideriamo quello che succede quando definiamo una funzione:

```
In[46]:= f[x_] := x^4 + Sin[x]
```

Quando andiamo ad usarla, al posto di x_ andiamo a mettere quello che ci serve:

```
In[47]:= f[{1, 45, 5}]
Out[47]= {1 + Sin[1], 4100625 + Sin[45], 625 + Sin[5]}

In[48]:= f[Cos[r]]
Out[48]= Cos[r]^4 + Sin[Cos[r]]

In[49]:= Clear[f]
```

Tuttavia, come abbiamo visto prima, quello che andiamo ad introdurre come argomento di una funzione, qualsiasi forma sia, è un'espressione: Anche i numeri stessi, sebbene non appaia in FullForm, hanno un head:

```
In[50]:= Head[5]
Out[50]= Integer

In[51]:= Head[4.6]
Out[51]= Real
```

Qualsiasi espressione è accetta come argomento, e questo è dovuto a x_ che rappresenta un pattern: in altre parole, pattern sta per qualsiasi cosa. Possiamo considerarlo, per chi programma, come la rappresentazione della radice dell'albero dell'espressione. Quando andiamo a definirlo in una funzione, come argomento, diciamo al programma che al posto dell'argomente può andare bene qualsiasi espressione. Quindi, il pattern rappresenta soltanto un'espressione qualsiasi.

Un caso in cui si possono usare i pattern riguarda le regole di sostituzione: supponiamo, per esempio, di avere la seguente lista:

```
In[52]:= lista = {a^r, b^(c^2 + e + b x), v, 34, 23^t}
Out[52]= {{1, 2^r, 3^r, 4^r, 5^r}, b^(c^2+e+b x), v, 34, 23^t}
```

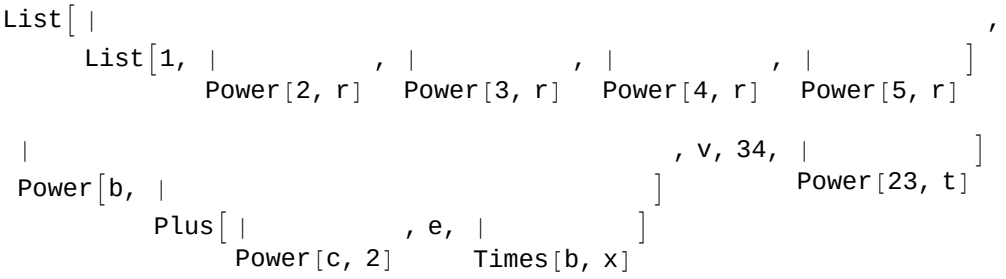
Possiamo trovare, in questo caso, sostituire tutti gli elementi in cui compare un esponente, di qualsiasi tipo esso sia:

```
In[53]:= lista /. x_^y_ → esponente
Out[53]= {{1, esponente, esponente, esponente, esponente},
          esponente, v, 34, esponente}
```

Quello che abbiamo fatto è abbastanza comprensibile: abbiamo utilizzato un'espressione, in cui era presente una regola di sostituzione così definita: se ci sono elementi formati da una qualsiasi espressione, elevata ad un'altra espressione qualsiasi, allora sostituisci l'elemento. In questo caso ci basiamo sulla struttura dell'espressione, pià che sul suo contenuto. Mentre prima potevamo sostituire casi in cui, per esempio, l'esponente era un incognita specifica, adesso possiamo introdurre il concetto di espressione generica nella sostituzione.

In[54]:=
```
TreeForm[lista]
```

Out[54]//TreeForm=



In questo caso, quello che *Mathematica* fa è trovare nell'albero della espressione le funzioni che combaciano col pattern: in questo caso x_^y_ sta per Power[x_,y_]. A questo punto, quando le trova, sostituisce tutta la funzione (non solo gli argomenti, quindi), con quello che definisce la regola. Imparare ad usare i pattern significa possedere strumenti molto potenti per poter effettuare manipolazioni sulle espressioni, concentrandoci sulla struttura delle ultime. In questo modo si possono definire, per esempio, regole di sostituzione se volessimo creare una funzione che faccia l'integrale dell'argomento. Nell'help di *Mathematica* c'è un esempio al riguardo, nella sezione Patterns, che vi consiglio di andare a vedere.

Possiamo anche usare le regole di sostituzione per poter modificare le espressioni mantenendo i pattern:

In[55]:=
```
f[x] + t[x] + c[x^2, r^5] + c[4] /. c[x_, y_] → x + y
```

Out[55]=
$$r^5 + x^2 + c[4] + f[x] + t[x]$$

Qua possiamo vedere come si usa una regola di sostituzione per modificare le funzioni usando i pattern stessi. *Mathematica* cerca la forma *c*[x_, y_] nell'espressione e, quando la trova, la modifica con $x + y$. Notate come la funzione *c*[4] non viene toccata, in quanto non corrispondono i pattern. Infatti, pur essendo la stessa funzione, viene chiamata solamente con un argomento.

Un'altra cosa importante da capire è il fatto che i patten si basano sulla struttura delle espressioni, e che bisogna impararle bene. supponiamo di avere la seguente lista:

In[56]:=
```
lista = {a, t / y, Sin[c] / (6 f), 1 / p}
```

Out[56]=
$$\left\{\{1, 2, 3, 4, 5\}, \frac{t}{y}, \frac{\text{Sin}[c]}{6\,f}, \frac{1}{p}\right\}$$

Eseguiamo, adesso, una sostituzione mediante i pattern:

```
In[57]:= lista /. x_ / y_ → x y
```

$$\text{Out[57]= } \left\{\{1, 2, 3, 4, 5\}, t\, y, \frac{1}{6} f \text{Sin}[c], \frac{1}{p}\right\}$$

Quello che succede non è esattamente quello che ci aspettavamo, volevamo che tutti i rapporti fossero stati modificati a prodotti: invece, soltanto il secondo elemento si è comportato come volevamo, mentre il terzo è stato modificato soltanto parzialmente (il 6 è rimasto al denominatore), mentre il quarto è rimasto tale e quale a prima. Questo accade perchè, sebbene in apparenza appaiano simili come struttura, in realtà non lo sono dal punto di vista delle espressioni: Andiamo a vedere la forma degli elementi:

```
In[58]:= TreeForm[t / y]
```

```
Out[58]//TreeForm=
Times[t, Power[y, -1]]
```

```
In[59]:= TreeForm[Sin[c] / (6 f)]
```

```
Out[59]//TreeForm=
Times[Rational[1, 6], Power[f, -1], Sin[c]]
```

```
In[60]:= TreeForm[1 / p]
```

```
Out[60]//TreeForm=
Power[p, -1]
```

Adesso, vediamo la struttura del pattern:

```
In[61]:= TreeForm[x_ / y_]
```

```
Out[61]//TreeForm=
Times[Pattern[x, Blank[]], Power[Pattern[y, Blank[]], -1]]
```

Vediamo che i pattern in Tree e FullForm, sono visti come funzioni che si chiamano Pattern, per l'appunto. Se si osservano bene tutte queste strutture, possiamo vedere come, effettivamente, solamente il secondo elemento della lista disponga di una struttura analoga a quella definita dal pattern: Infatti, è nella forma Times[x_, Power[y_, −1]]. E viene sostituita dalla regola del pattern in maniera corretta. Il terzo elemento della lista, invece, ha la seguente forma: Times[x_, Power[y_, −1], z_], il che non corrisponde esattamente a quanto visto in precedenza, dato che il pattern ha due argomenti, se pur qualsiasi, mentre l'espressione ha tre argomenti qualsiasi in Times. Allora esegue solo una sostituzione, dando quindi il risultato mostrato. Nel quarto elemento della lista, invece, sebbene appaia il simbolo di divisione, l'albero dell'espressione è differente, e non combacia con quello del pattern, vanificando quindi il nostro tentativo di sostituzione, perchè non è

nella forma corretta.

Si nota, quindi, come bisogna prestare attenzione, in alcuni casi, per la corretta manipolazione. In effetti, anche x ed x^2 possono essere visti come casi di x^n_, dove nel primo caso il pattern rappresenta l'unità; tuttavia, pur essendo matematicamente equivalenti, non lo sono dal punto di vista della struttura, per cui è consigliabile, prima, cercare di modificare l'espressione nella forma voluta, prima di effettuare le operazioni con i pattern, che sono, come abbiamo visto, molto sensibili alle differenti strutture (essendo, a tutti gli effetti, essi stessi delle strutture).

Inoltre, possiamo vedere che, anche se non esplicitamente, anche i valori numerici possono essere visti come strutture. Anche se la rappresentazione non lo da a vedere:

```
In[62]:= FullForm[45.3]

Out[62]//FullForm= 45.3`
```

In effetti anche i numeri hanno un head, che definisce il tipo di numero con cui abbiamo a che fare:

```
In[63]:= Head[5]

Out[63]= Integer

In[64]:= Head[5 / 9]

Out[64]= Rational

In[65]:= Head[1 - I]

Out[65]= Complex
```

e così via. Questo è utile, come vedremo più avanti, per poter definire particolari tipi di funzioni.

Ricerca di Pattern, e modifica

Una volta definiti i pattern, e capito cosa sono, una delle prime applicazioni che ci vengono in mente riguardano proprio la ricerca: per certi versi è simile a quanto visto finora, solo che nelle ricerche, al posto di rispondere alla domanda "Quali espressioni contengono questi valori?" si risponde alla domanda "Quali espressioni hanno questa struttura?". Il cambio della domanda è importante, e permette di trovare soluzioni difficilmente ottenibili per altra via.

Alcuni dei comandi più utilizzati sono i seguenti:

`Cases[`*list*`, `*form*`]`	restituisce gli elementi di *list* che corrispondono a *form*
`Count[`*list*`, `*form*`]`	restituisce il numero di elementi in *list* che corrispondono a *form*
`Position[`*list*`, `*form*`, {1}]`	restituisce le posizioni degli elementi in *list* che corrispondono a *form*
`Select[`*list*`, `*test*`]`	restituisce gli elementi di *list* per i quali *test* è `True`
`Pick[`*list*`, `*sel*`, `*form*`]`	restituisce gli elementi di *list* per i quali i corrispondenti elementi di *sel* corrispondono a *form*

Come possiamo vedere, sono pressochè analoghi alle funzioni già viste, con l'opprortuna modifica di trattare pattern invece che elementi.

Riprendiamo una lista che avevamo visto qualche pagina prima:

In[66]:=
```
lista = {a^r, b^(c^2 + e + b x), v, 34, 23^t};
```

Supponiamo di voler ottenere una lista degli elementi aventi la forma esponenziale:

In[67]:=
```
Cases[lista, x_^c_]
```

Out[67]=
$$\left\{b^{c^2+e+b\,x},\ 23^t\right\}$$

Abbiamo ottenuto la lista che volevamo. Naturalmente, abbiamo visto che queste funzioni non si applicano solamente alle liste:

In[68]:=
```
espr = x^3 + b^5 * c^(x + 9) + Sin[x] + Cos[s^3];
```

In[69]:=
```
Cases[espr, x_^c_]
```

Out[69]=
$$\{x^3\}$$

Se avete letto attentamente finora, dovreste aver capito che la risposta è corretta: infatti, nel secondo termine compare Times, con la c, e poi gli altri esponenziali sono nidificati in altre funzioni, mentre

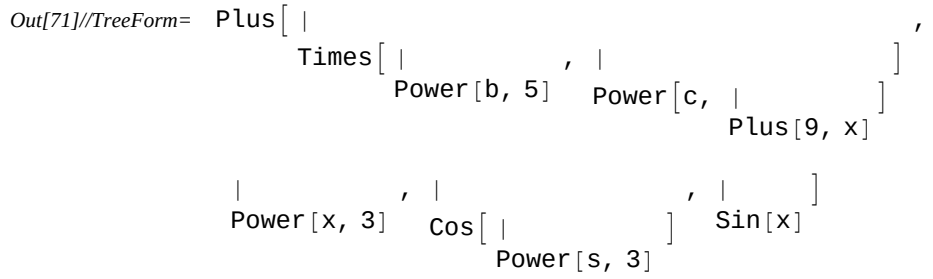
così com'è Cases si applica solo al primo livello dell'albero. Se vogliamo che funzioni su più livelli, dobbiamo specificarlo:

```
In[70]:= Cases[espr, x_^c_, Infinity]

Out[70]= {b^5, c^(9+x), x^3, s^3}
```

In questo caso, abbiamo definito che il comando cercata tutti i pattern in tutti il livelli dell'espressione; nel casi particolare, vediamo come un esponenziale compaia anche a sinistra della moltiplicazione, quindi ad un livello superiore, e compaiano esponenti anche all'interno di Cos:

```
In[71]:= TreeForm[espr]
```



Analizzandolo, vedrete come abbiamo potuto prendere tutti i casi possibili. Potevamo, naturalmente, fermarci al livello che ci interessava, specificando il numero, invece che ∞, che indica tutti i livelli dell'espressione. Tenete sempre conto, però, che non esiste numero di livelli che possa risolvere il problema di non corrispodenza di pattern che abbiamo visto poco fa.

```
In[72]:= Cases[espr, x_Integer, Infinity]

Out[72]= {5, 9, 3, 3}
```

Qua abbiamo fatto una cosa leggermente diversa; avevamo visto che qualsiasi cosa ha un head, anche i numeri. Specificando l'head nel pattern, gli abbiamo detto di estrapolare dall'espressione tutte le sottoespressioni aventi il corrispondente head. Questo può essere più semplice, a volte, che andare a ricostruire il pattern tramite gli operatori:

*x*_*h*	un espressinone avente head *h*
*x*_Integer	un numero intero
*x*_Real	numero reale approssimato
*x*_Complex	numero complesso
*x*_List	lista
*x*_Symbol	simbolo (incognita)
x_Rational	numero razionale

Possiamo vedere come possiamo riconoscere sia il tipo di valore numerico, sia il tipo di espressione da un punto di vista più generale.

```
In[73]:= Cases[espr, x_Power, Infinity]
Out[73]= {b^5, c^(9+x), x^3, s^3}
```

In questo caso siamo andati a beccare il pattern Power, cioè abbiamo estratto tutte le sottoespressioni aventi Power come head.

```
In[74]:= Position[espr, _Power]
Out[74]= {{1, 1}, {1, 2}, {2}, {3, 1}}
```

In questo caso, invece, abbiamo visto le posizioni che occupano nell'espressione le sottoespressioni Power. Avete anche notato come il pattern, in questo caso, sia privo di nome. Effettivamente, se non bisogna usarlo, ma solamente riconoscerlo, possiamo anche farne a meno: consideriamo questa nuova lista:

```
In[75]:= lista = {Sin[Cos[x]], c^3, 4 / (r^3)};
```

Supponiamo di voler sostituire i pattern esponenziali con un altro valore definito:

```
In[76]:= lista /. _^_ → cost
Out[76]= {Sin[Cos[x]], cost, 4 cost}
```

In questo caso, nella sostituzione non abbiamo avuto bisogno di usare i pattern che avevamo. Adesso, invece, supponiamo di voler scambiare mantissa ed esponente:

```
In[77]:= lista /. x_^y_ → y^x
Out[77]= {Sin[Cos[x]], 3^c, 4 (-3)^r}
```

Adesso, invece, avevamo due pattern distinti, e dovevamo riutilizzarli per poter modificare le espressioni nel modo che volevamo. In questo caso, dovevamo specificare quale pattern rappresentava la mantissa, quale l'esponente, e poi li riutilizzavo. Un poco come definire delle variabili ma, mentre le variabili hanno memorizzati dei dati o espressioni, i pattern siffatti memorizzano solamente strutture.

_	qualsiasi espressione
*x*_	un'espressione qualsiasi che verrà chiamata *x*
x:*pattern*	un'espressione che verrà chiamatra *x*, che soddisfa il *pattern*

L'ultima espressione delle tre è nuova, ed ancora non l'avevamo vista: mentre negli altri due casi avevo una rappresentazione di un pattern qualsiasi, nel terzo caso *x* rappresenta non un pattern, ma la struttura che lega i pattern fra di loro: se scriviamo *x* : r_ / e_, r ed e rappresentano i pattern, mentre x rappresenta l'espressione che li lega, ovvero la divisione.

In[78]:=
```
espr = 4 / (a + b)
```

Out[78]= $\left\{\frac{4}{1+b}, \frac{4}{2+b}, \frac{4}{3+b}, \frac{4}{4+b}, \frac{4}{5+b}\right\}$

In[79]:=
```
espr /. x : z_ + t_ → x / t
```

Out[79]= $\left\{\frac{4}{1+b}, \frac{8}{2+b}, \frac{12}{3+b}, \frac{16}{4+b}, \frac{20}{5+b}\right\}$

Quello che abbiamo fatto è questo: abbiamo scritto l'espressione, poi l'abbiamo modificata tramite le regole con il pattern. Abbiamo visto dove compariva una somma, chiamando x la somma stessa, e siamo andati a sostituirla con il rapporto fra la somma stessa ed il suo addendo. Dato che la somma compare al denominatore, il divisore della regola di sostituzione comparirà al numeratore. Il vantaggio consiste nel fatto che, invece di andare a riscrivere il pattern per intero nella parte destra della regola di sostituzione, è bastato usare il suo nome. Questo può semplificare parecchio la scrittura di queste regole avanzate.

Inoltre, pattern con lo stesso nome indicano pattern che sono identici in tutto:

In[80]:=
```
lista = {a Sin[a], b Sin[a], v, b a Sin[b]};
```

In[81]:=
```
Cases[lista, _ Sin[_]]
```

Out[81]= { }

In[82]:=
```
Cases[lista, x_ Sin[x_]]
```

Out[82]= { }

Come possiamo vedere qua sopra, nel primo caso il comando restituisce tutti gil elementi del tipo "seno di qualcosa che moltiplica qualcos'altro". Nel secondo caso, invece, i due pattern che usiamo hanno lo stesso nome, per cui non sono devono essere pattern, ma devono essere lo stesso pattern. Restituisce, in pratica, gli elementi del tipo "seno di un argomento che sono moltiplicati dallo stesso argomento". I pattern devono coincidere perfettamente, e ribadisco perfettamente, come potete notare dal fatto che l'ultimo comando non restituisce l'ultimo elemento, e neanche il secondo, per il banale motivo che non hanno un pattern corrispondente.

Inoltre, possiamol anche introdurre delle condizioni per il pattern; possiamo scegliere e modificare sì un determinato pattern, ma anche un pattern con delle limitazioni ben specifiche. Per esempio, sempre considerando gli esponenti, possiamo sostituire il pattern solo se l'esponente è un numero positivo, lasciando tutto inalterato quando è, invece, negativo oppure simbolico:

pattern `/;` *condition*	un pattern corrispondente soltanto quando la condizione è soddisfatta
lhs `:>` *rhs* `/;` *condition*	una regola che viene applicata solamente se sono soddisfatte le condizioni
lhs `:=` *rhs* `/;` *condition*	una definizione che viene applicata solamente sotto opportune condizioni

Le condizioni, a loro volta, possono essere espressioni logiche, oppure comunque funzioni che restituiscono come risultato True oppure False:

`IntegerQ[`*expr*`]`	verifica se l'espressione è un numero intero
`EvenQ[`*expr*`]`	verifica se l'espressione è un numero pari
`OddQ[`*expr*`]`	come sopra, ma se il numero è dispari
`PrimeQ[`*expr*`]`	verifica un numero primo
`NumberQ[`*expr*`]`	un valore numerico esplicito qualsiasi
`NumericQ[`*expr*`]`	una quantità numerica
`PolynomialQ[`*expr*`, {`x_1`, `x_2`, …}]`	verifica se l'espressione è una forma polinomiale in x_1, x_2, ...
`VectorQ[`*expr*`]`	verifica se ho una lista scritta in forma di vettore
`MatrixQ[`*expr*`]`	una lista di liste rappresentanti una matrice
`VectorQ[`*expr*`, NumericQ]`, `MatrixQ[`*expr*`, NumericQ]`	verifica se *l'* espressione è un vettore (oppure una matrice) numerica
`VectorQ[`*expr*`, `*test*`]`, `MatrixQ[`*expr*`, `*test*`]`	verifica i vettori e le matrici per cui *test* restituisce `True` per ogni elemento
`ArrayQ[`*expr*`, `*d*`]`	array con profondità *d*

Supponiamo di volere una funzione definita solamente per valori positivi dell'argomento; possiamo specificare questo nella funzione:

In[83]:=
```
f[x_] := x^2 + BesselJ[4, x] /; x > 0
```

Se, adesso, andiamo a scrivere la funzione con un numero positivo, otteniamo il risultato:

In[84]:=
```
f[4.754]
```

Out[84]=
```
22.9746
```

Se, invece, il numero è negativo, non soddisfa le condizioni, per cui la funzione non viene valutata:

```
In[85]:= f[-9.214]

Out[85]= f[-9.214]
```

In questo caso viene lasciata inalterata, perchè non è possibile valutarla.

```
In[86]:= f[g]

Out[86]= f[g]
```

Come possiamo vedere, anche in questo caso la funzione non viene valutata, perchè *g* rappresenta un'incognita, e *Mathematica* non è in grado di stabilire se essa rappresenta una quantità positiva, negativa o quant'altro, rispettando le regole stabilite, quindi:

```
In[87]:= f[Sin[40 °]]

Out[87]= BesselJ[4, Sin[40 °]] + Sin[40 °]^2
```

In questo caso, anche se abbiamo inserito una funzione, *Mathematica* è in grado di stabilire che il risultato è una quantità positiva, per cui riesce a valutare la funzione anche se non esplicita l'argomento, che rimane infatti in forma simbolica. Naturalmente, se volessimo, potremmo anche avere il valore approssimato, ma ormai siete troppo bravi (e pazienti...), per farvi vedere come si fa, vero????

Possiamo anche vedere come le funzioni si applicano alle liste:

```
In[88]:= f[{-3, 4, 2, -432, t, 5 r}]

Out[88]= f[{-3, 4, 2, -432, t, 5 r}]
```

Definiamo la stessa funzione, adesso, però senza la restrizione, ed applichiamo la stessa lista:

```
In[89]:= g[x_] := x^2 + BesselJ[4, x]

In[90]:= g[{-3, 4, 2, -432, t, 5 r}]

Out[90]= {9 + BesselJ[4, -3], 16 + BesselJ[4, 4], 4 + BesselJ[4, 2],
          186624 + BesselJ[4, -432], t^2 + BesselJ[4, t], 25 r^2 + BesselJ[4, 5 r]}
```

Come possiamo vedere, le due funzioni si comportano diversamente: effettivamente, uno si aspetterebbe che la funzione venisse applicata agli elementi della lista, valutandola soltanto per quegli elementi positivi. Tuttavia, possiamo notare che come argomento diamo una lista, non un numero, per cui rigorosamente non abbiamo una corrispondenza del pattern, dato che una lista non è un numero, e non possiamo quindi verificare se è maggiore o minore di 0, non viene valutato in

quando i pattern non coincidono. per risolvere questo problema possiamo usare il comando Map, che applica la funzione che ha come primo argomento, agli elementi della lista che ha come secondo argomento

```
In[91]:= Map[f, {-3, 4, 2, -432, t, 5 r}]
Out[91]= {f[-3], 16 + BesselJ[4, 4], 4 + BesselJ[4, 2], f[-432], f[t], f[5 r]}
```

Vedremo più avanti altri aspetti di Map. Per ora vi basti sapere che con questo comando possiamo applicare la funzione con le condizioni dei pattern anche ad elementi della lista.

```
In[92]:= espr = x^2 + t^5 x + 4 - 5^e;

In[93]:= espr /. x_ :> x + 6 /; IntegerQ[x]
Out[93]= 10 + 5 11^e + t^11 x + x^8
```

In quest'altro esempio abbiamo visto una sostituzione condizionata: avevamo un'espressione qualsiasi, dove comparivano anche dei numeri interi, e poi abbiamo applicato la seguente regola di sostituzione: "somma il valore 6 ad ogni elemento dell'espressione, che risulti essere un valore intero".

Tuttavia, a volte è possibile forzare il riconoscimento dei pattern, anche se in teoria non sarebbe possibile: per esempio, è possibile forzare *Mathematica* a riconoscere un incognita come un valore di un certo tipo:

```
In[94]:= h /: IntegerQ[h] = True
Out[94]= True
```

Si vede, in questo caso, che abbiamo forzato il valore h ad assumere un valore intero:

```
In[95]:= h^t /. x_ :> x + 6 /; IntegerQ[x]
Out[95]= (6 + h)^t
```

Applicando adesso la stessa regola di trasformazione di prima, vediamo che stavolta *Mathematica*, pur vedendo h sempre come un'incognita, adesso sa che, qualsiasi cosa rappresenti, sarà sempre un numero intero. In questo modo, riconosciuto il numero intero, gli va a sommare il valore 6 come da regola, mentre all'incognita t non succede niente, perchè è un incognita che rimane tale, in quanto non abbiamo specificato niente per quest'ultima.

In[96]:=
```
h^t + 3 /. x_ :> x + 6 /; NumberQ[x]
```

Out[96]=
$$9 + h^t$$

Qua possiamo vedere la rigidità di questo ragionamento: pur avendo assegnato ad h il significato di valore intero, non viene comunque riconosciuto come valore numerico, sebbene un numero intero sia anche, ovviamente, un numero; infatti la costante numerica, è riconosciuta sia come intero, sia come numero, e la regola viene applicata comunque. Questo perchè dobbiamo esplicitamente dichiarare ogni cosa che imponiamo all'incognita. Possiamo dire che un numero intero ha entrambi gli attributi, Number e Numeric, mentre h ne possiede uno soltanto. Per cui è necessario specificare ogni volta quello che vogliamo che l'incognita sia. Può essere estremamente utile quando trattiamo espressioni simboliche non definite, ma, che per esempio, sappiamo che l'argomento deve essere per forza di un certo tipo (diciamo intero). In questo modo possiamo applicare le regole anche se non abbiamo definito la funzione, permettendoci uno studio ed un'elaborazione delle formule più spedito, dato che non ci serve definire le funzioni completamente, ma solamente nelle loro proprietà che ci servono. Però, *Mathematica* non riesce, da sola, a propagare le proprietà: se, per esempio, h è intero, anche h^2 lo è, e tuttavia *Mathematica* non riesce a riconoscerlo:

In[97]:=
```
IntegerQ[h^2]
```

Out[97]=
```
False
```

Occorrono opportuni strumenti che permettano a *Mathematica* di fare queste asserzioni, cosa che vedremo sempre più avanti.

Possiamo anche fare test sulle strutture, un po' come accade negli operatori logici:

`SameQ[`*x*`, `*y*`]` or *x* `===` *y*	*x* e *y* sono identici
`UnsameQ[`*x*`, `*y*`]` or *x* `=!=` *y*	*x* e *y* non sono identici
`OrderedQ[{`*a*`, `*b*`, … }]`	*a*, *b*, ... sono in ordine standard
`MemberQ[`*expr*`, `*form*`]`	*form* combacia con un elemento di *expr*
`FreeQ[`*expr*`, `*form*`]`	*form* non compare in *expr*
`MatchQ[`*expr*`, `*form*`]`	*expr* corrisponde al pattern *form*
`ValueQ[`*expr*`]`	un valore definito per *expr*
`AtomQ[`*expr*`]`	*expr* non ha sottoespressioni

Possiamo considerare === come un operatore che, più che sui valori, lavora sulle strutture:

In[98]:=
```
q == t
```

Out[98]=
```
q == t
```

In questo caso, si lascia inalterata, perchè *Mathematica* non sa se il valore *q* corrisponde a quello di *t*:

```
In[99]:= q === t
Out[99]= False
```

In questo caso, i due pattern non sono identici (sebbene rappresentino entrambi delle incognite, avendo lettere diverse), per cui a questo test non importa il valore delle incognite, ma solamente la struttura, che è diversa:

```
In[100]:= q = 3; t = 3;

In[101]:= q === t
Out[101]= True
```

In questo caso, però, *Mathematica* esegue il confronto non fra i pattern delle espressioni, ma fra i pattern di quello che rappresentano:

```
In[102]:= Log[4^5] == 5 Log[4]

  — N::meprec : Internal precision limit $MaxExtraPrecision =
       49.99999999999999` reached while evaluating -5 Log[4] + Log[1024]. More…

Out[102]= Log[1024] == 5 Log[4]
```

OOPs..... Questo non me l'aspettavo... comunque, niente di difficile: proviamo a vederlo numericamente...

```
In[103]:= Log[4^5.] == 5. Log[4]
Out[103]= True
```

Ci siamo riusciti... Come potete vedere, il risultato è True, perchè entrambe le espressioni portano al medesimo risultato. Vediamo adesso, invece:

```
In[104]:= Log[4^5] === 5 Log[4]
Out[104]= False
```

Come potete vedere, in questo caso, sebbene abbiano lo stesso valore numerico, e rappresentino la stessa quantità, le espressioni hanno forme diverse:

```
In[105]:= FullForm[Log[4^5]]
Out[105]//FullForm= Log[1024]

In[106]:= FullForm[4 Log[5]]
Out[106]//FullForm= Times[4, Log[5]]
```

Non corrispondendo le espressioni, l'uguaglianza strutturale non è verificata, anche se lo è quella logica.

Un altro aspetto interessante, consiste nel fatto che possiamo considerare pattern alternativi: finora abbiamo considerato le regole e tutto quanto con un singolo pattern, per esempio possiamo sostituire tutti gli elementi di una lista del tipo Sin[_], con Cos[_]. Tuttavia, come possiamo fare se vogliamo che sia Sin[_], sia Log[_] siano sostutiti con la stessa quantità? I pattern alternativi esistono proprio per questo!!!

$patt_1$ | $patt_2$ | …	pattern che può assumere una delle forme specificate

Supponiamo, di avere un espressione dove compaiano sia seni che coseni, e che vogliamo cambiarli con un altra funzione:

```
In[107]:= lista = {Sin[x], Cos[s], v^u, Tan[Cos[e]], f[Sin[x]]};
```

Supponiamo di volerli cambiare con un logaritmo:

```
In[108]:= lista /. (Sin | Cos)[x_] → Log[x]
Out[108]= {Log[x], Log[s], v^u, Tan[Log[e]], f[Log[x]]}
```

Come abbiamo visto, in questo caso abbiamo sostituito tutte le funzioni trigonometriche seno e coseno. Notate, tuttavia, come la tangente, sebbene esprimibile come rapporto fra queste due funzioni, non venga toccata, grazie al fatto che nel suo pattern non compaiono le funzioni seno e coseno:

```
In[109]:= FullForm[Tan[x]]
Out[109]//FullForm= Tan[x]
```

Quindi, non è espressa in funzione di seno e coseno.

Notate anche come ho scritto la regola; dato che mi importava l'head della funzione, cioè Sin oppure Cos, l'ho scritta nella forma (Sin|Cos)[x_], per far capire meglio che intendevo uno dei due head.

Tuttavia, avrei avuto, naturalmente, lo stesso risultato anche se avessi specificato le due funzioni, cosa che magari avreste fatto voi:

```
In[110]:= lista /. Sin[x_] | Cos[x_] → Log[x]

Out[110]= {Log[x], Log[s], v^u, Tan[Log[e]], f[Log[x]]}
```

Anche in questo caso ho effettuato la sostituzione, ma ho posto maggior attenzione all'intero pattern, piuttosto che nel'head principale. Tuttavia, si tratta nella maggior parte dei casi solamente di differenti stili di scrittura. Ognuno, poi, sceglie il suo stile, esattamente come in qualsiasi linguaggio di programmazione.

Funzioni con argomenti variabili

Finora abbiamo considerato sempre funzioni specifiche, creando funzioni che abbiano sempre un numero ben determinato di funzioni: tuttavia, guardate qua:

```
In[111]:= FullForm[a + b]

Out[111]//FullForm= List[Plus[1, b], Plus[2, b], Plus[3, b], Plus[4, b], Plus[5, b]]
```

```
In[112]:= FullForm[a + b + c + d + e + f]

Out[112]//FullForm= List[Plus[1, b, c, d, e, f], Plus[2, b, c, d, e, f],
                     Plus[3, b, c, d, e, f], Plus[4, b, c, d, e, f], Plus[5, b, c, d, e, f]]
```

Come potete vedere, *Mathematica* è in grado anche di gestire funzioni che abbiano un arbitrario numero di argomenti al suo interno. Questo può essere utile, per esempio, quando si definiscono funzioni che devono lavorare su un arbitrario numero di elementi di una lista, quando si devono accodare varie operazioni etc.

Si possono specificare argomenti multipli mediante underscore multipli nella definizione dei pattern:

_	una singola espressione
*x*_	una singola espressione chiamata *x*
___	una sequenza di più espressioni
*x*___	una sequenza di più espressioni chiamata *x*
*x*___*h*	una sequenza di espressioni aventi tutti lo stesso head *h*
____	una sequenza nulla, con una o più espressioni
*x*____	una sequenza nulla, con una o più espressioni che viene chiamata *x*
*x*____*h*	una sequenza nulla, con una o più espressioni che hanno tutte come head *h*

Notate che __ è rappresentato da due underscore _ _ , mentre ___ da tre underscore _ _ _. Mi raccomando, state attenti a quanti ne mettete!!!

Naturalmente, questa generalizzazione degli argomenti permette di creare regole e definizioni più particolari e potenti. Per esempio, potremmo definire delle sostituzioni che usino funzioni con un numero diverso di argomenti:

```
In[113]:= r[a, g, b] /. r[x__] → f[x, x, x + 1, x^2, t[x]]

Out[113]= f[{1, 2, 3, 4, 5}, g, b, {1, 2, 3, 4, 5}, g,
           b, {2 + b + g, 3 + b + g, 4 + b + g, 5 + b + g, 6 + b + g},
           {1, 2^g^b^2, 3^g^b^2, 4^g^b^2, 5^g^b^2}, 3[{1, 2, 3, 4, 5}, g, b]]
```

Notiamo in questa espressione diverse cosucce: prima di tutto, vediamo come il pattern definisca, questa volta, tutti e tre gli argomenti. In questa maniera possiamo trattarli tutti in una volta. nella f, i primi due argomenti sono due volte la x, che si traduce nella ripetizione per due volte di *a*, *g*, *b*. Il terzo e quarto argomento della f, sono invece delle funzioni. Abbiamo detto a *Mathematica*, nel terzo argomento, di prendere tutto il pattern, e di sommargli uno:

```
In[114]:= r[a, g, b] /. r[x__] → x + 1

Out[114]= {2 + b + g, 3 + b + g, 4 + b + g, 5 + b + g, 6 + b + g}

In[115]:= FullForm[%]

Out[115]//FullForm= List[Plus[2, b, g], Plus[3, b, g],
                     Plus[4, b, g], Plus[5, b, g], Plus[6, b, g]]
```

In pratica, abbiamo introdotto nella funzione plus sia 1, sia il pattern, che è rappresentato dai tre argomenti. In questo modo ho ottenuto la funzione Plus con quattro argomenti, effettuando, così, la somma di tutti e quattro, come si evince dal risultato:

```
In[116]:= r[a, g, b] /. r[x__] → x^2

Out[116]= {1, 2^g^b^2, 3^g^b^2, 4^g^b^2, 5^g^b^2}

In[117]:= FullForm[%]

Out[117]//FullForm= List[1, Power[2, Power[g, Power[b, 2]]],
                     Power[3, Power[g, Power[b, 2]]],
                     Power[4, Power[g, Power[b, 2]]], Power[5, Power[g, Power[b, 2]]]]
```

In questo caso le cose sono andate diversamente. Infatto, Power è una funzione che richiete esattamente due argomenti: dato che con la nostra sostituzione non si poteva calcolare, allora sono

state gestite le cose in modo da avere una ripartizione fra gli argomenti esatta, in modo da poter creare una funzione così fatta. *Mathematica* applica iterativamente la funzione. Prima inserisce dentro la funzione Power il primo argomento, *a*, poi il secondo *b*. Dopo ancora inserisce il terzo ma, dato che non rientra nella definizione della funzione, combina il secondo ed il terzo in una nuova funzione Power, in modo che la funzione più esterna sia formata sempre da due argomenti. Infine ripete lo stesso procedimenti per il 2.

Mathematica effettua questa sostituzione solo quando sa il numero di argomenti di una funzione:

```
In[118]:= r[a, g, b] /. r[x__] → t[x]

Out[118]= 3[{1, 2, 3, 4, 5}, g, b]
```

In questo caso, infatti, la funzione *t*[*x*] non è stata definita, e *Mathematica* non è in grado di conoscere il numero dei suoi argomenti, per cui la lascia così com'è, non essendo in grado di valutarla.

Inoltre, quando si effettuano sostituzioni con pattern multipli, di solito sono possibili più combinazioni. Per esempio, se prendiamo *f*[*a*, *b*, *c*] e consideriamo *f*[x__, y__], allora posso avere sia x__ = *a*, y__ = *b*, *c*, sia x__ = *a*, *b*; y__ = *c*. In questo modo dobbiamo poter essere in grado di riconoscere tutte le combinazioni:

```
In[119]:= ReplaceList[t[a, b, c, d], t[x__, y__, z__] → {{x}, {y}, {z}}]

Out[119]= {{{{1, 2, 3, 4, 5}}, {b}, {c, d}},
           {{{1, 2, 3, 4, 5}}, {b, c}, {d}}, {{{1, 2, 3, 4, 5}, b}, {c}, {d}}}
```

Come possiamo vedere, abbiamo ottenuto in questo esempio tutte le partizioni che *Mathematica* riesce a fare con questi argomenti. Tuttavia, abbiamo considerato i pattern che devono contenere almeno un argomento. Se usiamo ___ invece di __, possiamo tener conto anche dei pattern vuoti:

```
In[120]:= ReplaceList[t[a, b, c, d], t[x___, y___, z___] → {{x}, {y}, {z}}]

Out[120]= {{{}, {}, {{1, 2, 3, 4, 5}, b, c, d}},
           {{}, {{1, 2, 3, 4, 5}}, {b, c, d}}, {{{1, 2, 3, 4, 5}}, {}, {b, c, d}},
           {{}, {{1, 2, 3, 4, 5}, b}, {c, d}}, {{{1, 2, 3, 4, 5}}, {b}, {c, d}},
           {{{1, 2, 3, 4, 5}, b}, {}, {c, d}}, {{}, {{1, 2, 3, 4, 5}, b, c}, {d}},
           {{{1, 2, 3, 4, 5}}, {b, c}, {d}}, {{{1, 2, 3, 4, 5}, b}, {c}, {d}},
           {{{1, 2, 3, 4, 5}, b, c}, {}, {d}}, {{}, {{1, 2, 3, 4, 5}, b, c, d}, {}},
           {{{1, 2, 3, 4, 5}}, {b, c, d}, {}}, {{{1, 2, 3, 4, 5}, b}, {c, d}, {}},
           {{{1, 2, 3, 4, 5}, b, c}, {d}, {}}, {{{1, 2, 3, 4, 5}, b, c, d}, {}, {}}}
```

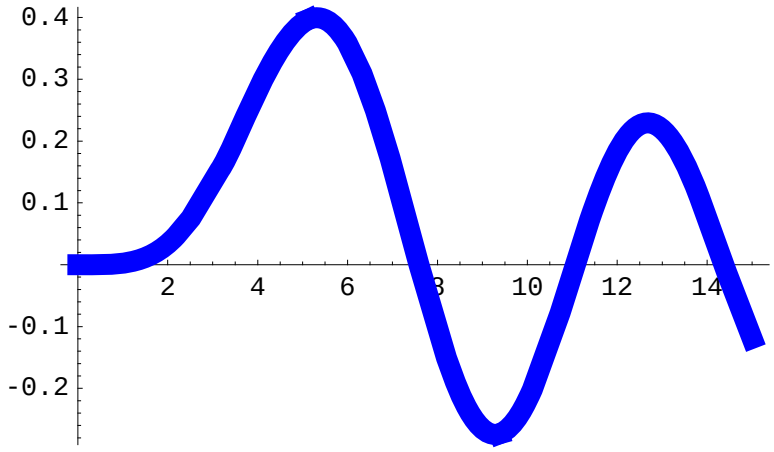
Come potete vedere, in questo caso compaiono anche pattern che non sono composti da nessun elemento, Per questo bisogna stare attenti quando si decide di usare tre underscore invece di due; è possibile infatti insorgere in qualche problema di rappresentazione, come è facile anche insorgere in

loop infiniti quandi si eseguono funzioni come ReplaceAll.

Possiamo che utilizzare degli altri modi per definire le funzioni con un numero variabile di argomenti. Per esempio, possiamo considerare il seguente esempio:

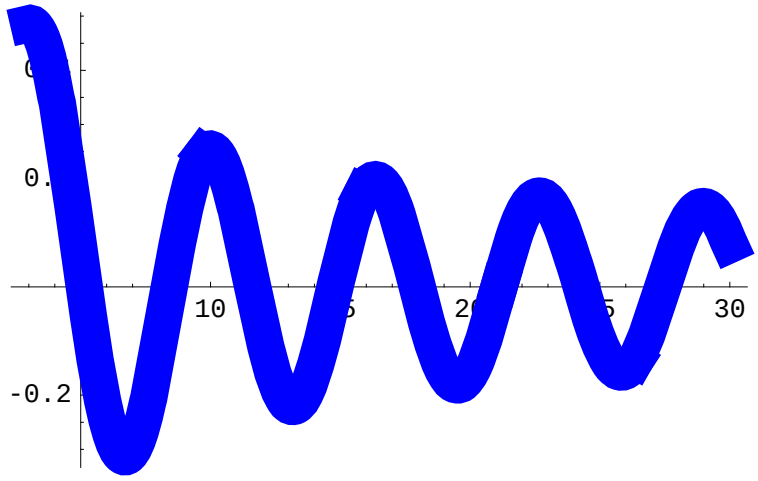
In[121]:=
```
disegno[order_: 4, start_: 0, end_: 15, thick_: 0.03] :=
 Plot[BesselJ[order, x],
  {x, start, end},
  PlotStyle → {Thickness[thick], Blue}, PlotRange → All
 ]
```

In[122]:=
```
disegno[]
```



Out[122]= - Graphics -

In[123]:=
```
disegno[2, 3, 30, .051]
```



Out[123]= - Graphics -

Come possiamo vedere, abbiamo definito una funizione con un numero variabile di argomenti. Tuttavia, ciò è vero soltanto in apparenza. In effetti, le variabili sono sempre in numero definito, solamente che abbiamo posto, questa volra, dei valori di default:

*x_*:*v*	espressione che, se omessa, viene definita con il valore di default *v*
x_h:*v*	un'espressioned con head *h* e valore di default pari a *v*
*x_*.	espressione con valore di default predefinito
x_ + y_.	valore di default pari al valore 0
x_ y_.	valore di default 1
x_^y_.	valore di default 1

Nella funzione di sopra, abbiamo definito degli argomenti aventi dei valori di default. Se, quando la funzione viene chiamata, non si specificano argomenti, allore viene disegnata con tutti i valori di default. Se invece andiamo a scrivere nuo o più argomenti, allora i valori di default nella definizione della funzione sono sostituiti da quelli che andiamo a scrivere. Inoltre, possiamo anche utilizzare, come si vede dalle ultime quattro definizioni della tabella, dei valori iniziali di default per *Mathematica*, pari all'elemento predefinito per l'operazione. Per esempio, se l'argomento viene sommato, come valore di default predefinito viene posto lo 0, mentre per la moltiplicazione viene posto pari ad 1; in pratica, è come se non lo considerassimo, dato che il risultato è lo stesso che si ha ignorando questi valori nell'espressione. Quindi la loro scelta non è stata casuale, come potete ben vedere.

Questo tipo di definizione è particolarmente utile, specialmente per le funzioni più complicate. Possono anche essere viste, sotto un determinato punto di vista, anche come delle opzioni per la funzione: certe volte è utile vederle sotto questi punto di vista, per esempio quando si definisce una funzione risolvibile con più algoritmi: se si omette l'argomento, allora si utilizza il metodo standard, altrimenti si sceglie esplicitamente il metodo. Possiamo anche esplicitare il fatto che si tratta di opzioni per la nostra funzione:

f[*x_*, *opts*___] := *value*	tipica definizione con zero, oppure più argomenti opzionali
name /. {*opts*} /. Options[*f*]	sostituzione usata per ottenere il valore di un argomento opzionale nel corpo della funzione

Possiamo definire in particolar modo esplicitamente le opzioni di default per una funzione:

```
In[124]:= Options[f] = {option1 → 12, option2 → q, option3 → 652 Sqrt[4]}
Out[124]= {option1 → 12, option2 → 3, option3 → 1304}
```

In[125]:=
```
Options[f]
```

Out[125]=
```
{option1 → 12, option2 → 3, option3 → 1304}
```

Come possiamo vedere, abbiamo le opzioni della funzione, sotto forma di regole di sostituzione. Possiamo vedere come si comportano, e in particolar modo, possiamo vedere il valore di default di un'opzione di una funzione, nella maniera solita delle regole di sostituzione:

In[126]:=
```
option2 /. Options[f]
```

Out[126]=
```
3
```

Abbiamo semplicemente sostituito il valore option2 con quello corrispondente nelle regole di sostituzione di Options[f].

Possiamo definire meglio la funzione, una volta definite le sue opzioni:

In[127]:=
```
f[x_, op___] := funzione[x,
  option1 /. {op} /. Options[f],
  option2 /. {op} /. Options[f],
  option3 /. {op} /. Options[f]]
```

In[128]:=
```
f[5]
```

Out[128]=
```
25 + BesselJ[4, 5]
```

Come possiamo vedere, se chiamiamo la funzione senza nessun argomento opzionale, cioè senza nessuna opzione, viene restituita con i valori standard:

In[129]:=
```
f[5, option1 -> "cambio!!!"]
```

Out[129]=
```
funzione[5, cambio!!!, 3, 1304]
```

In[130]:=
```
Clear[f]
```

In questo caso, sono state lasciate le opzioni di default, tranne la prima, che viene sostituita con il valore imposto da noi... Non vi sembra il modo con cui si definiscono le opzioni per le funzioni e comandi standard, come Plot? Adesso capite perchè li definiamo in quel modo? Certo che l'avete capito!!! Siete talmente bravi che mi fate paura!!!

Un modo per definire argomenti variabili, utili soprattutto nella ricerca di particolari pattern, sono i pattern ripetuti:

expr`..`	un pattern od un'altra espressione ripetuta una o più volte
expr`...`	un pattern od altra espressione ripetuta zero, una oppure più volte

Permettono di ricercare funzioni con pattern ripetuti come argomenti. Consideriamo la lista di funzioni:

```
In[131]:= lista = {f[q], f[a, a, a, a], f[f, e, e, g, h], f[e, e, r]};
```

Andiamo ad eseguire una ricerca vedendo gli elementi della lista dove compaiono patten ripetuti:

```
In[132]:= Cases[lista, f[a ..]]
Out[132]= {f[{1, 2, 3, 4, 5}, {1, 2, 3, 4, 5}, {1, 2, 3, 4, 5}, {1, 2, 3, 4, 5}]}
```

Come potete vedere, viene scelta solamente la funzione avente quel pattern ripetuto. Vediamo adesso:

```
In[133]:= Cases[lista, f[___, e .., ___]]
Out[133]= {f[f, e, e, g, h], f[e, e, r]}
```

In questo caso, siamo andati a trovare gli elementi della lista dove all'interno compare una ripetizione di argomenti *e*, in qualsiasi posizione, in mezzo, all'inizio od alla fine della dichiarazione degli argomenti. Infatti, come primo e terzo argomento del pattern ho usato il triplo underscore, che significa che possono anche esserci zerio elementi prima del pattern ripetuto, come infatti accade nell'ultimo elemento della lista, che viene comunque selezionato.

- **Operazioni su funzioni**

Introduzione

Abbiamo visto come manipolare le espressioni di qualsiasi tipo, siano esse liste, espressioni, considerato che tutte rappresentano, in fondo, la stessa cosa. Quello che vedremo adesso è una cosa analoga, fatta però per le funzioni. Al contrario delle espressioni, possono avere differenti tipi di manipolazioni. Per esempio, ci piacerebbe applicarle soltanto ad argomenti specifici, oppure iterativamente. Sebbene alcune cose si possano fare anche tramite le regole di sostituzione che abbiamo visto prima, ci sono alcune cose che possiamo fare solamente tramite altri tipi di operazioni, che adesso vedremo.

Manipolazioni base

Come possiamo vedere adesso, possiamo effettuare alcune operazioni sempre usando le regole di sostituzione:

In[134]:=
```
Sin[x] + Cos[Sin[x]] /. Sin → Log
```

Out[134]=
```
Cos[Log[x]] + Log[x]
```

In questo caso siamo andati semplicemente a cambiare l'head Sin con l'head Cos, trasformando di conseguenza tutti i seni in logaritmi. Quindi possiamo trattare gli head come espressioni. Possiamo anche assegnargli dei nomi, se vogliamo:

In[135]:=
```
pop = pip;
```

In[136]:=
```
pop[x]
```

Out[136]=
```
pip[x]
```

In questo caso, andando a memorizzare il nome della funzione, possiamo sostituirla ed usarla come ci piace. Questo può essere utile, ad esempio, quando usiamo spesso funzioni predefinite dal nome lungo, come InverseLaplaceTransform, che possiamo memorizzarla in ilp:

In[137]:=
```
ilp = InverseLaplaceTransform;
```

In[138]:=
```
ilp[s (s - 8) (s^2 - s + 3) / ((s - 4) (s^3 + 5)), s, t] // TraditionalForm
```

Out[138]//TraditionalForm=

$$-\frac{80\,e^{12}}{23}+\frac{1}{690}\,e^{-\frac{3}{2}\sqrt[3]{5}\left(2+i\sqrt{3}\right)}\left(\left(-350+215\sqrt[3]{5}-215\,i\sqrt{3}\,\sqrt[3]{5}+113\,5^{2/3}+113\,i\sqrt{3}\,5^{2/3}\right)e^{\frac{9\sqrt[3]{5}}{2}}-2\left(175+215\sqrt[3]{5}+113\,5^{2/3}\right)e^{\frac{3}{2}i\sqrt{3}\sqrt[3]{5}}+\left(-350+215\sqrt[3]{5}+215\,i\sqrt{3}\,\sqrt[3]{5}+113\,5^{2/3}-113\,i\sqrt{3}\,5^{2/3}\right)e^{\frac{3}{2}\sqrt[3]{5}\left(3+2i\sqrt{3}\right)}\right)$$

Come possiamo vedere, abbiamo usato un alias per la funzione, rendendola più corta, anche se non si può dire lo stesso del risultato...

Alla stessa maniera, possiamo utilizzare gli head come argomento di una funzione, cosa che può risultare estremamente utile in alcuni casi:

In[139]:=
```
pp[f_, x_] := f[x^2]
```

In[140]:=
```
pp[Log, 6]
```

Out[140]=
```
Log[36]
```

Esistono alcune funzioni predefinite in *Mathematica* che vogliono come argomento proprio un head:

In[141]:=
```
InverseFunction[Log]
```

Out[141]=
```
Exp
```

Questo, in particolare, restituisce la funzione inversa di quella scritta come argomento, se esiste. Puo funzionare anche con le funzioni che non sono state definite in precedenza, anche se in questo caso non restituisce l'inversa, dato che non può farlo, ma il simbolo dell'inversa:

In[142]:=
```
InverseFunction[f]
```

Out[142]=
```
f^(-1)
```

Un aspetto importante nello studio delle funzioni si ha quando bisogna iterativamente applicarle ad un argomento:

`Nest[f, x, n]`	applica la funzione *f* nidificandola *n* volte all'argomento *x*
`NestList[f, x, n]`	genera la lista `{x, f[x], f[f[x]], … }`, dove *f* è applicata iterativamente per ogni elemento, fino ad essere applicata *n* volte
`FixedPoint[f, x]`	Applica iterativamente la funzione fino *a* quando il risultato non varia più
`FixedPointList[f, x]`	genera la lista `{x, f[x], f[f[x]], … }`, fermandosi quando il risultato non varia più

Bisogna notare che *f* in questo caso rappresenta non tanto la funzione completa, quanto il suo argomento:

In[143]:=
```
Clear[g, z]
```

In[144]:=
```
Nest[g, z, 7]
```

Out[144]=
```
g[g[g[g[g[g[g[z]]]]]]]
```

Questo permette, come potete vedere, di applicare la funzione per un determinato numero di volte. E' necessario a volte, perchè, come potete vedere qua sotto, se la variabile z non è definita, l'assegnazione $z = g[z]$ porta ad un errore di ricorsione in *Mathematica*:

In[145]:=
```
z = g[z];
```

```
— $RecursionLimit::reclim : Recursion depth of 256 exceeded. More…
```

In[146]:=
```
Clear[z]
```

Perchè applica la sostituzione all'infinito (fino a quando non supera il numero massimo di ricorsioni, ovviamente). È anche possibile ottenere la lista di tutti gli elementi intermedi dell'operazione:

```
In[147]:= NestList[g, z, 7]

Out[147]= {z, g[z], g[g[z]], g[g[g[z]]], g[g[g[g[z]]]],
           g[g[g[g[g[z]]]]], g[g[g[g[g[g[z]]]]]], g[g[g[g[g[g[g[z]]]]]]]}
```

Un esempio classico di programmazione di questo tipo (presente anche nell'help di *Mathematica*, tanto è famoso), è dato dalla funzione reciproca, che abbiamo visto qualche tempo fa (in una galassia lontana, lontana...) per le sostituzioni:

```
In[148]:= inv[x_] := 1 / (1 + x)
```

Proviamo ad usare il comando:

```
In[149]:= Nest[inv, x, 6]
```

$$\text{Out[149]=}\quad \cfrac{1}{1+\cfrac{1}{1+\cfrac{1}{1+\cfrac{1}{1+\cfrac{1}{1+\cfrac{1}{1+x}}}}}}$$

I comandi FixedPoint e FixedPointList sono importanti specialmente quando abbiamo a che fare con problemi di convergenza. In questi problemi (come ad esempio l'algoritmo delle secanti), si applica ripetutamente la funzione al risultato ottenuto precedentemente, fino a quando tutto quanto non si stabilizza, ovvero fino a quando $x = f[x]$. In questo caso si dice che si è raggiunta la convergenza. Inoltre, con il comando che visualizza le liste, è anche possibile visualizzare tutti i passi intermedi.

Proviamo a fare l'esempio per il la funzione inv che abbiamo già definito:

```
In[150]:= FixedPoint[inv, 3.]

Out[150]= 0.618034
```

Abbiamo ottenuto il risultato corretto. Naturalmente possiamo anche vedere i passi intermedi, esattamente come prima

```
In[151]:= FixedPointList[inv, 3.]

Out[151]= {3., 0.25, 0.8, 0.555556, 0.642857, 0.608696, 0.621622, 0.616667,
           0.618557, 0.617834, 0.61811, 0.618005, 0.618045, 0.61803, 0.618036,
           0.618033, 0.618034, 0.618034, 0.618034, 0.618034, 0.618034, 0.618034,
           0.618034, 0.618034, 0.618034, 0.618034, 0.618034, 0.618034, 0.618034,
           0.618034, 0.618034, 0.618034, 0.618034, 0.618034, 0.618034,
           0.618034, 0.618034, 0.618034, 0.618034, 0.618034, 0.618034}
```

Abbiamo visto come siano necessari parecchi passaggi, prima di ottenere la convergenza numerica; notate come già dalla prima metà della lista i numeri compaiono tutti uguali: questo perchè la rappresentazione, di default, visualizza un numero di cifre significative minore di quelle che usa *Mathematica* per il calcolo interno, e quindi sono effettivamente diversi, tranne gli ultimi due che devono essere uguali per la condizione di raggiungimento della convergenza.

Bisogna notare come abbia esplicitamente richiesto un calcolo numerico, invece che esatto, scrivendo 3. invece che 3 ; questo è stato necessario perchè, in quest'ultimo caso, avrei ottenuto una serie di frazioni che sarebbero state s' convergenti, ma non avrei avuto mai due elementi consecutivi esattamente uguali, perchè numeratore e denominatore saranno sempre diversi. Possiamo vederlo, andando ad aggiungere agli argomenti di FixedPoint il numero massimo di iterazioni che vogliamo: è un parametro importante da aggiungere, se non siete sicuri della convergenza del metodo che state usando. Vediamo l'esempio di prima: la convergenza si ottiene dopo aver calcolato n elementi:

In[152]:=
```
n = Length[%]
```

Out[152]=
```
41
```

Vediamo di vedere lo stesso risultato senza però utilizzare l'approssimazione numerica:

In[153]:=
```
Take[FixedPointList[inv, 3, 41], -3]
```

Out[153]=
$$\left\{\frac{180510493}{292072113}, \frac{292072113}{472582606}, \frac{472582606}{764654719}\right\}$$

Come potete vedere in questo caso, anche se numericamente (quindi con approssimazione) i valori sono uguali, lo stesso non si può dire per le frazioni. Questo non cambia neanche se andiamo avanti con il calcolo:

In[154]:=
```
Take[FixedPointList[inv, 3, 100], -3]
```

Out[154]=
$$\left\{\frac{386165282814252014980}{624828552868675407173}, \frac{624828552868675407173}{1010993835682927422153}, \frac{1010993835682927422153}{1635822388551602829326}\right\}$$

Come potete vedere non si raggiunge mai la convergenza, perchè non potranno mai esistere due frazioni uguali per n che è finito, per quanto grande. Ponete sempre attenzione a questo, mi raccomando, perchè calcoli esatti possono portare problemi, come in questo caso, se non sono presi con la dovuta cautela, anche se sono sempre di più i vantaggi che gli svantaggi!!!!

Per casi come questo, piùttosto che verificare se gli ultimi elementi sono uguali, possono essere utili delle condizioni di terminazione, che bloccano il calcolo quando si verifica un determinato test: per esempio, se la differenza fra i due termini successivi è minore di una determinata tolleranza. Questo si può fare anche con numeri esatti:

`NestWhile[`*f*`,` *x*`,` *test*`]`	applica ripetitivamente *f* calcolando *test* ad ogni iterazione, e bloccandosi se non restituisce più `True`
`NestWhileList[`*f*`,` *x*`,` *test*`]`	genera la lista `{`*x*`,` *f*`[`*x*`],` *f*`[`*f*`[`*x*`]],` … `}`, fermandosi alle stesse condizioni della funzione di sopra
`NestWhile[`*f*`,` *x*`,` *test*`,` *m*`]`, `NestWhileList[`*f*`,` *x*`,` *test*`,` *m*`]`	usa gli *m* risultati più recenti per calcolare *test* ad ogni passo
`NestWhile[`*f*`,` *x*`,` *test*`, All]`, `NestWhileList[`*f*`,` *x*`,` *test*`, All]`	usa tutti i risultati calcolati come argomento per *test*

Possiamo, per esempio, ripetere l'esempio di prima, con risultati esatti, e porre una tolleranza fra gli ultimi due elementi, in modo da poter verificare la convergenza come faremmo in calcolo numerico, pur non utilizzando valori approssimati:

In[155]:=
```
tolleranza[x_, y_] := N[Abs[y - x]] > 10^-17
```

In[156]:=
```
NestWhile[inv, 3, tolleranza, 2]
```

Out[156]=
$$\frac{1237237325}{2001892044}$$

Come abbiamo potuto vedere, in questo caso abbiamo creato una funzione con due argomenti che calcola la differenza in modulo di due valori, e verifica se è maggiore della tolleranza. Quindi, restituisce True oppure False. Dopo, l'abbiamo utilizzata nel comando NestWhile, e abbiamo anche specificato che per utilizzare quella funzione abbiamo bisogno degli ultimi due elementi. Notate tuttavia che in questo esempio, anche se abbiamo ottenuto un risultato scritto in forma esatta, non sia detto che sia esatto, nel senso che per questo valore non è vero che $x = \text{inv}[x]$. Non confondete i due concetti, di numero esatto e risultato esatto. L'abbiamo ottenuto a meno di un'approssimazione, e va considerato sempre come tale.

Vediamo la lista dei valori delle varie iterazioni, adesso:

In[157]:=
```
NestWhileList[inv, 3, tolleranza, 2]
```

Out[157]=
$$\left\{3, \frac{1}{4}, \frac{4}{5}, \frac{5}{9}, \frac{9}{14}, \frac{14}{23}, \frac{23}{37}, \frac{37}{60}, \frac{60}{97}, \frac{97}{157}, \frac{157}{254}, \frac{254}{411}, \frac{411}{665}, \frac{665}{1076}, \frac{1076}{1741}, \frac{1741}{2817}, \frac{2817}{4558}, \frac{4558}{7375}, \frac{7375}{11933}, \frac{11933}{19308}, \frac{19308}{31241}, \frac{31241}{50549}, \frac{50549}{81790}, \frac{81790}{132339}, \frac{132339}{214129}, \frac{214129}{346468}, \frac{346468}{560597}, \frac{560597}{907065}, \frac{907065}{1467662}, \frac{1467662}{2374727}, \frac{2374727}{3842389}, \frac{3842389}{6217116}, \frac{6217116}{10059505}, \frac{10059505}{16276621}, \frac{16276621}{26336126}, \frac{26336126}{42612747}, \frac{42612747}{68948873}, \frac{68948873}{111561620}, \frac{111561620}{180510493}, \frac{180510493}{292072113}, \frac{292072113}{472582606}, \frac{472582606}{764654719}, \frac{764654719}{1237237325}, \frac{1237237325}{2001892044}\right\}$$

E che ci vuole, adesso, a ricopiarsi i valori nel quaderno???? :-)

Il limite di questo ragionamento è che si tratta di elaborare funzioni ad un solo argomento; invece, a volte sarebbe utile poter avere funzioni a due argomenti da trattare in maniera simile, cioè iterarle: per questo basta usare i seguenti comandi:

`FoldList[f, x, {a, b, ...}]`	crea la lista `{x, f[x, a], f[f[x, a], b], ...}`
`Fold[f, x, {a, b, ...}]`	restituisce l'ultimo elemento di `FoldList[f, x, {a, b, ...}]`

Supponiamo di avere la seguente funzione:

```
In[158]:= g[x_, y_] := inv[x] - inv[y]
```

```
In[159]:= Clear[g]
```

Vediamo il comportamento con FoldList. Supponiamo di voler usare i seguenti valori come secondo argomento nell'applicazione iterativa:

```
In[160]:= lista = {a, c, b, 2, 6};
```

Basta applicare alla lettera il comando:

```
In[161]:= FoldList[g, x, lista]
```

```
Out[161]= {x, g[x, {1, 2, 3, 4, 5}],
           g[g[x, {1, 2, 3, 4, 5}], c], g[g[g[x, {1, 2, 3, 4, 5}], c], b],
           g[g[g[g[x, {1, 2, 3, 4, 5}], c], b], 2],
           g[g[g[g[g[x, {1, 2, 3, 4, 5}], c], b], 2], 6]}
```

Vediamo qualcosa di più comprensibile, va':

```
In[162]:= Fold[f, x, lista]
```

```
Out[162]= f[f[f[f[f[x, {1, 2, 3, 4, 5}], c], b], 2], 6]
```

Come potete vedere, si applica iterativamente una funzione a due variabili, cosa che vi può tornare utile in alcuni algoritmi: io siceramente mi ricordo di averla usata solamente una volta in anni di utilizzo di *Mathematica*, ma ognuno è diverso dagli altri...

Un altro comando utile nel trattamento di funzioni, è il seguente:

`Apply[f, {a, b, … }]`	applica *f* ad una lista, restituendo *f*`[a, b, … ]`
`Apply[f, expr]` or *f* `@@` *expr*	applica *f* al livello più alto dell'espressione
`Apply[f, expr, {1}]` or *f* `@@@` *expr*	applica *f* al primo livello dell'espressione
`Apply[f, expr, lev]`	applica *f* ai livelli specificati dell'espressione

Può capitare, durante i nostri calcoli, che otteniamo una lista di valori da usare come argomento per una funzione: per esempio, potremmo ottenere una lista contenente come primo elemento l'ordine della funzione (per esempio, l'ordine di BesselJ), e come secondo argomento il punto dove vogliamo che sia calcolata.

```
In[163]:= Apply[BesselJ, {5, 87}] // N
Out[163]= -0.0855539
```

Naturalmente, potremmo applicare la funzione anche ad elementi che non siano quelli di una lista. Sappiamo, infatti, che le liste non sono altro che espressioni: di conseguenza, possiamo applicare il comando Apply anche alle espressioni generali:

```
In[164]:= f @@ (x^3 + x^2 + Sin[y])
Out[164]= f[x^2, x^3, Sin[y]]
```

Come potete vedere, è stato scambiato l'head principale, che è rappresentato da Plus, con l'head che abbiamo scelto noi, cioè f. Questo permette di poter organizzare e manipolare le espressioni in maniera avanzata. Possiamo anche cambiare il livello di nidificazione successivo:

```
In[165]:= f @@@ (x^3 + x^2 + Sin[y])
Out[165]= f[y] + f[x, 2] + f[x, 3]
```

Vediamo meglio quello che abbiamo fatto. Consideriamo l'espressione iniziale:

```
In[166]:= TreeForm[x^3 + x^2 + Sin[y]]
Out[166]//TreeForm=
Plus[Power[x, 2], Power[x, 3], Sin[y]]
```

Adesso, applicando il comando come per il primo esempio, otteniamo:

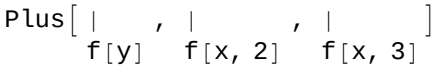
In[167]:=
```
TreeForm[f @@ (x^3 + x^2 + Sin[y])]
```

Out[167]//TreeForm=

Come possiamo vedere, abbiamo scambiato l'head principale, quello al livello più alto dell'albero rovesciato che abbiamo ottenuto. Di conseguenza i più se ne vanno. Consideriamo adesso il secondo esempio:

In[168]:=
```
TreeForm[f @@@ (x^3 + x^2 + Sin[y])]
```

Out[168]//TreeForm=



In questo caso, siamo andati ad applicare la f, sostituendola a 'qualsiasi' head che si trovava al primo livello. La potenza di questo comando risiede nel fatto che permette di sostituire gli head in base al livello in cui si trova, non in base al nome dell'head. Ovviamente, la necessità dell'uso di questo comando, come fra l'altro come quasi tutti quelli riguardanti la manipolazione avanzata, dipende da quello che volete fare.

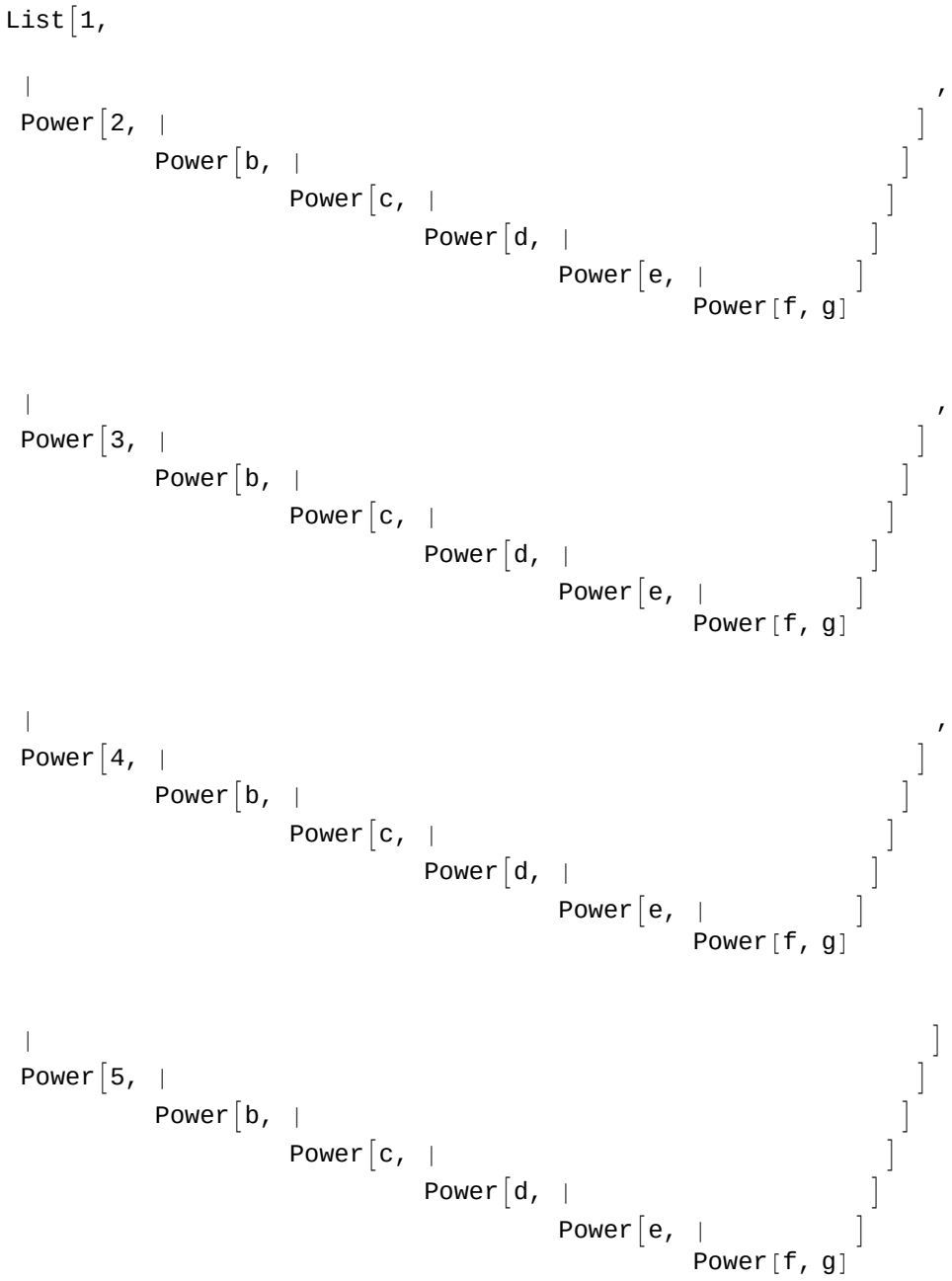
Possiamo anche applicarla a più livelli contemporaneamente. Consideriamo questa espressione:

In[169]:=
```
a^b^c^d^e^f^g
```

Out[169]=
$$\left\{1, 2^{b^{c^{d^{e^{f g}}}}}, 3^{b^{c^{d^{e^{f g}}}}}, 4^{b^{c^{d^{e^{f g}}}}}, 5^{b^{c^{d^{e^{f g}}}}}\right\}$$

In[170]:=
```
TreeForm[%]
```

Out[170]//TreeForm=



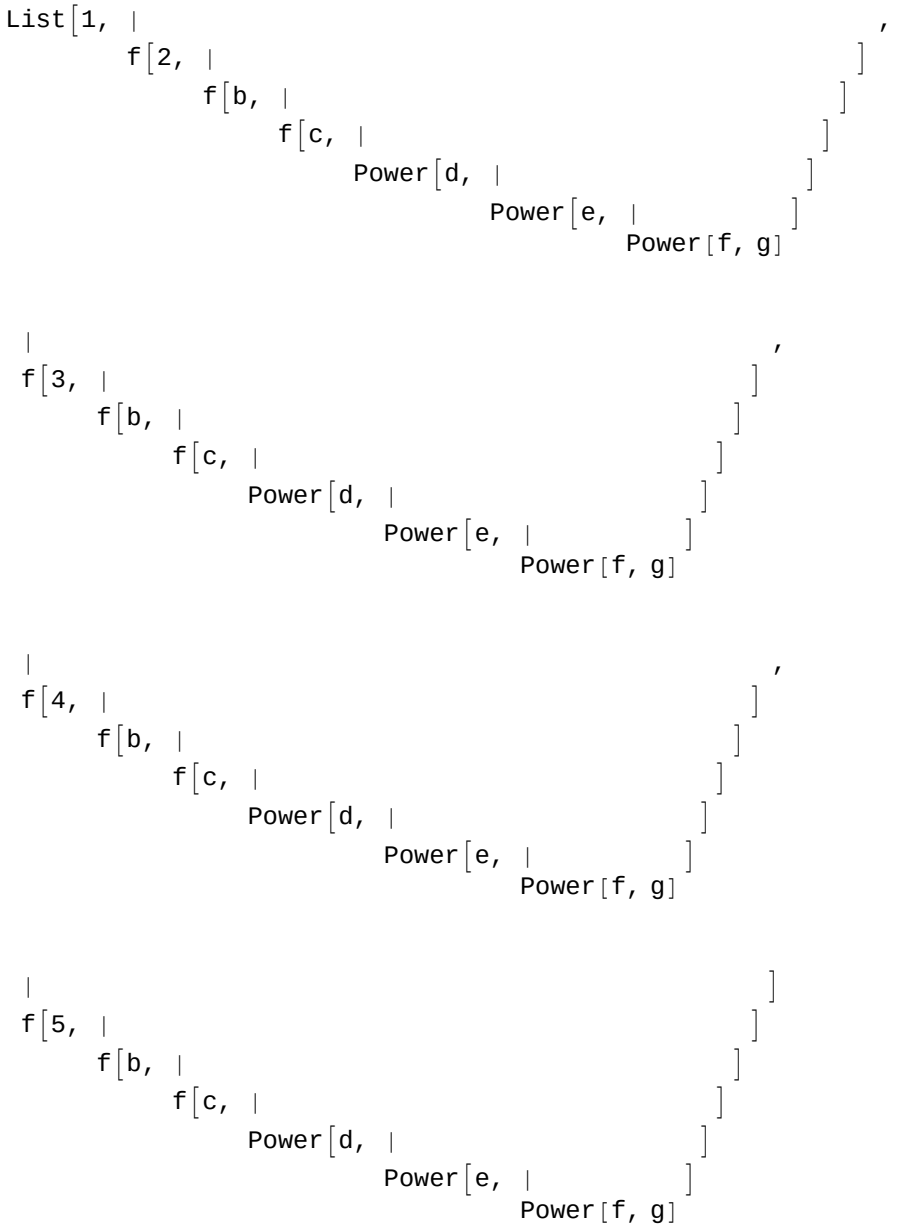
Ipotizziamo, adesso, di voler cambiare la funzione di potenza dal primo al terzo livello:

In[171]:=
```
Apply[f, %%, {1, 3}]
```

Out[171]=
$$\{1, f[2, f[b, f[c, d^{efg}]]], f[3, f[b, f[c, d^{efg}]]], f[4, f[b, f[c, d^{efg}]]], f[5, f[b, f[c, d^{efg}]]]\}$$

In[172]:=
```
TreeForm[%]
```

Out[172]//TreeForm=



Come possiamo vedere, abbiamo ottenuto proprio quello che desideravamo.

Tuttavia, a volte non è questo quello che desideriamo: a volte capita di volere che una funzione si *inserisca* nell'albero, cioè che la funzione vada ad agire sugli argomenti che abbiamo ad un certo livello, non che vada a sostituire gli head. Per esempio, ci piacerebbe che le incognite di un polinomio cubico siano espresse sotto forma di seni, da così:

In[173]:=
```
espr = a x^3 + b x^2 + c x + d;
```

a così:

In[174]:=
```
a Sin[x]^3 + b Sin[x]^2 + c Sin[x] + d
```

Out[174]=
```
{d + c Sin[x] + b Sin[x]^2 + Sin[x]^3,
 d + c Sin[x] + b Sin[x]^2 + 2 Sin[x]^3, d + c Sin[x] + b Sin[x]^2 + 3 Sin[x]^3,
 d + c Sin[x] + b Sin[x]^2 + 4 Sin[x]^3, d + c Sin[x] + b Sin[x]^2 + 5 Sin[x]^3}
```

Andando ad usare Apply, non otterremmo il risultato voluto:

In[175]:=
```
Sin @@ espr
```

```
— Sin::argx : Sin called with 5 arguments; 1 argument is expected. More…
```

Out[175]=
```
Sin[d + c x + b x^2 + x^3, d + c x + b x^2 + 2 x^3,
 d + c x + b x^2 + 3 x^3, d + c x + b x^2 + 4 x^3, d + c x + b x^2 + 5 x^3]
```

Come potete vedere, non otteniamo quello che vogliamo. Il risultato voluto si può applicare invece con il seguente comando (ne avevamo accennato qualche pagina fa, se ricordate...):

`Map[`*f*`, `*expr*`]` or *f* `/@` *expr*	applica *f* alle parti del primo livello di *expr*
`MapAll[`*f*`, `*expr*`]` or *f* `//@` *expr*	applica *f* a tutte le singole parti di *expr*
`Map[`*f*`, `*expr*`, `*lev*`]`	applica *f* ai livelli di *expr* spedificati da *lev*
`MapAt[`*f*`, `*expr*`, {`$part_1$`, `$part_2$`, … }]`	applica *f* in specifiche parti dell' espressione

Proviamo a ripetere il procedimento, con questo nuovo comando:

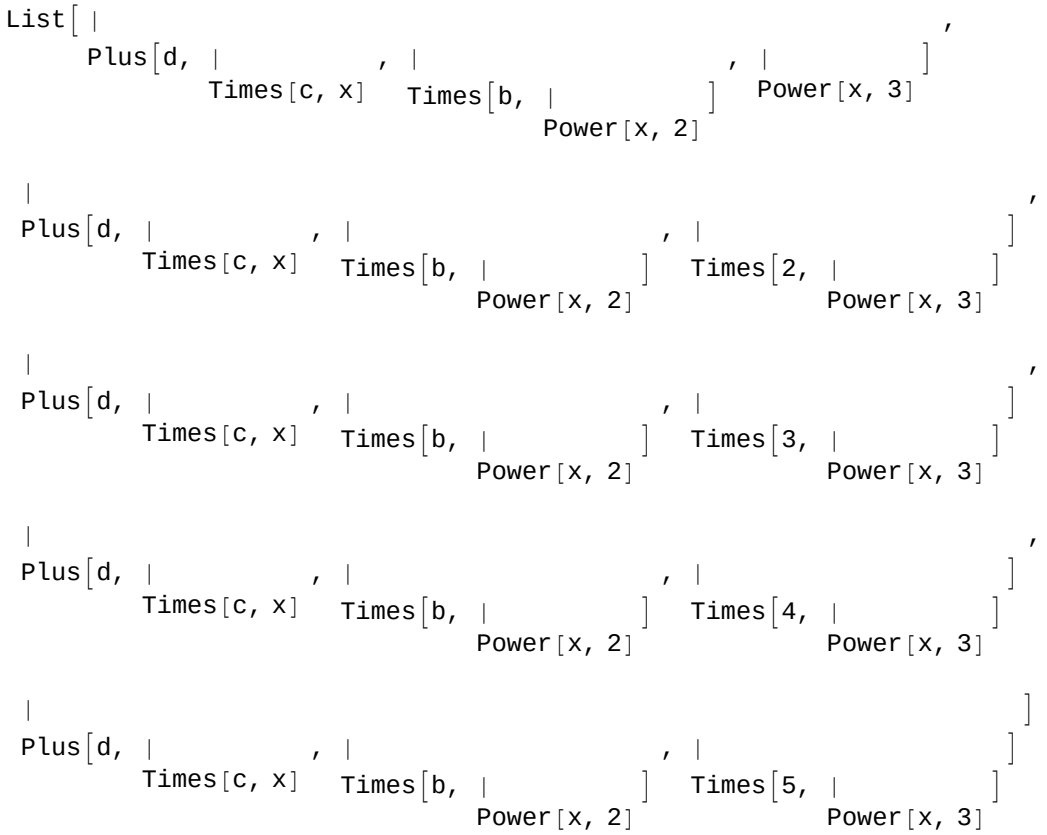
In[176]:=
```
Sin /@ espr
```

Out[176]=
```
{Sin[d + c x + b x^2 + x^3], Sin[d + c x + b x^2 + 2 x^3],
 Sin[d + c x + b x^2 + 3 x^3], Sin[d + c x + b x^2 + 4 x^3], Sin[d + c x + b x^2 + 5 x^3]}
```

Quello che abbiamo ottenuto è simile a quello che volevamo: non è uguale per il seguente motivo:

In[177]:=
```
TreeForm[espr]
```

Out[177]//TreeForm=



Come è possibile vedere, al primo livello compaiono le funzioni che eseguono le moltiplicazioni, non l'elevamento a potenza. Dato che ci interessa soltanto inserire il seno, abbiamo praticamente sbagliato livello. Invece che applicare la funzione agli elementi del primo livello, dobbiamo applicarla agli elementi specifici: vediamo prima dove compaiono gli esponenti:

In[178]:=
```
Position[espr, x]
```

Out[178]=
```
{{1, 2, 2}, {1, 3, 2, 1}, {1, 4, 1}, {2, 2, 2}, {2, 3, 2, 1},
 {2, 4, 2, 1}, {3, 2, 2}, {3, 3, 2, 1}, {3, 4, 2, 1}, {4, 2, 2},
 {4, 3, 2, 1}, {4, 4, 2, 1}, {5, 2, 2}, {5, 3, 2, 1}, {5, 4, 2, 1}}
```

Una volta visto dove compaiono le incognite, possiamo applicare le sostituzioni:

In[179]:=
```
MapAt[Sin, espr, %]
```

Out[179]=
```
{d + c Sin[x] + b Sin[x]^2 + Sin[x]^3,
 d + c Sin[x] + b Sin[x]^2 + 2 Sin[x]^3, d + c Sin[x] + b Sin[x]^2 + 3 Sin[x]^3,
 d + c Sin[x] + b Sin[x]^2 + 4 Sin[x]^3, d + c Sin[x] + b Sin[x]^2 + 5 Sin[x]^3}
```

Come possiamo vedere, questa volta abbiamo ottenuto il risultato voluto. Notate come si siano usato un comando descritto in precedenza, Position. Questo vi fa capire come bisogna sempre collegare

tutto quello che si sa per poter ottenere il risultato voluto velocemente. Se non l'avessi usato, avrei dovuto trovarmi a mano le singole posizioni delle varie incognite che compaiono nell'espressione, cosa già non facilissima con una cubica, figurarsi con un'espressione più complicata!

Naturalmente, se avessimo potuto, e soprattutto se ci sarebbe servito, avremmo potuto anche mappare ogni singolo elemento con la funzione seno:

In[180]:=
```
Sin //@ espr
```

Out[180]=
```
{Sin[Sin[Sin[d] + Sin[Sin[c] Sin[x]] +
    Sin[Sin[x]^Sin[3]] + Sin[Sin[b] Sin[Sin[x]^Sin[2]]]]],
 Sin[Sin[Sin[d] + Sin[Sin[c] Sin[x]] + Sin[Sin[b] Sin[Sin[x]^Sin[2]]] +
    Sin[Sin[2] Sin[Sin[x]^Sin[3]]]],
 Sin[Sin[Sin[d] + Sin[Sin[c] Sin[x]] + Sin[Sin[b] Sin[Sin[x]^Sin[2]]] +
    Sin[Sin[3] Sin[Sin[x]^Sin[3]]]],
 Sin[Sin[Sin[d] + Sin[Sin[c] Sin[x]] + Sin[Sin[b] Sin[Sin[x]^Sin[2]]] +
    Sin[Sin[4] Sin[Sin[x]^Sin[3]]]],
 Sin[Sin[Sin[d] + Sin[Sin[c] Sin[x]] + Sin[Sin[b] Sin[Sin[x]^Sin[2]]] +
    Sin[Sin[5] Sin[Sin[x]^Sin[3]]]]]}
```

Tuttavia, non capita spesso di dover usare comandi come questo. Di solito si usa Map, oppure MapAt.

Tuttavia, a volte ci capita di aver a che fare non con funzioni ad un argomento, ma con funzioni a due o più argomenti. Se vogliamo mappare nell'espressione queste funzioni, ci torna utile il seguente comando:

`MapThread[`*f*`, {`*expr*$_1$`, `*expr*$_2$`, …}]`	applica *f* facendo corrispondere fra di loro gli elementi di *expr*$_i$
`MapThread[`*f*`, {`*expr*$_1$`, `*expr*$_2$`, …}, `*lev*`]`	applica *f* alle varie parti di *expr*$_i$ nei livelli specificati

In questo caso abbiamo bisogno di tante espressioni quanti sono gli argomenti della funzione: in questo modo si creerà l'espressione corrispondente prendendo la funzione, il primo argomento che si trova nella prima espressione, il secondo che si trova nella seconda espressione e così via:

In[181]:=
```
lista1 = {6, 5, 6, 5}; lista2 = {8, 2, 9, 4};
```

In[182]:=
```
MapThread[HarmonicNumber, {lista1, lista2}]
```

Out[182]=
$$\left\{\frac{168646392872321}{167961600000000}, \frac{5269}{3600}, \frac{373997614931101}{373248000000000}, \frac{14001361}{12960000}\right\}$$

Come potete vedere, in questo caso abbiamo applicato la funzione con i corrispettivi argomenti:

In[183]:=
```
Table[HarmonicNumber[lista1[[n]], lista2[[n]]], {n, 4}]
```

Out[183]=
$$\left\{\frac{168646392872321}{167961600000000}, \frac{5269}{3600}, \frac{373997614931101}{373248000000000}, \frac{14001361}{12960000}\right\}$$

Come potete vedere le due liste corrispondono, per cui da qua si comprende il funzionamento di questo comando.

L'ultimo comando da vedere è questo:

`Scan[`*f*`, `*expr*`]`	valuta *f* applicata ad ogni elemento di *expr* in sequenza
`Scan[`*f*`, `*expr*`, `*lev*`]`	valuta *f* applicata alle parti di *expr* che si trovano nei livelli specificati da *lev*

Riconsideriamo il polinomio di prima:

In[184]:=
```
espr = a x^3 + b x^2 + c x + d;
```

Facendo qualcosa del tipo

In[185]:=
```
Map[f, espr]
```

Out[185]=
$$\{f[d + c\,x + b\,x^2 + x^3],\ f[d + c\,x + b\,x^2 + 2\,x^3],\ f[d + c\,x + b\,x^2 + 3\,x^3],\ f[d + c\,x + b\,x^2 + 4\,x^3],\ f[d + c\,x + b\,x^2 + 5\,x^3]\}$$

Otteniamo, in pratica, una nuova espressione. Tuttavia con Scan possiamo valutare, al posto dell'intera espressione, solamente i risultati che si hanno applicando la funzione ai vari elementi dell'espressione:

In[186]:=
```
Scan[Print, espr]
```

$d + c\,x + b\,x^2 + x^3$

$d + c\,x + b\,x^2 + 2\,x^3$

$d + c\,x + b\,x^2 + 3\,x^3$

$d + c\,x + b\,x^2 + 4\,x^3$

$d + c\,x + b\,x^2 + 5\,x^3$

In[187]:=
```
Scan[f, espr]
```

Come possiamo vedere, nel secondo caso manca l'output. Questo perchè Scan di suo non scrive il risultato delle funzioni, ma le valuta solamente: questo può essere utile quando ci sono determinate

assegnazioni e funzioni. Nel primo caso Print valutato restituisce una cosa scritta su schermo, quindi non è *Mathematica* stessa che scrive l'output, ma è una conseguenza della valutazione del comando Print.

Modifiche strutturali delle espressioni e funzioni

Finora abbiamo visto come possiamo andare a sostituire e modificare parti di un'espressione. Tuttavia, la struttura delle espressioni rimaneva pressochè invariata: è vero che a volte andiamo a sostituire degli argomenti con altre funzioni, modificando l'albero, ma è pur sempre vero che quelle funzioni rappresentano comunque dei valori; la struttura originale dell'espressione viene comunque preservata. Tuttavia ci sono casi (specialmente nelle liste), in cui invece non vogliamo andare a manipolare i vari valori memorizzati nell'epressione, ma vogliamo, invece, andare a modificarne direttamente la struttura.

Una cosa che si può fare, per esempio, è considerare le operazioni non tanto sugli argomenti, quanto sugli head: possiamo considerare questo ragionamento se li consideriamo come se fossero degli operatori che agiscono sul loro argomento, e quindi, come per gli operatori, possiamo appiccicargli un algebra, propria degli operatori:

`Composition[`f`, `g`, … ]`	composizione degli operatori f, g, …
`InverseFunction[`f`]`	l'inverso della funzione f
`Identity`	la funzione identità
`Through[`p`[`f_1`, `f_2`][`x`], `q`]`	restituisce $p[f_1[x], f_2[x]]$ se p è lo stesso di q
`Operate[`p`, `f`[`x`]]`	restituisce $p[f][x]$
`Operate[`p`, `f`[`x`], `n`]`	applica *l'* operatore p nel livello n di f
`MapAll[`p`, `*expr*`, Heads->True]`	applica p in tutte le parti di f, head incluso

Supponiamo di avere degli operatori e di volerli comporre:

```
In[188]:= operatore = Composition[op1, op2, op3]
Out[188]= Composition[op1, op2, op3]
```

Se adesso vado ad applicare l'operatore ottenuto ad un valore, ottengo:

```
In[189]:= operatore[x]
Out[189]= op1[op2[op3[x]]]
```

Come possiamo vedere, si può considerare come funzione di funzione. Possiamo anche effettuare le opportune operazione. Per esempio possiamo scrivere la somma di due operatori in questa maniera:

In[190]:=
```
(op1 + op2)[x]
```

Out[190]=
```
(op1 + op2)[x]
```

E, se abbiamo definito da qualche parte questi due operatori, possiamo scrivere il tutto in maniera esplicita, permettendo a *Mathematica* di calcolarlo:

In[191]:=
```
Through[%, Plus]
```

Out[191]=
```
op1[x] + op2[x]
```

Toh, abbiamo appena definito la linearità fra gli operatori....

Possiamo anche espandere e creare manipolazioni per gli operatori: per esempio:

In[192]:=
```
(op1 + op2 + op3)^2[x]
```

Out[192]=
```
(op1 + op2 + op3)^2[x]
```

Può essere esteso facilmente:

In[193]:=
```
MapAll[Expand, %, Heads → True]
```

Out[193]=
```
(op1^2 + 2 op1 op2 + op2^2 + 2 op1 op3 + 2 op2 op3 + op3^2)[x]
```

Come possiamo vedere, le manipolazioni che possiamo eseguire sono veramente tante...

Oltre che dal punto di vista degli operatori, quindi eseguendo espressioni e modifiche tramite heads, possiamo anche correggere direttamente la struttura delle espressioni:

`Sort[`*expr*`]`	ordina gli elementi dell'espressione in ordine standard
`Sort[`*expr*`, `*pred*`]`	ordina gli elementi usando *pred* per determinare l'ordine degli elementi
`Ordering[`*expr*`]`	restituisce la lista contenente gli indici di ordinamento dell'espressione
`Ordering[`*expr*`, `*n*`]`	come sopra, ma per i primi *n* elementi
`Ordering[`*expr*`, `*n*`, `*pred*`]`	usa *pred* per determinare il tipo di ordine
`OrderedQ[`*expr*`]`	restituisce `True` se gli elementi di *expr* sono ordinati in maniera standard
`Order[`$expr_1$`, `$expr_2$`]`	restituisce `1` se $expr_1$ precede $expr_2$ nell'ordine standard, e -1 altrimenti
`Flatten[`*expr*`]`	appiattisce le funzioni nidificate dell'albero delle espressioni con lo stesso head di *expr*
`Flatten[`*expr*`, `*n*`]`	appiattisce al massimo *n* livelli di nidificazione
`Flatten[`*expr*`, `*n*`, `*h*`]`	appiattisce tutte le funzioni con head *h*
`FlattenAt[`*expr*`, `*i*`]`	appiattisce soltanto l' *i-simo* elemento di *expr*
`Distribute[`*f*`[`*a* + *b* + … , … `]]`	destribuisce *f* nella somma per avere *f*`[`*a*`, … ] + `*f*`[`*b*`, … ] + …`
`Distribute[`*f*`[`*args*`], `*g*`]`	distribuisce *f* su qualsiasi argomento avente head *g*
`Distribute[`*expr*`, `*g*`, `*f*`]`	distribuisce solamente quando l'head è *f*
`Distribute[`*expr*`, `*g*`, `*f*`, `*gp*`, `*fp*`]`	distribuisce *f* su *g*, sostituendoli con *fp* e *gp*, rispettivamente
`Thread[`*f*`[{`a_1`, `a_2`}, {`b_1`, `b_2`}]]`	inserisce *f* nella lista per ottenere `{`*f*`[`a_1`, `b_1`], `*f*`[`a_2`, `b_2`]}`
`Thread[`*f*`[`*args*`], `*g*`]`	inserisce *f* negli oggetti con head *g* in *args*
`Outer[`*f*`, `$list_1$`, `$list_2$`]`	prodotto esterno generalizzato
`Inner[`*f*`, `$list_1$`, `$list_2$`, `*g*`]`	prodotto interno generalizzato

Come potete vedere, le funzioni per manipolare le strutture non sono in fondo così poche, ma vi posso assicurare che non sono difficili da capire e da usare, sempre, ovviamente, se si sa quello che si sta facendo.

Vediamo prima di tutto la prima parte della lista dei comandi, cioè quelli riguardanti l'ordinamento. Sappiamo che Sort ordina gli argomenti di una funzione in maniera standard:

In[194]:=
```
Sort[{a, f, r, t, i, b, x}]
```

Out[194]=
```
{3, b, f, i, r, x, {1, 2, 3, 4, 5}}
```

In[195]:=
```
Sort[f[a, v, f, e, q, v, c, h, b]]
```

Out[195]=
```
f[3, b, c, e, f, h, v, v, {1, 2, 3, 4, 5}]
```

Come potete vedere, il risultato è questo, ed è ovvio. Tuttavia, supponiamo di aver bisogno di un ordinamneto inverso, come possiamo fare? Naturalmente, conoscete benissimo le funzioni pure... Aggiungiamo il fatto che possiamo usare Sort con la funzione di ordinamento, ed abbiamo ottenuto quello che volevamo:

In[196]:=
```
Sort[f[a, v, f, e, q, v, c, h, b],
 Sort[ToString /@ {#1, #2}][[1]] === ToString[#2] &]
```

Out[196]=
```
f[v, v, h, f, e, c, b, 3, {1, 2, 3, 4, 5}]
```

Vediamo di capire bene quello che abbiamo appena fatto, anche per riassumere un poco quello che abbiamo fatto (e che ho pazientemente spiegato...) sinora. Come sappiamo, *Mathematica* non è in grado di ordinare due simboli, perchè li dovrebbe ordinare in base al contenuto, e lascia le cose come sono se non contengono niente:

In[197]:=
```
a > b
```

Out[197]=
```
{1, 2, 3, 4, 5} > b
```

Analogamente, senza la funzione Sort, non riesce neanche a distinguere se una stringa viene prima o dopo di un'altra: in C questa disuguaglianza darebbe il risultato corretto, invece:

In[198]:=
```
"grge" > "asdaf"
```

Out[198]=
```
grge > asdaf
```

Però, siamo comunque in grado di ordinare le stringhe, anche se non direttamente, utilizzando la funzione Sort:

In[199]:=
```
Sort[{"grge", "asdaf"}]
```

Out[199]=
```
{asdaf, grge}
```

Quello che ci serve, allora, è convertire i nomi delle variabili, usate come argomenti nella funzione, in stringhe, ed ordinarle. A questo ci pensa il comando

In[200]:=
```
ToString /@ {#1, #2}
```

Out[200]=
```
{#1, #2}
```

In questo caso, al posto di scrivere due volte ToString, abbiamo mappato il comando, in modo che risultasse, alla fine, in ogni argomento della lista; in effetti è equivalente a scrivere:

```
In[201]:= {ToString[#1], ToString[#2]}
Out[201]= {#1, #2}
```

Ed abbiamo scritto in quella maniera per essere più brevi e coincisi. Adesso, ottenuta la lista di stringhe, la ordiniamo mediante Sort:

```
In[202]:= Sort[ToString /@ {#1, #2}]
Out[202]= {#1, #2}
```

In questo modo riusciamo a riorganizzare la lista di stringhe in modo che risultino in ordine alfabetico. Una volta fatto questo, si tratta di vedere se l'ordine era giusto oppure no: prendiuamo il primo elemento della lista ordinata che abbiamo appena ottenuto:

```
In[203]:= Sort[ToString /@ {#1, #2}][[1]]
Out[203]= #1
```

Adesso, se il primo elemento di questa lista coincide con il secondo elemento della lista non ordinata, allora gli elementi nella lista ordinata erano in ordine inverso: guardate qua:

```
In[204]:= lista = {3, 1}; listaordinata = Sort[lista]
Out[204]= {1, 3}

In[205]:= listaordinata[[1]] == lista[[2]]
Out[205]= True
```

Come abbiamo visto, questo risultato è True, perchè gli elementi della lista originale erano in ordine inverso. Facciamo lo stesso identico ragionamento per la coppia ordinata di stringhe. Per vedere se il primo elemento della lista ordinata di stringhe che abbiamo ottenuto è uguale al secondo elemento della coppia non ordinata (che corrisponde al secondo argomento della funzione pura), ne verifichiamo l'uguaglianza:

```
In[206]:= Sort[ToString /@ {#1, #2}][[1]] === ToString[#2]
Out[206]= False
```

Notate come, dato che la lista è ora formata da stringhe, anche il secondo argomento debba essere una stringa, e come, inoltre, abbia eseguito l'uguaglianza esatta fra pattern ===, e non ==, in quanto in questo secondo caso, non sapendo *Mathematica* ordinare da sola le stringhe, non saprebbe se sono uguali, per cui devo eseguire un' uguaglianza fra pattern. Infine aggiungo il simbolo della funzione pura, per completare il tutto:

```
In[207]:= Sort[ToString /@ {#1, #2}][[1]] === ToString[#2] &
Out[207]= Sort[ToString /@ {#1, #2}][[1]] === ToString[#2] &
```

Visto quante cose siamo riusciti a fare in molto meno di un rigo? Provate a farlo dichiarando tutte le funzioni etc etc, e vedete quanto dovrete scrivere....

Comunque, una volta ottenuta la nostra funzione di ordinamento, vediamo che effettivamente funziona, ed ordina tutti gli argomenti in modo inverso. Naturalmente, se avessimo avuto valori numerici, le cose sarebbero state molto più semplici:

```
In[208]:= Sort[f[3, 1, 6, 7, 2, 4, 2, 9, 2, 6], #1 > #2 &]
Out[208]= f[9, 7, 6, 6, 4, 3, 2, 2, 2, 1]
```

In questo caso basta fare il confronto diretto fra gli argomenti, dato che non c'è bisogno di convertire alcunchè, perchè gli operatori logici naturalmente lavorano direttamente sui numeretti...

Aaahhh!!!! Scommetto che dopo questo esempio andrete a prendervi un bel caffè, oppure vi tufferete al manre o vi rinchiuderete in qualche pub... Fate bene, bisogna uscire!!!! Comunque, sempre qua dovete tornare!!! Aspetterò con calma...

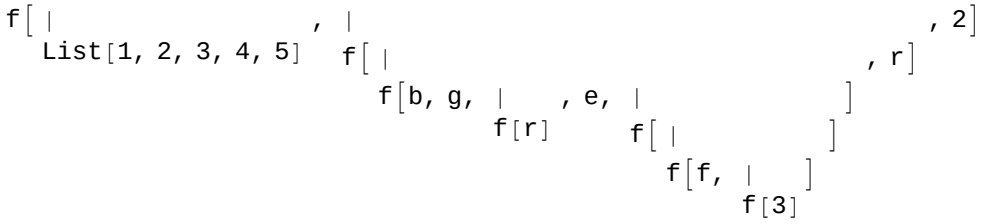
Ancora qui? Bene, vuol dire che continuerò a rubarti quel poco di vita sociale che ti resta, succhiandoti la linfa vitale... Benvenuto!!!

Abbiamo visto come possiamo usare Sort per modificare il pattern di una funzione. Adesso vediamo il comando Flatten. Come dice la definizione, serve per appiattire l'espressione. Vediamo cosa intendo per appiattire. Considera questa espressione:

```
In[209]:= f[a, f[f[b, g, f[r], e, f[f[f, f[3]]]], r], 2];
```

In[210]:=
```
TreeForm[%]
```

Out[210]//TreeForm=



Mamma mia, quanti livelli ha questa funzione cattiva cattiva!!!!!

Vediamo di applicare al risultato il comando:

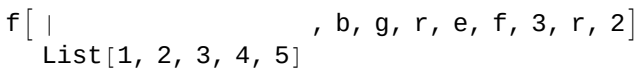
In[211]:=
```
Flatten[%%]
```

Out[211]=
```
f[{1, 2, 3, 4, 5}, b, g, r, e, f, 3, r, 2]
```

E che è successo??? Esattamente quello che ho detto: abbiamo appiattito l'espressione:

In[212]:=
```
TreeForm[%]
```

Out[212]//TreeForm=



Anche in questo caso abbiamo modificato (abbastanza pesantemente, direi) la struttura dell'espressione originale. Possiamo considerarlo, in un certo modo, come se avessimo dato alla funzione f che non è definita, la proprietà dell'associatività, esattamente come per l'addizione o la moltiplicazione. Vedendola in questa maniera, si nota già un certo modo di operare del programma che è propenso al calcolo puramente simbolico, basato sulle proprietà delle funzioni. Lo è anche per questo motivo.

In[213]:=
```
f[g, r, t[r, t], t[f[g, f[g]]]];
```

In[214]:=
```
Flatten[%]
```

Out[214]=
```
f[g, r, 3[r, 3], 3[f[g, f[g]]]]
```

In questo caso, abbiamo visto che il lavoro di Flatten si blocca non appena trova una sottoespressione che ha un head non conforme a quello principale. Appiattisce, insomma, solamente le sottoespressioni con lo stesso head. Un comando che fa una cosa in più, ma anche una in meno, è Sequence:

In[215]:=
```
Sequence[a, b, f]
```

Out[215]=
```
Sequence[{1, 2, 3, 4, 5}, b, f]
```

In apparenza da solo non fa niente. L'abbiamo visto a proposito della programmazione, ed effettivamente fa qualcosa... accoda fra di loro i suoi argomenti. Possiamo vederlo meglio qua:

In[216]:=
```
f[a, b, Sequence[g, h, t], g]
```

Out[216]=
```
f[{1, 2, 3, 4, 5}, b, g, h, 3, g]
```

L'utilità di questa funzione, però, si vede soprattutto quando dobbiamo andare a fare delle sostituzioni. Si può vedere l'utilità (sempre se usato al momento giusto!!!) da questo esempio:

In[217]:=
```
f[g[x, d], g[f[g[f, g[h]], u, r], e]] /. g → Sequence
```

Out[217]=
```
f[x, d, f[f, h, u, r], e]
```

Come abbiamo potuto vedere, possiamo appiattire varie funziono con Sequence. Il vantaggio rispetto a Flatten è che possiamo farlo, usando i pattern alternativi, anche per più funzioni in una volta sola:

In[218]:=
```
f[g[r, t, w[e, t], u, g[f[g[t, y]], y], w[t]]] /. (w | g) → Sequence
```

Out[218]=
```
f[r, 3, e, 3, u, f[3, y], y, 3]
```

Lo svantaggio, invece, rispetto a Flatten, consiste nel fatto che se si appiattisce una funzione che compare anche nell'head principale (cosa per cui è fatto Flatten), sostituisce anche quello:

In[219]:=
```
f[f[c, r], t] /. f → Sequence
```

Out[219]=
```
Sequence[c, r, 3]
```

E non è quello che vogliamo. Quindi, se bisogna appiattire più funzioni, tenendo sempre conto dell'head, possiamo anche usarli in maniera combinata, appiattendo le funzioni con head uguale a quello principale con Flatten, e gli altri con Sequence:

In[220]:=
```
f[f[t, g[r, e, f[r]], r, t, g[e]]];
```

In[221]:=
```
Flatten[%] /. g → Sequence
```

Out[221]=
```
f[3, r, e, f[r], r, 3, e]
```

Notate come in questo caso permane all'interno della funzione un'altra annidata. Questo perchè, come abbiamo visto prima, quando compare un'altra funzione con head diverso da quello usato da

Flatten, il comando si interrompe. Da come è scritto il comando, *Mathematica* prima esegue la funzione Flatten, quindi facendo restare la f residua, e dopo applica la sostituizione. Se vogliamo invece che applichi prima la regola, dobbiamo sostituirla e metterla all'interno di Flatten, in modo che lo scavalchi in precedenza il comando

```
In[222]:= Flatten[%% /. g → Sequence]

Out[222]= f[3, r, e, r, r, 3, e]
```

Stavolta, prima abbiamo effettuato la sostituzione con Sequence, e poi abbiamo applicato Flatten, quindi la f dentro la g stavolta era scoperta e il comando l'ha appiattita.

Adesso, dopo aver visto Flatten, ed averlo considerato coma proprietà associativa, vediamo come possiamo considerare quella distributiva. Come? Qualcuno ha detto Distribute??? Spero di no, non vorrei che foste giunti al punto che parlate da soli...

Comunque, distribute, come avete letto, fa proprio questo; nella sua forma più semplice, distribuisce la funzione, avente come argomento una somma, fra gli addendi stessi:

```
In[223]:= Distribute[f[a + b + f + g + e]]

Out[223]= f[{1 + b + e + f + g, 2 + b + e + f + g,
              3 + b + e + f + g, 4 + b + e + f + g, 5 + b + e + f + g}]
```

Esattamente quello che ci aspettavamo, vero?

```
In[224]:= Distribute[f[a, b, c]]

Out[224]= f[{1, 2, 3, 4, 5}, b, c]
```

Sembra che questo non lavori con gli argomenti generici, ma guardate qua:

```
In[225]:= Distribute[f[a + b, c + d, e + f]]

Out[225]= f[{1 + b, 2 + b, 3 + b, 4 + b, 5 + b}, c, e] +
          f[{1 + b, 2 + b, 3 + b, 4 + b, 5 + b}, c, f] +
          f[{1 + b, 2 + b, 3 + b, 4 + b, 5 + b}, d, e] +
          f[{1 + b, 2 + b, 3 + b, 4 + b, 5 + b}, d, f]
```

Come possiamo vedere, esegue la proprietà distributiva fra gli argomenti della funzione. Prima non si vedeva perchè gli argomenti non erano somme. Effettivamente, nel primo esempio, il comando eseguiva esattamente la stessa cosa, ma l'argomento era uno soltanto. Nel secondo, avevamo tre argomenti, ciascuno però formato, possiamo dire, da un unico addendo, per cui il risultato era uguale a prima. Nel terzo, invece, abbiamo visto la proprietà distributiva nel senso più generale, dove compaiono più argomenti, ognuno somma di più addendi.

Quindi, Distribute viene eseguito di default sulla somma o, più in generale, nelle espressioni con head Plus. Adesso, vorremmo definire questa proprietà anche per altre funzioni; naturalmente possiamo farlo, esplicitando di quale funzione stiamo parlando, naturalmente:

```
In[226]:= Distribute[f[g[a, b], g[c, d]], g]

Out[226]= g[f[{1, 2, 3, 4, 5}, c], f[{1, 2, 3, 4, 5}, d], f[b, c], f[b, d]]
```

In questo caso abbiamo dichiarato che g, pur non essendo definita, gode della proprietà distributiva, e *Mathematica* si è comportato di conseguenza.

```
In[227]:= Distribute[f[g[a, b], g[a, b, c]], g]

Out[227]= g[f[{1, 2, 3, 4, 5}, {1, 2, 3, 4, 5}], f[{1, 2, 3, 4, 5}, b],
           f[{1, 2, 3, 4, 5}, c], f[b, {1, 2, 3, 4, 5}], f[b, b], f[b, c]]
```

Qua vediamo come si comporta bene, anche se gli argomenti sono in numero diverso. D'altronde, la proprietà distributiva permette, per definizione, di non tener conto del numeri degli argomenti della funzione. Un uso più avanzato si può avere anche considerando di sostituire gli head:

```
In[228]:= Distribute[f[g[a, b], g[c, d]], g, f, nuovaf, nuovag]

  — General::spell1 : Possible spelling error: new symbol
       name "nuovag" is similar to existing symbol "nuovaf". More…

Out[228]= nuovaf[nuovag[{1, 2, 3, 4, 5}, c],
           nuovag[{1, 2, 3, 4, 5}, d], nuovag[b, c], nuovag[b, d]]
```

In questo caso, non solo abbiamo sfruttato la proprietà distributiva, ma abbiamo anche cambiato gli head con quelli nuovi. Questa manipolazione è, effettivamente, abbastanza inusuale, e serve principalmente per effettuare operazioni con funzioni predefinite che non consentirebbero di fare. Però credo che la userete veramente poco, indipendentemente da quanto diventerete bravi con *Mathematica*.

Uno degli ultimi comando che analizziamo in questa sezione (non manca molto, coraggio!) è quello che permette di 'infilare' una funzione fra le altre, cioè Thread:

```
In[229]:= Thread[f[{a, b}, {c, d}]]

Out[229]= {f[{1, 2, 3, 4, 5}, c], f[b, d]}
```

In pratica, abbiamo delle liste di argomenti, e tramite il comando Thread siamo in grado di trasportare la f all'interno delle liste, permettendo quindi di valutare la funzione avendo come argomento gli elementi delle liste.

```
In[230]:= Thread[f[{a, b}, {c, d}, f, {t, y}]]

Out[230]= {f[{1, 2, 3, 4, 5}, c, f, 3], f[b, d, f, y]}
```

Come possiamo vedere in questo esempio, notiamo come le costanti vengano sempre ripetute nei vari pattern. Questa è la caratteristica con cui possiamo dare in pasto a molte fiunzioni predefinite delle liste:

```
In[231]:= Sin[{1, 2, 3, 4}]

Out[231]= {Sin[1], Sin[2], Sin[3], Sin[4]}
```

Vediamo come abbiamo fatto agire il seno ad ogni elemento della lista

```
In[232]:= Thread[f[{1, 2, 3, 4}], List]

Out[232]= {f[1], f[2], f[3], f[4]}
```

Per l'esattezza, questo è il modo in cui le funzioni predefinite in *Mathematica* agiscono quando gli diamo come argomento delle liste; possiamo anche fare in modo che funzionino in questa maniera funzioni che hanno bisogno di più argomenti:

```
In[233]:= Thread[f[{1, 2, 3, 4}, {a, b, c, d}], List]

Out[233]= {f[1, {1, 2, 3, 4, 5}], f[2, b], f[3, c], f[4, d]}
```

Come possiamo vedere, abbiamo preso il primo argomento dalla prima lista, ed il secondo dalla seconda lista. Anche questo metodo è standard per le funzioni di *Mathematica*:

```
In[234]:= BesselI[{1, 2, 3, 4}, {a, b, c, d}]

Out[234]= {{BesselI[1, 1], BesselI[1, 2], BesselI[1, 3], BesselI[1, 4],
            BesselI[1, 5]}, BesselI[2, b], BesselI[3, c], BesselI[4, d]}
```

Questo modo di combinare gli argomenti è molto pulito: tuttavia a volte ci serve qualcosa di più efficace, come creare funzioni che abbiamo tutte le combinazioni fra primo e secondo argomento, invece di prendere in considerazione soltanto le coppie ordinate: primo con il primo, secondo con il secondo e così via. Per ottenere tutte le combinazioni ci vuole Outer:

```
In[235]:= Outer[f, {a, b, c, d}, {1, 2, 3}]

Out[235]= {{{f[1, 1], f[1, 2], f[1, 3]}, {f[2, 1], f[2, 2], f[2, 3]},
            {f[3, 1], f[3, 2], f[3, 3]}, {f[4, 1], f[4, 2], f[4, 3]},
            {f[5, 1], f[5, 2], f[5, 3]}}, {f[b, 1], f[b, 2], f[b, 3]},
           {f[c, 1], f[c, 2], f[c, 3]}, {f[d, 1], f[d, 2], f[d, 3]}}
```

Notiamo che, dato che esegue tutte le combinazioni possibili, e non li ordina in sequenza, non è più necessario in questo caso che le liste siano di lunghezza uguale. D'altronde, Outer funziona anche se non tratto liste, ma espressioni con lo stesso head:

```
In[236]:= Outer[f, g[a, b, c, d], g[1, 2, 3]]

Out[236]= g[g[f[{1, 2, 3, 4, 5}, 1], f[{1, 2, 3, 4, 5}, 2], f[{1, 2, 3, 4, 5}, 3]],
           g[f[b, 1], f[b, 2], f[b, 3]],
           g[f[c, 1], f[c, 2], f[c, 3]], g[f[d, 1], f[d, 2], f[d, 3]]]
```

Otteniamo esattamente la stessa cosa, e magari possiamo accorgercene meglio così:

```
In[237]:= % /. g → List

Out[237]= {{f[{1, 2, 3, 4, 5}, 1], f[{1, 2, 3, 4, 5}, 2], f[{1, 2, 3, 4, 5}, 3]},
           {f[b, 1], f[b, 2], f[b, 3]},
           {f[c, 1], f[c, 2], f[c, 3]}, {f[d, 1], f[d, 2], f[d, 3]}}
```

Come vedete, i risultati sono identici...

Riconsideriamo Outer:

```
In[238]:= Outer[f, {a, b}, {c, d}]

Out[238]= {{{f[1, c], f[1, d]}, {f[2, c], f[2, d]}, {f[3, c], f[3, d]},
            {f[4, c], f[4, d]}, {f[5, c], f[5, d]}}, {f[b, c], f[b, d]}}
```

E vediamo adesso il suo duale, Inner:

```
In[239]:= Inner[f, {a, b}, {c, d}]

Out[239]= f[b, d] + f[{1, 2, 3, 4, 5}, c]
```

Vediamo, in questo caso, un comportamento simile a Thread (mentre prima era simile a Distribuite). Però fa anche qualcosa in più:

```
In[240]:= Inner[f, {a, b}, {c, d}, g]

Out[240]= g[f[{1, 2, 3, 4, 5}, c], f[b, d]]
```

In questo caso abbiamo anche specificato l'head principale, mentre nel casi di Thread avevamo per forza di cose una lista di risultati. Quindi è più generale, e si può anche usare al posto di Thread, dato che il risultato, se hanno i medesimi argomenti, è uguale.

Funzioni pure

Le funzioni pure sono un aspetto abbastanza avanzato in *Mathematica*, specificato quando occorrono usi particolari. Se appresi bene, permettono di risolvere problemi scrivendo molto meno, anche se in genere il codice risulerà alquanto criptico, dato che si fa uso di simboli, anche se, come tutto il resto, si possono scrivere come funzioni:

`Function[`*x*`, `*body*`]`	una funzione pura in cui *x* è sostituita dall'argomento voluto
`Function[{`x_1`, `x_2`, … }, `*body*`]`	una funzione pura con più argomenti
body `&`	una funzione pura i cui argomenti sono indicati da # oppure (se sono più di uno) da `#1`, `#2`, `#3`, etc.

Le funzioni pure hanno il compito di usare delle definizioni di funzioni, senza che sia stata definita la funzione stessa. Per esempio, se ci occorre una funzione a due argomenti, che sarebbe definita come il prodotto di due funzioni note ognuna avente un argomento, potremmo scrivere così:

```
In[241]:= f[x_, y_] := q[x] w[y]
```

Per ora, tuttavia cominciamo a considerare il caso di un solo argomento:

```
In[242]:= Clear[f]
```

```
In[243]:= f[x_] := q[x] w[x]
```

Una volta definita la funzione, possiamo utilizzarla dove più ci serve:

```
In[244]:= Map[f, a + b + v + d + e]
```

```
Out[244]= {3[1 + b + d + e + v] w[1 + b + d + e + v],
           3[2 + b + d + e + v] w[2 + b + d + e + v], 3[3 + b + d + e + v] w[3 + b + d + e + v],
           3[4 + b + d + e + v] w[4 + b + d + e + v], 3[5 + b + d + e + v] w[5 + b + d + e + v]}
```

Tuttavia possiamo anche evitare di utilizzare la definizione, ed utilizzare direttamente il corpo della funzione:

```
In[245]:= Map[q[#] w[#] &, a + b + v + d + e]
```

```
Out[245]= {3[1 + b + d + e + v] w[1 + b + d + e + v],
           3[2 + b + d + e + v] w[2 + b + d + e + v], 3[3 + b + d + e + v] w[3 + b + d + e + v],
           3[4 + b + d + e + v] w[4 + b + d + e + v], 3[5 + b + d + e + v] w[5 + b + d + e + v]}
```

In questo caso particolare, abbiamo definito la funzione pura utilizzando il modo abbreviato; prima abbiamo definito il corpo della funzione, che è dato dal prodotto delle altre due. Come argomento abbiamo utilizzato quello delle funzioni pure, che è rappresentato da #. Dopo aver definito il corpo,

abbiamo fatto capire a *Mathematica* che si tratta di una funzione pura, facendolo seguire dal simbolo &, che rappresenta la fine della definizione del corpo della funzione. Tuttavia, potevamo anche utilizzare la forma esplicita:

```
In[246]:= Map[Function[x, q[x] w[x]], a + b + v + d + e]

Out[246]= {3[1 + b + d + e + v] w[1 + b + d + e + v],
           3[2 + b + d + e + v] w[2 + b + d + e + v], 3[3 + b + d + e + v] w[3 + b + d + e + v],
           3[4 + b + d + e + v] w[4 + b + d + e + v], 3[5 + b + d + e + v] w[5 + b + d + e + v]}
```

Ottenendo, in questa maniera, lo stesso risultato. Notate come adesso gli argomenti debbano essere chiamati (e non siano definiti coma pattern x_, ma solo con il nome x), e che il primo modo di scriverlo sia più breve.

Function[*x*, *q*[*x*] *w*[*x*]] rappresenta soltanto il corpo della funzione, che corrisponderebbe al suo head. Infatti abbiamo sostituito l'intera espressione dove, normalmente, ci starebbe soltanto l'head. se vogliamo calcolarla esplicitamente, dobbiamo usare la seguente notazione:

```
In[247]:= Function[x, q[x] w[x]][arg]

  — General::spell1 : Possible spelling error: new
       symbol name "arg" is similar to existing symbol "Arg". More…

Out[247]= 3[arg] w[arg]
```

Come potete vedere, in contesti di uso 'normale' della funzione questo metodo di scrittura è poco pratico, ed è sicuramente meglio definirci prima la funzione. Anche se a volte è meglio usare le funzioni pure anche in questo caso, se la funzione la utilizzeremo soltanto una volta, perchè altrimenti dovremo scrivere ogni volta tutto quanto, invece di un semplice nome mnemonico.

Inoltre, anche come abbiamo visto per le funzioni normali, possiamo prendere in considerazione gli argomenti della funzione pura, singoli oppure in sequenza:

#	il primo argomento della funzione pura
#*n*	l'argomento *n*−simo della funzione pura
##	la sequenza di tutte le variabili nella funzione pura
##*n*	la sequenza di tutti gli argomento in sequenza a partire dall' *n*−simo

È importante non sottovalutare la potenza delle funzioni pure, anche se rappresentano operazioni semplici. Per esempio, possiamo facilmente scrivere qualcosa del tipo:

```
In[248]:= Clear[y]
```

In[249]:=
```
Map[# Sin[#^2] &, {a, v, b, d, r, y}]
```

Out[249]=
```
{{Sin[1], 2 Sin[4], 3 Sin[9], 4 Sin[16], 5 Sin[25]},
 v Sin[v^2], b Sin[b^2], d Sin[d^2], r Sin[r^2], y Sin[y^2]}
```

Se non avessimo avuto le funzioni pure, avremmpo dovuto prima definire una funzione che facesse quello che volevamo, dato che, per dire, semplicemente l'espressione di #^2 sarebbe stata difficile da implementare direttamente in Map, dato che richiede un head, e Power richiede due argomenti. Notiamo anche che l'operatore & che definsce la funzione pura ha una precedenza piuttosto bassa nel calcolo, per cui possiamo tranquillamente definire funzioni pure senza l'uso delle parentesi, come nel caso #1 + #2 &.

Possono essere utili, per esempio, nella costruzione di tavole e di array:

In[250]:=
```
Array[# Log[#] &, 8]
```

Out[250]=
```
{0, 2 Log[2], 3 Log[3], 4 Log[4], 5 Log[5], 6 Log[6], 7 Log[7], 8 Log[8]}
```

Ed ecco veloce veloce la tabellina delle elementari...

In[251]:=
```
Array[#1 #2 &, {10, 10}] // TableForm
```

Out[251]//TableForm=

1	2	3	4	5	6	7	8	9	10
2	4	6	8	10	12	14	16	18	20
3	6	9	12	15	18	21	24	27	30
4	8	12	16	20	24	28	32	36	40
5	10	15	20	25	30	35	40	45	50
6	12	18	24	30	36	42	48	54	60
7	14	21	28	35	42	49	56	63	70
8	16	24	32	40	48	56	64	72	80
9	18	27	36	45	54	63	72	81	90
10	20	30	40	50	60	70	80	90	100

Vediamo come l'uso permetta, effettivamente, di snellire parecchie operazioni, permettendoci di scrivere meno codice.

Per esempio, possiamo selezionare in maniera più specifica e potente parti di un'espressione, utilizzando le funzioni:

`Select[`*list*`, `*f*`]`	seleziona gli elementi della lista se *f* restituisce True
`Select[`*expr*`, `*f*`, `*n*`]`	si limita *a* restituire *i* primi *n* elementi della lista che soddisfano la funzione

Con questo comando, e con le funzioni pure, possiamo selezionare velocemente parti della lista o di un'espressione in generale:

```
In[252]:= lista = {2, 4, 6, r, h, hj, 47, 2, 46, 1};

In[253]:= Select[lista, # < 20 &]

Out[253]= {2, 4, 6, 2, 1}

In[254]:= Array[# &, 1000];

In[255]:= Select[%, PrimeQ[# + Floor[Sqrt[#]]] &]

Out[255]= {1, 2, 5, 10, 14, 19, 26, 32, 37, 41, 47, 52, 54, 60, 65, 71, 75, 88, 92,
           94, 98, 103, 117, 126, 128, 138, 140, 145, 151, 155, 161, 167, 178,
           180, 184, 186, 197, 209, 213, 215, 219, 226, 236, 242, 248, 254, 261,
           265, 267, 277, 290, 294, 296, 300, 314, 320, 329, 331, 335, 341, 349,
           355, 364, 370, 378, 382, 390, 401, 411, 413, 419, 423, 429, 437, 442,
           446, 458, 466, 470, 478, 482, 487, 499, 501, 519, 525, 534, 540,
           546, 548, 554, 564, 570, 577, 583, 589, 593, 595, 607, 617, 619,
           623, 628, 634, 636, 648, 652, 658, 666, 683, 693, 701, 707, 713,
           717, 725, 730, 734, 742, 746, 760, 770, 782, 793, 795, 799, 801,
           811, 825, 829, 831, 835, 848, 852, 854, 858, 878, 882, 890, 907,
           911, 917, 923, 937, 941, 947, 953, 966, 978, 982, 988, 990, 1000}
```

In quest'ultimo esempio abbiamo prima, con le funzioni pure, creato un array contenente i primi 1000 numeri interi, e poi con Select abbiamo effettuato il test verificando quali elementi della lista sono in grado di soddisfare il test (che verifica se una manipolazione di un elemento della lista risulta essere un numero primo). Notate ancora una volta come abbiamo utilizzato il corpo di una funzione, nel posto dove a tutti gli effetti dovrebbe starci un head. È questa la potenza principale in cui risiedono le funzioni pure, permettendovi (ripeto di nuovo per chi si fosse appena collegato) di scrivere meno codice soprattutto se utilizzate raramente la funzione.

Definizioni di espressioni e funzioni

Vediamo adesso, un poco più in dettaglio, come e cosa fare quando creiamo delle definizioni, siano esse di funzioni, oppure di variabili.Abbiamo già visto come si creano le definizioni:

```
In[256]:= x = Binomial[20, 3]

Out[256]= 1140
```

E che adesso, ogni volta che andremo a scrivere il nome della variabile, userà sempre il valore memorizzato (anche per l'ordinamento, come abbiamo visto):

```
In[257]:= Gamma[x] // Short

Out[257]//Short= 44308842923032903576493714 04 <<2933>> 00000000000000000000000000000
```

Abbiamo anche visto che, esattamente come in ogni linguaggio di programmazione, possiamo anche modificarli, una volra definiti:

```
In[258]:= x = 4
Out[258]= 4

In[259]:= x
Out[259]= 4

In[260]:= x = x^2 + 1
Out[260]= 17

In[261]:= x
Out[261]= 17
```

E, se vogliamo che diventino solamente incognite senza nessun contenuto, dobbiamo cancellarlo:

```
In[262]:= x =.
```

Oppure

```
In[263]:= Clear[x]
```

Clear, in particolar modo, funziona anche con le definizioni delle funzioni.

Possiamo inoltre definire più variabili contemporaneamente, se devono contenere lo stesso valore:

```
In[264]:= q = w = e = r = 13;

In[265]:= Print[q, " ", w, " ", e, " ", r]
          13 13 13 13
```

Possiamo anche definire con un unico comando più variabili con valori diversi, utilizzando le liste:

```
In[266]:= {q, w, e} = {41, 54215, 42};

In[267]:= Print[q, " ", w, " ", e]
          41 54215 42
```

Come nella maggior parte dei linguaggi di programmazione odierni, possiamo modificare il contenuto della variabile con una sintassi coincisa, presa direttamente dal C:

x++	incrementa il valore di *x* di 1
x--	decrementa il valore di *x* di 1
++*x*	pre-incremento di *x*
--*x*	pre-decremento di *x*
x+= *dx*	somma *di* all'attuale valore di *x*
x -= *dx*	sottrare *dx* dall'attuale valore di *x*
x *= *c*	moltiplica *x* con *c*
x /= *c*	divide *x* con *c*

Come potete vedere, sono esattamente i comandi che si usano in C per scrivere meno codice. Supposto:

```
In[268]:= x = 5;
```

Possiamo incrementarlo mediande:

```
In[269]:= x++
Out[269]= 5

In[270]:= x
Out[270]= 6

In[271]:= ++x
Out[271]= 7

In[272]:= x
Out[272]= 7
```

Come potete vedere, l'incremento lavora in maniera differente, a seconda se consideriamo l'incremento oppure il pre-incremento, esattamente come succede in C. Nella prima operazione, infatti, l'incremento viene applicato 'dopo' che viene usata la variabile per l'operazione corrente. Quindi, prima ha dato in output il contenuto della variabile, e poi l'ha incrementata, come si può vedere che, dopo il comando, ha valore pari a 6.

Invece, il pre-incremento, 'prima' effettua l'incremento della variabile di un'unità, e dopo esegue l'operazione assegnata. Infatti, ++*x* prima incrementa x, e poi lo manda all'output, come abbiamo

potuto vedere. Bisogna fare attenzione a questo diverso modo di procedere di questi comandi, che altrimenti sarebbero equivalenti:

```
In[273]:= x = y = 3;

In[274]:= x++ + 5

Out[274]= 8

In[275]:= ++y + 5

Out[275]= 9
```

Come avete capito, utilizzare l'uno oppure l'altro ha differenti effetti sulle operazioni, di cui bisogna tenere conto specialmente nella programmazione.

Le altre forme, invece, sono modi abbreviati di scrivere assegnamenti:

```
In[276]:= x = y = 10;

In[277]:= x += 5

Out[277]= 15
```

Vediamo che il valore di x viene aggiornato. Questo è equivalente a scrivere:

```
In[278]:= y = y + 5

Out[278]= 15
```

L'operazione eseguita è la stessa. Analogamente succede con le altre tre operazioni fondamentali:

```
In[279]:= x = 6;

In[280]:= x /= 100

Out[280]= 3/50
```

E così via...

Sappiamo anche come realizzare definizioni delle funzioni:

```
In[281]:= f[x_] := x^3 Sin[x]
```

Abbiamo, in questo caso, definito una semplic funzione:

```
In[282]:= f[t]
Out[282]= 27 Sin[3]

In[283]:= f[7] // N
Out[283]= 225.346

In[284]:= f[7] = 5
Out[284]= 5

In[285]:= f[7]
Out[285]= 5
```

Cos'è successo, stavolta? $f[7]$ non restituisce più lo stesso risultato di prima. Questo perchè abbiamo assegnato ESPLICITAMENTE un valore alla funzione quando è presente l'argomento pari esattamente a 7:

```
In[286]:= f[5] := 3

In[287]:= f[5]
Out[287]= 3
```

Vediamo che possiamo usare in questo caso entrambe le assegnazioni, anche se io preferisco mettere sempre = . Definire valori espliciti ha particolare significato specialmente per le funzioni ricorsive, in quanto impongono delle condizioni di terminazione:

```
In[288]:= g[1] = 1;

In[289]:= g[x_] := 3 Sin[g[x - 1]]

In[290]:= g[5]
Out[290]= 3 Sin[3 Sin[3 Sin[3 Sin[1]]]]
```

Supponiamo, di dimenticarsi di definire la condizione di terminazione:

```
In[291]:= gg[x_] := 3 Sin[gg[x - 1]]
```

In[292]:=
```
gg[5];
```

```
– $RecursionLimit::reclim : Recursion depth of 256 exceeded. More…
```

Ho deciso di non visualizzare il risultato perchè sarebbe troppo lungo: comunque *Mathematica* restituisce un messaggio d'errore, dicendo che la profondità massima della ricorsione è stato raggiunto, ed il calcolo si è fermato. Può capitare, però, anche se la funzione è chiamata legittimamente, se la profondità di ricorsione supera quella consentita:

In[293]:=
```
g[300];
```

```
– $RecursionLimit::reclim : Recursion depth of 256 exceeded. More…
```

In questo caso, occorre aumentare la profondità della ricorsione, modificando il valore originale nella variabile corrispondente:

In[294]:=
```
$RecursionLimit = 330;
```

In[295]:=
```
N[g[300], 23]
```

```
– N::meprec : Internal precision limit $MaxExtraPrecision = 49.99999999999999`
    reached while evaluating 3 Sin[3 Sin[3 Sin[3 Sin[ <<1>> ]]]]. More…
```

Out[295]=
```
2.313705137424
```

Come potete vedere, possiamo alzare facilmente i limiti di calcolo di *Mathematica*, modificando le opporune variabili di sistema. Ovviamente, questo va fatto solamente se è strettamente necessario, mi raccomando!!!

I valori standard valgono anche per funzioni a più variabili. Supponiamo di definire, per esempio:

In[296]:=
```
Clear[q]
```

In[297]:=
```
q[x_Integer, y_] := Sin[x^2 y] BesselJ[x, y]
```

In[298]:=
```
q[2, 9.6]
```

Out[298]=
```
0.153515
```

In questo caso, possiamo creare dei valori espliciti che modifichino la definizione della funzione quando è presente un solo paramentro corrispondente ad un'espressione specificata:

In[299]:=
```
q[2, x_] := x^2
```

In[300]:=
```
q[2, 9.6]
```

Out[300]=
```
92.16
```

In[301]:=
```
q[2, 3]
```

Out[301]=
```
9
```

Possiamo vedere la definizione della funzione così creata:

In[302]:=
```
?q
```

```
Global`q

q[2, x_] := x^2

q[x_Integer, y_] := Sin[x^2 y] BesselJ[x, y]
```

Vediamo che le assegnazioni esplicite compaiono prima, perchè hanno precedenza sulla funzione generica:

In[303]:=
```
q[2, x_] =.
```

In[304]:=
```
q[2, 3]
```

Out[304]=
```
BesselJ[2, 3] Sin[12]
```

In questo caso, esattamente come per le variabili, abbiamo cancellato la definizione che avevamo usato in precedenza:

In[305]:=
```
?q
```

```
Global`q

q[x_Integer, y_] := Sin[x^2 y] BesselJ[x, y]
```

Come vediamo, adesso non compare più. Inoltre, quello che abbiamo detto vale anche per pattern generici, non soltanto per argomenti; per esempio posso avere qualcosa come

In[306]:=
```
Clear[f, g]
```

In[307]:=
```
f[x_] := Sin[x] + 5
```

In[308]:=
```
f[g[x_]] := Cos[x]
```

In[309]:=
```
?f
```

```
Global`f

f[g[x_]] := Cos[x]

f[x_] := Sin[x] + 5

Options[f] = {option1 → 12, option2 → 3, option3 → 1304}
```

Vedete come riconosco adesso un pattern generico specificato da me, e che inoltre posso anche utilizzare l'uguaglianza 'ritardata'. Vedremo fra poco l'utilità di questa definizione esplicita con quest'altro simbolo. Così, se *Mathematica* riconosce il pattern, esegue la sostituzione voluta:

In[310]:=
```
f[r]
```

Out[310]=
```
5 + Sin[13]
```

In[311]:=
```
f[g[r]]
```

Out[311]=
```
Cos[13]
```

Vediamo come in quest'ultimo caso abbia riconosciuto correttamente il pattern dell'argomento.

Un altro modo per definire valori standard alle funzioni, e, soprattutto, utile quando dobbiamo velocizzare calcoli ripetutti, è il seguente:

`f[`*x*`_] := f[`*x*`] = `*rhs*	definisce una funzione che ricorda i valori che trova

Questo può essere più utile di quanto si pensi: Supponiamo di avere una funzione ricorsiva, scritta in maniera poco efficiente, come per esempio la funzione per trovare il numero di Fibonacci:

In[312]:=
```
fibo[x_Integer] := fibo[x - 1] + fibo[x - 2]
```

Inizializziamo i due valori che terminano la ricorsività:

In[313]:=
```
fibo[0] = fibo[1] = 1;
```

Vediamo che questi compaiono nella definizione della funzione:

In[314]:=
```
?fibo
```

```
Global`fibo

fibo[0] = 1

fibo[1] = 1

fibo[x_Integer] := fibo[x - 1] + fibo[x - 2]
```

Come sappiamo dai nostri studi di fondamenti di informatica (perchè saper programmare un poco è una base indispensabile per ogni ingegnere, sappiatelo...). questa implementazione è estremamente inefficiente, perchè il numero di volte che viene richiamata la funzione ricorsiva all'interno della stessa cresce in maniera esponenziale al crescere dell'argomento:

In[315]:=
```
Timing[fibo[37]]
```

Out[315]=
```
{128.609 Second, 39088169}
```

Supponiamo, adesso, di definire un'altra funzione di Fibonacci, equivalente a quella che abbiamo appena scritto, ma implementandola nel modo appena visto:

In[316]:=
```
fibo2[x_Integer] := fibo2[x] = fibo2[x - 1] + fibo2[x - 2]
```

In[317]:=
```
fibo2[0] = fibo2[1] = 1;
```

Vediamo la sua definizione:

In[318]:=
```
?fibo2
```

```
Global`fibo2

fibo2[0] = 1

fibo2[1] = 1

fibo2[x_Integer] := fibo2[x] = fibo2[x - 1] + fibo2[x - 2]
```

Appare quasi equivalente a quella che abbiamo scritto prima. Tuttavia, adesso, guardate qua cosa succede:

In[319]:=
```
Timing[fibo2[37]]
```

Out[319]=
```
{0. Second, 39088169}
```

Mizzica, quanto abbiamo risparmiato!!! L'algoritmo usato era lo stesso inefficiente di prima, ma qua addirittura il tempo si è praticamente azzerato!!!

Quello che è successo è semplice, in effetti. Invece di calcolare innumerevoli volte sempre gli stessi valori della funzione per gli stessi argomenti (compariranno, per esempio, un tot volte fibo2[3], fibo2[5] e così via), calcoliamo il valore per un argomento una sola volta, e lo memorizziamo come valore predefinito. In questo modo il numero di valutazioni della funzione che dobbiamo fare cala esponenzialmente, valutandola soltanto 37 volte, invece che $a^{y\,37}$, come invece ha fatto nel caso precedente. Se vediamo la deinizione di fibo2, vediamo che effettivamente spuntano tutte le definizioni della funzione per tutti i valori preceddenti:

In[320]:=
```
?fibo2
```

```
Global`fibo2

fibo2[0] = 1

fibo2[1] = 1

fibo2[2] = 2

fibo2[3] = 3

fibo2[4] = 5

fibo2[5] = 8

fibo2[6] = 13

fibo2[7] = 21

fibo2[8] = 34

fibo2[9] = 55

fibo2[10] = 89

fibo2[11] = 144

fibo2[12] = 233

fibo2[13] = 377

fibo2[14] = 610

fibo2[15] = 987

fibo2[16] = 1597

fibo2[17] = 2584

fibo2[18] = 4181
```

```
fibo2[19] = 6765

fibo2[20] = 10946

fibo2[21] = 17711

fibo2[22] = 28657

fibo2[23] = 46368

fibo2[24] = 75025

fibo2[25] = 121393

fibo2[26] = 196418

fibo2[27] = 317811

fibo2[28] = 514229

fibo2[29] = 832040

fibo2[30] = 1346269

fibo2[31] = 2178309

fibo2[32] = 3524578

fibo2[33] = 5702887

fibo2[34] = 9227465

fibo2[35] = 14930352

fibo2[36] = 24157817

fibo2[37] = 39088169

fibo2[x_Integer] := fibo2[x] = fibo2[x - 1] + fibo2[x - 2]
```

Come possiamo vedere, abbiamo aggiunto, appena venivano calcolati per la prima volta, definizioni per gli argomenti calcolati, in modo da usare risorse computazionali solamente una volta, e salvando i risultati. Questo ha ovviamente, come abbiamo visto, un notevole vantaggio nel caso di algoritmi ricorsivi, specialmente se inefficienti come questo, ma dobbiamo comunque ricordare che il tempo risparmiato ha l'altra faccia della medaglia nella maggior memoria occupata, che per algoritmi e funzioni più serie può rappresentare un problema. Occorre sempre decidere se spendere più tempo, oppure se rischiare di bloccare il programma tentando un uso intensivo della memoria. In entrambi i casi, vi consiglio sempre di salvare una copia di backup del file... Comunque, almeno per calcoli tutto sommato semplici, e se avete un computer, non per forza dell'ultima generazione, ma almeno attuale, con 512 MB di memoria almeno, potete sempre tentare questa strada se volete. Non credo che vi capiteranno molti casi di blocco per mancanza di memoria.

Possiamo applicare casi specifici anche quando, al posto di valori specifici, abbiamo particolari pattern. Supponiamo, per esempio, di avere una generica funzione g, che però non è definita:

```
In[321]:= g[x]
Out[321]= g[3/50]
```

Lavoriamo, per semplificarci le cose, con un solo argomento, anche se quello che diciamo vale anche per più argomenti della funzione. Così fatta, la nostra *g* non gode di nessuna proprietà particolare. Tuttavia, vogliamo che goda della proprietà distributiva rispetto all'addizione. Per far questo possiamo esplicitare direttamente il pattern da utilizzare:

```
In[322]:= g[x_ + y_] := g[x] + g[y]
```

Se andiamo a riscrivere un argomento generico, non cambia ancora niente:

```
In[323]:= g[f[r]]
Out[323]= g[5] + g[Sin[13]]
```

Però, se andiamo ad inserire un argomento con il pattern corrispondente, possiamo facilmente vedere come applichi la proprietà che gli abbiamo imposto:

```
In[324]:= g[a + b + c]
Out[324]= g[{1 + b + c, 2 + b + c, 3 + b + c, 4 + b + c, 5 + b + c}]
```

Notate come, anche se il pattern è con due addendi, la proprietà distributiva si applica a tutti gli addendi. Questo perchè ho:

```
In[325]:= g[a + b + c] == g[a + b] + g[c] == g[a] + g[b] + g[c]
Out[325]= g[{1 + b + c, 2 + b + c, 3 + b + c, 4 + b + c, 5 + b + c}] ==
            g[c] + g[{1 + b, 2 + b, 3 + b, 4 + b, 5 + b}] == g[b] + g[c] + g[{1, 2, 3, 4, 5}]
```

Come vedete, i tre modi di scrivere l'espressione sono equivalenti. Dal primo si arriva al secondo caso ma, dato che il primo addendo ha ancora il pattern corrispondente, viene a sua volta riapplicata la proprietà distributiva. Possiamo, adesso, anche definirla rispetto alla moltiplicazione:

```
In[326]:= g[x_ y_] := g[x] g[y]
```

Vediamo adesso:

In[327]:=
```
g[a b + c d]
```

Out[327]=
```
g[{b + c d, 2 b + c d, 3 b + c d, 4 b + c d, 5 b + c d}]
```

Le proprietà e le forme particolari che abbiamo scritto vengono memorizzate nella definizione della funzione, esattamente come per i valori espliciti:

In[328]:=
```
?g
```

```
Global`g

g[x_ + y_] := g[x] + g[y]

g[x_ y_] := g[x] g[y]
```

A questo punto, potete vedere come, applicando le giuste proprietà, possiamo creare una funzione generica che si comporti esattamente come vogliamo noi, semplicemente andando ad applicare le regole invece che una definizione, restituendo sempre il risultato corretto, anche se in forma simbolica.

Abbiamo anche visto che, pur specificando un parametro, davamo in questo caso sempre una funzione, per cui abbiamo divuto utilizzare := invece di = .

lhs = *rhs*	*rhs* viene considerato il valore finale di *lhs* (e.g., `f[x_] = 1 - x^2`)
lhs := *rhs*	*rhs* restituisce un comando, oppure un programma, che deve essere eseguito quando inserisco un valore per *lhs* (e.g., `f[x_] := Expand[1 - x^2]`)

Un esempio utile per poter capire se la funzione è calcolata subito o quando viene chiamato il programma, consiste nell'usare il comando Random:

In[329]:=
```
a = Random[]; b = Random[];
```

Quando chiamiamo i due numeri, otteniamo dei numeri casuali:

In[330]:=
```
{a, b}
```

Out[330]=
```
{0.635399, 0.441706}
```

Se adesso li richiamiamo, riotteniamo gli stessi numeri:

In[331]:=
```
{a, b}
```

Out[331]=
```
{0.635399, 0.441706}
```

Questo perchè in *a* e *b* sono memorizzati non Random, ma il loro valore, cioè due numeri casuali fissi: scriviamo, adesso, la seguente definizione, che usa l'altro simbolo:

In[332]:=
```
a := Random[]; b := Random[];
```

In questo caso, non memorizziamo subito un valore, ma la funzione Random viene valutata ogni volta che *a* e *b* sono richiamati:

In[333]:=
```
{a, b}
```

Out[333]=
```
{0.535825, 0.450193}
```

Se adesso li chiamiamo un'altra volta, le funzioni sono rivalutate ancora, dando quindi dei numeri diversi:

In[334]:=
```
{a, b}
```

Out[334]=
```
{0.595409, 0.634603}
```

Come potete vedere, tutto si è svolto come previsto. Questo è in genere anche vero se, nella definizione, si vanno ad utilizzare delle funzioni che cambieranno man mano nel notebook: per esempio:

In[335]:=
```
Clear[f, g, h]
```

In[336]:=
```
h[x_] := Sin[x]
```

In[337]:=
```
f[x_] = h[x^2]
```

Out[337]=
$$\text{Sin}\left[\frac{9}{2500}\right]$$

In[338]:=
```
g[x_] := h[x^2]
```

Le definizioni di f e *g* sembrano uguali:

In[339]:=
```
f[4]
```

Out[339]=
$$\text{Sin}\left[\frac{9}{2500}\right]$$

In[340]:=
```
g[4]
```

Out[340]=
```
Sin[16]
```

Supponiamo, adesso, di voler ridefinire la funzione *h*:

In[341]:=
```
Clear[h]; h[x_] := Cos[x]
```

Vediamo come si comportano adesso le funzioni:

In[342]:=
```
f[4]
```

Out[342]=
$$\text{Sin}\left[\frac{9}{2500}\right]$$

In[343]:=
```
g[4]
```

Out[343]=
```
Cos[16]
```

Come possiamo vedere, nel caso della *f* non cambia niente, perchè tutto era stato valutato nella definizione: nel caso di *g*, invece, prima di calcolare la funzione nel punto 4, abbiamo valutato l'espressione della funzione, e, dato che *h* era cambiato, l'espressione, ricalcolata dopo la modifica di *h* risulta pure modificata di conseguenza. Nella maggior parte dei casi conviene utilizzare l'assegnazione ritardata := anche perchè permette di modificare una funzione semplicemente andando a modificarne una sua parte, cosa utile se, per esempio, la definizione è un programma con molte sottofunzioni definite a parte

Regole di trasformazione

Come abbiamo visto anche prima, le regole di trasformazione sono un metodo abbastanza potente e veloce per poter dotare le nostre funzioni di particolari proprietà, oltre che poter manipolare le espressioni in maniera avanzata e personalizzata.

Alla luce di quanto imparato finora, tuttavia, come ad esempio i pattern, possiamo vedere nuove potenzialità delle regole di trasformazione. Un esempio classico di trasformazione è la seguente:

In[344]:=
```
Sin[x y] y x / z + z /. x → z
```

Out[344]=
$$z + \frac{9\,\text{Sin}\left[\frac{9}{10}\right]}{10\,z}$$

Possiamo anche creare, mediante pattern, altre modifiche, che comportino lo stesso cambiamento a più parti dell'espressione aventi la stessa struttura:

In[345]:=
```
Sin[x y] y x / z + z /. (x | y | z) → Cos[x]
```

Out[345]=
$$\text{Cos}\left[\frac{3}{50}\right] + \frac{9}{10}\,\text{Sec}\left[\frac{3}{50}\right]\,\text{Sin}\left[\frac{9}{10}\right]$$

Le manipolazioni ottenibili sono parecchie.

Dato che tutto quanto possiede un nome, in quanto tutto può essere scritto sotto forma di espressione, possiamo anche definire variabili che contengano regole di sistituzione:

In[346]:=
```
regola = Sin[x_] → Cos[x];
```

In[347]:=
```
Sin[x^2] Sin[x + 1] /. regola
```

Out[347]=
$$\text{Cos}\left[\frac{3}{50}\right]^2$$

Inoltre, abbiamo anche visto come fare per effettuare sostituzioni ripetute:

In[348]:=
```
a + b /. {a → b, b → c}
```

Out[348]=
```
0.409006
```

In questo caso, si vede che la b non è stata restituita correttamente, perchè applichiamo le definizioni una volta sola. Se vogliamo che siano ripetute fino a quando l'espressione non varia più, dobbiamo usare l'altra sintassi:

In[349]:=
```
a + b //. {a → b, b → c}
```

Out[349]=
```
0.950903
```

Come potete vedere, in questo caso le sostituzioni vengono effettuate iterativamente.

Tuttavia, ci possono essere casi in cui le iterazioni non portano mai ad un'espressione che rimane invariata, e questo ne è un esempio banalissimo:

In[350]:=
```
a //. {a → b, b → c + a}
```

Out[350]=
```
0.852091
```

Se proviamo ad eseguire questo comando, *Mathematica* si blocca in un loop infinito. Possiamo usare la forma completa del comando, aggiungendo un opzione che tenga conto del massimo numero di iterazioni:

In[351]:=
```
ReplaceRepeated[a, {a → b, b → c + a}, MaxIterations → 20]
```

Out[351]=
```
0.769572
```

In questo caso, vediamo che il programma ci restituisce un warning, dicendo che non è riuscito a trovare l'espressione definitiva (quella, cioè, che non cambia più applicando le regole), entro il numero di iterazioni previste, restituendo il risultato ottenuto con l'ultima iterazione. Questo, però, se ben usato, può essere un modo per poter effettuare delle trasformazioni cicliche a delle espressioni senza andare ad usare cicli Do e For, sempre, ovviamente, che si sa quello che si sta facendo, e fregandosene del warning, che in questo caso non serve a niente.

Questo può essere vero anche quando, per esempio, si ha una lista di regole che trasformano uno stesso pattern in espressioni diverse. In questo caso *Mathematica* applica solamente la prima che trova:

In[352]:=
```
a + b /. {x_ + y_ → x^y, _ + _ → c}
```

Out[352]=
```
1.5096
```

Si vede che prima applica la prima regola di sostituzione, poi, non essendo più il pattern una somma, la seconda regola viene ignorata. Se vogliamo invece avere una lista che comprenda entrambe le sostituzioni, possiamo usare il seguente comando:

`ReplaceList[`*expr*`, `*rules*`]`	applica *rules* in tutti i modi possibili

Vediamo l'effetto di questo comando nell'espressione di prima:

In[353]:=
```
ReplaceList[a + b, {x_ + y_ → x^y, x_ + y_ → c}]
```

Out[353]=
```
{}
```

Vediamo che ha effettivamente applicato tutti i modi possibili di eseguire la lista di sostituzioni: prima di tutto ha effettuato la prima regola, sfruttando il fatto che la somma è commutativa, quindi $a + b == b + a$, usando quindi entrambi i modi per scrivere la funzione: dopo ha eseguito nello stesso modo la seconda regola, dando in entrambi i casi, come risultato, c. Naturalmente questo può essere utile anche per espressioni più complicate:

In[354]:=
```
espr = x y z + a b c + w e r;
```

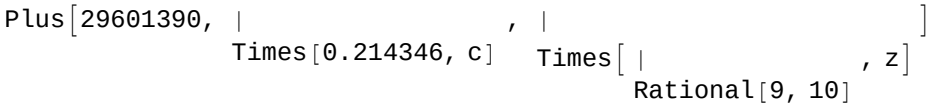
In[355]:=
```
ReplaceList[espr, {x_ y_ → Sin[x y], x_ + y_ → Cos[x y]}]
```

Out[355]=
$$\left\{\cos\left[\frac{9}{10}\right], \cos\left[\frac{9}{10}\right], \cos\left[\frac{9}{10}\right], \cos\left[\frac{9}{10}\right], \cos\left[\frac{9}{10}\right], \cos\left[\frac{9}{10}\right]\right\}$$

Vediamo che qualcosa non va. Infatti, sebbene applichi la seconda regola correttamente, non lo fa altrettando per la prima:

In[356]:=
```
TreeForm[espr]
```

Out[356]//TreeForm=



Il problema consiste nel fatto che ReplaceList agisce solamente, come Replace, nel primo livello. Mentre però con Replace possiamo specificare la profondità di livello a cui applicare le sostituzioni, non si fa altrettanto con ReplaceList, dato che il terzo argomento (opzionale) decide solamente il massimo numero di elementi di cui deve essere composta la lista (che di default è Infinity). Per applicare ReplaceList a qualsiasi profondità, dobbiamo inventarci la nostra funzione, che creiamo in questo modo:

In[357]:=
```
ReplaceAllList[expr_, rules_] := Module[{i},
  Join[
   ReplaceList[expr, rules],
   If[
    AtomQ[expr], {},
    Join @@ Table[ReplacePart[expr, #, i] & /@ ReplaceAllList[
        expr[[i]], rules
       ]
      , {i, Length[expr]}
     ]
   ]
  ]
 ]
```

Module, vedremo più avanti, serve per creare una variabile locale, in questo caso i, che verrà usata dalle espressioni e comandi successivi. Dopo di che, vediamo il funzionamento della funzione. Join serve a concatenare liste fra di loro, oppure espressioni con lo stesso head. Il primo argomento di Join è la lista ottenuta dal comando semplice ReplaceList. Come secondo argomento, invece, abbiamo un'altra funzione.

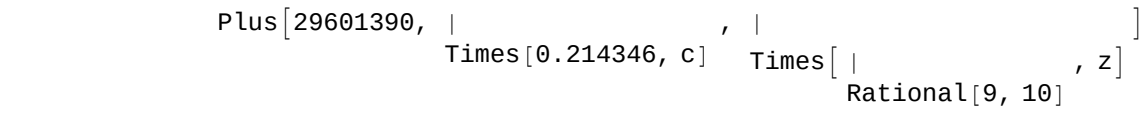
L'If serve per vedere se l'espressione ha un solo livello. In caso affermativo, l'operazione è finita, e restituisce una lista nulla che, unita a quella ottenuta in precedenza, da la lista come avremmo avuto con ReplaceList. In caso contrario, invece, eseguiamo il terzo argomento del comando If.

Nel terzo argomento, applichiamo Join (@@ rappresenta, vi ricordo, il comando Apply), agli elementi della lista ottenuta mediante Table. Quest'ultima lista sarà formata dall'espressione, con l'i-simo elemento (del primo livello), con l'elemento stesso a cui è stata applicata la stessa funzione. Come possiamo vedere, è una funzione ricorsiva.

Quello che fa, in pratica, è questo: sostituisce il primo livello con ReplaceList; poi prende il primo elemento, il primo elemento del primo elemento e così via ricorsivamente, fino all'ultimo livello. A questo punto applica ReplaceList a questo elemento, e sale di un livello, applicando ReplaceList pure a questo livello, e così via tornando su fino al primo livello. A questo punto ottengo una lista di espressioni con tutte le sostituzioni del primo elemento, e passo al secondo.

In[358]:=
```
TreeForm[espr]
```

Out[358]//TreeForm=



In questo caso, prima applico le regole al primo livello. Poi le applico al primo elemento del secondo livello, e così via.

Da spiegare, effettivamente, è difficile (come lo sono molte funzioni ricorsive complicate), ma imparare questo esempio, e capire come funziona serve a darvi un quadro generale su come funziona *Mathematica*. Se capite questo esempio, vuol dire che siete già abbastanza bravi col programma. Meglio non ve lo so spiegare, mi dispiace.

In[359]:=
```
ReplaceAllList[espr, {x_ y_ → Sin[x y], x_ + y_ → Cos[x y]}]
```

Out[359]=
$$\left\{\text{Cos}\left[\frac{9}{10}\right], \text{Cos}\left[\frac{9}{10}\right], \text{Cos}\left[\frac{9}{10}\right], \text{Cos}\left[\frac{9}{10}\right], \text{Cos}\left[\frac{9}{10}\right], \text{Cos}\left[\frac{9}{10}\right], 29601390 + \frac{9\,z}{10} + \text{Sin}\left[\frac{9}{10}\right], 29601390 + \frac{9\,z}{10} + \text{Sin}\left[\frac{9}{10}\right], 29601390 + 0.214346\,c + \text{Sin}\left[\frac{9}{10}\right], 29601390 + 0.214346\,c + \text{Sin}\left[\frac{9}{10}\right]\right\}$$

Vediamo come ci siano più combinazioni rispetto a prima, anche se ancora non ci sono tutte quelle disponibili (mancano, per esempio, quelle con Sin e Cos assieme nidificati). Dubito comunque che vi servirete spesso di queste manipolazioni; per la maggior parte dei casi il primo livello basta e avanza (altrimenti sicuramente *Mathematica* avrebbe avuto un comando per far meglio quello che abbiamo appena fatto...).

Attributi

Mediante gli attributi, siamo in grado di conferire particolari proprietà alle funzioni che andiamo a definire. Questo è importante quando vogliamo attribuire particolari proprietà matematiche a delle funzioni che, magari non sono definite, oppure se vogliamo che *Mathematica* applichi automaticamente alcune trasformazioni e regole definite dalle proprietà della funzione. Tuttte le funzioni predefinite di *Mathematica* hanno una serie di attributi che descrivono alcune delle loro propietà:

`Attributes[`*f*`]`	restituisce una lista contenente gli attributi di *f*
`Attributes[`*f*`] = {`*attr*$_1$`,` *attr*$_2$`,` … `}`	definisce degli attributi per *f*
`Attributes[`*f*`] = {}`	togliamo ad *f* tutti *i* suoi attributi
`SetAttributes[`*f*`,` *attr*`]`	aggiunge *attr* agli altri attributi della *f*
`ClearAttributes[`*f*`,` *attr*`]`	rimuove *attr* dagli attributi della *f*

Se vediamo una qualsiasi funzione, vediamo i suoi attrributi:

In[360]:=
```
Attributes[Plus]
```

Out[360]=
```
{Flat, Listable, NumericFunction, OneIdentity, Orderless, Protected}
```

Vediamo che restitusce una lista di attributi. Elenchiamo, adesso, i vari attributi, ed il loro significato:

- **Ordeless**: una funzione i cui argomenti possono scambiarsi di posizione. Saranno ordinati, per il calcolo, in posizione standard.
- **Flat:** definisce una funzione associativa, passando, quindi, il comando Flatten ai suoi argomenti, se la funzione viene chiamata come argomento di se stessa
- **OneIdentity:** nel matching dei pattern, $f[f[a]]$, e così via, viene visto equivalente ad *a*
- **Listable:** la funzione viene automaticamente mappata nelle liste che compaiono come argomento; quindi la funzione applicata ad una lista diventa la lista della funzione applicata ad ognuno degli argomenti
- **Constant:** tutte le derivate della funzione sono nulle
- **NumericFunction:** si assume che la funzione restituisca un valore numerico, quando i suoi argomenti sono pure quantità numeriche
- **Protected:** i valori della funzione non possono essere cambiati
- **Locked:** gli attributi della funzione non possono essere cambiati

- **ReadProtected:** i valori della funzione non possono essere letti
- **HoldFirst:** il primo argomento della funzione non viene valutato
- **HoldRest:** tutti gli argomenti, tranne il primo, non vengono valutati
- **HoldAll:** nessuno degli argomenti della funzione vengono valutati
- **HoldAllComplete:** gli argomenti della funzione sono considerati completamente inerti
- **NHoldFirst:** al primo argomeno della funzione non viene applicato il comando N
- **NHoldRest:** a tutti gli argomenti della funzione, tranne che al primo, non viene applicato il comando N
- **NHoldAll:** N non viene valutato in nessun argomento della funzione
- **SequenceHold:** gli oggetti Sequence che appaiono negli argomenti della funzione non vengono appiattiti
- **Temporary:** la funzione viene trattata come variabile locare, venendo rimossa quando non è più usata
- **Stub:** il comando Needs viene automaticamente chiamatato persino se la funzione ha un input esplicito

Se adesso, andiamo a rivedere gli attributi della funzione Plus, possiamo vedere che è Flat, cioè che gode della proprietà associativa:

```
In[361]:= Plus[a, Plus[b, c]] // FullForm
Out[361]//FullForm= Plus[1.1704507214396762`, c]
```

Vediamo che è Listable, quindi che viene mappata nelle liste:

```
In[362]:= Plus[{a, b, c}, {d, e, f}] // FullForm
Out[362]//FullForm= List[Plus[0.9376438006411553`, d], 42.11352905166384`, Plus[c, f]]
```

Gode dell'attributo NumericFunction, quindi, se ha quantità numeriche come argomenti, la funzione viene definita quantità numerica:

```
In[363]:= NumericQ[Plus[4, 7, 4]]
Out[363]= True
```

Ha anche l'attributo OneIdentity, cioè gode dell'identità nel pattern matching

```
In[364]:= Plus[Plus[a]] === a
Out[364]= False
```

Viene attribuito a questa funzione l'attributo Ordeless, che fa godere alla funzione la proprietà commutativa:

```
In[365]:= Plus[n, f, u, a] // FullForm
Out[365]//FullForm= Plus[41.855646347601024`, f, u]
```

Notate come, avendo questo attributo, gli argomenti vengono automaticamente riordinati.

Infine, come capita per tutte le funzioni predefinite di *Mathematica*, gode dell'attributo Protected, che impedisce di modificare la funzione, a meno che non si specifichi diversamente in maniera esplicita:

```
In[366]:= Plus[3, 5, 3] = 5
  – Set::write : Tag Plus in 3 + 3 + 5 is Protected. More…
Out[366]= 5
```

Supponiamo di voler dare delle particolari proprietà ad una funzione, senza definirla per forza; per esempio, vogliamo che una generica funzione f sia associativa e commutativa:

```
In[367]:= SetAttributes[f, {Orderless, Flat}]
```

Vediamo se funziona:

```
In[368]:= f[v, d, c, y, t]
Out[368]= f[3, 15, c, d, v]

In[369]:= f[a, f[v, b]]
Out[369]= f[0.588522, 0.671335, v]
```

Uao!!!!!!!! E neanche sappiamo cosa sia la f!!!

Come potete vedere, riusciamo in questa maniera a definire particolari proprietà per una funzione generica, che magari viene difficile oppure impossibile da definire con le corrispondenze fra

argomenti e pattern, che abbiamo visto sopra. Per esempio, in questa maniera è estremamente più semplice applicare l'attributo per la proprietà commutativa, che andare a creare una regola complicata (che comunque abbiamo visto prima), per l'ordinamento degli argomenti della funzione.

- **Programmazione avanzata**

Introduzione

In questa sezione affrontiamo alcuni aspetti particolari della programmazione, che non abbiamo ancora visto. Sebbene quello che già sapete dovrebbe bastarvi, per alcuni casi (cresceranno in numero, man mano che diventerete più bravi), occorre porre particolare attenzione a quello che si fa, e trovare modi per sbrigarsela e risolvere problemi noti in programmazione classica, come, ad esempio, quello dello scoping delle variabili definite...

Visibilità di variabili locali

Ogni volta che andiamo a definire qualsiasi espressione, funzione etc, vengono considerati come variabili globali; cioè, una volta che sono definite potete utilizzarle in qualsiasi punto del vostro lavoro. Potete anche definire una funzione in un notebook, se volete, ed andarla ad utilizzare in un altro notebook, senza doverla riscrivere, dato che sarà già in memoria.

Tuttavia, come ogni ragazzo per bene che abbia programmato qualcosa di più del famigerato "Hello World!" che vi insegnano sempre, non è buona cosa, nei programmi, affidarsi alle variabili locali. Questo è vero anche in *Mathematica*, specialmente quando si devono definire dei programmi, oppure delle funzioni un poco più complicate. Per esempio, supponete di dover utilizzare un ciclo For all'interno di una funzione. Dovete definire un iteratore ma, se caricate la funzione da qualche altra parte, niente vi dice che il nome usato per l'iteratore non sia già stato utilizzato, incasinandovi le cose anche di parecchio. Per questo è importante il concetto di visibilità, sia delle variabili che di tutto il resto: d'altronde, come avrete capito, in *Mathematica* 'tutto' è un espressione....

Come tutti sapete, uno degli ambiti principali in cui viene applicata la visibilità consiste nella definizione delle variabili locali. Supponete di avere qualcosa del genere:

In[370]:=
```
i = 3; i = i^3 + t; i = i^3 + t
```

Out[370]=
```
27003
```

In[371]:=
```
fun[x_] := For[i = 0; y = 0, i < 20, i++, y = y + x^i]
```

In[372]:=
```
fun[500]
```

In[373]:=
```
y
```

Out[373]=
```
191117097476202404809619238476953907815631262525050l
```

In[374]:=
```
i
```

Out[374]=
```
20
```

Vediamo che il valore della variabile *i*, che abbiamo usato all'interno della funzione, ha modificato il valore della *i* che se ne stava fuori per i fatti suoi. Effettivamente, dato che parliamo di variabili globali, per *Mathematica* sono la stessa identica variabile. Però, probabilmente, non vogliamo che questo succeda, perchè la *i* che avevamo definito conteneva un espressione che magari ci serviva per effettuare vari calcoli, mentre quella della funzione non conteneva informazioni, ma era solamente un indice per poter effettuare una determinata funzione. Per cui, non vogliamo che siano la stessa cosa.

Quando però si comincia a creare un notebook che superi il paio di schermate, diventa difficile ricordarsi ciò che avevamo definito, quello che avevamo cancellato e così via. Inolte, potremmo voler memorizzare questa funzione in un package personale, da richiamare quando vogliamo, e potremmo non ricordarci più che all'interno era stata ridefinita la variabile i, complicandoci la situazione.

La soluzione a questo problema, naturalmente, c'è, ed è data dal seguente comando:

`Module[{`*x*`, `*y*`, … }, `*body*`]`	un blocco contenente le variabili locali *x*, *y*, …

Module, in altre parole, definisce una porzione di programma, di funzione o di qualsiasi altra cosa, in cui si definiscono delle variabili locali. A questo punto, le eventuali variabili globali avente lo stesso nome di quelle locali vengono preservate e non vengono modificate, ma vengono modificate le rispettive variabili locali. Inoltre, non appena si esce dal blocco, le variabili sono cancellate. Ripetiamo l'esempio di prima, utilizzando questa volta le variabili locali:

In[375]:=
```
i = 3; i = i^3 + t; i = i^3 + t
```

Out[375]=
```
27003
```

In[376]:=
```
fun[x_] := Module[{i},
  For[i = 0; y = 0, i < 20, i++, y = y + x^i]
 ]
```

In[377]:=
```
fun[500]
```

In[378]:=
```
y
```

Out[378]=
```
191117097476202404809619238476953907815631262525050l
```

In[379]:=
```
i
```

Out[379]=
```
27003
```

Vediamo che abbiamo ottenuto lo stesso risultato, e che, inoltre, il valore della variabile *i* che si trova fuori dal blocco non viene modificato. Questo perchè, all'interno del blocco stesso, la i utilizzata era quella locale, non quella globale. Guardando più attentamente il codice ed il risultato, notiamo però un'altra cosa: la variabile *y* continua ad essere vista come variabile globale. Infatti, al di fuori del blocco ha lo stesso valore di quello che ha all'interno. Questo perchè non l'avevamo definita esplicitamente come variabile locale, rimanendo quindi, a tutti gli effetti, una variabile globale. Quindi, se avevamo in precedenza definito già la *y*, il suo valore verrebbe comunque riscritto. Bisogna stare attenti, quindi, a definire come locale tutto quello che ci serve, perchè capiterà di salvare delle funzioni e dei programmi, poi di ricaricarli e riutilizzarli altre volte senza che ci preoccupiamo di come l'abbiamo fatte, e sapendo solo il risultato che danno.

Vi lascio come esercizietto la modifica alla funzione, che vi permetta di definire anche la *y* come variabile locale, ma che, una volta chiamata la funzione, invece di non restituire niente, restituisca direttamente il valore calcolato. Ormai sapete farlo sicuramente, e fidatevi che è una cavolata!!! Ma d'altronde, dovete sbatterci almeno una volta la vostra fida testolina, no?

Inotre, all'interno della lista delle variabili locali, potete direttamente porre un valore di inizializzazione, se vi serve:

In[380]:=
```
Module[{x, i = 0}, x = i + 12; i = x * 234]
```

Out[380]=
```
2808
```

In[381]:=
```
i
```

Out[381]=
```
27003
```

Notate come i sia rimasto ancora invariato....

Inoltre, potete anche definire dei blocchi dove compaiano delle costanti, invece che delle variabili:

`With[{`$x = x_0$`, `$y = y_0$`, … }, `*body*`]`	definisce delle costanti locali x, y, …

Rispetto ai linguaggi di programmazione normali, però, le costanti non devono essere per forza associate ad un numero, ma possono essere anche delle espressioni:

In[382]:=
```
With[{q = t + 23}, Sin[q]]
```

Out[382]=
```
Sin[26]
```

Visto in un certo modo, possiamo considerare questo comando come l'equivalente di /. quando andiamo a scrivere dei programmi, che permette di sostituire ad opportune variabili quello che vogliamo, se presenti nel blocco.

Naturalmente questi blocchi possono anche essere nidificati fra loro; in questo caso le variabili locali che si useranno saranno quelle definite nello stesso blocco: se non sono definite nello stesso blocco, saranno le variabili locali del blocco immediatamente superiore e così via:

In[383]:=
```
Module[{i = 5},
 Module[{i = 4},
  Module[{i = 3}, Print[i]];
  Print[i]
 ];
 Print[i]
]
```

```
3

4

5
```

In questo esempio, si procede con ordine. Scorrendo il flusso di programma, il primo comando Print che incontriamo si trova nel blocco più interno; in questo blocco la variabile locale *i* è pari a 3, ed è questo il valore che viene stampato. Dopo, si incontra il secondo Print, che è conenuto nel blocco Module dove la variabile *i* è inizializzata a 4, e viene stampato questo valore. Infine, l'ultimo Print si trova nel blocco dove *i* vale 5, stampando il corrispondente valore.

Volendo, però, potremmo vedere in un blocco nidificato il valore di una variabile globale, o locale definita in un blocco superiore, anche se una variabile locale nel nostro blocco ne mina la visibilità. Infatti, *Mathematica*, nel risolvere le variabili locali, le rinomina:

```
In[384]:= Module[{i},
           Module[{i},
            Module[{i}, Print[i]];
            Print[i]
           ];
           Print[i]
          ]

          i$105

          i$104

          i$103
```

Come potete vedere, ogni variabile è rinominata in maniera univoca, venendo numerata. possiamo, quindi, utilizzare questa numerazione per accedere direttamente alle variabili che ci interessano. Tuttavia non si fa mai (almeno io non l'ho mai fatto), perchè, come potete notare, la numerazione non parte dall'inizio, ma dipende da quando si è chiamato il blocco etc. Il mio consiglio è quello di utilizzare nomi diversi per variabili che vi servono in diversi blocchi annidati, anche perchè in questo caso potrete dargli nomi più sensati, e che non variano a seconda di quando e dove si richiama la funzione oppure il programma.

E, mentre con Module possiamo trattare i nomi delle variabili come locali, con Block possiamo trattarne soltanto i valori:

`Block[{`*x*`, `*y*`, … }, `*body*`]`	calcola *body* usando valori locali per *x*, *y*, …
`Block[{`*x* `=` x_0`, `*y* `=` y_0`, … }, `*body*`]`	assegna valori iniziali ad *x*, *y*, …

In apparenza questo blocco si comporta come Module:

```
In[385]:= x = 20; y = 30;

In[386]:= Block[{x = 4, y = 5}, x + y]

Out[386]= 9

In[387]:= x

Out[387]= 20

In[388]:= y

Out[388]= 30
```

Tuttavia bisogna notare un'importante differenza concettuale. Module rende univoci i nomi delle variabili, mentre Block rende univoci i valori.

Abbiamo visto che, all'interno di Module, le variabili sono rinominate, assumento un altro nome. Questo è importante quando si pensa di utilizzare espressioni definite fuori dai blocchi, che contengono il nome delle variabili. consideriamo, per esempio, la seguente espressione:

```
In[389]:= es = a + Cos[b];

In[390]:= a = 30; b = 50;
```

Se vogliamo utilizzare questa espressione all'interno di un blocco, dobbiamo stare attenti. Verrebbe da usare Module, ma ciò sarebbe sbagliato:

```
In[391]:= Module[{a = 3, b = 4}, es]
Out[391]= 0.627023
```

Come possiamo vedere, il risultato non viene modificato: ma, in fondo, non deve stupirci più di tanto, perchè nella valutazione di es non utilizziamo i valori definiti con Module, perchè abbiamo visto che corrispondono a qualcosa coma x$15. Allora *Mathematica*, non riconoscendo il nome delle veriabili di es definite all'interno del blocco, usa quelle globali, che erano state definite prima, per il calcolo dell'espressione. Per poter utilizzare l'espressione con valori locali, occorre che le variabili locali definite all'interno del blocco abbiano lo stesso nome di quelle globali, ed a questo pensa Block:

```
In[392]:= Block[{a = 3, b = 4}, es]
Out[392]= 0.627023
```

In questo caso vediamo che i nomi sono rimasti quelli, mentre abbiamo assunto come locali i valori assunti. Infatti, nonostante il risultato sia corretto, dal punto di vista globale i valori delle variabili non sono cambiati:

```
In[393]:= {a, b}
Out[393]= {30, 50}
```

Occorre prestare attenzione, nella progettazione dei programmi, quale dei due blocchi è più consono all'utilizzo che se ne vuole fare, utilizzando di volta in volta quello che semplifica maggiormente le cose.