Red (autore: Vittorio Albertoni)

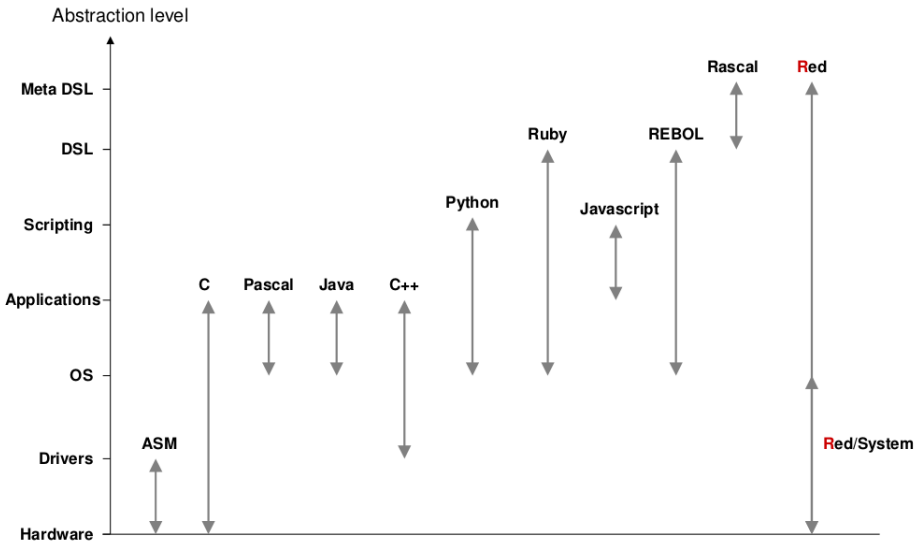
Premessa

Red, come si capirà meglio leggendo queste pagine, è, per molti aspetti, un linguaggio rivoluzionario.

Occorre però riconoscere che di questa caratteristica esso è debitore ad un altro linguaggio al quale si ispira, il linguaggio REBOL, acronimo di Relative Expression Based Object Language. Questo linguaggio, creato dal veterano della programmazione per personal computers Carl Sassenrath, primo sviluppatore di AmigaOS, ha un nome la cui pronuncia inglese richiama il termine «ribelle» e, a detta del suo creatore, esso infatti si ribella al concetto che il software moderno debba essere grande e complesso.
In REBOL, creato verso la fine del secolo scorso, tutto è all'insegna del minimalismo: risibile impiego di risorse, file di dimensioni minime, eliminazione della verbosità che caratterizza più o meno tutti i linguaggi di programmazione.
REBOL si è affermato come domain-specific language (in sigla DSL), linguaggio di dominio specifico nello sviluppo software e nell'ingegneria di dominio.

Basandosi su questo linguaggio Nenad "DocKimbel" Rakočević ha creato il linguaggio Red, che vuole essere un linguaggio all purpose.

Il suo stesso creatore, in una conferenza tenutasi nel 2013, dopo un paio d'anni dall'avvio dei lavori per il nuovo linguaggio, così collocava il suo futuro linguaggio, con una lodevole ambizione, tra i principali linguaggi esistenti



E dopo molti anni di lavoro e di continui affinamenti molto probabilmente l'obiettivo sta per essere raggiunto.

Come si vede dal grafico, il linguaggio si presenta in due edizioni: Red e Red/System.

Lasciando Red/System ai professionisti che si occupano di software di sistema, noi qui ci occuperemo di Red e scopriremo che è un linguaggio adatto anche per principianti e dilettanti e sa essere anche molto divertente.

Indice

1 Predisposizione del computer

Per usare Red non dobbiamo effettuare installazioni come si usa fare con gli altri linguaggi.

Dobbiamo semplicemente copiare sul computer alcuni file, sistemarli in un luogo comodo per poterli utilizzare e, se siamo su Linux, fare alcune cose per predisporre il computer.

Red è all'indirizzo *https://www.red-lang.org/*.

Qui, oltre a trovare notizie sul linguaggio, documentazione e tutorial, nella pagina DOWNLOAD, troviamo i file che ci servono per lavorare con il linguaggio stesso.

I file sono due:

. uno dei due, classificato CLI Red, ha un nome che incomincia per `red` ed è l'interprete per lavorare su normale console attraverso il terminale,

. l'altro, classificato GUI Red, ha un nome che incomincia per `red-view` ed è l'interprete per lavorare su una console diversa dal terminale e, volendo, in modalità grafica.

Per i sistemi Linux e Windows abbiamo un terzo file, classificato Red Toolchain, con un nome che incomincia, appunto, con `red-toolchain` ed è il compilatore e fa parte di Red/System.

Possiamo semplificare le cose ribattezzando questi file con nomi più semplici, per esempio nominando `red` quello per lavorare su terminale, redv quello per lavorare in modalità grafica e `redc` il compilatore.

Un mio suggerimento per poter lavorare come si deve con Red è di creare una directory in cui collocare i file scaricati e ribattezzati e in cui posizionarci per lavorare.

Se lavoriamo su Linux non dimentichiamo di rendere eseguibili i file ricorrendo alla scheda PROPRIETÀ ▷ PERMESSI del gestore di file o da terminale con `chmod 555`.

Ancora, per chi lavora su Linux a 64 bit, è necessario abilitare il sistema ad operare a 32 bit. In proposito troviamo istruzioni dettagliate per le principali distro Linux nella stessa pagina da cui scarichiamo i file.

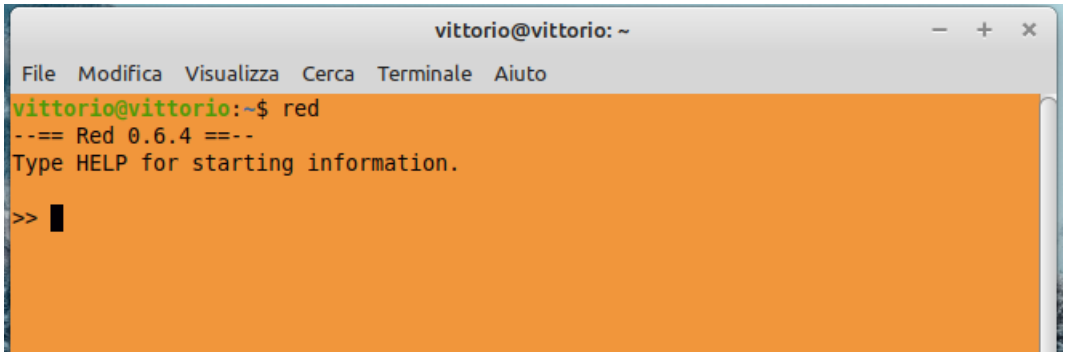
Consultando ricorrentemente la home page di Red potremo avere notizie su come evolverà tutto questo, in particolare su quando Red sarà disponibile per sistemi a 64 bit.

Un primo segnale sul fatto che siamo in presenza di qualche cosa di rivoluzionario l'abbiamo riscontrando la dimensione dei file che fanno funzionare tutto: nessuno di questi supera i 2 MB.

2 Come funziona

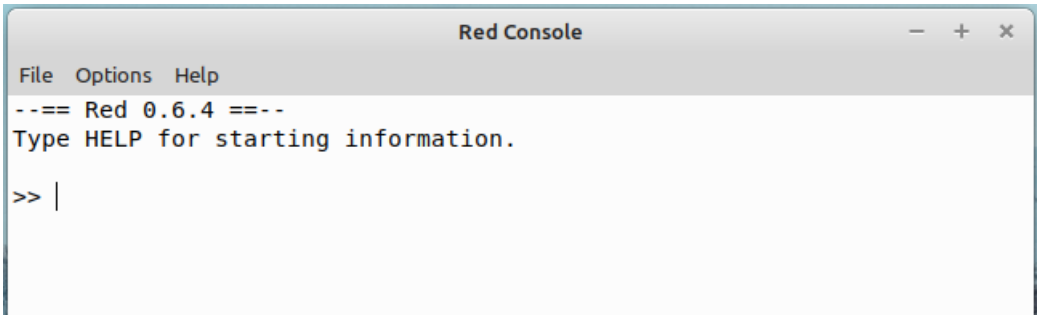
Abbiamo visto che disponiamo di due file eseguibili per avviare Red.

Se eseguiamo quello che ho suggerito di ribattezzare `red` trasformiamo il nostro terminale in una shell per il linguaggio, così



Scrivendo al prompt istruzioni in linguaggio Red, alla pressione di INVIO ne otteniamo l'esecuzione nello stesso terminale

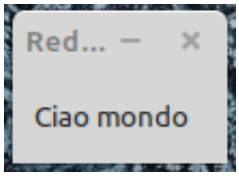
Se eseguiamo quello che ho suggerito di ribattezzare `redv` apriamo una console grafica che si presenta così



Anche in questo caso possiamo inserire al prompt istruzioni per vederle eseguite nella console alla pressione di INVIO ma, in più, possiamo lavorare con la grafica

```
>> print "Ciao mondo"
Ciao mondo
>> 6 * 8
== 48
>> view [text "Ciao mondo"]
```

Come vediamo, le prime due istruzioni vengono eseguite nella console come prima venivano eseguite nel terminale, ma l'esecuzione della terza produce questo



La shell ci consente interattività con il linguaggio, utile per imparare e sperimentare istruzioni oppure per eseguire brevi programmini, come calcoli al volo.

Veri e propri programmi vanno scritti, memorizzati in un file per eseguirli successivamente, magari compilati in codice macchina.

Per scrivere un programma in linguaggio Red ci basta un qualsiasi editor di testo (non un word processor salvo sia utilizzato per memorizzare il file in testo semplice non formattato).

Vedremo poi che cosa sia esattamente un programma in linguaggio Red e come si debba scrivere perché funzioni.

Per intanto ci basti sapere che Red è case insensitive, cioè non distingue lettere minuscole e maiuscole, per cui è indifferente, per esempio, scrivere l'istruzione `print` come `Print` o `PrInT`. Fa eccezione ciò che dobbiamo scrivere nella prima riga di un programma, dove la parola Red va scritta proprio `Red` non, per esempio, `RED` o `Red`.

Questa prima riga deve almeno essere

`Red [ ]` per i programmi da interpretare con l'interprete ribattezzato `red` (senza grafica),
`Red [needs: view]` per i programmi da interpretare con l'interprete ribattezzato `redv`.

Una volta scritto il programma lo salviamo in un file con estensione `.red` nella directory di lavoro che abbiamo dedicato a Red.

Per eseguirlo, posizionati nella directory di lavoro di Red, possiamo scrivere a terminale il nome dell'interprete (`red` o `redv`) seguito dal nome del file che contiene il programma.

Possiamo commentare una riga del programma scrivendo il commento dopo un punto e virgola (`;`). Per commenti più lunghi abbiamo a disposizione la funzione `comment` che possiamo usare con queste sintassi:

```
comment { .......<commento>........}
comment [ .......<commento>........]
comment " .......<commento>........"
```

Compilazione del programma

Come ho detto nel Capitolo 1, per i sistemi Linux e Windows abbiamo a disposizione un terzo file, classificato Red Toolchain, con un nome che incomincia, appunto, con `red-toolchain`, che è il compilatore di Red/System e che ho suggerito, una volta che l'abbiamo scaricato, di rinominarlo semplicemente come `redc` e di piazzarlo nella nostra directory di lavoro dedicata a Red.

Questo file ci consente di compilare il programma scritto in linguaggio Red in modo che esso possa essere eseguito sul nostro computer con velocità molto maggiore ed anche su computer ove non è installato l'interprete Red, proprio generando un file eseguibile come possiamo fare quando utilizziamo linguaggi come C o Pascal.

Penso sia il compilatore più leggero che esista: infatti pesa solo 1,5 MB.

Vi sono tanti modi di utilizzare il compilatore. Per noi principianti ritengo basti e avanzi citarne tre, ognuno dei quali corrisponde ad una opzione da indicare in questa istruzione:

`redc` <opzione> <nome_file`.red`>

dove

<nome_file`.red`> è il file che contiene il programma da compilare

<opzione> può essere espressa con

`-c` per generare un eseguibile attraverso la libreria `libRedRT`,

`-r` per generare un eseguibile stand alone,

`-t` per generare un eseguibile stand alone in cross compilation.

Nel primo caso, la prima volta che usiamo il comando, viene generata la libreria `libRedRT` nella nostra directory di lavoro e la presenza di questa libreria velocizza la compilazione di successivi file.

L'eseguibile così generato può però essere utilizzato solo su un computer in cui sia presente questa libreria (cioè un computer che abbia installato l'interprete Red e sul quale sia avvenuta almeno una compilazione con l'opzione `-c`).

Nel secondo caso generiamo un vero e proprio eseguibile stand alone, utilizzabile anche su un computer ove non sia installato l'interprete Red.

In entrambi i casi, se siamo su Linux generiamo un eseguibile per Linux, che avrà lo stesso nome del file del programma senza l'estensione .red, e se siamo su Windows generiamo un eseguibile per Windows, che avrà lo stesso nome del file del programma con l'estensione `.exe`.

Nel terzo caso, su qualsiasi sistema operativo stiamo lavorando, aggiungendo all'opzione `-t` il nome del sistema operativo di destinazione, generiamo un eseguibile stand alone per questo sistema operativo.

Limitandoci ai tre più diffusi, i nomi dei sistemi operativi riconosciuti per questa operazione sono i seguenti

`Linux` per uno dei sistemi operativi GNU/Linux,

`Windows` per il sistema operativo Windows,

`Darwin` per il sistema Mac OS Intel e solo per applicazioni console,

`macOS` per il sistema operativo Mac OS e per applicazioni bundles.

Per esempio, se abbiamo un file di programma denominato `prova.red` e stiamo lavorando su Linux, con l'istruzione

```
redc -t Windows prova.red
```

generiamo l'eseguibile `prova.exe` che gira su Windows, anche senza che vi sia installato l'interprete Red.

I file compilati, fino a nuovo ordine, sono destinati a sistemi a 32 bit.

Pertanto, se siamo su un sistema Linux a 64 bit, per utilizzare l'eseguibile che abbiamo compilato, dobbiamo preventivamente installare le librerie di supporto per l'ambiente a 32 bit secondo le istruzioni che troviamo nella pagina DOWNLOAD all'indirizzo *https://www.red-lang.org/*, di cui ho già parlato nel Capitolo 1.

3 Il linguaggio in pillole.

Un programma Red è una serie di parole, che possono essere azioni (per esempio `print`) o valori (per esempio `8`).

Red vede la serie di parole come continua, ignorando i new-line che possiamo inserire con la pressione di INVIO per andare a capo quando scriviamo il programma su un editor di testo.

Red legge la serie di parole ed esegue le azioni indicate partendo da sinistra e procedendo verso destra.

Le azioni devono corrispondere ad una funzione prevista dal linguaggio e indicate usando la sintassi prevista dal linguaggio.

Possiamo conoscere la sintassi delle varie funzioni utilizzando il ricco help di Red, scrivendo nella shell di Red un punto di domanda, seguito, dopo uno spazio, dal nome della funzione.

Per esempio, se scriviamo nella shell

```
?  print
```

otteniamo

```
USAGE: PRINT value
DESCRIPTION: Outputs a value followed by a newline.
PRINT is a native!  value.
ARGUMENTS: value [any-type!]
```

da cui veniamo a sapere che

. la funzione si usa richiamandone il nome seguito da un valore,

. in output otteniamo il valore indicato seguito da a capo,

. si tratta di una funzione nativamente prevista dal linguaggio,

. il valore da indicare come argomento può essere di uno qualunque dei tipi riconosciuti.

I valori devono essere di un tipo riconosciuto dal linguaggio.

La loro indicazione può avvenire attraverso espressioni algebriche. Red sostituirà alle espressioni un valore. Attenzione al fatto che la valorizzazione delle espressioni viene sempre fatta procedendo da sinistra a destra senza rispettare la precedenza algebrica degli operatori. Possiamo modificare ciò con l'uso di parentesi tonde.

Esempi:

`print "Ciao"` scrive la parola `Ciao` ricavata dal tipo stringa `"Ciao"` e va a capo,

`print 8` scrive il valore di tipo intero `8` e va a capo,

`print 8 + 5 * 2` scrive il valore `26` e va a capo.

Per ottenere il corretto valore dell'espressione secondo la precedenza algebrica degli operatori, che sarebbe 18, dobbiamo indicare

```
print 8 + (5 * 2)
```

Le parole che compongono il programma vanno sempre separate da uno spazio.

Dobbiamo, per esempio, scrivere `print "Ciao"` e non `print"Ciao"`.

Così come dobbiamo scrivere `8 + 5 * 2` e non `8+5*2` o `8 + 5*2`.

E' possibile organizzare le serie di parole che compongono il programma in blocchi elencando le parole tra parentesi quadre. Questo può essere un modo per creare quelle che in altri contesti vengono chiamate routine. L'esecuzione di ciò che è racchiuso tra le parentesi può avvenire attraverso la funzione `do` oppure, in presenza di più valutazioni, `reduce`.

Se scriviamo nella shell

`do [print "Ciao" 7 + 5 8 * 4]` otteniamo `Ciao 32`

in quanto `do` esegue le istruzioni ma valuta solo l'ultima espressione contenuta nel blocco

Se scriviamo nella shell

`reduce [print "Ciao" 7 + 5 8 * 4]` otteniamo `Ciao [unset 12 32]`

4 Parole

La parola è la base del linguaggio Red e, abbiamo detto, può rappresentare un'azione o un valore.

In questo manualetto verremo a conoscere le principali parole chiave che identificano le funzioni proprie del linguaggio. Una di queste, che identifica la funzione `print`, l'abbiamo vista nel precedente Capitolo.

Ma possiamo noi stessi creare parole assegnando loro o un'azione o un valore.

La creazione avviene con la sintassi

<parola>: <assegnazione>

L'assegnazione di un'azione può avvenire attraverso un blocco di istruzioni o attraverso la creazione di una funzione.

Per esempio, per avere a disposizione una routine che mi rivolga un saluto potrei creare la parola `salutami` in questo modo

```
salutami:  [print "Ciao, Vittorio, spero tu stia bene!"]
```

ed ottenere l'azione del saluto nella forma indicata ricorrendo alla funzione `do`, così

```
do salutami
```

Oppure, per avere a disposizione una funzione che raddoppi un numero, ricorrendo alla sintassi della funzione `func`, potrei creare la parola raddoppia in questo modo

```
raddoppia:  func [n] [2 * n]
```

in modo che con l'istruzione

```
raddoppia 5
```

possa ottenere il risultato `10`.

Interessante notare che nella shell dove abbiamo creato queste due parole possiamo ricorrere all'help per sapere come si usano.

Scrivendo

```
?  salutami
```

otteniamo

```
SALUTAMI is a block!  value.  length:  2 [print "Ciao, Vittorio, spero tu sti...
```

e scrivendo

```
?  raddoppia
```

otteniamo

```
USAGE: RADDOPPIA n
DESCRIPTION: RADDOPPIA is a function!  value.
ARGUMENTS: n
```

L'assegnazione di un valore, utilizzando un tipo di dato riconosciuto dal linguaggio, può avvenire indicando un valore o una espressione per calcolarlo. In questo modo la nostra parola diventa quella che in altri linguaggi di programmazione viene chiamata variabile.

Per esempio

`x:  5` assegna alla parola `x` il valore `5`

`y:  5 * 8` assegna alla parola y il valore `40`

Con l'espressione

`z:  x + y` assegniamo a `z` il valore dell'espressione (`45`)

esattamente come avviene utilizzando le variabili in altri linguaggi.

Esiste una sintassi alternativa per creare parole e assegnare ad esse un'azione o un valore:

`set '`<parola>: <assegnazione>

che contiene anche il modo per annullare l'assegnazione

`unset '`<parola>

Per esempio

`set 'a 45` assegna alla parola a il valore `45`

`unset 'a` annulla il valore assegnato alla parola `a`.

5 Tipi

Esaminiamo i principali tipi di dato riconosciuti da Red.

Il tipo viene attribuito automaticamente in relazione a ciò che scriviamo.

Esiste una funzione che ci indica il tipo attribuito:

`type?`

seguita da un dato o dalla parola cui è stato assegnato un dato.

Per esempio, se scriviamo al prompt della shell

`type? "Ciao"` otteniamo in risposta `string!`

Avendo una parola `x` con assegnato il valore `4`

`type? x` ci dà in risposta `integer!`

5.1 Numeri

I dati numerici sono

integer!

Gli interi sono a 32 bit con segno.

Ciò significa che il valore massimo dei numeri interi con cui possiamo lavorare è compreso tra –2.147.483.648 e 2.147.483.647.

Se un numero sta fuori da questo range, Red gli attribuisce il tipo float!

float!

I numeri in virgola mobile sono a 64 bit.

Ciò significa che la precisione è tale per cui abbiamo a disposizione 16 cifre significative, virgola compresa.

Se a un intero che esce dai limiti visti prima viene attribuito il tipo float! possiamo arrivare a valutare numeri fino ad oltre 300 cifre, ma solo le prime quindici sono precise.

Vengono riconosciuti come float! i numeri scritti con una parte decimale, sia essa indicata con il separatore virgola (,) sia essa indicata con il separatore punto (.), sia essa indicata in notazione scientifica.

Scrivendo 12,6 oppure 12.6 oppure 0.126e2 oppure 0,126e2 otteniamo sempre lo stesso numero di tipo float! che, nell'usanza italiana, è 12,6.

Un interessante esercizio per sperimentare tutto ciò che riguarda i numeri può essere quello di costruirci una funzione per calcolare il fattoriale.

Non scrivo la funzione in forma ricorsiva (Red supporta benissimo la ricorsione) in quanto non mi consentirebbe di fare ciò che voglio e che poi, subito qui sotto, spiegherò ma la scrivo utilizzando un ciclo in questo modo:

```
fattoriale: func [n] [either n = 0 [1][f: 1.0 repeat i n [f: f * i]]]
```

A suo tempo vedremo i cicli e scopriremo che quello qui utilizzato fa corrispondere un ciclo a ciascun intero inferiore al numero `n` di cui si vuole calcolare il fattoriale, a partire dal numero più piccolo e in ciascun ciclo moltiplica il valore ottenuto prima con il numero successivo: proprio come si fa per calcolare il fattoriale.

La parola `f` cui viene via via assegnato il risultato del calcolo è stata inizializzata ad un valore di tipo float! (`f: 1.0`) in quanto, trattandosi di un valore locale, all'interno della funzione, Red non ne varierebbe il tipo in presenza di un intero oltre il suo limite fisiologico e, se la parola cui assegnare i risultati del calcolo fosse di tipo intero, ci dovremmo fermare al calcolo del fattoriale di 12, in quanto il calcolo del fattoriale di 13 genererebbe stack overflow.

Utilizzando la nostra funzione, con la forzatura verso un risultato in virgola mobile, sul computer che sto usando arrivo a calcolare il fattoriale di 170 che viene indicato con 7.257415615307994e306

un bel numero di 307 cifre, di cui sono precise solo le prime sedici, virgola compresa e già l'ultima cifra indicata (4) è sbagliata."

Sempre sul computer su cui sto lavorando, il calcolo del fattoriale di 171 utilizzando la funzione indica come risultato un infinito, senza andare in stack overflow.

5.2 Caratteri

I dati che rappresentano caratteri alfabetici sono

char!

Racchiuso tra doppi apici e preceduto dal simbolo # è un carattere Unicode.

In realtà si tratta del codice dei caratteri Unicode in numero intero da 0 a 1.114.111.
#"A"
è il carattere A maiuscolo e Red lo riconosce con l'intero in base decimale 65, corrispondente al codice esadecimale 0041 della tabella Unicode UTF-8.

string!

E' una serie di caratteri racchiusa tra doppi apici o tra parentesi graffe.
`"Ciao bellezza"` è una stringa, come `{Ciao bellezza}`.

5.3 Valori diversi

Altri tipi che identificano valori sono i seguenti

none!

Quello che in altri linguaggi è `null`. Indica un dato inesistente.

logic!

Ricomprende i valori `true` e `false` e gli omologhi `on` e `yes` (che valgono `true`) e off e `no` (che valgono `false`).

time!

Esprime un'ora che può essere indicata come ora:minuti:secondi.sottosecondi
`ora: 8:45` attribuisce alla parola `ora` il valore `8:45:00`
e, in maniera intelligente:
`ora: 0:75` assegna alla parola `ora` il valore `1:15:00`

date!

Esprime una data che può essere indicata come giorno/mese/anno o anno/mese/giorno separati con barra o lineetta.

```
data: 31/12/2022
data: 2022/12/31
data: 31-12-2022
data: 2022-12-31
```

assegnano tutti alla parola `data` il valore `31-Dec-2022`

Red controlla che la data sia una data reale, controllando pure gli anni bisestili.
Se si scrivono i valori 29/2/2023, 31/11/2022 ecc. vengono rifiutati.

pair!

Esprime le coordinate di un punto in un sistema di assi cartesiane x y e si indica con la x e la y separate dal carattere `x` senza spazi di separazione.
`a: 12x23` assegna alla parola `a` le coordinate cartesiane `12` e `23`.

percent!

Serve per calcolare una percentuale e si scrive aggiungendo al numero indicante la percentuale il simbolo %.
`a: 5%` attribuisce alla parola `a` il valore `5%`, cioè `0,05`
`a: 30 * 5%` attribuisce alla parola a il valore corrispondente al `5%` di `30`, cioè `1,5`

tuple!

E' una lista contenente da 3 a 12 elementi separati da un punto.
`bianco: 255.255.255` è una tupla che identifica il colore bianco in RGB.

5.4 Altri tipi

url!

E' l'indirizzo di un sito web che si scrive nella forma
<protocollo>://<indirizzo>
per esempio
`a: https://www.vittal.it`
assegna alla parola `a` l'indirizzo del mio blog.

L'unico controllo che fa Red è che contenga i caratteri `://` non come caratteri iniziali.

email!

E' un indirizzo email.

L'unico controllo che fa Red è che contenga un carattere @ non come primo carattere.

5.5 Tipi contenitori

block!

Successione di valori di qualsiasi tipo riconosciuto racchiusa tra parentesi quadrate.

Abbiamo visto nel precedente Capitolo 3 che possiamo comporre blocchi anche con istruzioni.

Red utilizza il blocco anche per quelli che in altri linguaggi si chiamano array o matrici.
`a: [1 2 3 4]` costruisce l'array `[1 2 3 4]`
`m: [[3 2] [4 1] [6 2]]` costruisce la matrice

$$\begin{vmatrix} 3 & 2 \\ 4 & 1 \\ 6 & 2 \end{vmatrix}$$

Gli elementi del blocco sono indicizzati a partire da 1 e si ritrovano con la sintassi qui esemplificata applicata ai blocchi a e m che abbiamo appena costruito:
`a/2` ritorna `2`
`a/4` ritorna `4`
`a/5` ritorna `none`
`m/2` ritorna `[4 1]`
`m/1/1` ritorna `3`
`m/3/2` ritorna `2`

vector!

L'array che abbiamo costruito prima come blocco non è direttamente assoggettabile a operazioni matematiche.
Se, utilizzando l'array a che abbiamo costruito nel paragrafo precedente per l'operazione
`a * 2`
generiamo un errore.

Per operazioni di questo tipo dobbiamo utilizzare il tipo vettore che si crea con
`make vector! [<dato> <dato> .....]`
dove <dato> può essere di tipo integer!, float!, char! o percent!

Con
`v: make vector! [1 2 3 4]`
creiamo il vettore `1 2 3 4` e con
`v * 2`
otteniamo il vettore `2 4 6 8`

map!

E' quello che in altri linguaggi si chiama dizionario ed è una sequenza di dati accoppiati ad una chiave.

Si costruisce con
`make map! [<chiave> <dato> <chiave> <dato> ......]`

Per ritrovare un dato si usa la funzione `select`.

Creata la map!
`m: make map! ["Italia" "Roma" "Francia" "Parigi" "Spagna" "Madrid"]`
con
`select m "Francia"` otteniamo `"Parigi"`

5.6 Conversioni di tipo

E' possibile effettuare conversioni da un tipo ad un altro utilizzando la funzione `to` con la seguente sintassi
`to` <tipo_a> <da>
dove
<tipo_a> è il tipo di destinazione della conversione,
<da> è il dato o la parola cui è assegnato il dato da convertire.

Esempi:
`to integer! 3.4` ritorna `3`
con `x` che vale `5,9` `to integer! x` ritorna `5`
`to float! 12` ritorna `12.0`
`to string! 14.6` ritorna `"14.6"`
con `s` che vale `"13.7"` `to float! s` ritorna `13.7`

e per arrivare all'intero occore fare
`to integer! to float! s`, che ritorna `13`

Esiste anche la funzione `to-time` per convertire nel formato time!
`to-time [12 78 85]` ritorna `13:19:25`

Esiste poi la funzione `as-pair` per convertire una coppia di numeri in una coppia di coordinate intere.
`as-pair 12 25` ritorna `12x25`
`as-pair 3.2 5.67` ritorna `3x5`
`as-pair 88 12.7` ritorna `88x12`

6 Accesso ai dati e qualche formattazione

Ogni parola, sia nativa del linguaggio sia creata da noi, è inserita in un dizionario.

E' il dizionario cui accede il sistema di help che abbiamo visto nel Capitolo 3.

Per vedere come il dizionario definisce una parola abbiamo anche a disposizione la funzione `get`, che si usa fornendo come argomento il nome della parola preceduto da apice semplice.

Se scriviamo nella shell

```
get 'get
```

otteniamo in risposta

```
== make native!  [[
   "Returns the value a word refers to"
```

Se scriviamo

```
get 'print
```

otteniamo la risposta

```
== make native!  [[
   "Outputs a value followed by a newline"
```

Se abbiamo creato la variabile `x` assegnandole il valore `15,5`, con

```
get 'x
```

otteniamo la risposta

```
== 15.5
```

Cioè, se la parola rappresenta un'azione otteniamo la descrizione di questa azione, se la parola rappresenta un dato otteniamo il dato.

Abbiamo anche due funzioni per formattare i dati assegnati alla parola.

La funzione `mold` trasforma il dato in stringa e così lo ritorna.

Data la variabile `x` che abbiamo appena visto, con

```
mold x
```

otteniamo

```
== "15.5"
```

Data una variabile `y` con assegnato il blocco `[4 6 7]`, con

```
mold y
```

otteniamo

```
== "[4 6 7]"
```

La funzione `form` fa la stessa cosa ma depura il dato da segni come parentesi o altro.

Nel caso dell'ultimo esempio, con

```
form y
```

otteniamo

```
== "4 6 7"
```

Altro esempio di formattazione lo abbiamo con il seguente script:

```
Red []
print "---------MOLD----------"
print mold {La mia casa
è veramente bella}
print "---------FORM----------"
print form {La mia casa
è veramente bella}
```

che, eseguito, fornisce il seguente risultato

```
---------MOLD----------
"La mia casa^/è veramente bella"
---------FORM----------
La mia casa
è veramente bella
```

evidenziando come l'uso di form sviluppi qualche eleganza in più.

7 Operazioni matematiche e logiche

Ricordare sempre che Red esegue le operazioni procedendo da sinistra a destra senza rispettare alcuna precedenza degli operatori. Per precedenze diverse occorre usare le parentesi tonde.

Il separatore decimale per i numeri in virgola mobile può essere indifferentemente la virgola o il punto.

Se fa comodo, si possono inserire numeri da tastiera usando come separatore delle migliaia un apice singolo che Red ignora.

Le espressioni per eseguire le operazioni si possono rappresentare come stringa e la stringa che le contiene può essere valorizzata utilizzando la funzione `do`.

Esempi:

```
s:  "7'425,5 - 2 * 8"
```

`do s` fornisce il risultato `59388.0`

```
ss:  "7'425,5 - (2 * 8)"
```

`do ss` fornisce il risultato `7409.5` che è quello che si otterrebbe se si rispettasse la precedenza algebrica degli operatori (qui forzata utilizzando le parentesi tonde).

Le funzioni e gli operatori che vedremo in questo Capitolo possono essere utilizzati con tutti i dati che Red rappresenta con caratteri numerici, ivi compreso il tipo char!.

Ciò nei limiti in cui l'operazione ha senso, in caso contrario Red restituirà un errore.

Per esempio sommare o elevare a potenza valori di tipo date! non ha senso e Red non lo fa ma la sottrazione tra due di questi valori viene eseguita e fornisce il numero di giorni che separa due date.

Aritmetica di base

Per le quattro operazioni aritmetiche e l'elevamento a potenza abbiamo a disposizione operatori e funzioni.

Le funzioni si usano in notazione prefissa ed accettano solo due argomenti operandi.

Gli operatori si inserisono tra gli operandi anche componendo lunghe espressioni.

funzione	operatore	
`add`	`+`	addizione
`subtract`	`-`	sottrazione
`multiply`	`*`	moltiplicazione
`divide`	`/`	divisione
`remainder`	`//`	resto della divisione
`power`	`**`	elevamento a potenza

Matematica

Gli argomenti passati a queste funzioni possono essere numeri, espressioni o parole con assegnato un valore.

`absolute` ritorna il valore assoluto

`exp` ritorna *e* elevato alla potenza indicata come argomento

`log-10` ritorna il logaritmo decimale

`log-2` ritorna il logaritmo in base 2

`log-e` ritorna il logaritmo naturale

`negate` ritorna il valore con segno invertito

`random` ritorna un numero casuale compreso tra 1 e il valore indicato come argomento se questo è un integer! oppure tra 0 e il valore indicato se questo è un float!

`round` ritorna l'intero più vicino

con l'opzione `/to` possiamo indicare fino a quale cifra decimale arrotondare

`round/to pi 0.001` arrotonda la costante di sistema π alla terza cifra decimale (3.142)

`square-root` estrae la radice quadrata

Trigonometria

Le funzioni dirette `sin`, `cos`, `tan` accettano l'argomento in radianti.

Le funzioni dirette `sine`, `cosine`, `tangent` accettano l'argomento in gradi.

Le funzioni inverse `asin`, `acos`, `atan` accettano l'argomento in float! e ritornano l'angolo in radianti.

Le funzioni inverse `arcsine`, `arccosine` e `arctangent` ritornano l'angolo in gradi.

Logica

Anche in questo caso abbiamo la possibilità di usare funzioni prefisse o operatori infissi.

funzione	operatore	
`and~`	`and`	e logico
`or~`	`or`	o logico

Confronto

Sempre con funzioni prefisse o operatori infissi.

funzione	operatore	
`equal?`	`=`	uguale
`not-equal?`	`<>`	non uguale, diverso
`greater?`	`>`	maggiore
`greater-or-equal?`	`>=`	maggiore o uguale
`lesser?`	`<`	minore
`lesser-or-equal?`	`<=`	minore o uguale

Abbiamo poi

. la funzione `same?` (corrispondente all'operatore `=?`) che segnala verità quando i due elementi confrontati si riferiscono allo stesso dato, cioè occupano lo stesso spazio di memoria.

Per esempio se abbiamo la parola `x: "Ciao"` e la parola `y: x`, dal momento che y punta allo stesso dato assegnato a x avremo che
`same? x y` (oppure `x =? y`) ritorneranno `true`.

. la funzione `strict-equal?` (corrispondente all'operatore `==`) che segnala verità quando vi è perfetta corrispondenza tra i due elementi confrontati. Per esempio, in un contesto dove maiuscole e minuscole non si distinguono, come avviene con Red, se abbiamo le due stringhe
`a: "Ciao"` e `b: "ciao"`
con `equal? a b` (oppure `a = b`) avremo `true`
con `strict-equal? a b` (oppure `a == b`) avremo `false`

8 Manipolazione di testo e stringhe

Qui elenco le principali funzioni per lavorare con testi e stringhe.

L'argomento da assegnare che indico con <a> può essere una stringa o una parola cui è assegnata una stringa.

Ricordo che l'indice dei caratteri che formano la stringa parte da sinistra con il numero 1.
`length?` <a> ritorna un intero che indica il numero di caratteri della stringa,
`insert` <a> <stringa> aggiunge una stringa all'inizio di un'altra,
`append` <a> <stringa> aggiunge una stringa alla fine di un'altra,
`insert at` <a> <indice> <stringa> inserisce una stringa in un'altra nel punto indicato,
`rejoin [`<a> <a> <a> `...]` concatena stringhe,
`trim` <a> rimuove spazi eventualmente presenti all'inizio e alla fine di una stringa,
`trim/head` <a> rimuove spazi eventualmente presenti all'inizio di una stringa,
`trim/tail` <a> rimuove spazi eventualmente presenti alla fine di una stringa,
`trim/lines` <a> rimuove indicazioni di end of line da una stringa,
`trim/with` <a> "<carattere/i>" rimuove da una stringa il carattere o i caratteri indicati.

9 Input e output standard

L'input e l'output standard sono rispettivamente quelli dalla tastiera e verso lo schermo.

Per l'input abbiamo a disposizione due funzioni:

`input` legge quanto scritto su tastiera, dopo premuto INVIO, come stringa,

`ask` <stringa> lo stesso, con possibilità di scrivere un messaggio per l'input.

Se si vuole registrare l'input in tipo diverso da string! occorre anteporre alla chiamata della funzione una istruzione per la conversione voluta.

Esempi:

`nome: input` assegna a nome il valore scritto su tastiera come stringa,

`nome: ask "Come ti chiami? "` lo stesso, premettendo la richiesta del nome,

`x: to float! input` assegna a x il valore scritto su tastiera come numero in virgola mobile,

`y: to integer! ask "Scegli un numero intero "` assegna a y il numero intero digitato.

Per l'output le principali funzioni sono le seguenti.

`print` riceve come argomento ciò che vogliamo scrivere sullo schermo, lo scrive a va a capo.

L'argomento può essere una stringa o un'espressione, che viene valutata prima di scriverla;

oppure può essere una parola e verrà scritto il valore ad essa assegnato.

`prin` lo stesso di print con la differenza che scrive e non va a capo,

`probe` scrive l'argomento senza effettuare alcuna valutazione.

Esempi:

`print "Ciao"` scrive la parola Ciao e va a capo,

print 3 * 4 scrive 12 e va a capo,

prin [4 + 5] scrive 9 e non va a capo,

`probe` [4 + 5] scrive [4 + 5]

10 Input e output da e su file

Per lavorare con un file occorre sempre indicare il percorso per arrivare al file stesso anteponendo il simbolo `%` alla serie di directory separate dalla barra `/`.

Per arrivare al file `prova` che si trova nella mia directory `home` nella sottodirectory `Documenti` dovrò indicare

`%/home/vittorio/Documenti/prova`

Leggiamo il contenuto di un file con la funzione

`read` <percorso>

Se assegniamo quanto letto a una parola, stampando il valore di quella parola vediamo l'intero contenuto del file così come è.

Scriviamo in un file con la funzione

`write` <percorso> <stringa_di_testo> Se il file non c'è viene creato. Se c'è viene sovrascritto.

`write/append` <percorso> <stringa_di_testo> Aggiunge contenuto a un file esistente.

11 Controllo del flusso

11.1 Esecuzione condizionale

`if` <test> <blocco_istruzioni>

se <test> è vero viene eseguito il blocco di istruzioni, altrimenti si passa oltre,

`either` <test> <blocco_istruzioni_per_vero> <blocco_istruzioni_per_falso>

se <test> è vero viene eseguito il primo blocco, altrimenti viene eseguito il secondo,

`switch` <valore> <blocco_alternative>

esegue il blocco di istruzioni corrispondente a un certo valore,

`switch/default` <valore> <blocco_alternative> <blocco_default>

esegue il blocco di default se nessuna alternativa è verificata,

`case` [<test> <blocco> <test> <blocco> <test> <blocco>.....]

fa una serie di test ed esegue il blocco corrispondente alla prima verità,

`case/all` [<test> <blocco> <test> <blocco> <test> <blocco>.....]

fa una serie di test ed esegue tutti i blocchi corrispondenti a verità.

Esempi:

```
>> if 4 < 5 [print "OK"]
OK
>> either 4 > 5 [print "OK"] [print "NO"]
NO
>> switch 20 [10 [print "dieci"] 20 [print "venti"]]
venti
>> switch/default 30 [10 [print "dieci"] 20 [print "venti"]] [print "non so"]
non so
>> case [1 > 2 [print "male"] 2 > 1 [print "bene"] 3 > 1 [print "benissimo"]]
bene
>> case/all [1 > 2 [print "male"] 2 > 1 [print "bene"] 3 > 1 [print "benissimo"]]
bene
benissimo
```

11.2 Ripetizione

`loop n` <blocco_istruzioni>

esegue le istruzioni contenute nel blocco `n` volte,

`repeat i n` <blocco_istruzioni>

lo stesso di loop gestendo un indice che si incrementa ad ogni giro,

`foreach i a` <blocco_istruzioni>

data una parola `a` con assegnato un blocco esegue istruzioni per ogni elemento del blocco,

`while` [<condizione>] <blocco_istruzioni>

esegue le istruzioni fino a quando una condizione è vera,

`until` [<blocco_istruzioni> <condizione>]

esegue le istruzioni fino a quando il blocco restituisce verità.

Esempi:

```
>> loop 2 [print "Ciao"]
Ciao
Ciao
>> repeat i 2 [prin i print " Ciao"]
1 Ciao
2 Ciao
>> a:  [1 2 3]
>> foreach i a [print i * i]
1
4
9
>> i:  1
>> while [i < 3] [print "Ciao" i:  i + 1]
Ciao
Ciao
>> i:  2
>> until [print "Ciao" i:  i - 1 i = 0]
Ciao
Ciao
== true
```

12 Piccoli programmi console

Ciò che abbiamo visto sin qui ci mette in condizione di scrivere programmi in linguaggio Red per console, cioè eseguibili su terminale.

Questo programma chiede il nome all'utente per salutarlo:

```
Red []
nome:  ask "Come to chiami?  "
print rejoin ["Ciao " nome "!"]
```

Quest'altro chiede il raggio di un cerchio per calcolare circonferenza e area del cerchio ed espone i risultati con tre cifre decimali:

```
Red []
r:  to float!  ask "Indica il raggio di un cerchio:  "
c:  2 * pi * r
a:  r * r * pi
prin "Per un cerchio di raggio " print r
prin "la circonferenza è " print round/to c 0.001
prin "l'area è           " print round/to a 0.001
```

Infine un programmino che traduce in inglese i giorni della settimana:

```
Red []
giorno:  ask "Scrivi un giorno della settimana in italiano:  "
switch/default giorno [
"lunedì" [print "in inglese si dice Monday"]
"martedì" [print "in inglese si dice Tuesday"]
"mercoledì" [print "in inglese si dice Wednesday"]
"giovedì" [print "in inglese si dice Thursday"]
"venerdì" [print "in inglese si dice Friday"]
"sabato" [print "in inglese si dice Saturday"]
"domenica" [print "in inglese si dice Sunday"]
]
[print "Non ho capito"]
```

13 Conclusione e rimando

In questo manualetto ho cercato di divulgare le basi del linguaggio Red per console e ciò ci mette in grado di produrre programmi del tipo di quelli esemplificati nel precedente Capitolo.

Dei due file che nel Capitolo 1 ho indicato comporre il sistema Red, questo manualetto è dedicato a ciò che si può fare con il primo file, quello classificato CLI Red, che ho suggerito di rinominare semplicemente come `red`.

Penso sia risultato evidente come il linguaggio Red sia diverso da tutti gli altri, soprattutto nella stringata sintassi, magari con una predisposizione poco orientata al calcolo.

Ma probabilmente la parte più spettacolare di Red la ritroviamo in ciò che si può fare con la grafica, sia finalizzata a programmi dotati di interfaccia utente grafica (GUI) sia finalizzata al puro e semplice disegno.

Per queste cose si usa il secondo file di cui ho parlato nel Capitolo 1, quello classificato GUI Red e mi propongo di produrre un manualetto a ciò dedicato.