Capitolo 4

■ 4.1 Uso di If e Which

Mathematica fornisce vari strumenti per specificare che una particolare espressione venga calcolata solo se determinate condizioni sono soddisfatte.

Il comando **If[p,then,else]** fornisce il valore **then** se **p** e' *True* e il valore **else** se **p** e' *False*. Se **p** non e' ne' *True* ne' *False* non fornisce nessun valore. **If[p,then,else,other]** ha invece come risultato **other** se **p** non e' ne' *True* ne' *False*. Osservate i seguenti esempi

```
f[x_]:=If[x>0,x^2,-x^2]
Plot[f[x],{x,-2,2},AxesLabel->{x,y}]
```

```
Clear[f]
f[x_] := If[(x < 0 || x > 1), x^2, -x^3]
Plot[f[x], {x, -2, 2}]
```

Il comando **If** permette di scegliere tra due alternative. Se si hanno piu' alternative o si usano piu' **If** incapsulati oppure si usa la funzione **Which[cond1,result1,cond2,result2,...,condn,resultn]** che valuta *condi* e ritorna il valore *resulti* corrispondente alla prima condizione verificata. Se nessuna condizione e' verificata non assegna nessun valore.

```
h[x_]:=Which[x<0,x^2,x>1,x^3,True,1];Plot[h[x],{x,-1,2}]
```

La stessa funzione si poteva anche definire usando due **If** annidati

```
g[x_] := If[x < 0, x^2, If[x > 1, x^3, 1]]; Plot[g[x], {x, -1, 2}]
```

Esercizio. Definite (in due modi diversi) la funzione a tratti che vale -1 se x<-2, vale cos(xπ/2) se x appartiene a [-2,2] e vale -sin[xπ/4] se x>2. Disegnate la funzione in [-3,3], specificando 2 opzioni a scelta.■

Esempio. Usando il comando **If (Which)**, ridefiniamo la funzione **EvenQ[n]** (**OddQ[n]**) e definiamo sugli interi la funzione **treq[n]** che vale **True** se n e' multiplo di 3 e **False** altrimenti.

```
? EvenQ
? OddQ

EvenQ[expr] gives True if expr is an even integer, and False otherwise.
 More…

OddQ[expr] gives True if expr is an odd integer, and False otherwise.
 More…
```

```
evenq[n_] := If[IntegerQ[n / 2], True, False]
oddq[n_] := Which[IntegerQ[(n + 1) / 2], True, True, False]
treq[n_Integer] := If[IntegerQ[n / 3], True, False]
```

■ 4.2 Uso di Do, While, For

Ci sono diversi modi di chiedere a *Mathematica* di ripetere molte volte la stessa operazione. Una possibilita' e' usare il comando **Do**, che agisce come l'analogo comando in Fortran o in C. La sintassi esatta e' **Do[*expr*,{n}]** che esegue *expr* ripetutamente n volte;

Do[*expr*,{i,imax}] che esegue *expr* ripetutamente per i=1,...,imax;

Do[*expr*,{i,imin,imax}] che esegue *expr* per i=imin,...,imax;

Do[*expr*,{i,imin,imax,di}] con i che varia da imin a imax con step *di*.

```
Clear["Global`*"]
Do[Print[i!], {i, 3}]
```

Ripetiamo tre volte l'istruzione $t = \frac{1}{2+t}$

```
t = x; Do[t = 1 / (2 + t), {3}]; t
```

Definiamo la funzione $f(x) = \sum_{i=1}^{7} \frac{1}{(x+i)}$ (in realta' esiste un modo piu' semplice ...)

```
a = 0; Do[a = a + 1 / (x + i), {i, 7}]; f[x_] := a
```

Il comando **While[test,body]** esegue ripetutamente *test* e *body* fino a che il *test* non e' verificato. Se *body* e' costituito da piu' termini, questi si separano da punto e virgola **;** . Guardate i seguenti esempi

```
n = 0; While[n < 10, n = n + 1; Print[n]]
a = 0; n = 0; While[(n = n + 1) < 10, a = a + 1 / n]; a
a = 0; n = 0; While[n^3 < 197, n = n^2 + 1; a = a + n]; a
a = 0; n = 0; While[(n = (n + 1)^2) < 10, a = a + 1 / n]; a
a = 0; n = 18; While[ n ≠ 0, n = Floor[n / 2]; a = a + n; Print[n]]; a
```

```
?Floor
```

```
Floor[x] gives the greatest integer less than or equal to x. More...
```

Il comando **For[start,test,incr,body]** esegue *start* poi ripetutamente esegue *test* e quindi calcola *body* e *incr* e si ferma appena *test* non e' piu' verificato. Se *start* e *body* sono costituiti da piu' termini, questi si separano da **;** Sia **While** che **For** prima eseguono il *test* di verifica e poi eseguono il comando. Per esempio calcoliamo 1/5! e poi 1/5!! **i++** incrementa i di 1 cioe' i=i+1

```
For[i = 1; a = 1, i < 6, i++, a = a / i]; a
```

```
For[i = 1; a = 1, i ≤ 5, i = i + 2, a = a / i]; a
```

Il seguente ciclo termina quando $i^2 < 10$

```
For[i = 1; t = x, i^2 < 10, i++, t = t^2 + i; Print[t]]
```

i-- decrementa i di 1 cioe' i=i-1, **i*=c** moltiplica i per c cioe' i=i*c

```
For[t = 10, t > 0, t--, Print[t]]
```

```
For[t = 1, t < 19, t *= 2, Print[t]]
```

Consultate l'**Help Browser** per maggiori informazioni sui comandi descritti in questo paragrafo**.**

Esempio. Costruiamo una funzione di due variabili x,y che assume il valore 0 in (0,0), vale -1 lungo l'asse x, vale -2 lungo l'asse y e vale *i* se (x,y) appartiene al quadrante *i*. Possiamo costruiamo questa funzione in piu' modi.

```
f1[0, 0] := 0
f1[x_, 0] := -1
f1[0, y_] := -2
f1[x_, y_] := 1 /; (x > 0 && y > 0)
f1[x_, y_] := 2 /; (x < 0 && y > 0)
f1[x_, y_] := 3 /; (x < 0 && y < 0)
f1[x_, y_] := 4 /; (x > 0 && y < 0)
{f1[0, 0], f1[1, 0], f1[0, 4], f1[1, 1], f1[-1, 2], f1[-4, -4], f1[1, -3]}
```

Osservate che la definizione specifica prevale sulla definizione generale.

Usiamo ora il comando **If**

```
f2[x_, y_] := If[(x == 0 && y == 0), 0,
  If[y == 0, -1,
   If[x == 0, -2,
    If[(x > 0 && y > 0), 1,
     If[(x < 0 && y > 0), 2,
      If[(x < 0 && y < 0), 3,
       4]]]]]]
{f2[0, 0], f2[1, 0], f2[0, 4], f2[1, 1], f2[-1, 2], f2[-4, -4], f2[1, -3]}
```

Usiamo il comando **Which**

```
f3[x_, y_] := Which[(x == 0 && y == 0), 0,
  y == 0, -1,
  x == 0, -2,
  (x > 0 && y > 0), 1,
  (x < 0 && y > 0), 2,
  (x < 0 && y < 0), 3, True, 4]
{f3[0, 0], f3[1, 0], f3[0, 4], f3[1, 1], f3[-1, 2], f3[-4, -4], f3[1, -3]}
```

Oppure possiamo combinare l'uso di **If**, **Which**, **/;**

Per esempio

```
f4[0, 0] := 0
f4[x_, 0] := -1
f4[0, y_] := -2
f4[x_, y_] := If[x < 0, 2, 1] /; y > 0
f4[x_, y_] := If[x < 0, 3, 4] /; y < 0
{f4[0, 0], f4[1, 0], f4[0, 4], f4[1, 1], f4[-1, 2], f4[-4, -4], f4[1, -3]}
```

Esercizio. Definire la funzione segno(x), che vale 0 in 0, vale 1 se x>0 e vale -1 se x <0, in tre modi diversi.

Esempio. Per ciascuno intero da 100 a 110 controlliamo se e' pari, dispari e se e' divisibile per 3.

```
For[i = 100, i ≤ 110, i++,
 If[evenq[i], Print[i, " e' pari"], Print[i, " e' dispari"]];
 If[treq[i], Print[i, " e' multiplo di 3"],
  Print[i , "  non e' multiplo di 3"]]]
```

Esempio. Definiamo una funzione **test[matrice]** che verifichi se la matrice e' quadrata e simmetrica, se e' quadrata ma non simmetrica e se non e' quadrata.

```
Clear["Global`*"]
test[m_ /; MatrixQ[m]] := If[(Dimensions[m][[1]] == Dimensions[m][[2]]),
  If[m == Transpose[m], "la matrice e' quadrata e simmetrica",
   "la matrice e' quadrata ma non simmetrica"],
  "la matrice non e' quadrata"]

testBIS[m_ /; MatrixQ[m]] :=
 Which[(Dimensions[m][[1]] == Dimensions[m][[2]]) &&
    (m == Transpose[m]), "la matrice e' quadrata e simmetrica",
   (Dimensions[m][[1]] == Dimensions[m][[2]]) &&
    (m != Transpose[m]), "la matrice e' quadrata ma non simmetrica",
   (Dimensions[m][[1]] != Dimensions[m][[2]]), "la matrice non e' quadrata"]
```

■ 4.3 Uso di Module

Se la funzione da costruire e' piu' complicata, puo' essere utile usare il comando **Module**, che usa anche variabili locali cioe' variabili che non interferiscono al di fuori del corpo del comando **Module**. La sintassi e' **f[x_,y_,...]:=-Module[{lista di variabili locali}, espr1;...;esprn].** Dei vari calcoli effettuati (separati da **;**) l'ultimo e' quello ritornato effettivamente cioe' **exprn**. Per esempio il seguente comando considera **t** variabile globale

```
t = 10
```

mentre **t** all'interno del comando **Module** e' una variabile locale che puo' essere trattata indipendentemente dalla variabile globale **t**

```
Module[{t}, t = 25; t]
```

La variabile globale **t** ha sempre il valore assegnato inizialmente:

```
t
```

Prima di eseguire i seguenti comandi, intuite l'output:

```
t = 11;
pp[x_] := Module[{t}, t = 25; t * x]; pp[x]

t = 5;
gg[x_] := Module[{t}, t = x^2 + 4; Log[t]]; gg[x]

a = 3; q = 5; hh[x_] := Module[{a, q}, a = Sin[x]; q = Exp[a]]; hh[x]

ff[x_] := Module[{t}, t = (1 + x + x^2)^4; Expand[t]]; ff[x]

kk[x_] := Module[{t}, t = Sin[x]; Log[t] /; (Sin[x] > 0)]
kk[Pi / 3]
kk[0]
kk[-1]
```

Il modo in cui **Module** lavora e' molto semplice: ogni volta che si richiama il comando, ad ogni variable locale e' associato un nome della forma "nome variabile locale" seguito da $ e seguito da un numero identificativo cioe' **x$nnn**. Quindi x variabile globale e **x$nnn** variabile locale sono variabili distinte per *Mathematica*. Per esempio

```
Module[{t}, Print[t]]

Module[{t, s}, Print[t]; Print[s]]
```

Esempio. Costruiamo l'equazione cartesiana della retta perpendicolare a una retta data (non verticale) e passante per un punto assegnato. Il comando **Last[expr]** fornisce l'ultimo elemento di *expr*.

```
Clear["Global`*"]
rettacoefang[{x0_, y0_}, m_] :=
 y - y0 == m (x - x0)(*eq. della retta per (x0,y0) e coefficiente angolare m*)

rettanormale[retta_, {x0_, y0_}] := Module[{r, m, r2, r3},
  r = Roots[retta, y];
  r2 = Last[r];
  m = Coefficient[r2, x];
  If[m == 0, x == x0,
   r3 = rettacoefang[{x0, y0}, -1 / m]; Roots[r3, y]]
 ]
rettanormale[3 x + 5 y == 11, {1, 2}]
rettanormale[5 y + 11 == 0, {0, 0}]
```

Esempio. Costruiamo la funzione **diagonalmatrix[{lista}]** che ricopia la funzione predefinita **DiagonalMatrix**

```
? DiagonalMatrix

DiagonalMatrix[list] gives a matrix with the elements
   of list on the leading diagonal, and 0 elsewhere. More...

diagonalmatrix[x_ /; VectorQ[x]] := Module[{v, n, i},
  n = Length[x];
  v = Table[0, {n - 1}];
  Table[Insert[v, x[[i]], i], {i, 1, n}]]
```

Esempio. Definiamo la funzione **primeq[x]** che ricopia **PrimeQ[x]**

```
? PrimeQ

PrimeQ[expr] yields True if expr is
   a prime number, and yields False otherwise. More...

primeq[1] = False;
primeq[n_Integer /; n > 1] := Module[{i, done = True},
  For[i = 2, i < n, i++,
   If[IntegerQ[n / i], done = False]]; done]

Table[{primeq[i] == PrimeQ[i]}, {i, 1, 100}]
```

Esempio. Adesso scriviamo una funzione che dato un numero, scriva il numero primo successivo a questo:

```
succprime[x_ /; (NumberQ[x] && Im[x] == 0 && x > 0)] := Module[{i = 0, n},
  n = Floor[x];
  While[primeq[n + i] == False, i++]; n + i]
```

Esempio. Adesso scriviamo una funzione che ricopi **PrimePi,** funzione definita sui reali

```
? PrimePi

PrimePi[x] gives the number of primes less than or equal to x. More...

primepi[x_ /; (NumberQ[x] && Im[x] == 0 && x ≤ 1)] = 0;
primepi[x_ /; (NumberQ[x] && Im[x] == 0 && x > 1)] := Module[{i = 1, l = 0},
  While[i <= x,
   If[primeq[i], l = l + 1]; i++];
  l]
```