MAPLE – SCHEDE SINTETICHE

Introduzione

Maple è un *Computer Algebra System* sviluppato inizialmente in ambito accademico e attualmente prodotto dalla MapleSoft (`www.maplesoft.com`), che fornisce un ambiente integrato per il calcolo simbolico e numerico, il trattamento di dati strutturati, la programmazione, la visualizzazione grafica, le presentazioni multimediali interattive.

Possiede una sofisticata interfaccia amichevole con un elevato grado di interattività per la manipolazione di espressioni e grafici, e un linguaggio di alto livello basato su una sintassi relativamente semplice, anche se non sempre pulita e coerente, che consente la rappresentazione di oggetti matematici con notazioni simili a quelle standard.

Dispone di strumenti per l'esportazione/importazione di dati in parecchi formati e per l'integrazione con altri software/linguaggi, che ne consentono un utilizzo immediato a diversi livelli e in diversi ambiti (didattico, scientifico, applicativo).

In particolare, mentre ha funzioni di calcolo simbolico molto sviluppate, per il calcolo numerico, oltre alle funzioni proprie, consente di utilizzare quelle di *MatLab*, sistema prodotto dalla MathWorks (`www.mathworks.com`) specializzato negli aspetti modellistico-numerici e diffuso soprattutto in ambito ingegneristico.

L'altro *Computer Algebra System* commerciale con prestazioni analoghe è *Mathematica*, prodotto dalla Wolfram (`www.worlfram.com`). Ci sono poi diversi *Computer Algebra System* non commerciali sviluppati in ambito accademico, tra i quali: Maxima (`www.maxima.sourceforge.net`), Sage (`www.sagemath.org`), SciLab (`www.scilab.org`).

Principi generali

Ambienti di lavoro

Maple consente due diverse modalità di lavoro: *Worksheet* e *Document*. Quando si crea un nuovo documento (usando il comando `New` del menù `File`), si sceglie quale modalità usare per quel documento. In entrambi i casi vi sono due tipologie di input principali disponibili nella barra dei comandi: `Text` e `Math`.

Nella modalità *Worksheet*, `Text` serve per immettere comandi alfanumerici nel formato *Maple* originario, mentre `Math` consente l'immissione nel formato grafico *2D-Math* di simboli e formule (come per esempio frazioni, radici, esponenti e indici). Nella modalità *Document*, `Text` serve invece per inserire effettivamente del testo (non comandi o espressioni da valutare), mentre `Math` ha esattamente la stessa funzionalità detta sopra.

Indipendentemente dalla modalità scelta è comunque possibile usare il menù `Insert` per inserire: testi (`Text`), comandi/espressioni in formato *Maple* (`Maple Input`), comandi/espressioni in formato *2D-Math* (`2D Math`), disegni (`Canvas`), immagini (`Image`), grafici (`Plot`) e altro. È inoltre possibile organizzare tutto il materiale in sezioni e più livelli di sottosezioni, ciascuna collassabile al solo testo iniziale (titolo ed eventuale sommario).

I comandi/espressioni (sia nel formato *Maple* che in quello *2D-Math*) possono essere distribuiti su più righe (usando il tasto `Return` con opportuno modificatore per andare a capo), nel qual caso i ritorni a capo sono interpretati come spazi. In ciascuna riga è possibile inserire un commento preceduto dal simbolo `#`. L'esecuzione/valutazione del comando/espressione si ottiene digitando un semplice `Return`. Nel formato *Maple* ogni comando/espressione deve terminare con `;` (punto e virgola).

Per l'immissione di comandi/espressioni in formato *2D-Math* si può usare il tasto `esc` dopo i singoli comandi testuali per convertirli in grafica (per esempio `sqrt esc` per ottenere la radice quadrata), dopo di che i tasti `tab` e frecce consentono di spostarsi da un elemento all'altro. In alternativa si possono usare numerose *Palette*, visualizzabili nella barra laterale sinistra usando l'apposito comando nel menù `View`.

Calcoli algebrici e grafici

L'uso più elementare di *Maple* consiste nell'esecuzione immediata di calcoli algebrici (come la semplificazione di espressioni, la soluzione di equazioni, la determinazione di limiti, derivare e integrali).

Il modo standard di procedere è quello di immettere e far valutare l'espressione iniziale e applicare poi a questa in sequenza gli operatori necessari all'esecuzione di ogni singolo passo del calcolo da eseguire. Ciò consente di visualizzare e controllare anche tutti i risultati intermedi.

In questo caso si ottiene una sequenza alternata di input e di output: a ogni espressione in input segue la corrispondente espressione in output ottenuta dalla sua valutazione. Le espressioni in output vengono automaticamente contraddistinte da un'etichetta numerica progressiva.

In ciascun input ci si può riferire agli ultimi tre output precedenti usando **%**, **%%** e **%%%** rispettivamente per l'ultimo, il penultimo e il terzultimo output. Se si usa il formato di immissione *2D-Math*, è possibile richiamare anche gli output precedenti usando l'etichetta corrispondente, che si può inserire con il comando `Label` del menù `Insert` (o con la corripondente combinazione di tasti). Altrimenti, gli output intermedi possono essere associati a nomi simbolici usando l'operatore di assegnazione `:=` e quindi richiamati negli input successivi per nome.

Un modo alternativo di manipolare espressioni algebriche è quello di utilizzare il menù pop-up che appare cliccando su un'espressioni di output (con il tasto destro del mouse o con un opportuno tasto modificatore). Tale menù presenta diversi comandi relativi alle possibili trasformazioni applicabili all'espressione considerata. Selezionando uno di questi comandi si ottiene un ulteriore output preceduto da $\rightarrow$ o $=$ (a seconda dei casi) con sopra indicata la trasformazione eseguita.

Nelle espressioni di output *Maple* usa le seguenti convenzioni: 1) $\_C1, \_C2, \ldots$ per le costanti arbitrarie (se le costanti variano in insiemi numerici particolari, al posto della lettera C c'è il riferimento a tali insiemi, come per esempio N per i naturali, NN per gli interi non negativi, Z per gli interi); 2) un nome seguito da `~` significa che il simbolo con quel nome è soggetto ad assunzioni (per esempio è assunto intero, positivo, ecc.); 3) $\%1, \%2, \ldots$ per indicare sottoespressioni rappresentate per convenienza in modo simbolico.

Nel valutare le espressioni algebriche, incluse quelle numeriche, tutti i calcoli sono eseguiti in modo simbolico fin dove possibile e lascianti indicati altrimenti.

Per quanto riguarda la rappresentazione grafica di dati, funzioni, curve, superfici e altri oggetti geometrici, si può usare il comando generale `plot`. Altri comandi specifici sono contenuti nei Package `plots`, `plottools`, `geometry` e `geom3d`. Nel menù `Plot` e nella barra degli strumenti sono disponibili molti strumenti per modificare interattivamente le proprietà dei grafici generati.

Valutazione delle espressioni

Più in generale quanto detto sopra può essere esteso alla manipolazione di qualunque tipo di espressioni, incluse quelle che rappresentano dati strutturati (come liste, insiemi, vettori, matrici, tabelle), quelle che definiscono funzioni e procedure, e quelle che esprimono comandi.

Ogni espressione *Maple* è atomica (numero o *simbolo*) o ha la struttura `head(op1,op2,...)`, dove sia l'*intestazione* `head` che gli (eventuali) *operandi* `op1,op2,...` sono a loro volta espressioni.

Ciò vale anche per espressioni contenenti operatori "infissi", che vanno considerate con la corrispondente *notazione prefissa* (per esempio l'espressione `a + b + c` va considerata come `+(a,b,c)`). Il passaggio alla notazione infissa viene effettuato tenendo conto della priorità determinata dalle

parentesi (solo tonde) e, in assenza di parentesi, secondo un ordine standard di *priorità degli operatori*. Per esempio l'espressione `a + b * c` viene interpretata come `a + (b * c)` (in quanto `*` ha priorità maggiore di `+`) e quindi considerata come `+(a,*(b,c))`.

La definizione ricorsiva delle espressioni è alla base dell'algoritmo generale di valutazione.

La valutazione di un'espressione atomica numerica da come risultato il numero stesso, mentre la valutazione di un simbolo può dare come risultato qualunque altra espressione precedentemente assegnata ad esso come valore o, in assenza di assegnazioni, il simbolo stesso.

La valutazione di un'espressione del tipo `head(op1,op2,...)` consiste nei seguenti passi ricorsivi: 1) si valuta l'espressione `head`; 2) si valutano nell'ordine tutte le espressioni `op1,op2,...`, tenendo conto delle eventuali direttive e/o eccezioni dipendenti dal valore di `head`; 3a) se il valore di `head` è una funzione/procedura applicabile agli argomenti/parametri `op1,op2,...`, si applica (e si eseguono le eventuali azioni collaterali che essa comporta, come assegnazioni, operazioni di lettura/scrittura, visualizzazioni di grafici), ottenendo così una nuova espressione; 3b) se `head` non ha un tale valore, allora l'espressione mantiene la forma `head(op1,op2,...)`, dove `head` resta come *etichetta* che determina il *tipo* dell'espressione; 4) dopo 3a si itera il processo di valutazione a partire dall'espressione ottenuta, mentre dopo 3b la valutazione ha termine e l'espressione ottenuta è il risultato finale della valutazione.

È possibile modificare il processo standard di valutazione appena descritto usando le virgolette `''` per racchiudere una espressione di cui si voglia impedire la valutazione (per esempio per passarla non valutata come argomento/parametro di una funzione/procedura). La valutazione di un'espressione tra virgolette può essere ottenuta con la funzione speciale `eval`.

Alcune funzioni di sistema ammettono anche una *versione inerte* (che si comporta cioè come se fosse racchiusa tra virgolette `''`), denominata nello stesso modo della corrispondete funzione attiva, ma con l'iniziale maiuscola invece che minuscola. Per le funzioni di sistema elencate più avanti, queste sono le versioni inerti esistenti: `Sum`, `Product`, `Irreduc`, `Factor`, `Factors`, `Divide`, `Quo`, `Rem`, `Gcd`, `Lcm`, `Roots`, `Interp`, `Randpoly`, `Expand`, `Normal`, `Limit`, `Diff`, `Int`, `Eval`.

Programmazione

La manipolazione/valutazione di espressioni (algebriche e non) consente di eseguire un calcolo specifico e/o risolvere un problema particolare. Programmare in *Maple* significa definire nuove funzioni/procedure, che consentano invece si eseguire un'intera classe di calcoli e/o risolvere un'intera classe di problemi, al variare di certi parametri. Analogamente eseguire un programma significa applicare una tale funzione/procedura a certi valori specifici dei parametri.

In generale la distinzione tra funzione e procedura è legata al fatto che nel primo caso l'attenzione è rivolta all'espressione risultato finale della valutazione, mentre nel secondo caso è rivolta alle azioni collaterali prodotte dalla valutazione. Va detto però che sia le funzioni che le procedure possono dar luogo ad entrambi gli effetti.

Nell'ambito di *Maple* funzioni e procedure si differenziano sia nella definizione che nella valutazione.

Le *funzioni* sono definite con una sintassi simile alla notazione matematica sia nella forma implicita (per esempio `f(x) := sin(x)/x`) che in quella esplicita (per esempio `f := x -> sin(x)/x`) e ammettono anche definizioni in stile *dichiarativo* mediante l'operatore `define`. Le *procedure* adottano invece una sintassi analoga a quella dei linguaggi di programmazione di tipo *imperativo* (come il linguaggio C). Sia per le funzioni (solo nella forma esplicita) che per le procedure si può dichiarare il tipo degli argomenti/parametri. Inoltre l'operatore `define` permette di associare diverse definizioni a seconda dei tipi degli argomenti.

Per quanto riguarda la valutazione, le funzioni consentono la manipolazione simbolica, anche con argomenti simbolici per cui non è possibile la determinazione del valore, mentre ciò vale solo in parte per le procedure. D'altra parte le procedure, a differenza delle funzioni, consentono una

gestione flessibile dei parametri e l'uso dell'operatore di assegnazione (a variabili locali o globali) nei calcoli intermedi e/o per l'esecuzione di azioni collaterali alla valutazione.

La palette `Components` consente di inserire nel foglio di lavoro elementi interattivi con i quali si può realizzare facilmente una semplice interfaccia grafica per i propri programmi.

Interfacce grafiche più sofisticate in grado di funzionare anche come applicazioni indipendenti (chiamate *Maplet*) possono essere realizzate utilizzando il package `Maplets`. In questo caso però l'inserimento degli elementi interattivi e la defininizione delle loro proprietà (caratteristiche e azioni ad essi associati) deve essere fornita in modo testuale con il formato di input *Maple*.

Strumenti

Maple è dotato di un *Help* in linea molto esteso. In particolare è possibile ottenere informazioni su tutte le funzioni di sistema mediante il comando `?nomefunz`.

Il menù `Tools` contiene diversi strumenti utili: `Tasks`, `Assistants`, `Math Apps` e `Tutor`.

I `Tasks` sono template da incollare nel foglio di lavoro, per lo svolgimento di una grande varietà di calcoli. Gli `Assistants` sono invece *Maplet* che forniscono interfacce interattive per effettuare diverse attività, come l'approssimazione analitica (`Curve fitting`) e l'analisi statistica (`Data Analysis`) di dati numerici, la manipolazione di equazioni (`Equation manipulator`), la ricerca di massimi/minimi vincolati di funzioni reali (`Optimization`), la realizzazione di grafici (`Plot Builder`).

`Math Apps` e `Tutor` sono rispettivamente documenti *Maple* con componenti interattive e *Maplet* finalizzati ad illustrare concetti matematici a scopo didattico.

Infine lo stesso nemù `Tools` contiene i sottomenù `Load Package` e `Unload Package`, per caricare e scaricare estensioni del sistema *Maple* dedicate ad aspetti specifici. Quelli menzionati di seguito sono solo alcuni dei numerosi *package* disponibili.

`combinat`	funzioni combinatorie, permutazioni e combinazioni, partizioni di interi
`CurveFitting`	approssimazione analitica di dati numerici
`geometry`	costruzione e visualizzazione di oggetti geometrici nel piano euclideo
`GraphTheory`	costruzione, analisi e visualizzazione di grafi
`LinearAlgebra`	manipolazione di vettori e matrici
`ListTools`	manipolazione di liste
`Maplets`	strumenti per la creazione di *Maplet*
`Optimization`	determinazione numerica di massimi/minimi vincolati di funzioni reali
`plots`	generazione di diversi tipi di grafici
`plottools`	generazione e manipolazione di oggetti grafici
`StringTools`	manipolazione delle stringhe
`Statistics`	analisi statistica di dati
`Student`	collezione di package didattici

Oggetti

Numeri

• Ci sono diversi tipi di oggetti numerici: `integer`, `rational`, `float` e `complex`. • Il tipo `integer` rappresenta i numeri interi con un numero arbitrario di cifre. • Il tipo `rational` è codificato come frazione nella forma `±n/m`. • Il tipo `float`, per i numeri in virgola mobile, si suddivide in due sottotipi: `sfloat` (software float, con precisione arbitraria) e `hfloat` (hardware float, rappresentato in 8 byte). • I tipi `integer`, `rational` e `float` formano il tipo `numeric`. • Il tipo `complex` è rappresentato nella forma `x + yI`, dove `x` e `y` sono di tipo `numeric`.

Simboli

• Tutti gli oggetti simbolici sono inclusi nel tipo `name`, che si divide in due sottotipi: `symbol` per gli oggetti simbolici atomici e `indexed` per gli oggetti simbolici con (multi)indici. • In entrambi i casi l'oggetto è rappresentato da un nome formato da una parola contenente caratteri alfanumerici (con distinzione tra lettere maiuscole e minuscole), soggetto alle sole considizioni di iniziare con una lettera e di non contenere spazi e caratteri speciali riservati agli operatori (`+ - * / _ , ; : . ^ ' " ` % ! ? # $ & @ < > ~ \`). • Ogni oggetto di tipo `symbol` può essere usato come variabile, associandogli un valore (altrimenti il valore è lui stesso). • Analogamente a un oggetto di tipo `indexed` possono essere associati più valori, uno per ogni scelta del (multi)indice.

Strutture

• Le strutture fondamentali sono i tipi `set`, `list`, `table` e `rtable`. • Il tipo `set` è usato per rappresentare insiemi finiti. Gli elementi sono ordinati in modo standard (in base alla complessità e poi all'ordine numerico/alfabetico) e le ripetizioni sono eliminate. Ogni elemento è accessibile mediante un indice intero che ne rappresenta la posizione. • Il tipo `list` è usato per rappresentare liste di oggetti, cioè insiemi finiti ordinati, con possibili ripetizioni. Le liste possono avere una struttura gerarchica ad albero, nel senso che gli elementi di una lista possono a loro volta essere liste. Ogni elemento è accessibile mediante un (multi)indice intero che ne rappresenta la posizione. • Il tipo `table` è usato per rappresentate tabelle. È analogo al tipo `list`, con la differenza che i (multi)indici non sono necessariamente interi, ma possono essere oggetti arbitrari. • Il tipo `rtable` rappresenta in modo efficiente tabelle "rettangolari" con (multi)indici interi. • Le strutture `Vector`, `Matrix` e `Array`, che rappresentano vettori, matrici e array multidimensionali, sono implementate come sottotipi di `rtable`. • Le analoghe strutture `vector`, `matrix` e `array` sono invece implementate come `table`, ma restano solo per compatibilità con le versioni precedenti.

Espressioni

Costanti numeriche

`I`	i	unità immaginaria
`Pi`	π	pi greco
`exp(1)`	e	numero di Nepero
`infinity`	∞	infinito positivo

Operatori aritmetici

`a + b + ...`	$a+b+\ldots$	addizione
`a * b * ...` `a b ...`	$a\,b\,\ldots$	moltiplicazione
`a - b`	$a-b$	sottrazione
`a / b`	a/b	divisione
`a ** b` `a ^ b`	a^b	potenza
`a mod b`	$a \bmod b$	modulo
`n!`	$n!$	fattoriale

Relazioni aritmetiche

`a = b`	$a=b$	uguaglianza
`a >= b`	$a\geq b$	maggiore o uguale
`a <= b`	$a\leq b$	minore o uguale

`a > b`	$a > b$	maggiore
`a < b`	$a < b$	minore
`a <> b`	$a \neq b$	disuguaglianza

Costanti logiche

`true`	T	vero
`false`	F	falso

Operatori logici

`not p`	$\neg p$	negazione
`p and q and ...`	$p \wedge q \wedge \ldots$	congiunzione
`p or q or ...`	$p \vee q \vee \ldots$	disgiunzione
`p xor q xor ...`	$p \veebar q \veebar \ldots$	disgiunzione esclusiva
`p implies q`	$p \Rightarrow q$	implicazione

Operatori insiemistici

`A union B`	$A \cup B$	unione
`A intersect B`	$A \cap B$	intersezione
`A minus B`	$A - B$	differenza

Relazioni insiemistiche

`a in A`	$a \in A$	appartenenza
`A subset B`	$A \subset B$	inclusione

Operatori funzionali

`x -> fx`		funzione che associa a x l'espressione $f(x)$
`g @ f`	$g \circ f$	composizione
`f @@ n`	$f^n = f \circ f \circ \ldots$	iterazione n-esima
`f', f'', ...`	$f', f'', \ldots$	funzioni derivate

Operatori vettoriali

`V1 + V2`	$V_1 + V_2$	somma di vettori
`a * V` `a V`	$a\,V$	prodotto di uno scalare per un vettore
`V1 . V2` `V1 V2`	$V_1 \cdot V_2$	prodotto scalare tra vettori
`M . V` `M V`	$M\,V$	prodotto tra una matrice e un vettore (colonna)
`M1 + M2`	$M_1 + M_2$	somma di matrici
`a * M` `a M`	$a\,M$	prodotto di uno scalare per una matrice
`M1 . M2` `M1 M2`	$M_1\,M_2$	prodotto righe per colonne tra matrici
`M ^ n`	M^n	matrice potenza
`M^(-1)`	M^{-1}	matrice inversa
`M^+`	M^+	matrice trasposta
`⟨A | B⟩`	$(A\|B)$	concatenazione orizzontale di vettori e matrici
`⟨A , B⟩`	$\left(\frac{A}{B}\right)$	concatenazione verticale di vettori e matrici

Operatore sequenza

`$ n..m`	$m, \ldots, n$	sequenza numerica
`a $ n`	$a, \ldots, a$	sequenza costituita da n volte l'espressione a
`ai $ i = m..n`	$a_m, \ldots, a_n$	sequenza di espressioni per i da m a n

Operatore assegnazione

`name := expr`	assegnazione a un simbolo
`name[i,j,...] := expr`	assegnazione a un elemento di una struttura
`f(x,y,...) := expr`	definizione implicita di una funzione
`f := (x,y,...) -> expr`	definizione esplicita di una funzione

Unità di misura

`⟦m⟧`	metro
`⟦s⟧`	secondo
`⟦kg⟧`	kilogrammo
`⟦K⟧`	kelvin
`⟦A⟧`	ampere

Parentesi

`( )`	per gli argomenti di funzioni o procedure per indicare la precedenza nelle espressioni
`[ ]`	per rappresentare le liste e le tabelle per gli indici degli elementi di insiemi, liste, tabelle, vettori e matrici
`{ }`	per rappresentare gli insiemi
`⟨ ⟩`	per rappresentare i vettori e le matrici

Punteggiatura

`,`	tra gli elementi di insiemi (dentro parentesi graffe) tra gli elementi di liste e tabelle (dentro parentesi quadre) tra gli elementi di vettori colonna e di righe di matrici (dentro parentesi angolari) tra gli indici di insiemi liste, tabelle, vettori e matrici (dentro parentesi quadre) tra gli argomenti di funzioni (dentro parentesi tonde) tra gli elementi di sequenze (senza parentesi)
`;`	per separare le righe di una matrice (dentro parentesi angolari) per terminare e concatenare diverse espressioni ...
`:`	... sopprimendo l'output di quelle precedute dai due punti
`::`	per specificare il tipo di un simbolo
`.`	per separare la parte decimale di un numero (senza spazi) per indicare il prodotto tra array (per esempio prodotto scalare e "righe per colonne")
`..`	per indicare un intervallo di valori

Virgolette

`f'`	per indicare la funzione derivata
`'expr'`	per ritardare la valutazione dei nomi di un passo
`` `name` ``	per simboli il cui nome contenga spazi o caratteri speciali
`"string"`	per rappresentare le stringhe

Spaziatura

• Spazi consecutivi valgono come uno solo. • Uno spazio tra due espressioni vale come moltiplicazione, eccetto il caso in cui la seconda sia un numero. • Gli spazi prima e dopo gli operatori, le parentesi e la punteggiatura sono trascurati, tranne che prima e dopo il punto (nessuno spazio per il punto decimale, altrimenti vale come operatore dot) e prima delle parentesi tonde aperte

(`f(x,y...)` significa applicare la funzione f a $x, y, \ldots$, mentre `f (x,y...)` vale come prodotto $f \cdot (x, y, \ldots)$). • Un'espressione può essere scritta su più righe (i ritorni a capo valgono come spazi).

Funzioni di sistema

Numeri interi

`isprime(n)`		n è un numero primo?
`ifactor(n)`		fattorizzazione del numero intero n
`ifactors(n)`		lista dei fattori primi di n
`isqrfree(n)`		n non ha fattori primi multipli?
`ithprime(i)`		i-esimo numero primo
`iquo(a,b)`		quoziente della divisione tra interi $a : b$
`irem(a,b)`		resto della divisione tra interi $a : b$ (con segno)
`igcd(a,b,...)`	$(a, b, \ldots)$	massimo comune divisore
`ilcm(a,b,...)`	$[a, b, \ldots]$	minimo comune multiplo
`igcdex(a,b,x,y)`		assegna a x e y valori interi t.c. $ax + by = (a, b)$
`isqrt(n)`		radice quadrata intera (approssimata)
`issqr(n)`		n è un quadrato perfetto?
`n mod m` `mod(n,m)`		} n modulo m (a valori in $0, \ldots, m-1$)
`chrem([n1,...,nk],[m1,...,mk])`		n t.c. $n = n_i \bmod m_i$ (teorema cinese del resto)

Numeri razionali

`Fraction(n,m)`	frazione n/m
`numer(q)`	numeratore del numero razionale q
`denom(q)`	denominatore del numero razionale q

Numeri "reali"

`xen` `xEn`	} numero "reale" $x \times 10^n$
`Float(x,n)` `SFloat(x,n)`	} numero sfloat $x \times 10^n$
`HFloat(x,n)`	numero hfloat $x \times 10^n$
`HFloat(x,n,2)`	numero hfloat $x \times 2^n$
`round(x)`	arrotondamento intero di x
`trunc(x)`	parte intera di x
`frac(x)`	parte frazionaria di x
`frem(x,y)`	resto della divisione $x : y$ con quoziente intero
`fnormal(expr)`	normalizza i numeri float nell'espressione *expr*
`fnormal(expr,n)`	... valutandoli con n cifre decimali

Numeri complessi

`Complex(x,y)`	$x + y\,i$	numero complesso
`Re(z)`	$\mathrm{Re}\, z$	parte reale del numero complesso z
`Im(z)`	$\mathrm{Im}\, z$	parte immaginaria del numero complesso z
`abs(z)`	$\lvert z \rvert$	modulo del numero complesso z
`argument(z)`	$\arg z$	argomento del numero complesso z
`surd(x,n)`	$\sqrt[n]{z}$	radice complessa di z

Conversioni numeriche

`convert(x,rational)`		approssimazione razionale di x
`convert(x,rational,n)`		approssimazione razionale di x con n cifre
`convert(x,rational,exact)`		approssimazione razionale di x con tutte le cifre
`identify(x)`		approssimazione di x in forma chiusa
`convert(x,float)`		conversione di x in float
`convert(x,float,n)`		conversione di x in float con n cifre decimali
`Float(x)`	}	conversione di x in sfloat
`SFloat(x)`	}	
`HFloat(x)`		conversione di x in hfloat

Funzioni aritmetiche

`add(ai,i = m..n)`	$\sum_{i=m}^{n} a_i$	sommatoria per i da m a n (con m e n numeri)
`add(ai,i = S)`	} $\sum_{i\in S} a_i$	sommatoria al variare di i in S (insieme o lista finiti)
`add(ai,i in S)`	}	
`sum(ai,i = m..n)`		sommatoria anche con limiti simbolici o infiniti
`mul(ai,i = m..n)`	$\prod_{i=m}^{n} a_i$	produttoria per i da m a n
`mul(ai,i = S)`	} $\prod_{i\in S} a_i$	produttoria al variare di i in S (insieme o lista)
`mul(ai,i in S)`	}	
`product(ai,i = m..n)`		produttoria anche con limiti simbolici o infiniti
`factorial(n)`	$n!$	fattoriale
`max(a,b,...)`	$\max\{a, b, \ldots\}$	massimo
`min(a,b,...)`	$\min\{a, b, \ldots\}$	minimo

Polinomi

`ispoly(p,n,x)`	p è un polinomio di grado n in x?
`ispoly(p,n,x,a0,a1,...,an)`	... se sì, assegna i coefficienti alle variabili $a_0, \ldots, a_n$
`degree(p,x)`	grado del polinomio p rispetto ad x
`ldegree(p,x)`	minimo esponente di x del polinomio p
`coeff(p,x,n)`	coefficiente di x^n nel polinomio p
`sort(p,opts)`	ordina i termini del polinomio p
`irreduc(p)`	il polinomio p è irriducibile?
`irreduc(p,K)`	p è irriducibile sul campo K?
`factor(p)`	fattorizzazione del polinomio p
`factor(p,K)`	fattorizzazione di p sul campo K
`factors(p)`	lista dei fattori irriducibili del polinomio p
`factors(p,K)`	lista dei fattori irriducibili di p sul campo K
`quo(p,q,x)`	quoziente della divisione $p(x) : q(x)$
`rem(p,q,x)`	resto della divisione $p(x) : q(x)$
`divide(p,q)`	il polinomio p è divisibile per il polinomio q?
`gcd(p,q,...)`	massimo comune divisore di polinomi
`lcm(p,q,...)`	minimo comune multiplo di polinomi
`discrim(p,x)`	discriminante del polinomio $p(x)$
`compoly(p,x)`	cerca q e r polinomi t.c. $p(x) = q(r(x))$
`roots(p,x)`	lista delle radici del polinomio $p(x)$
`roots(p,x,K)`	lista delle radici di $p(x)$ sul campo K
`RootOf(p,x)`	radici formali del polinomio $p(x)$
`interp([x1,...,xn],[y1,...yn],x)`	polinomio interpolante $p(x)$ t.c. $p(x_i) = y_i$
`randpoly(x,y,...,opts)`	polinomio random nelle indeterminate $x, y, \ldots$

Espressioni algebriche

`indets(expr)`	insieme delle indeterminate in *expr*
`indets(expr,type)`	insieme delle indeterminate in *expr* di tipo *type*
`depends(expr,x)`	*expr* dipende dall'indeterminata x?
`numer(expr)`	numeratore dell'espressione *expr*
`denom(expr)`	denominatore dell'espressione *expr*
`normal(expr)`	forma normale di *expr* come espressione razionale
`normal(expr,expanded)`	... con numeratore e denominatore espansi
`radnormal(expr,opts)`	forma normale di *expr* espressione con radicali
`rationalize(expr)`	razionalizza il denominatore di *expr*
`expand(expr)`	espande l'espressione *expr* (in particolare i prodotti)
`expand(expr,expr1,...,exprn)`	espande *expr*, ma non le sottoespressioni $expr_i$
`collect(expr,x)`	raccoglie le potenze di x nell'espressione *expr*
`collect(expr,{x1,...,xn})`	raccoglie le potenze di $x_1, \ldots, x_n$ in *expr*
`combine(expr)`	combina termini nell'espressione *expr*
`combine(expr,type)`	combina termini di tipo *type* in *expr*
`combine(expr,type,symbolic)`	... consente combinazioni simboliche
`algsubs(a = b,expr,opts)`	sostituzione algebrica di a con b in *expr*
`subsindets(expr,type,f)`	applica f alle occorrenze di tipo *type* in *expr*
`simplify(expr)`	semplifica *expr* usando regole di sistema
`simplify(expr,r1,r2,...)`	... usando solo le regole di sistema $r_1, r_2, \ldots$
`simplify(expr,a1 = b1,a2 = b2,...)`	... usando le equazioni $a_i = b_i$
`simplify(expr,assume = prop)`	... assumendo la proprietà *prop* per le indeterminate
`simplify(expr,symbolic)`	... consentendo semplificazioni simboliche
`match(expr = pat,{v1,v2,...},opts)`	tenta di identificare l'espressione *expr* con il pattern *pat* assumendo $v_1, v_2, \ldots$ come variabili libere
`expr assuming p1,p2,...`	valuta l'espressione *expr* assumendo le proprietà p_i

Funzioni elementari

`abs(x)`	$\lvert x \rvert$	funzione valore assoluto
`signum(x)`	$\operatorname{sgn}(x)$	funzione segno
`floor(x)`	$\lfloor x \rfloor$	massimo intero $\leq x$
`ceil(x)`	$\lceil x \rceil$	minimo intero $\geq x$
`sqrt(x)`	$\sqrt{x}$	radice quadrata
`root(x,n)` `root[n](x)`	$\sqrt[n]{x}$	radice n-esima
`exp(x)`	e^x	funzione esponenziale naturale
`a^x`	a^x	funzione esponenziale con base a
`ln(x)` `log(x)`	$\log x$	logaritmo naturale
`log[a](x)`	$\log_a x$	logaritmo in base a
`sin(x) cos(x) tan(x)` `sec(x) csc(x) cot(x)`		funzioni goniometriche
`arcsin(x) arccos(x) arctan(x)` `arcsec(x) arccsc(x) arccot(x)`		funzioni goniometriche inverse
`arctan(y,x)`		arcotangente di y/x a valori in $(-\pi, +\pi]$
`sinh(x) cosh(x) tanh(x)` `sech(x) csch(x) coth(x)`		funzioni goniometriche iperboliche
`arcsinh(x) arccosh(x) arctanh(x)` `arcsech(x) arccsch(x) arccoth(x)`		funzioni goniometriche iperboliche inverse

Operatori funzionali

`vars -> expr` `unapply(expr,vars,opts)`		} funzione data dall'espressione *expr* con variabili *vars*
`apply(f,vars)`		applica la funzione f alle variabili *vars*
`define(f,r1,r2,...)`		definisce la funzione f mediante le regole $r_1, r_2, \ldots$
`definemore(f,r1,r2,...)`		... aggiunge alla definizione le ulteriori regole $r_1, r_2, \ldots$
`undefine(f)`		elimina tutte le regole associate a f

Calcolo

`limit(expr,x = a)`	$\lim_{x\to a} expr$	limite per $x \to a$ dell'espressione *expr*
`limit(expr,x = a,dir)`	$\lim_{x\to a^{\pm}} expr$	limiti destro e sinistro di *expr*
`iscont(expr,x = a..b)`		l'espressione *expr* è continua rispetto a x in (a, b)?
`iscont(expr,x = a..b,closed)`		l'espressione *expr* è continua rispetto a x in $[a, b]$?
`discont(expr,x)`		insieme delle discontinuità di *expr* rispetto a x
`isdifferentiable(expr,x,k)`		l'espressione *expr* è di classe C^k rispetto a x?
`isdifferentiable(expr,x,k,s)`		... se no, assegna a s la classe $h < k$ e le singolarità
`diff(expr,x1,...,xk)`	$\partial^k expr/\partial x_k \ldots \partial x_1$	derivata parziale di *expr* rispetto a $x_1, \ldots, x_k$
`abs(k,x)`	$d^k \lvert x \rvert / dx^k$	derivata k-esima della funzione valore assoluto
`signum(k,x)`	$d^k \mathrm{sgn}(x)/dx^k$	derivata k-esima della funzione segno
`floor(k,x)`	$d^k \lfloor x \rfloor / dx^k$	derivata k-esima della funzione $\lfloor\ \rfloor$
`ceil(k,x)`	$d^k \lceil x \rceil / dx^k$	derivata k-esima della funzione $\lceil\ \rceil$
`D[i1,...,ik](f)`	$\partial^k f/\partial x_{i_k} \ldots \partial x_{i_1}$	funzione derivata di f rispetto a $x_{i_1}, \ldots, x_{i_k}$
`D[i1,...,ik](f)(x1,...,xn)`		... applicata al punto $(x_1, \ldots, x_n)$
`minimize(expr,vars,opts)`		minimizza *expr* rispetto alle variabili *vars*
`maximize(expr,vars,opts)`		massimizza *expr* rispetto alle variabili *vars*
`extrema(expr,{eq1,...,eqm},vars)`		estremi vincolati di *expr* rispetto alle variabili *vars*
`extrema(expr,{eq1,...,eqm},vars,p)`		... assegna i corrispondenti punti del dominio a p
`taylor(expr,x = a,n)`		serie di Taylor di *expr* in $x = a$ fino all'ordine n
`coeftayl(expr,x = a,k)`		k-esimo coefficiente della serie di Taylor
`order(s)`		ordine di troncamento della serie s
`int(expr,x,opts)`	$\int expr\, dx$	integrale indefinito di *expr* rispetto a x
`int(expr,x = a..b,opts)`	$\int_a^b expr\, dx$	integrale definito di *expr* rispetto a x in $[a, b]$
`int(expr,[x,y,...],opts)`		integrale indefinito multiplo di *expr*
`int(expr,[x = a..b,y = c..d,...],opts)`		integrale definito multiplo di *expr*

Vettori e matrici

`Vector([x1,...,xn])`	vettore $(x_1, \ldots, x_n)$
`Vector(n,init,opts)`	vettore di dimensione n inizializzato con *init*
`Matrix([[x11,...],...,[...,xnm]])`	matrice $(x_{i,j})_{i=1,\ldots,n,j=1,\ldots,m}$
`Matrix(n,m,init,opts)`	matrice $n \times m$ inizializzata con *init*
`v[i]`	i-esima componente del vettore v
`v[i..j]`	componenti del vettore v dall'i-esima alla j-esima
`m[i,j]`	elemento di indice (i, j) nella matrice m
`m[i1..i2,j1..j2]`	sottomatrice di m con indici da (i_1, j_1) a (i_2, j_2)

Package LinearAlgebra

`Dimension(V)`		dimensione del vettore V
`DotProduct(V1,V2)`	$V_1 \cdot V_2$	prodotto scalare
`CrossProduct(V1,V2)`	$V_1 \times V_2$	prodotto vettoriale

`Dimension(M)`		dimensioni della matrice M
`Determinant(M)`	$\det M$	determinante
`DotProduct(M1,M2)`	$M_1\, M_2$	prodotto righe per colonne
`DotProduct(M,V)`	$M\, V$	prodotto matrice vettore

Equazioni

`lhs(eqn)`	primo membro dell'equazione *eqn*
`rhs(eqn)`	secondo membro dell'equazione *eqn*
`isolate(eqn,expr,opts)`	isola *expr* al primo membro dell'equazione *eqn*
`eliminate(eqns,vars)`	elimina le variabili *vars* dalle equazioni *eqns*
`solve(eqns,vars,opts)`	soluzioni delle equazioni *eqns* nelle incognite *vars*
`fsolve(eqns,vars,opts)`	ricerca le soluzioni mediante metodi numerici
`isolve(eqns)`	soluzioni intere delle equazioni *eqns*
`msolve(eqns,m)`	soluzioni intere modulo m delle equazioni *eqns*
`rsolve(eqns,vars,opts)`	soluzione delle equazioni *eqns* per ricorrenza
`dsolve(eqns,y(x),opts)`	soluzione delle equazioni differenziali *eqns*
`odetest(sol,eqns,y(x),opts)`	... verifica della soluzione *sol* per *eqns*
`intsolve(eqns,y(x),opts)`	soluzione delle equazioni integrali *eqns*
`Equate(a,b)`	uguaglia le componenti di liste/vettori/matrici/array

Grafica

`plot(fx,x = a..b,opts)`	grafico di $f(x)$ al variare di x in $[a,b]$
`plot(⟨x1,...,xn⟩,⟨y1,...,yn⟩,opts)`	grafico per punti $(x_1,y_1),\ldots,(x_n,y_n)$
`plot([x,y,t = a..b],opts)`	curva piana $(x(t),y(t))$ al variare di t in $[a,b]$
`plot([[x1,y1],...,[xn,yn]],opts)`	curva piana per punti $(x_1,y_1),\ldots,(x_n,y_n)$
`plot3d(fxy,x = a..b,y = c..d,opts)`	grafico di $f(x,y)$ al variare di (x,y) in $[a,b]\times[c,d]$
`plot3d([x,y,z],t = a..b,s = c..d,opts)`	superficie nello spazio $(x(t,s),y(t,s),z(t,s))$ al variare di (t,s) in $[a,b]\times[c,d]$

Package plots

`display(gr1,gr2,...,opts)`	visualizza i plot o oggetti grafici $gr_1, gr_2, \ldots$
`animate(p,args,t = a..b,opts)`	animazione del plot $p(args)$ al variare di $t \in [a,b]$
`contourplot(expr,x = a..b,y = c..d,opts)`	curve di livello di *expr* con $(x,y)\in[a,b]\times[c,d]$
`contourplot3d(expr,...,opts)`	... versione 3D (con valori nella terza dimensione)
`fieldplot([vx,vy],vars,opts)`	campo di vettori (vx,vy) nel piano
`fieldplot3d([vx,vy,vz],vars,opts)`	campo di vettori (vx,vy,vz) nello spazio
`implicitplot(eq,vars,opts)`	curva di equazione *eq* nel piano
`implicitplot3d(eq,vars,opts)`	superficie di equazione *eq* nello spazio
`spacecurve([x,y,z],t = a..b,opts)`	curva $(x(t),y(t),z(t))$ al variare di t in $[a,b]$
`tubeplot(c,opts)`	superficie tubolare della curva c nello spazio

Package plottools

`point([x,y],opts)`	punto nel piano di coordinate (x,y)
`point([x,y,z],opts)`	punto nello spazio di coordinate (x,y,z)
`curve([[x1,y1],...,[xn,yn]],opts)`	poligonale nel piano di vertici (x_i,y_i)
`curve([[x1,y1,z1],...,[xn,yn,zn]],opts`	poligonale nello spazio di vertici (x_i,y_i,z_i)
`polygon([[x1,y1],...,[xn,yn]],opts)`	poligono nel piano di vertici (x_i,y_i)
`polygon([[x1,y1,z1],...,[xn,yn,zn]],opts)`	poligono nello spazio di vertici (x_i,y_i,z_i)
`line([x1,y1],[x2,y2],opts)`	segmento di estremi (x_1,y_1) e (x_2,y_2)

`rectangle([x1,y1],[x2,y2],opts)`	rettangolo di vertici (x_1, y_1) e (x_2, y_2)
`circle([x0,y0],r,opts)`	circonferenza di centro (x_0, y_0) e raggio r
`disk([x0,y0],r,opts)`	cerchio di centro (x_0, y_0) e raggio r
`arc([x0,y0],r,a..b,opts)`	arco circolare di centro (x_0, y_0) e raggio r
`sector([x0,y0],r1..r2,a..b,opts)`	settore circolare di centro (x_0, y_0) tra i raggi $r1$ e $r2$
`transform(f)(gr)`	applica la trasformazione f all'oggetto grafico gr

Simboli

`cat(a,b,...)`	simbolo $ab\ldots$ ottenuto per concatenazione
`assign(s1 = expr1,s2 = expr2,...)`	effettua le assegnazioni $s_i := expr_i$
`unassign('s1','s2',...)`	rimuove le assegnazioni ai simboli $s_1, s_2, \ldots$
`assigned(s)`	il simbolo s ha assegnato un valore?
`anames();`	sequenza dei simboli ai quali è associato un valore
`anames(user);`	sequenza dei simboli definiti dall'utente
`anames(type);`	sequenza dei simboli definiti di tipo *type*
`protect(s1,s2,...)`	protegge i simboli $s_1, s_2, \ldots$ da modifiche
`unprotect(s1,s2,...)`	elimina la protezione dei simboli $s_1, s_2, \ldots$
`setattribute(s,a1,a2,...)`	associa al simbolo s gli attributi $a_1, a_2, \ldots$
`attributes(s)`	sequenza degli attributi del simbolo s
`assume(x1,p1,x2,p2,...)`	assume che il simbolo x_i soddisfi la proprietà p_i
`assume(x1::t1,x2::t2,...)`	assume che il simbolo x_i sia del tipo t_i
`assume(rel1,rel2,...)`	assume le relazioni rel_i valide per i simboli presenti

`additionally(x1,p1,x2,p2,...)`
`additionally(x1::t1,x2::t2,...)`
`additionally(rel1,rel2,...)`
	ulteriori assunzioni (come sopra)

`about(s)`	informazioni sulle assunzioni associate al simbolo s
`hasassumptions(s)`	il simbolo s ha assunzioni associate?
`getassumptions(s)`	insieme delle assunzioni associate al simbolo s
`AndProp(p1,p2,...)`	proprietà data dalla congiunzione di $p_1, p_2, \ldots$
`OrProp(p1,p2,...)`	proprietà data dalla come disgiunzione di $p_1, p_2, \ldots$
`Non(p)`	proprietà data dalla negazione di p

Tabelle e array

`table([v1,v2,...])`	*table* con valore iniziale $v_1, v_2, \ldots$
`table([i1 = v1,i2 = v2,...])`	*table* con valore iniziale $v_{i_1}, v_{i_2}, \ldots$
`rtable(dims,init,opts)`	*rtable* di dimensioni *dims* con valori iniziali *inits*
`Array(dims,init,opts)`	*array* di dimensioni *dims* con valori iniziali *inits*
`a[i]`	valore di indice i nell'array/rtable/table a
`a[i..j]`	valori di indici i a j nell'array/rtable a
`a[i,j,...]`	valore di indice $(i, j, \ldots)$ nell'array/rtable a
`copy(t)`	genera una copia della table/rtable/array t
`numelems(t)`	numero degli elementi della table/rtable/array t
`indices(t)`	sequenza degli indici della table/rtable/array t
`entries(t,opts)`	sequenza dei valori della table/rtable/array t
`lowerbound(a,n)`	indice minimo nella n-esima dimensione dell'array a
`upperbound(a,n)`	indice massimo nella n-esima dimensione dell'array a
`ArrayNumDims(a)`	numero delle dimensioni dell'array a
`ArrayDims(a)`	range di variabilità degli indici i dell'array a
`ArrayNumElems(a,opts)`	numero degli valori v_i dell'array a

`ArrayElems(a)`		insieme degli elementi di a nella forma $\{i = v_i\}_i$
`comparray(a,b,opts)`		confronta i valori degli array/rtable/table a e b

Sequenze, liste e insiemi

`x1,x2,...`	$x_1, x_2, \ldots$	sequenza ordinata
`seq(m..n)`	$m, \ldots, n$	sequenza di interi da m a n
`seq(f,i = m..n)`	$f(m), \ldots, f(n)$	sequenza di espressioni $f(i)$ al variare di i da m a n
`seq(f,i = S)` `seq(f,i in S)`	$f(i) \mid i \in S$	sequenza di espressioni $f(i)$ al variare di i in S
`[x1,x2,...]`	$(x_1, x_2, \ldots)$	lista (= insieme ordinato) degli elementi $x_1, x_2, \ldots$
`{x1,x2,...}`	$\{x_1, x_2, \ldots\}$	insieme degli elementi $x_1, x_2, \ldots$
`l[i]`		i-esimo elemento della lista/insieme/sequenza l
`l[i..j]`		elementi di l dall'i-esimo al j-esimo
`l[i,j,...]`		elemento di posizione $(i, j, \ldots)$ nella lista l
`sort(l)`		ordina la lista l rispetto all'ordine standard
`sort(l,ord)`		ordina la lista l rispetto all'ordine *ord*
`zip(f,l1,l2)`		lista $[f(x_i, y_i)]$ con x_i nella lista l_1 e y_i nella lista l_2
`union(S1,S2,...)`	$S_1 \cup S_2 \cup \ldots$	unione degli insiemi $S_1, S_2, \ldots$
`intersect(S1,S2,...)`	$S_1 \cap S_2 \cap \ldots$	intersezione degli insiemi $S_1, S_2, \ldots$
`symmdiff(S1,S2,...)`		differenza simmetrica degli insiemi $S_1, S_2, \ldots$
`minus(S1,S2)`	$S_1 - S_2$	differenza tra gli insiemi S_1 e S_2
`subset(S1,S2)`	$S_1 \subset S_2$	S_1 è sottoinsieme di S_2?
`x in S`	$x \in S$	x appartiene all'insieme/lista S?
`x in SetOf(t)`		x appartiene all'insieme degli oggetti di tipo t?

Espressioni

`NULL`	espressione nulla
`length(expr)`	lunghezza (= complessità) dell'espressione *expr*
`op(expr)`	sequenza degli operando dell'espressione *expr*
`nops(expr)`	numero degli operandi dell'espressione *expr*
`op(i,expr)`	i-esimo operando dell'espressione *expr*
`op(i..j,expr)`	operandi di *expr* dall'i-esimo allo j-esimo
`op([i,j,...],expr)`	operando di *expr* nella posizione $(i, j, \ldots)$
`member(x,expr)`	*expr* ha x tra i suoi operandi?
`membertype(t,expr)`	*expr* ha un operando di tipo t?
`has(expr,x)`	*expr* contiene x?
`hasfun(expr,f)`	*expr* contiene f come funzione?
`hasfun(expr,f,x)`	*expr* contiene f funzione con x nell'argomento?
`numboccur(expr,x)`	numero di occorrenze di x nell'espressione *expr*
`select(p,expr,t1,t2,...)`	elimina dall'espressione *expr* gli operandi x che non soddisfano la proprietà $p(x, t_1, t_2, \ldots)$
`remove(p,expr,t1,t2,...)`	elimina dall'espressione *expr* gli operandi x che soddisfano la proprietà $p(x, t_1, t_2, \ldots)$
`subsop(i1 = x1,i2 = x2,...,expr)`	sostituisce l'i_k-esimo operando di *expr* con x_k
`applyop(f,i,expr,opts)`	applica la funzione f all'i-esimo operando di *expr*
`map(f,expr)`	applica la funzione f agli operandi di *expr*
`map(f,expr,x1,x2,...)`	applica $x \to f(x, x_1, x_2, \ldots)$ agli operandi di *expr*
`map[i](f,x1,...,expr,xi,...)`	applica $x \to f(x_1, \ldots, x, x_i, \ldots)$ agli operandi di *expr*

`subs(a1 = b1,a2 = b2,...,expr)`	sostituisce ogni occorrenza di a_i in *expr* con b_i
`applyrule(a1 = b1,a2 = b2,...,expr)`	applica iterativamente le regole $a_i \to b_i$ in *expr*
`if(cond,expr1,expr2)`	$expr_1$ o $expr_2$ a seconda che *cond* sia vera o falsa
`piecewise(c1,expr1,c2,expr2,...,exprn)`	espressione con valore $expr_i$ se vale la condizione c_i
`andmap(p,expr,x1,x2,...)`	$p(x, x_1, x_2, \dots)$ vale per ogni operando x di *expr*?
`ormap(p,expr,x1,x2,...)`	$p(x, x_1, x_2, \dots)$ vale per qualche operando x di *expr*?
`convert(expr,form,opts)`	converte l'espressione *expr* nella forma *form*

Tipi e pattern

`x::t`	simbolo x di tipo t
`subtype(s,t)`	s è un sottotipo del tipo t?
`whattype(expr)`	tipo principale dell'espressione *expr*
`type(expr,t)`	l'espressione *expr* è di tipo t?
`hastype(expr,t,opts)`	*expr* contiene una sottoespressione di tipo t?
`patmatch(expr,pat,opts)`	l'espressione *expr* corrisponde al pattern *pat*?
`typematch(expr,pat,opts)`	*expr* corrisponde al pattern *pat* in base ai tipi?
	In base alle assunzioni sulle indeterminate ...
`is(expr,p)`	... l'espressione *expr* soddisfa la proprietà p?
`is(expr::t)`	... l'espressione *expr* è di tipo t?
`is(rel)`	... la relazione *rel* è soddisfatta?
	Per qualche valore delle indeterminate ...
`coulditbe(expr,p)`	... l'espressione *expr* può soddisfare la proprietà p?
`coulditbe(expr::t)`	... l'espressione *expr* può essere di tipo t?
`coulditbe(rel)`	... la relazione *rel* può essere soddisfatta?

Valutazione

`expr assuming p1,p2,...`	*expr* valutata assumendo le proprietà p_i
`eval(expr)`	valutazione completa dell'espressione *expr*
`eval(expr,{a1 = b1,a2 = b2,...})`	valutazione dell'espressione *expr* ponendo $a_i = b_i$
`evalb(expr)`	valutazione di *expr* come espressione booleana
`evalf(expression)`	valutazione di *expr* usando l'aritmetica *float*
`evalf[n](expression)`	... con n cifre decimali
`evalhf(expr)`	valutazione di *expr* usando l'aritmetica *hfloat*
`evalc(expr)`	valutazione simbolica di *expr* sui complessi
`evalindets(expr,t,f,x1,x2,...)`	valuta l'espressione *expr* sostituendo le sottoespressioni x di tipo t con $f(x, x_1, x_2, \dots)$
`value(expr)`	valuta *expr* rendendo attive le funzioni inerti
`freeze(expr)`	versione congelata dell'espressione *expr*
`thaw(expr)`	valuta *expr* scongelando le parti congelate

Stringhe

`length(s)`	lunghezza della stringa s
`cat(s1,s2,...)`	concatenazione delle stringhe $s_1, s_2, \dots$
`s[m..n]` `substring(s,m..n)`	} sottostringa di s dalla posizione m alla posizione n
`searchtext(t,s)`	cerca il testo t nella stringa s (maiuscole = minuscole)
`SearchText(t,s)`	cerca il testo t nella stringa s (maiuscole $\neq$ minuscole)
`parse(s,opts)`	converte la stringa s in espressione

Package StringTools

`Char(n)`	carattere con codice ASCII n
`Ord(c)`	codice ASCII del carattere c
`Search(t,s)`	prima posizione della stringa t come sottostringa di s
`SearchAll(t,s)`	posizioni della stringa t come sottostringa di s
`Substitute(s,t1,t2)`	sostituisce la prima sottostringa t_1 con t_2 in s
`SubstituteAll(s,t1,t2)`	sostituisce tutte le sottostringhe t_1 con t_2 in s
`Select(p,s)`	sottostringa dei caratteri di s con la proprietà p
`Remove(p,s)`	... sottostringa degli altri caratteri di s
`Split(s,opts)`	lista delle sottostringhe parole di s
`LengthSplit(s,n,opts)`	spezza s in sottostringhe di lunghezza n

Strutture dati

`stack[new](x1,x2,...)`	genera un nuovo stack inzializzato con $x_1, x_2, \ldots$
`stack[push](x,s)`	aggiunge l'elemento x allo stack s
`stack[pop](s)`	estrae l'elemento in cima allo stack s
`stack[top](s)`	elemento in cima allo stack s
`stack[depth](s)`	altezza dello stack s
`stack[empty](s)`	lo stack s è vuoto?
`heap[new](ord,x1,x2,...)`	genera un nuovo heap inzializzato con $x_1, x_2, \ldots$ che utilizza la relazione d'ordine totale *ord*
`heap[insert](x,h)`	inserisce l'elemento x all'heap h
`heap[extract](h)`	estrae l'ultimo elemento dall'heap h
`heap[max](h)`	ultimo elemento dell'heap h
`heap[size](h)`	numero di elementi nell'heap h
`heap[empty](h)`	l'heap h è vuoto?
`queue[new](x1,x2,...)`	genera una nuova queue inzializzata con $x_1, x_2, \ldots$
`queue[enqueue](q,x)`	inserisce x in fondo alla queue q
`queue[dequeue](q)`	astrae il primo elemento dalla queue q
`queue[front](q)`	primo elemento della queue q
`queue[clear](q)`	svuota la queue q
`queue[reverse](q)`	inverte l'ordine della queue q
`queue[length](q)`	lunghezza della queue q
`queue[empty](q)`	la queue q è vuota?

Input-Output

`currentdir()`	directory corrente
`currentdir(dir)`	imposta *dir* come directory corrente
`open(f,READ)`	apre il file f in lettura senza buffer
`open(f,WRITE)`	apre il file f in scrittura senza buffer
`close(f1,f2,...)`	chiude i file $f_1, f_2, \ldots$
`fopen(f,RAED,opts)`	apre il file f in lettura con buffer
`fopen(f,WRITE,opts)`	apre il file f in scrittura con buffer
`fopen(f,APPEND,opts)`	apre il file f in scrittura con buffer
`fflush(f1,f2,...)`	scrive il contenuto dei buffer nei file $f_1, f_2, \ldots$
`fclose(f1,f2,...)`	chiude i file $f_1, f_2, \ldots$
`fremove(f1,f2,...)`	rimuove i file $f_1, f_2, \ldots$
`feof(f)`	la posizione corrente nel file f è terminale?
`readbytes(f,n,opts)`	legge n byte dal file f
`readbytes(f,rtable)`	legge il contenuto della struttura *rtable* dal file f

`readline(f)`	legge una riga dal file f
`fscanf(f,fmt)`	legge dal file f secondo il formato *fmt*
`sscanf(s,fmt)`	legge dalla stringa s secondo il formato *fmt*
`fprintf(f,fmt,a,b,...)`	scrive $a, b, \ldots$ nel file f secondo il formato *fmt*
`sprintf(fmt,a,b,...)`	scrive $a, b, \ldots$ come stringa secondo il formato *fmt*
`ImportVector(f,opts)`	importa un vettore dal file f
`ImportMatrix(f,opts)`	importa una matrice dal file f
`ExportVector(f,V,opts)`	esporta il vettore V nel file f
`ExportMatrix(f,M,opts)`	esporta la matrice M nel file f
`read f`	legge comandi dal file f
`save a,b,...,f`	salva i valori delle variabili $a, b, ...$ nel file f

Procedure

`proc(paramSeq)::t;`	procedura di tipo t con argomenti descritti dalla sequenza *paramSeq*
`  local localSeq;`	variabili locali descritte dalla sequenza *localSeq*
`  global globalSeq;`	variabili global descritte dalla sequenza *globalSeq*
`  option optionSeq;`	sequenza delle opzioni (`remember` per generare una *remember table*)
`  description descrSeq;`	descrizione opzionale della procedura
`  uses usesSeq;`	*usesSeq* è la sequenza dei moduli e package usati dalla procedura
`  statements`	comandi che formano il corpo eseguibile della procedura
`end proc;`	

`if expr1 then`
`  statments1`
`elif expr2 then`
`  statments2`
`...`
`else`
`  statementn`
`end if`
	struttura *if*

`while expr` `  do statements end do;`	struttura *while*
`for i from m by j to n` `  do statements end do;`	ciclo con i che varia da m a n con incremento j (opzionale)
`for i in I` `  do statements end do;`	ciclo con i che varia in I (insieme o lista)
`break`	per uscire da un ciclo prima della fine
`next`	per saltare al passo successivo del ciclo
`return expr1,expr2,...`	per uscire dalla procedura con la sequenza di valori $expr_1, expr_2, \ldots$

Moduli

`module()`	modulo (= contenitore di definizioni)
`  export s1,s2,...;`	$s_1, s_2, \ldots$ sono i simboli le cui definizioni sono visibili all'esterno
`  local localSeq;`	variabili locali descritte dalla sequenza *localSeq*
`  global globalseq;`	variabili global descritte dalla sequenza *globalSeq*
`  option optionSeq;`	sequenza delle opzioni (`remember` per generare una *remember table*)
`  description descrSeq;`	descrizione opzionale della procedura
`  uses usesSeq;`	*usesSeq* è la sequenza dei moduli e package usati dalla procedura
`  statements`	corpo eseguibile del modulo contenente le definizioni di $s_1, s_2, \ldots$
`end module`	
`exports(mod)`	sequenza dei simboli definiti ed esportati dal modulo *mod*
`mod:-fp`	funzione/procedura esportata dal modulo *mod*

Package

`packages()`	lista dei package caricati
`with(pack)`	carica il package *pack*
`with(pack,fp1,fp2,...)`	carica le funzioni/procedure *fp1*, *fp2*, ... del package *pack*
`unwith(pack)`	rimuove le definizioni caricate dal package *pack*
`pack[fp]`	funzione/procedura *fp* del package *pack* (richiamata anche senza aver caricato il package *pack*)