di Roberto Rossi

www.redchar.net

Il Mio Lisp

Roberto Rossi
http://www.redchar.net
Edizione 08.2016.1

Copyright

Contributi e ringraziamenti

Un doveroso ringraziamento va a tutte le persone che hanno contribuito, in modo diretto o indiretto, alla realizzazione di questo libro :

- **Margherita Monti**
  Prima lettura e revisione, correzioni e suggerimenti vari, revisione di diversi altri capitoli. In seguito, correzione edizione 2016. Questo testo non sarebbe mai esistito senza il suo contributo.

- **Roberto Corona**
  Revisione completa del testo per l'edizione 2014/2015.

- **Elena Lombardo**
  Autrice della grafica di copertina dall'edizione 2015 e successivi. Ha inoltre collaborato alla realizzazione della nuova impaginazione del testo.
  http://www.elenalombardo.eu

- **Fabio Guerrera**
  Correzioni e suggerimenti vari;
  Autore del capitolo 16 (Entità non grafiche, xrecord e dizionari);

- **Cono La Galia**
  Correzioni e suggerimenti vari;

- **Flavio Tabanelli**
  Correzioni e suggerimenti vari;

- **Stefano Ferrario**
  Correzioni e suggerimenti vari.

Script Lisp presentati nel testo

Tutti gli script LISP presentati in questo testo possono essere scaricati dal sito dell'autore:

http://www.redchar.net

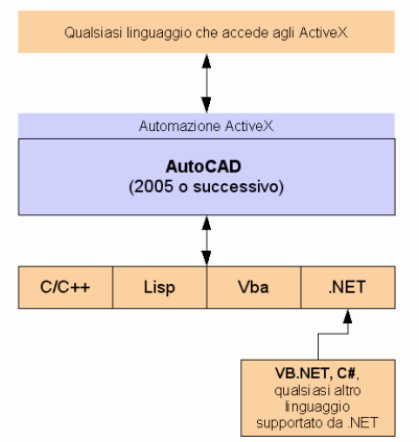
Indice generale

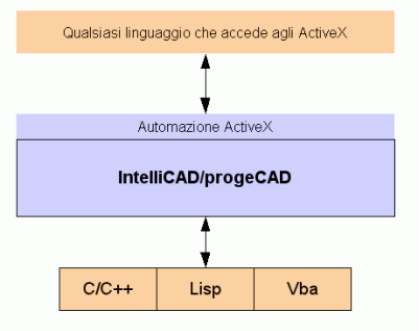
Capitolo 1
I CAD e il linguaggio Lisp

I sistemi di sviluppo

Quale linguaggio scegliere per sviluppare con questi ambienti CAD ? Chiunque intenda sviluppare applicativi per AutoCAD® o software simili, si sarà certamente posto questa domanda.

Per rispondere adeguatamente è necessario, prima di tutto, esaminare i sistemi di sviluppo disponibili in questi software.





Come si può facilmente notare, i linguaggi ufficialmente supportati sono, in larga parte comuni. A questo punto non ci rimane che esaminarli uno ad uno, per scoprirne i pregi e i difetti.

C/C++

Prima di tutto abbiamo C/C++, che meglio di tutti gli altri si presta alla realizzazione di applicazioni performanti, di qualsiasi tipo. Tutti gli applicativi maggiori sono scritti completamente, o almeno in parte, sfruttando questo linguaggio.

Le librerie di sviluppo, in AutoCAD® e negli altri CAD simili, mettono a disposizione il massimo per quanto riguarda velocità di esecuzione e flessibilità di gestione, sia del disegno sia delle funzionalità del sistema operativo sul quale il CAD si poggia. Con queste caratteristiche è adatto alla realizzazione di qualsiasi tipo di applicazione sia interna che esterna al CAD.

Il vero inconveniente con questo linguaggio, è rappresentato prima di tutto dalla curva di apprendimento molto ripida, inoltre dalla scarsa produttività raggiungibile nella

realizzazione di piccolo e medie applicazioni. Nonostante sia, in assoluto, il più potente, il suo apprendimento è decisamente più difficoltoso e il suo utilizzo, soprattutto su progetti medio/piccoli, risulta controproducente. Questo avviene, non solo in fase di stesura del programma, ma anche in fase di debug (correzione).

C/C++ diviene indispensabile in due sole situazioni:

- quando si richiedono funzioni non disponibili in altri ambienti;
- quando l'applicazione da realizzare risulta di grandi dimensioni, o richiede prestazioni elevate.

Si può comunque affermare, senza paura di essere smentiti, che qualsiasi software professionale non può prescindere dall'uso di C/C++.

Lisp

Presente sin dalle prime versioni, sia di AutoCAD®, sia di IntelliCAD®, Lisp è un linguaggio interpretato ed eseguito senza alcun processo intermedio, come ad esempio la compilazione, tipica di altri sistemi (come C/C++).

I suoi maggiori pregi sono la semplicità e l'immediatezza di utilizzo, che si traduce in una curva di apprendimento particolarmente favorevole.

E' possibile, con Lisp, iniziare a scrivere le prime procedure, dopo poche ore di studio. La sua potenza consente lo sviluppo di un ampio spettro di applicazioni, garantendo sempre una elevatissima produttività, con tempi ridotti e un processo di debug (correzione errori) molto rapido.

Anche la velocità di esecuzione è notevole, anche se non raggiunge i livelli di C/C++.

Ovviamente, sono presenti anche alcuni "difetti". Il primo nasce proprio dalla semplicità di Lisp, infatti questo linguaggio, che nella versione adottata dai CAD viene chiamato AutoLISP® è un sottoinsieme del più famoso Lisp originale (Lisp) e non presenta molte delle caratteristiche comuni ai moderni linguaggi di programmazione (Lisp compreso).

Un esempio può essere la mancanza del supporto alla programmazione orientata agli oggetti, limitandosi a quella procedurale.

Altro inconveniente è dato dalla gestione delle interfacce grafiche (GUI). In questo caso, l'uso di un apposito linguaggio, DCL (Dialog Control Language) limita fortemente lo sviluppo, rendendo accessibili solamente finestre di dialogo modali, con un numero ristretto di tipologie di controlli inseribili al loro interno (bottoni, caselle di testo, ecc).

Nonostante ciò, AutoLISP® è la scelta migliore quando si devono realizzare semplici procedure di interazione con il CAD. Grazie alla sua immediatezza e produttività, riduce drasticamente i tempi di sviluppo e consente a chiunque la creazione di procedure personalizzate.

Visual Basic for Application (VBA) ®

In questo caso siamo di fronte ad una via di mezzo tra AutoLISP® e C/C++.

Più semplice di C/C++, più potente di AutoLISP®, consente la realizzazione di complesse applicazioni, potendo realizzare sofisticate interfacce grafiche e potendo accedere, in maniera immediata al mondo dei database.

VBA è, in pratica, una versione semplificata del famoso Visual Basic, dal quale eredita un linguaggio sufficientemente completo per la realizzazione di applicazioni di medie dimensioni, garantendo discrete prestazioni, paragonabili a quelle di AutoLISP, ma sempre distanti da C/C++.

Per contro, VBA è più complesso da apprendere e su piccole procedure richiede una quantità di codice maggiore rispetto ad AutoLISP, a parità di programma.

Solitamente VBA è coadiuvato da piccole procedure Lisp, utili per compiti particolari, come ad esempio la definizione di nuovi comandi, cosa che VBA non è in grado di fare.

Microsoft .NET®

A Partire dalla versione 2005, AutoCAD® consente di caricare direttamente moduli sviluppati con la tecnologia .NET.

Due sono i linguaggi supportati ufficialmente da AutoDESK® (sviluppatrice di AutoCAD®):

- Visual Basic .NET;
- C#.

Entrambe, ovviamente, si basano sulla “nuova” piattaforma .NET. Ciò consente ai due di avere il medesimo potenziale, distinguendosi per la sintassi del linguaggio e per alcune semplificazioni inserite nel solo VB.NET.

Solitamente la scelta dell’uno o dell’altro è una questione di gusti personali, anche se la mia esperienza mi porta a ritenere VB.NET superiore a C#, soprattutto per quanto riguarda la semplicità e la produttività della sintassi.

Rispetto ad AutoLISP e VBA, i linguaggi basati su .NET sono molto più potenti e flessibili riuscendo, da questo punto di vista, ad eguagliare C/C++.

Dal punto di vista delle prestazioni, .NET necessita di una notevole quantità di memoria anche se, disponendo di un hardware adeguato, la velocità di esecuzione è di tutto rispetto, riuscendo a renderlo più performante di VBA e Lisp, anche se non riesce ad eguagliare C/C++.

Interfaccia ActiveX

Questa non è un linguaggio di programmazione, ma è una tecnologia (targata Microsoft) che consente ai software di interagire fra loro.

Sia AutoCAD, sia IntelliCAD/progeCAD, possiedono un'interfaccia ActiveX che fornisce una gerarchia di oggetti a tutti i linguaggi in grado di utilizzare questa tecnica. In pratica è possibile pilotare l'ambiente CAD da un'applicazione esterna, come se questa fosse integrata nell'ambiente stesso.

Questo ci consente di sviluppare software con Delphi, PHP, Python, Visual FoxPro, ecc...

Nonostante le possibilità offerte siano allettanti, si presentano alcuni rilevanti problemi. Prima di tutto è impossibile definire comandi all'interno dell'ambiente CAD, cosa che ci costringe ad appoggiarci a linguaggi interni (come Lisp). Abbiamo poi un grosso problema prestazionale. Purtroppo, ogni operazione effettuata attraverso l'interfaccia ActiveX, risulterà parecchie volte più lenta della medesima operazione fatta con uno dei linguaggi interni (VBA, Lisp, .NET, ecc...), pregiudicando il funzionamento di applicazioni, anche molto semplici.

Tutto ciò, consente di scrivere solamente software con una limitata interazione con il CAD.

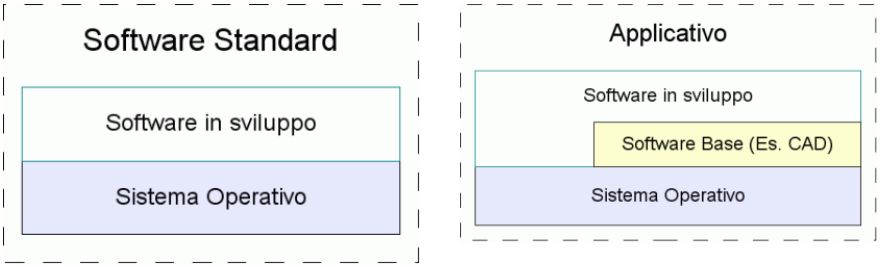
Alcune considerazioni

Iniziamo col dire che, non esiste il linguaggio perfetto. Ogni applicazione che si sviluppa è mondo a se. Se un linguaggio è adatto per la realizzazione di un determinato software, non è affatto detto che sia la scelta migliore per la realizzazione di un'altra tipologia di programma.

Ci sono però, alcuni fatti da considerare durante la scelta del sistema di sviluppo.

Iniziamo col valutare, cos'è lo sviluppo di un'applicativo CAD. Questa tipologia di software, possiede un livello di complessità superiore, rispetto ad un normale programma.

Vediamo una rappresentazione grafica per confrontare un software applicativo ed un normale programma:



Nel caso di un “Software Standard”, l’interazione avviene esclusivamente con il sistema operativo, di conseguenza qualsiasi linguaggio che supporti adeguatamente quel SO è un buon candidato per la realizzazione di programmi efficaci.

Quando invece parliamo dello sviluppo di applicativi, che estendono le funzionalità di altri software (come nel caso di AutoCAD o IntelliCAD), siamo di fronte ad un livello di complessità decisamente superiore. Non solo dobbiamo scegliere un linguaggio in grado di lavorare bene con il sistema operativo (SO), ma anche in grado di operare, nel miglior modo possibile, con il programma da estendere.

Un altro fattore da considerare è la difficoltà, per un programmatore, nell’utilizzare più di un linguaggio o di un ambiente di sviluppo. Ovviamente, più cose ci saranno da apprendere e utilizzare, più confusione si genererà, riducendo in maniera progressiva, sia la produttività, sia la qualità del software prodotto.

Ne consegue che, un limitato numero di strumenti scelti attentamente, rappresenta la soluzione migliore.

Il Futuro

Come se tutto ciò non bastasse, è necessario avere un’idea di come si evolveranno, nei prossimi anni, i sistemi di sviluppo in AutoCAD e in IntelliCAD/progeCAD.

Per quanto riguarda IntelliCAD/progeCAD, tutti i linguaggi attuali saranno, certamente, supportati ancora per diverso tempo. Inoltre, come è sempre accaduto per questo CAD, il linguaggio maggiormente supportato, sviluppato e corretto sarà C/C++.

Nel caso di AutoCAD, le cose si fanno più complesse. Fermo restando che C/C++ e AutoLISP saranno certamente supportati e sviluppati, VBA e l’automazione ActiveX subiranno certamente un fermo nel loro sviluppo. Soprattutto VBA, sarà mantenuto ma non potenziato, in quanto NON più sviluppato da Microsoft, che lo fornisce come ambiente integrabile in altro software (nel nostro caso AutoCAD e

IntelliCAD/progeCAD).

E' evidente che, nelle intenzioni di AutoDESK ci sia la sostituzione di VBA e ActiveX con la "nuova" tecnologia .NET ed in particolare con VB.NET e C#, decisamente più potenti e performanti.

Il Supporto

Un ultimo aspetto da considerare riguarda il supporto, che i vari linguaggi possiedono.

Le società che sviluppano IntelliCAD e quella che produco i suoi derivati, forniscono ufficialmente un supporto professionale per quello che riguarda C/C++, VBA e Lisp, così come AutoDESK che, in più, supporta la piattaforma .NET con VB.NET e C#.

Per quanto invece concerne, le forme di assistenza gratuite, su Internet è possibile trovare una quantità di informazioni e di codice sorgente, per risolvere molti dei normali problemi che si incontrano durante lo sviluppo. Soprattutto i siti specializzati, il sito di AutoDESK e i gruppi di discussione, offrono un ottimo supporto in forma assolutamente gratuita.

Riassumendo

Come abbiamo già detto, il miglior linguaggio non esiste!

Il miglior linguaggio per la realizzazione di un'applicativo, può dimostrarsi il peggiore per lo sviluppo di un software di tipo differente. Ad esempio, una piccola procedura di interazione con il CAD, è più produttivo realizzarla in Lisp, piuttosto che scomodare C/C++. Al contrario, un grosso software architettonico è molto meglio realizzarlo con C/C++, grazie alla sua potenza e rapidità di esecuzione.

Da ciò deriva la necessità di apprendere ed utilizzare più di un linguaggio. Con IntelliCAD/progeCAD è quindi meglio apprendere, in tutto o solo in parte, tutti e tre i sistemi di sviluppo:

- C/C++;
- VBA;
- Lisp.

Con AutoCAD, la scelta è più flessibile in quanto, visto l'abbandono di VBA, si hanno a disposizione i due linguaggi ufficiali di .NET, VB.NET o C#. In quest'ultimo caso, chi ha esperienze con Visual Basic o con VBA prediligerà il primo, mentre chi arriva da esperienze con C/C++, Java o linguaggi con sintassi simile al C, prediligerà C#.

In ultima analisi, la scelta, per i motivi esposti finora (supporto, documentazione,

efficienza) non può che essere fatta all'interno dei linguaggio ufficiali, evitando strade alternative, forse più stimolanti, ma decisamente controproducenti.

Linguaggi Alternativi

In alternativa ai linguaggi ufficiali, tramite l'interfaccia ActiveX e la tecnologia .NET è possibile utilizzare una vasta schiera di strumenti di sviluppo. Tanto per fare alcuni esempi, è certamente possibile utilizzare Delphi o Python. Purtroppo questo crea una serie di problemi, di non poca importanza:

1. Nessun supporto ufficiale. Le aziende sviluppatrici dei software CAD, per limitare il numero di problematiche possibili, supportano solamente una piccola schiera di linguaggi, ignorando gli altri. Ovviamente questo fatto incide negativamente quando ci si trova ad affrontare i vari problemi legati allo sviluppo del software;
2. Scarse Informazioni. Vista la ridotta comunità di sviluppatori che segue i linguaggi non ufficiali, anche il supporto che arriva da Internet è scarso e, alle volte, di poco aiuto;
3. Scarse prestazioni. Sia Python, sia Delphi possono interagire con l'ambiente CAD, attraverso l'interfaccia ActiveX, cosa che però pregiudica fortemente le prestazioni dell'applicativo;
4. Come avrete certamente capito, non è possibile ignorare i linguaggi ufficiali, che comunque devono essere studiati ed utilizzati, soprattutto se si desidera assistere adeguatamente i propri clienti durante lo sviluppo delle loro personalizzazioni. Aggiungere a questi altre tecnologie e sistemi di sviluppo, porterebbe ad un aumento significativo di informazioni da apprendere ed utilizzare.

Un ultimo appunto va fatto a quei linguaggi che possiedono un'implementazione per la piattaforma .NET. Nel nostro esempio, sia Delphi, sia Python, possiedono un apposito compilatore. Ciò risolverà il precedente punto 3), lasciando comunque inalterati i punti 1) 2) e 4), inoltre, come per tutti i linguaggi basati su .NET, il loro utilizzo è limitato al solo AutoCAD (versione 2005 o successiva).

Cos'è e cosa può fare Lisp

Lisp è uno dei linguaggi di programmazione utilizzati per la personalizzazione degli ambienti AutoCAD®, BricsCAD®, IntelliCAD®, progeCAD®, ZwCAD®, NanoCAD®, Draftsight®, ecc...

Lisp accompagna l'ambiente AutoCAD fin dalla sua nascita e permette di controllare, creare e manipolare un disegno, consentendo la realizzazione di sofisticati

automatismi.

Lisp è a disposizione di qualsiasi utente che abbia la necessità di costruire piccole o grandi procedure per automatizzare gli aspetti più ripetitivi del disegno tecnico in ambiente CAD.

Il linguaggio Lisp utilizzato in AutoCAD è una variante dell'originale LISP, linguaggio di programmazione studiato principalmente per l'applicazione nel campo dell'intelligenza artificiale, la caratteristica principale che ha permesso la sua adozione in ambito CAD, è la capacità di trattare le liste dinamiche di dati. Questa capacità consente il trattamento di un disegno tecnico come una grossa lista composta dagli oggetti inseriti (linee, cerchi, archi ecc...), rendendo la loro modifica e creazione molto semplice ed immediata.

In questo testo si vedrà come, utilizzando Lisp, sia molto semplice manipolare un disegno o generarne uno nuovo, scrivendo pochissime righe di codice.

Le versioni di Lisp

Esistono attualmente 2 versioni di Lisp. Quella base, che si trova all'interno di tutti gli AutoCAD (fino alla versione 13) e in tutte le versioni di IntelliCAD, poi quella introdotta dalla versione 14/2000 di AutoCAD denominata "Visual LISP". La base del linguaggio è comune a tutte le versioni di Lisp, con la differenza che, all'interno delle ultime versioni sono state introdotte alcune nuove funzionalità, questo comunque senza eliminare nessuna delle caratteristiche delle versioni precedenti.

Ciò significa che, utilizzando la prima versione, tutti i programmi scritti saranno certamente pienamente compatibili con qualsiasi versione dell'ambiente CAD.

All'interno di questo testo viene spiegato l'utilizzo della prima versione di Lisp, proprio per consentire la portabilità del codice.

Sistemi operativi alternativi

Tutto ciò che è stato detto finora è valido a patto che, la piattaforma che stiamo utilizzando, sia Windows, in una qualsiasi delle sue versioni recenti (Xp, Vista, 7, 8 o successivo).

Nel panorama attuale è possibile però imbattersi in sistemi alternativi, come Os X (Mac) oppure in una delle incarnazioni di Linux, come Ubuntu, Suse, Redhat, Debian, Mint, ecc...

In questo caso, le possibilità rispetto all'ambiente Windows si riducono e variano parecchio, nonostante alcuni aspetti rimangano comuni.

Sicuramente, vi sono alcuni linguaggi che, per la loro natura sono presenti su tutti i

sistemi operativi. In particolare, C/C++ e Lisp sono disponibili su tutte e tre le piattaforme.

Prendendo come esempio AutoCAD, la sua incarnazione per Mac Os X (introdotta con la v.2011) permette lo sviluppo in C/C++ e Lisp, eliminando invece qualsiasi supporto a VBA e .NET, tecnologie Microsoft disponibili solo su Windows.

Stesso discorso può essere fatto per i CAD alternativi ad AutoCAD, che soffrono, più o meno delle stesse limitazioni/differenze.

Quando le necessità lo consente, la realizzazione di procedure automatiche scritte in linguaggio Lisp è sicuramente un buon modo per avere software multi piattaforma, senza un eccessivo dispendio di risorse.

A chi è destinato questo testo

Lavorando nel campo del CAD, ci si rende presto conto del fatto che, tutti o quasi gli utenti, tendono a utilizzare una bassa percentuale delle funzionalità messe a disposizione dallo strumento che hanno a loro disposizione, in questo caso AutoCAD o IntelliCAD. A volte questo accade per mancanza di tempo, di voglia o solamente perché si ritiene che l'utilizzo di un linguaggio di programmazione come Lisp sia difficile da apprendere e richieda una grossa quantità di tempo.

Al contrario penso che, se da un lato è certamente vero che ci voglia una certa quantità di tempo per imparare Lisp, dall'altro ritengo che il suo apprendimento sia abbastanza semplice e consenta all'utente di risparmiare molto tempo automatizzando tutti quelle piccole procedure ripetitive che ogni giorno devono essere eseguite manualmente.

Ritengo che, un utente esperto di CAD, cioè colui che lavora quotidianamente con questo strumento e che ne conosca a fondo le funzionalità (anche solo per ciò che concerne la parte 2D), possa apprendere e rendere produttivo Lisp nel giro di circa 50 ore.

Questo testo vuole essere un primo approccio con Lisp, per consentire all'utente la stesura delle sue prime procedure automatiche, in modo da poter lavorare più velocemente e meglio, dimenticandosi tutti quei passaggi tediosi che ogni giorno "affliggono" l'utente CAD.

Per rendere comprensibile l'argomento a tutte le fasce di utenza, verrà utilizzata una terminologia non sempre tecnica ma, verranno utilizzate semplici espressioni a volte di tipo figurato.

Nel leggere questo testo, si tenga presente che, per sperimentare direttamente il codice dimostrativo è possibile digitare sulla linea di comando le espressioni Lisp, questo permette la verifica immediata di quanto qui esposto.

Capitolo 2
Concetti Base

In questo capitolo verranno fornite le nozioni base per poter affrontare agevolmente l'apprendimento del linguaggio e delle tecniche di programmazione.

Le variabili

In un qualsiasi linguaggio di programmazione, le variabili, sono dei "cassettini" all'interno dei quali, il programmatore, memorizza dei dati. Questi ultimi possono essere di carattere numerico (numeri interi o reali) oppure semplici stringhe (frasi di testo). Ogni variabile potrà poi, essere scritta o letta in qualsiasi momento.

Gli utilizzi possibili sono praticamente infiniti, si pensi ad esempio alla memorizzazione delle risposte che l'utente fornisce a domande poste da un programma, le quali serviranno poi per una elaborazione successiva. Oppure semplicemente si pensi alle variabili per contenere risultati numerici di calcoli complessi.

In Lisp, così come in qualsiasi altro linguaggio, ogni variabile deve possedere un nome che la distingue dalla altre, in modo da poterla utilizzare semplicemente indicandone l'identificativo (nome). Questo nome è una sequenza alfanumerica di caratteri, l'unica limitazione è data dal fatto che non è possibile chiamare una variabile utilizzando solamente delle cifre numeriche, per esempio sono nomi validi:

```
punto
Valore1
1richiesta
PNT1
A
```

al contrario sono identificativi errati:

```
1
26372
```

Per convenzione solitamente è preferibile utilizzare identificativi chiari in base alla funzione svolta, ad esempio una variabile che conterrà il punto di inizio di una linea potrebbe chiamarsi "PuntoIniziale" in modo che sia chiaro casa contiene.

Lisp definisce 4 tipologie differenti per le variabili. Queste differiscono le una dalle altre in base al tipo di dato contenuto. I dati immagazzinabili in una variabile sono:

- Numeri Reali;
- Numeri Interi;
- Stringhe di Testo;
- Liste di valori, cioè variabili contenenti più dati contemporaneamente. Di

questo tipo si parlerà più a fondo in un capitolo successivo.

Oltre a differenziarsi in queste 4 categorie, è possibile dividere le variabili in due grosse famiglie, "Variabili di Sistema" e "Variabili Utente". Le prime sono definite nell'ambiente CAD e vengono utilizzate per settare i suoi parametri o per contenere informazioni di utilizzo generale, si pensi ad esempio alla variabile "DWGNAME" che contiene il nome del disegno corrente, tutte le "Variabili di Sistema" sono elencate all'interno dell'help in linea di AutoCAD. Al contrario le seconde sono tutte quelle variabili create e gestite interamente da un programma sviluppato dall'utente.

Variabili di sistema

Come indicato nella sezione precedente questo tipo di variabile è definita dall'ambiente CAD. Alcuni esempi sono:

- LASTPOINT, contiene l'ultimo punto indicato dall'utente;
- OSMODE, contiene il tipo di snap ad oggetto attivo;
- SCREENSIZE, contiene la dimensione della finestra di disegno corrente;
- GRIDMODE, permette di conoscere lo stato della griglia, attiva o spenta;
- DWGPREFIX, contiene il percorso del disegno corrente;
- TEXTSTYLE, contiene il nome dello stile di testo attivo.

Di queste alcune si possono sia scrive che leggere, altre sono in sola lettura. L'elenco completo è possibile trovarlo nell'help in linea dell'ambiente CAD.

Per poter leggere una variabile di sistema all'interno di un programma Lisp si utilizza una sintassi del genere:

```
(getvar "DWGNAME")
```

questa riga legge la variabile DWGNAME e ne restituisce il valore. Per sperimentare quanto detto è sufficiente digitare la riga seguente sulla linea di comando di AutoCAD, si otterrà un risultato del genere:

```
Comando: (getvar "DWGNAME")
"Drawing1.dwg"
```

il valore visualizzato (restituito) sotto la riga di comando indica il nome del file di disegno corrente.

La forma di scrittura utilizzata sopra, per ricavare il valore di una variabile di sistema, è comune a tutte le istruzioni Lisp. Infatti ogni istruzione è formata da:

```
([comando] [argomento] ...)
```

prima di tutto una parentesi tonda iniziale, quindi il nome del comando (che chiameremo d'ora in poi funzione) da eseguire, uno o più argomenti, ed infine una parentesi tonda chiusa. Nella lettura di una variabile è stata utilizzata la funzione "getvar" alla quale è stato indicato come primo argomento il nome della variabile di cui restituire il valore.

Variabili utente

Questo tipo di variabile viene creata, modificata ed eliminata esclusivamente all'interno di un programma o di un'istruzione Lisp. E' possibile creare un numero indefinito di variabili utente, le quali possono contenere uno qualsiasi dei tipi di dato considerati in precedenza.

All'interno di un programma Lisp è possibile leggere il valore di una variabile semplicemente indicando il suo nome, senza la necessità di utilizzare la funzione "getvar", indispensabile per la lettura delle "variabili di sistema".

Settaggio delle variabili

Per capire meglio la funzione e l'utilizzo delle variabili utente esaminiamo il seguente codice:

```
(setq Lunghezza 23)
```

questa istruzione si limita ad inserire nella variabile "Lunghezza" il numero intero 23. Con questa istruzione si ottiene un duplice effetto, si crea la variabile e le si assegna un valore. Bisogna tener presente che le variabili vengono automaticamente create durante la loro prima assegnazione.

Si noti che in questo caso si è utilizzata la funzione "setq" che permette l'inserimento di un valore all'interno di una qualsiasi "variabile utente".

Ovviamente è anche possibile assegnare il valore contenuto in una variabile esistente ad un'altra semplicemente utilizzando la seguente sintassi:

```
(setq Destinazione Sorgente)
```

in questo caso, in "Destinazione" verrà inserito il valore già presente in "Sorgente", naturalmente sia "Destinazione" che "Sorgente" sono delle "variabili utente".

Con la stessa metodologia è possibile inserire all'interno di una variabile numeri reali e stringhe come nell'esempio di seguito:

```
(setq valoreS "Questa è una stringa")
```

```
(setq valoreR 12.34)
```

la prima istruzione assegna una stringa alla variabile "valoreS", mentre la seconda istruzione esegue la medesima operazione con un numero reale. Si tenga presente che il separatore dei decimali all'interno di Lisp è sempre e soltanto il punto (.).

Molto importante è sapere che, se una variabile non viene assegnata, il suo valore è pari a "nil". Questo particolare valore indica un dato nullo, cioè mai assegnato, inoltre è possibile settare le variabili a "nil" quando si vuole eliminare il loro contenuto e liberare la memoria occupata dalle stesse. Infatti impostando una qualsiasi variabile a "nil" lo spazio di memoria occupato dai suoi dati viene reso disponibile per altre informazioni. Ad esempio per eliminare "valoreS":

```
(setq valoreS nil)
```

Le funzioni e il loro utilizzo

Una funzione è un programma che può essere utilizzato più di una volta semplicemente indicandone il nome, inoltre una funzione può restituire un valore al programma dal quale viene eseguita.

Ad esempio:

```
(setq valoreS "Stringa")
```

qui si utilizza la funzione "setq", la quale, inserisce un dato valore all'interno della variabile specificata (nel nostro caso "valoreS").

In Lisp qualsiasi operazione che l'utente compie avviene con l'uso di funzioni. Anche la semplice somma di due numeri utilizza una funzione:

```
(+ 3 5)
```

questa espressione restituisce la somma di 3 più 5, tramite la funzione "+". Ovviamente esistono anche le altre tre operazioni, la sottrazione "-", la divisione "/", e la moltiplicazione "*". Ovviamente con questi operatori è possibile utilizzare, oltre che numeri, anche variabili (contenenti valori numerici), si consideri questo piccolo esempio che dimostra l'uso degli operatori e delle variabili:

```
(setq valore1 3)
(setq valore3 5.6)
(setq risultato (* valore1 valore3))
```

in questo esempio vengono inseriti due valori numerici in altrettante variabili, infine viene assegnato ad una terza variabile il risultato del prodotto delle prime due.

In Lisp esistono moltissime funzioni, parte delle quali esaminate in questo testo. Se si desidera un elenco completo si può fare riferimento all'help in linea di AutoCAD 14

oppure 15, i quali contengono un ottimo elenco con la spiegazione sull'utilizzo di tutte le funzioni disponibili.

La funzione Command

Questa è una delle più importanti funzioni, il suo scopo è quello di eseguire i normali comandi del CAD permettendo di rispondere automaticamente alle domande che il comando stesso pone.

AutoCAD e i CAD suoi simili, consentono la digitazione dei comandi e dei dati da tastiera. Ad esempio, se volessimo disegnare una semplice linea tra due punti, potremo inserire, sulla linea di comando (in basso) :

```
Command: _LINE
Start of line: 0,0
Angle/Length/<End point>: 10,10
Angle/Length/Follow/Undo/<End point>:
Command:
```

Prima di tutto è stato inserito il comando la lanciare "_LINE", digitato in lingua inglese e preceduto da un "_", in modo che venga riconosciuto da tutti i CAD indipendetemente dalla lingua di quest'ultimo. Infatti, se con un CAD in lingua italiano possiamo certamente utilizzare il comando "LINEA", questo è riconosciuto solo su un CAD nella stessa lingua, invece, usando la sua forma inglese preceduta da un "_", il comando sarà riconosciuto indipendentemente dalla lingua del software.

Dopo aver specificato il comando e premuto invio, il software ci chiederà il punto di inizio della linea, e noi andremo a digitare le sue coordinate, nel nostro esempio "0,0". Stessa cosa quando ci verrà chiesto il secondo punto, in questo caso "10,10". Dopo aver digitato questi dati, il comando linea continuerà a chiederci di specificare coordinate finché non risponderemo con un semplice invio, cosa che terminerà la procedura riportandoci nello stato iniziale, dove il CAD attendo un nuovo comando.

Se volessimo trasformare questi inserimenti, in una singola istruzione Lisp, da inserire in un nostro script, ci basterà passare i dati che normalmente andremo a specificare manualmente, alla funzione "command". Ad esempio, per riprodurre il disegno della linea tra due punti (dal punto 0,0 al punto 10,10) potremmo scrivere:

```
(command "_line" "0,0" "10,10" "")
```

questa riga esegue il comando linea, indica come primo punto 0,0, quindi 10,10, infine per terminare la linea viene indicata una stringa vuota "" che corrisponde alla pressione del tasto Invio sulla tastiera. Come si può facilmente notare ogni parte del comando è racchiusa tra doppi apici e separata dalle altre da uno spazio, in realtà non esiste un numero limite di comandi consecutivi, quindi è possibile realizzare anche procedure piuttosto complesse.

Nella pratica, la funzione “command” replica, in modo automatico, le operazioni che normalmente faremmo manualmente.

Con la funzione "Command" è possibile utilizzare il parametro speciale "PAUSE". Questa istruzione ferma l’esecuzione di "Command" attendendo l’input dell’utente. Si esamini la seguente riga:

```
(command "_line" PAUSE "10,10" "")
```

in questo caso il primo punto che definisce la linea viene richiesto all’utente e non indicato direttamente. Per comprendere meglio, digitare la precedente riga sulla linea di comando di AutoCAD ed osservare ciò che accade.

Con "Command" è possibile poi utilizzare le variabili. Queste ultime possono essere utilizzate per passare alla funzione punti, comandi ecc...

Nell’esempio che segue vengono utilizzate due variabili per indicare a "Command" i punti di inizio e fine della linea:

```
(setq p1 "0,0")
(setq p2 "10,10")
(command "_line" p1 p2 "")
```

come si vede è sufficiente indicare le variabili come parametri di "Command" perché quest’ultimo li utilizzi come dati, riconoscendo il loro contenuto, che in questo caso è di tipo stringa e rappresenta dei punti espressi come x,y.

N.B.: negli esempi precedenti è stato utilizzato il comando AutoCAD "_LINE", se si utilizzasse una versione italiana dell’ambiente CAD questo potrebbe essere sostituito dal comando "LINEA". La scelta fatta in questo testo di utilizzare comandi inglesi è dettata dal fatto che, tutti i comandi digitati in lingua inglese e preceduti da un segno "_" sono comandi internazionali che funzionano su qualsiasi versione nazionalizzata di AutoCAD, ciò consente l’utilizzo delle stesse istruzioni in qualunque AutoCAD a partire dalla release 12.

Scrivere, caricare ed eseguire un programma

Fino ad ora ci siamo limitati a utilizzare Lisp per scrivere semplici linee di codice direttamente sulla linea di comando di AutoCAD, questo certo non è il modo migliore per utilizzare Lisp. Ora vedremo come si può scrivere un programma LISP con più linee di codice, come è possibile caricarlo nell’ambiente CAD e come viene eseguito.

Per prima cosa occorre sapere che tutto ciò che serve per scrivere un programma Lisp è un semplice editor di testo, va benissimo anche Notepad (Blocco Note) fornito con Windows. Infatti un programma non è altro che una serie di linee di codice inserite in un semplice file di testo, l’unica accortezza è quella di salvare questo file con estensione LSP anziché TXT.

Facciamo subito un esempio, ipotizzando di lavorare in ambiente Windows apriamo Notepad e digitiamo il seguente codice:

```
(setq p1 "0,0")
(setq p2 "10,0")
(command "_line" p1 p2 PAUSE p1 "")
```

questo piccolo programmino, formato solo da 3 linee disegna un triangolo chiedendo all'utente il punto che identifica uno degli angoli. Dopo averlo salvato in una cartella qualsiasi, ricordandosi di assegnargli l'estensione LSP (es.: TEST.LSP), siamo pronti per caricarlo all'interno di AutoCAD.

Per compiere questa operazione vi sono due strade. La prima prevede l'utilizzo del comando _APPLOAD, che possiamo digitare direttamente da tastiera sulla linea di comando. Questa procedura ci chiede il file da caricare e quindi ne esegue il caricamento. La seconda soluzione è quella di utilizzare la funzione Lisp "load". Questa istruzione esegue la stessa operazione di _APPLOAD senza però fare alcuna richiesta tramite finestre, infatti per utilizzare "load" è possibile scrivere:

```
(load "c:/cad/test.lsp")
```

in questo caso verrà caricato il file "test.lsp" posizionato in "c:\cad". Notare, che nello specificare il percorso, è necessario utilizzare la barra semplice "/" al posto della consueta rovesciata "\", questo poiché l'AutoCAD utilizza quest'ultima per indicare l'inizio un carattere speciale (che vedremo più avanti nel testo) e non il separatore di percorso. Se comunque si desidera utilizzare questo carattere è possibile sostituire a "/" la coppia "\\", ciò significa che l'istruzione precedente diventerebbe:

```
(load "c:\\cad\\test.lsp")
```

questa regola per l'indicazione dei percorsi è valida non solo in questo caso ma, con tutte le funzioni che richiedono l'indicazione di un percorso.

Utilizzando "load" è possibile omettere il percorso nel quale risiede il file. Ad esempio

```
(load "miofile.lsp")
```

In questo caso, il file specificato verrà cercato all'interno della cartella corrente e in tutte le cartelle presenti nei 'Percorsi di ricerca dei file di supporto' (in AutoCAD), oppure nei percorsi 'Percorsi Utente' (in progeCAD/IntelliCAD).

Addirittura è possibile omettere l'estensione (.lsp):

```
(load "miofile")
```

Così facendo il CAD cercherà nei suddetti percorsi il file con estensione ".lsp". Nel caso di AutoCAD, quest'ultimo comportamento è leggermente differente, infatti, prima vengono ricercati i file con estensione ".vlx", poi i file ".fas" e infine i file ".lsp".

Sia _APPLOAD che "load" caricano i file LSP, inoltre dopo essere stati caricati,

vengono automaticamente eseguiti da AutoCAD, quindi, ogni qual volta volessimo eseguire un file LISP come quello sopra descritto, non dovremo fare altro che caricarlo.

Scrivere messaggi per l'utente

Fino ad ora ci siamo limitati ad utilizzare istruzioni senza fornire alcun messaggio all'utente. Normalmente però, la comunicazione con l'operatore è di fondamentale importanza per la riuscita di un buon software.

Esistono, in AutoCAD, due modi per mandare messaggi all'utente. Scrivere sulla linea di comando oppure far comparire una finestra contenente il testo che l'utente deve leggere. Questi due sistemi sono supportati da Lisp tramite alcune semplici funzioni che analizzeremo in questa sezione.

La visualizzazione di messaggi tramite linea di comando è disponibile per mezzo di quattro funzioni:

- "prin1", che stampa una stringa;
- "princ", identica a "prin1" con la differenza che qui è possibile utilizzare i caratteri di controllo (esaminati più avanti);
- "print", identica a "prin1" con la differenza che viene inserito un ritorno a capo prima del messaggio visualizzato ed uno spazio subito dopo;
- "prompt", stampa un testo sulla linea di comando, riconoscendo i caratteri di controllo.

La sintassi da utilizzare per le rispettive funzioni è sempre la stessa:

```
([funzione] [stringa])
```

facendo alcuni esempi, è possibile scrivere la frase "Procedura di disegno triangoli", sulla linea di comando, con le seguenti espressioni:

```
(prin1 "Procedura di disegno triangoli")
(princ "Procedura di disegno triangoli")
(print "Procedura di disegno triangoli")
(prompt "Procedura di disegno triangoli")
```

ogni riga avrà un comportamento diverso a seconda di quanto detto sopra.

Nell'esporre le differenze tra le 4 funzioni si è parlato dei "caratteri di controllo", questi sono dei particolari caratteri che, quando utilizzati, non vengono stampati ma interpretati in maniera diversa, di seguito vengono indicati quelli disponibili:

\\ Carattere \

*	Carattere *
\e	Carattere Escape (ESC)
\n	Carattere di riga nuova
\r	Carattere di ritorno a capo
\t	Carattere di tabulazione
\nnn	Carattere il cui codice ottale è nnn
\U+XXXX	Sequenza Unicode
\M+NXXXX	Sequenza Unicode

si noti il carattere "\\" già utilizzato in precedenza. Un esempio di utilizzo può essere il seguente:

```
(princ "\nEssere o non essere\nQuesto è il problema.")
```

in questo caso verranno inseriti dei ritorni a capo prima della frase e a metà, quindi verranno stampate a video due righe di testo.

Altra differenza tra le funzioni esaminata sta nel fatto che "prin1", "princ" e "print" accettano facoltativamente anche un secondo parametro subito dopo la frase da stampare, in questa forma:

```
([funzione] [stringa] [file])
```

[file] non è altro che l'identificatore di un file aperto nel quale verranno inserite le stringhe stampate. Questo sistema, che verrà esaminato nel dettaglio nel capitolo relativo alla modifica dei file, è la via più semplice per scrivere file di testo.

Tutte le funzioni indicate, così come qualsiasi altra, dopo aver visualizzato il messaggio restituiscono esattamente il contenuto di ciò che è stato stampato. Quindi la riga:

```
(print "ciao")
```

stampa la parola "ciao" e restituisce "ciao" come valore di ritorno all'eventuale funzione chiamante.

Per comprendere meglio si esamini il codice seguente:

```
(setq linea (princ "ciao"))
(print linea)
```

in questo caso viene stampata una prima volta "ciao" e quindi il suo valore viene restituito alla funzione "setq" la quale, a sua volta, lo assegnerà alla variabile "linea". Dopo di che "linea" viene stampata da "print".

Quello visto finora è il primo, dei due sistemi, per comunicare con l'utente. Il secondo metodo consiste nell'utilizzo di una funzione che permette la creazione di piccole

finestre di messaggio:

```
(alert [messaggio])
```

se ad esempio utilizzassimo la riga:

```
(alert "Messaggio di prova")
```

comparirebbe a video una piccola finestra di dialogo dotata di un pulsante per chiuderla con, al suo interno, il messaggio indicato.

Nel massaggio indicato è possibile utilizzare i "caratteri di controllo" per ottenere una formattazione migliore del testo.

Quelli visti sono i sistemi di messaggistiche più immediati e semplici. In realtà AutoCAD mette a disposizione anche un terzo sistema per comunicare con l'utente, le DCL. Queste non sono altro che finestre componibili di dialogo interamente realizzate e controllate dal programma Lisp.

Le funzioni matematiche

In questa sezione esamineremo la parte di Lisp che permette la gestione dei numeri, siano questi interi o reali.

Di seguito si ha lista delle funzioni disponibili per l'elaborazione numerica:

```
+ (addizione)
- (sottrazione)
* (moltiplicazione)
/ (divisione)
~ (NOT a livello bit)
1+(incremento unitario)
1-(decremento unitario)
abs
atan
cos
exp
expt
fix
float
gcd
log
logand
logior
```

```
lsh
max
min
minusp
rem
sin
sqrt
zerop
numberp
```

Se ad esempio si volesse eseguire un'addizione tra i numero 10 e 7.5 sarebbe sufficiente scrivere:

```
(+ 10 7.5)
```

in questo modo la funzione "+" restituisce il dato 17.5.

La sintassi utilizzata:

```
([funzione] [valore1] [valore2])
```

è applicabile solamente alla quattro operazioni "+", "-", "*", "/". In tutti gli altri casi ciò non è corretto.

Di seguito vengono esaminate una ad una le funzioni di trattamento numerico in modo da poterne comprendere il funzionamento.

- "~", questa funzione esegue l'operazione di negazione logica (NOT) a livello di bit su un numero, ad es:

  ```
  (~ 3) restituisce -4
  ```

- "1+" e "1-" funzionano in maniera simile, salvo che il primo incrementa un numero di un valore 1 mentre il secondo lo decrementa, per esempio:

  ```
  (1+ 5) restituisce 6
  (1- 5) restituisce 4
  ```

- "abs" si limita a restituire il valore assoluto di un numero, ad esempio:

  ```
  (abs 100) restituisce 100
  (abs -100) restituisce 100
  ```

- "atan" ritorna l'arcotangente del valore specificato come primo parametro in radianti, se si specifica anche un secondo parametro, viene ritornato l'arcotangente di valore1/valore2, dove valore1 e valore2 corrispondono al primo e al secondo argomento specificato, come ad esempio:

```
(atan 0.5) restituisce 0.463648
(atan 2.0 3.0) restituisce 0.588003
```

- "cos" calcola il coseno, in radianti, di un numero come nell'esempio:

```
(cos 0.0) restituisce 1.0
```

- "exp" restituisce la costante e elevata alla potenza indicata come parametro:

```
(exp 1.0) restituisce 2.71828
```

- "expt" ritorna un numero elevato ad una determinata potenza, se i due parametri indicati sono entrambi interi verrà restituito un dato intero altrimenti un numero reale:

```
(expt 2 4) ritorna 16
```

- "fix" restituisce la parte intera di un numero:

```
(fix 4.8) ritorna 4
(fix 3) ritorna 3
```

- "float" converte un numero in valore reale:

```
(float 3) ritorna 3.0
```

- "gcd" restituisce il massimo comune denominatore di due numeri interi:

```
(gcd 10 35) ritorna 5
```

- "log" calcola il logaritmo naturale di un numero:

```
(log 35) ritorna 3.55535
```

- "logand" ritorna il risultato di un AND logico eseguito sui valori passati come argomenti. Il numero di argomenti utilizzabili è libero, ad esempio:

```
(logand 3 35 7) ritorna 3
(logand 12 45) ritorna 12
(logand 3 8 13 47 89) ritorna 0
```

- "logior" esegue un operatore logico a livello bit OR inclusivo su una serie di numeri passati come argomenti. Anche in questo caso il numero di argomenti è libero:

```
(logior 3 35 7) ritorna 39
(logior 12 45) ritorna 45
(logior 3 8 13 47 89) ritorna 127
```

- "lsh" ritorna lo spostamento logico di un numero intero (primo parametro) per un certo numero di bit (secondo parametro), per esempio:

```
(lsh 2 1) ritorna 4
(lsh 40 2) ritorna 160
```

- "max" e "min" ritornano rispettivamente il valore massimo e minimo scelti tra i numeri passati come argomenti. Il numero di argomenti, anche in questo caso, è libero. Per esempio:

```
(max 1 3 6 9 2 18) ritorna 18
(min 3 87 64 1 82 9) ritorna 1
```

- "minusp" restituisce "T" se il valore specificato come argomento è negativo in caso contrario ritorna "nil", ad esempio:

```
(minusp -3) ritorna T
(minusp 34) ritorna nil
```

- "numberp" restituisce "T" se il valore specificato come argomento è un numero reale o intero valido, in caso contrario ritorna "nil", ad esempio:

```
(numberp -3) ritorna T
(numberp "test") ritorna nil
```

- "rem" ritorna il resto della divisione effettuata tra il primo argomento ed il secondo. Se si specificano più di due argomenti , "rem" divide il primo per il secondo, il risultato della divisione viene poi diviso per il terzo numero, e così via. Ad esempio:

```
(rem 3 5) ritorna 3
(rem 3 5 8.0) ritorna 3
(rem 12 4 8 22) ritorna 0
```

- "sin" ritorna il seno di un angolo espresso in radianti:

```
(sin 0) ritorna 0
(sin 1) ritorna 0.841471
```

- "sqrt" ritorna la radice quadrata di un numero:

```
(sqrt 4) ritorna 2.0
```

- “zerop" verifica che l’argomento passato sia uguale a 0, in questo caso ritorna "T", altrimenti "nil":

```
(zerop 0) ritorna T
```

```
(zerop 0.000) ritorna T
(zerop 0.1) ritorna nil
```

Le stringhe

Abbiamo visto nei paragrafi precedenti che una stringa non è altro che un gruppo di caratteri racchiusi tra doppi apici, come per esempio "Indicare un punto". Inoltre si è visto che il carattere \ utilizzato all'interno di una stringa indica la presenza di un "carattere di controllo".

Tutte le stringhe simili a quelle usate finora, cioè stringhe con una lunghezza costante vengono dette stringhe letterali o costanti e la loro lunghezza massima è pari a 132 caratteri.

In Lisp esistono diverse funzioni che permettono la manipolazione delle stringhe. Di seguito vengono indicate le più importanti:

"strcase", permette di convertire tutte le lettere facenti parte di una stringa in maiuscolo o minuscolo. Questa funzione richiede due parametri, il primo è la stringa da convertire, il secondo vale T se il risultato deve essere minuscolo altrimenti verrà generata una stringa completamente maiuscola. Ad esempio:

```
(strcase "Test") ritorna "TEST"
(strcase "Test" T) ritorna "test"
```

- "strcat" consente di unire più stringhe in una sola. Questa funzione permette di specificare N parametri, che verranno concatenati per generare la stringa risultante. Ad esempio:

  ```
  (strcat "pro" "va") ritorna "prova"
  (strcat "Test" " Uni" "one") ritorna "Test Unione"
  ```

- "strlen" ritorna la lunghezza della stringa specificata come parametro. Se vengono specificati più parametri "strlen" ritorna la somma delle lunghezza. Ad esempio:

  ```
  (strlen "test") ritorna 4
  (strlen "prova strlen") ritorna 12
  (strlen "prova" "strlen") ritorna 11
  ```

- "substr" permette di estrarre un parziale di stringa, specificando come primo parametro la stringa da cui estrarre il parziale, la posizione del primo carattere da estrarre e infine la lunghezza del parziale. Se non viene specificato il terzo parametro verranno restituiti tutti i caratteri a partire da quello specificato. Si tenga inoltre presente che il primo carattere della stringa è sempre il numero

1. Ad esempio:

```
(substr "Stringa Test" 2 3) ritorna "tri"
(substr "1234" 1 2) ritorna "12"
(substr "Stringa Test" 4 1) ritorna "i"
(substr "Prova" 2) ritorna "rova"
(substr "Test" 5) ritorna ""
```

Oltre alle funzioni viste esiste anche "wcmatch" che permette di controllare la presenza, all'interno di una stringa di un particolare modello. Questa funzione ha una complessità tale da non permettere il suo inserimento in questo testo che vuole essere soltanto un primo approccio con Lisp, per questa ragione una spiegazione approfondita di "wcmatch" può essere reperita nella documentazione in linea del software CAD utilizzato.

In aggiunta a quanto visto, Lisp dispone anche di una serie di funzioni per la conversione di valori numerici in stringhe e viceversa. Tali funzioni risultano molto utili in tutte quelle situazioni in cui è necessario utilizzare un tipo di dato diverso da quello che si ha al momento, si pensi per esempio a "command" che richiede come parametri delle stringhe.

La lista delle maggiori funzioni di conversione tra numeri e stringhe è riportata di seguito:

- "angtof" converte una stringa contenente un angolo espresso in radianti in un numero reale. Es.:

  ```
  (angtof "23.8") ritorna 0.415388
  ```

- "angtos" converte un valore angolare espresso in radianti in stringa. Es.:

  ```
  (angtos 0.415388) ritorna 24
  ```

- "ascii" ritorna il codice ASCII del primo carattere della stringa passata come argomento. Es.:

  ```
  (ascii "a") ritorna 97
  ```

- "atof" converte una stringa in un numero reale. Es.:

  ```
  (atof "123.2") ritorna 123.2
  ```

- "atoi" converte una stringa in un numero intero. Se come parametro viene passata una stringa contenente un numero reale, "atoi" prende solamente la parte intera. Es.:

  ```
  (atoi "23") ritorna 23
  (atoi "12.5") ritorna 12
  ```

- "chr" ritorna il carattere corrispondente al numero di codice ASCII inserito come parametro. Es.:

  ```
  (chr 75) ritorna "K"
  ```

- "distof" converte una stringa in un valore reale. Questa funzione richiede un secondo argomento che rappresente le unità in cui viene convertita la stringa passata come primo parametro. Le unità permesse sono:
  Modalità 1 -> Scientifica
  Modalità 2 -> Decimale
  Modalità 3 -> Ingegneristica (piedi e pollici decimali)
  Modalità 4 -> Architettonica (piedi e pollici frazionari)
  Modalità 5 -> Frazionario
  Es.:

  ```
  (distof "23.3" 1) ritorna 23.3
  ```

- "itoa" converte un valore intero in una stringa. Es.:

  ```
  (itoa 3) ritorna "3"
  ```

- "rtos" converte un numero in una stringa permettendo di specificare la modalità (secondo parametro) e la precisione (terzo parametro) della conversione. I valori validi per la modalità sono gli stessi validi per "distof", mentre la precisione è un numero intero. Es.:

  ```
  (rtos 17.5 1 2) ritorna 1.75E+01
  ```

Esempio riassuntivo

In questa sezione viene esaminato un piccolo programma che riassume brevemente i concetti esposti nel capitolo.

La funzione del programma esposto è quella di, dato un disegno vuoto, disegnare la squadratura di un foglio A4 verticale inserendo nell'angolo sinistro inferiore il nome del disegno completo di percorso.

Codice programma:

```
(alert "Procedura squadratura foglio A4")
(setvar "TEXTSTYLE" "standard")
(setq larghezza 210)
(setq altezza 297)
(setq punto (strcat (itoa larghezza) "," (itoa altezza)))
(command "_rectang" "0,0" punto)
```

```
(setq larghezza (- larghezza 5))
(setq altezza (- altezza 5))
(setq punto (strcat (itoa larghezza) "," (itoa altezza)))
(command "_rectang" "5,5" punto)
(setq NomeDisegno (getvar "dwgname"))
(setq Percorso (getvar "dwgprefix"))
(setq NomeCompleto (strcat Percorso NomeDisegno))
(command "_text" "10,10" "2.5" "" NomeCompleto "" "")
(alert "Procedura conclusa con successo.")
```

Il sorgente esposto imposta le misure del foglio inserendole in due variabili, compone il punto convertendo i valori numerici in stringhe, disegna il rettangolo esterno della squadratura del foglio, quindi calcola le dimensioni della squadratura interna distante 5 mm da quella esterna e la disegno, infine, estraendo il nome del disegno e il suo percorso da due variabili di sistema disegna un testo nell'angolo sinistro inferiore.

Per utilizzare il codice esposto:

- Aprire un editor di testo (es.: "blocco note");
- Digitare il codice del programma esattamente come sopra riportato;
- Salvare il file assegnandogli estensione .LSP (es.: test.lsp);
- Aprire AutoCAD/progeCAD/IntelliCAD;
- Digitare il comando "_APPLOAD";
- Selezionare il file salvato (es.: test.lsp) e quindi premere il tasto per il suo caricamento.

Capitolo 3
Gestire un programma

Questo capitolo introdurrà alla gestione del flusso dei programmi, assolutamente indispensabile per poter realizzare qualsiasi tipologia di software.

Le espressioni di confronto

Le espressioni di confronto utilizzano le seguenti funzioni di base:

- "=" valuta se due o più valori, passati come argomenti, sono uguali, in questo caso ritorna T altrimenti nil. Es.:

```
(= 3 4) ritorna nil
(= "test" "test") ritorna T
(= 1 3 4) ritorna nil
(= "a" "a" "a") ritorna T
```

- "/=" valuta se due o più valori sono diversi, in questo caso ritorna T altrimenti nil. Es.:

```
(/= 3 4) ritorna T
(/= "test" "test") ritorna nil
(/= 1 3 4) ritorna T
(/= "a" "a" "a") ritorna nil
```

- "<" e ">" valutano se un valore è minore o maggiore dell'elemento alla sua destra, in questo caso viene restituito T, altrimenti nil. Es.:

```
(< 10 5) ritorna nil
(> "test" "a") ritorna T
```

E' possibile poi combinare le espressioni di confronto utilizzando gli operatori logici AND, OR e NOT.

- "AND" ritorna T se tutti i suoi argomenti sono T (veri):

```
(AND (= 3 5) (= 3 3)) ritorna nil
(AND (= 1 1) (= 1 1)) ritorna T
```

- "OR" ritorna T se uno o più argomenti sono veri (T):

```
(OR (= 3 5) (= 3 3)) ritorna T
(OR (= 0 1) (= 1 3)) ritorna nil
```

- "NOT" ritorna T se il suo argomento è nil (falso) e nil se il suo argomento è T (vero):

```
(NOT T) ritorna nil
(NOT nil) ritorna T
(NOT 10) ritorna nil
```

- "NULL" ritorna T se il suo argomento è nil (falso) e nil se il suo argomento è T (vero):

```
(NULL T) ritorna nil
(NULL nil) ritorna T
(NULL 10) ritorna nil
```

Apparentemente, non esiste differenza tra le funzioni NOT e NULL. Normalmente però, NULL viene utilizzata con liste ed elenchi in genere, mentre NOT con tutti gli altri tipi di dato.

Le strutture di controllo If e Cond

Le funzioni "If" e "Cond" permettono di eseguire parti di codice sole se accade una determinata condizione. Tale condizione viene stabilita da espressioni di confronto tra elementi.

Analizziamo ora nel dettaglio la struttura del costrutto "if":

```
(if [condizione]
      (progn
      [codice eseguito se condizione = vero]
      )
      (progn
      [codice eseguito se condizione = falso]
      )
)
```

Importante è notare che il codice eseguito deve essere racchiuso all'interno di un costrutto PROGN che permette l'esecuzione di più linee di codice. Se non si utilizzasse PROGN, verrebbe eseguita una sola riga di codice per ognuno dei due stati.

Esaminiamo ora un piccolo esempio per l'utilizzo di quanto esposto per il costrutto "if":

```
(if (= a 3)
      (progn
                  (setq b 3)
      )
      (progn
                  (setq b 0)
```

```
      )
)
```

In questo esempio se inizialmente la variabile ’a’ valesse 3 verrebbe settata ’b’ al valore 3, in caso contrario ’b’ verrebbe impostata a 0. In questo caso le istruzioni PROGN risultano superflue in quanto, ognuna delle due possibili parti di codice è composta da una sola funzione, si potrebbe quindi riscrivere il tutto come:

```
(if (= a 3)
      (setq b 3)
      (setq b 0)
)
```

La seconda, importante funzione di scelta è "cond". Essa permette di valutare una o più espressioni eseguendo il codice associato alla prima condizione risultante vera. Si potrebbe considerare "cond" come un if multiplo, dove vengono valutate n espressioni anziché una soltanto.

Ecco di seguito un esempio sulla struttura del costrutto "cond":

```
(cond
      ( [condizione]
      [codice eseguito se condizione = vero]
      )
      ( [condizione2]
      [codice eseguito se condizione2 = vero]
      )

      (T
      [codice eseguito se nessuna condizione risulta vera]
      )
)
```

In questo costrutto si possono aggiungere un numero qualsiasi di condizioni che, se vere, eseguono un determinato codice. Se però nessuna condizione risultasse verificata verrebbe automaticamente eseguito il codice appartenente alla condizione T, ovviamente la presenza di quest’ultima è facoltativa.

Facciamo ora un esempio:

```
(cond
      ((= a 1)
            (setq b 1)
      )
      ((= a 2)
```

```
            (setq b 2)
      )
      (T
            (setq b 0)
      )
)
```

Questo spezzone di codice prevede 2 diversi confronti ed inoltre viene utilizzato un default (T) nel caso nessuna condizione risulti verificata.

I Cicli While, Foreach e Repeat

In molti casi, durante la programmazione si ha la necessità di eseguire ripetutamente una parte di codice finché non si verificano determinate condizioni. Per soddisfare questa ricorrente esigenza Lisp mette a disposizione tre strutture sintattiche "While", "Foreach" e "Repeat".

Dei tre il più utilizzato è il costrutto "While" che permette l'esecuzione di una parte di programma finché si verifica una certa condizione. La sua struttura è la seguente:

```
(while [condizione]
      [codice da eseguire]
)
```

Il codice da eseguire viene ripetuto finché la condizione risulta vera, in caso contrario il programma termina il ciclo e prosegue il programma.

Si osservi ora l'esempio seguente:

```
(setq a 25)
(while (/= a 20)
      (setq a (1- a))
)
```

Questo codice esegue le righe presenti all'interno del ciclo per 5 volte, cioè finché la condizione dello stesso risulta vera.

Quello vista ora è l'utilizzo classico della funzione "while", che prevede di specificare una condizione ed un codice, che viene eseguito finché la condizione risulta Vera. Per ribadire il concetto facciamo un ulteriore esempio:

```
(setq x 1)
(while (< x 100)
      (print x)
      (setq x (1+ x))
```

```
)
```

In questo caso la condizione prevede che il codice venga eseguito finché “x” sia minore di 100.

“While” può presentare un piccolo inconveniente. Non è possibile eseguire il codice interno se il valore iniziale di x è uguale o maggiore di 100. Al contrario, in altri linguaggi, come ad esempio Visual Basic o PHP, è possibile utilizzare cicli il cui codice viene eseguito sempre, almeno una volta. Per esempio, in Visual Basic esiste il costrutto do..loop e in PHP troviamo “do..while”. Questi cicli, simili al ’while’ del Lisp, differiscono per il fatto che il valore dell’espressione viene controllato alla fine di ogni iterazione anziché all’inizio, ciò significa che la prima iterazione di un blocco “do..while” (PHP) o do..loop (VB) verrà sempre eseguita.

In Lisp, questa possibilità non è preclusa, ma semplicemente è nascosta. Infatti se osserviamo la definizione della funzione ’while’ sul manuale ufficiale:

```
(while testexpr [expr ...])
```

Ci accorgeremo del fatto che solo la condizione “testexpr” è obbligatoria (infatti tutto ciò che è racchiuso tra [ ] è facoltativo). Partendo da questa osservazione, e utilizzando la funzione “progn”, la quale permette di raggruppare più istruzioni come se si trattasse di una soltanto, possiamo realizzare un ciclo che esegua del codice, sempre, almeno una volta:

```
(setq x 100)
(while
      (progn
            (print x)
            (setq x (1+ x))
            (if (< x 100) T nil)
      )
)
```

In questo caso, anche se x vale 100, il codice interno viene eseguito una volta, prima di terminare il ’while’. Il ciclo si conclude poiché, l’ultima istruzione presente all’interno del “progn”, risulta essere quella che in realtà determina il valore restituito dalla condizione al costrutto ’while’.

Nell’osservare l’esempio precedente, si tenga presente che ’progn’ può racchiudere al suo interno un numero qualsiasi di altre istruzione, restituendo l’ultimo valore ritornato dall’ultima espressione in esso contenuto. Nel nostra caso l’ultima istruzione è un “if” che, a seconda della sua condizione ritorna “t” (vero) oppure “nil” (falso).

Lasciando per ora “while”, se si volesse eseguire una parte di codice un numero preciso di volte, senza dover porre condizioni precise, Lisp fornisce “Repeat”.

```
(repeat [numero di iterazioni]
```

```
      [codice da eseguire]
)
```

In questo caso il codice verrà eseguito un numero di volte fisso pari al numero di iterazioni.

Si osservi questo esempio:

```
(setq Nvolte 5)
(repeat Nvolte
      (print "esecuzione ciclo")
)
```

Questo codice stampa semplicemente un messaggio per 5 volte. Il costrutto "Repeat" equivale al classico "For" presente nella maggior parte di linguaggi di programmazione comuni (es. c, basic, pascal ...).

L'ultimo ciclo fondamentale in Lisp è "Foreach". Per capirne l'utilizzo ed il funzionamento è necessario introdurre le liste di valori. Le liste non sono altro che variabili, all'interno delle quali sono presenti più valori, diversamente dalla comuni variabili (che contengono un solo valore), le liste possono contenere n elementi, anche di tipo differente le une dalle altre.

Le funzioni che permettono di agire sulle liste sono parecchie e verranno trattate in apposito capitolo, per ora ci limiteremo alla funzione che permette di costruire una lista semplice, il suo nome è "List". Osserviamo l'esempio che segue:

```
(setq listaTest
      (list "a" "b" "c" "d")
)
```

Questa riga crea una variabile chiamata "listaTest" inserendo al suo interno la sequenza di valori "a", "b", "c", "d".

Per chi abbia già esperienze con altri linguaggi di programmazione si può dire che una lista sia simile ad un array (o vettore), nel quale è possibile inserire un numero n di elementi.

Veniamo ora a "Foreach", il quale permette di scorrere un elemento alla volta una qualsiasi lista, ritornando, uno alla volta gli elementi della lista stessa.

(foreach [variabile] [lista] [codice da eseguire])

Il codice verrà eseguito n volte una per ogni elemento della lista, ad ogni iterazione la variabile conterrà un elemento della lista stessa.

Ad esempio:

```
(setq listaTest (list "a" "b" "c" "d"))
```

```
(foreach elemento listaTest
      (print elemento)
)
```

Questo semplice programma stampa, uno per volta tutti gli elementi presenti nella variabile listaTest.

Nell'utilizzo di qualsiasi ciclo è necessario ricordarsi che, come qualsiasi funzione lisp, anche questi ritornano un valore, questo valore è il risultato dell'ultima operazione compiuta dal ciclo stesso.

Ad esempio:

```
(setq a 25)
      (print (while (/= a 20)
            (setq a (1- a))
      )
)
```

Questo codice stampa, sulla linea di comando 20, poiché l'ultima operazione eseguita dal ciclo "while" da come risultato proprio questo valore.

Funzioni personalizzate

All'interno di un qualsiasi programma, anche molto semplice, un ruolo determinante lo svolgono le "funzioni" definite dallo sviluppatore.

Una funzione in Lisp così come in tutti gli altri linguaggi di programmazione non è altro che una parte di codice richiamabile attraverso l'utilizzo di un identificatore, si pensi ad esempio alla funzione "print" che stampa un testo sulla linea di comando dell'ambiente CAD.

I vantaggi nell'uso delle funzioni sono sostanzialmente tre:

- rendono più efficiente il lavoro di programmazione evitando di ripetere parti di programma più volte all'interno del codice sorgente;
- possono essere utilizzate per costruire librerie di funzioni riutilizzabili all'interno di più programmi senza doverlo riscrivere ogni volta;
- permettono di rendere più chiara la struttura di un programma dividendolo in piccole parti, ognuna delle quali svolge un determinato compito. Questo inoltre aiuta durante la fase di correzione del software.

Vediamo ora come, Lisp, permette di definire una semplice funzione:

```
(defun SempliceFunzione ( )
```

```
      (print "Messaggio sulla linea di comando.")
)
```

In questo caso viene semplicemente stampato un messaggio fisso sulla linea di comando del CAD. Questa funzione può essere richiamata all'interno di una qualsiasi parte del programma inserendo:

```
(SempliceFunzione)
```

Una caratteristica molto importante delle funzione è quella di poter accettare parametri ai quali assegnare dei valori. Quindi se volessimo modificare la "SempliceFunzione" facendo in modo di poter cambiare il messaggio stampato:

```
(defun SempliceFunzione ( msg / )
      (print msg)
)
```

Così facendo potremmo scrivere:

```
(SempliceFunzione "Messaggio da stampare")
```

a questo punto verrebbe stampato il nostro nuovo messaggio. Ogni funzione ovviamente può avere un numero n di parametri ai quali è possibile assegnare qualsiasi tipo di dato (nel nostro caso una stringa).

Finora, tutte le variabili che sono state dichiarate possono essere viste, ed utilizzate, in qualsiasi punto del nostro programma. Le funzioni permettono invece, di definire delle variabili visibili solo ed esclusivamente al loro interno, quindi, una volta terminata la funzione queste variabili spariscono, inoltre, tutto il codice all'esterno della nostra funzione non può avere accesso a tali variabili.

Facciamo ora un esempio, nel quale la "Semplice funzione" stamperà la somma di due valori numerici:

```
(defun SempliceFunzione( valore1 valore2 / somma)
      (setq somma (+ valore1 valore2))
      (print (strcat "Somma = " (rtos somma)))
)
```

richiamandola in questo modo

```
(SempliceFunzione 3.24 5.67)
```

la funzione stamperà sulla linea di comando del CAD

```
Somma = 8.9100
```

la variabile utilizzata per immagazzinare il risultato della somma è locale rispetto alla funzione, quindi, viene creata automaticamente all'inizio di "SempliceFunzione" e distrutta al suo termine. Ogni riferimento a "risultato" agirà sulla variabile locale. Allo

stesso modo anche "valore1" e "valore2" sono variabili locali, che si differenziano per il fatto che, all'inizio della funzione vengono settate con i valori passati come parametri. Anche nel caso delle variabili locali non esiste un limite al loro numero.

Lisp permette al programmatore, di creare particolari funzioni che possono essere utilizzate come comandi nell'ambiente CAD. Si pensi al comando "LINE" che, nell'ambiente CAD, permette di disegnare una linea retta. Allo stesso modo il programmatore potrà creare suoi comandi utilizzabili direttamente dall'operatore.

Di seguito verrà creato un semplice comando che disegna un quadrato a dimensione e posizione fissa:

```
(defun C:DisegnaQuadrato ()
      (command "_rectang" "0,0" "10,10")
)
```

Dopo aver caricato questo piccolissimo programma, se l'utente digitasse sulla linea di comando "DisegnaQuadrato" verrebbe immediatamente disegnato un quadrato tra il punto 0,0 e il punto 10,10.

Ciò che differenzia questo comando, utilizzabile dall'operatore CAD, da una semplice funzione è l'indicazione "C:" prima del nome della funzione, infatti tale prefisso trasforma qualsiasi funzione in un comando CAD. L'unico limite è l'impossibilità di avere parametri nella definizione della funzione, al contrario sarà possibile utilizzare le variabili locali.

Così come tutte le funzioni Lisp, anche quelle personalizzate ritornano al programma che le chiama un valore.

Si osservi il seguente codice

```
(defun PrimaFunzione ( / risultato)
      (setq risultato (raddoppia 123.23))
      (print (strcat "Valore = " (rtos risultato)))
)
(defun raddoppia (valore / ris)
      (setq ris (* valore 2))
ris
)
```

Caricando questo programma e utilizzando la "PrimaFunzione" verrà stampato il valore 123.23 moltiplicato per due. Il raddoppio avviene tramite l'utilizzo di "raddoppia" che, preso il dato passatogli come parametro lo moltiplica per due e ne restituisce il risultato al codice chiamante, il quale a sua volta, lo inserisce nella variabile (locale) "risultato". Come si può constatare l'ultima riga di "raddoppia" è semplicemente il nome di una variabile, questo poiché il valore restituito da una funzione è il dato ritornato dell'ultima espressione presente nella funzione stessa.

Funzioni personalizzate e variabili locali

Come si è visto, ogni volta che si definisce una funzione, questa è utilizzabile in tutto il programma che la contiene.

Questo, può arrivare a generare software con decine o addirittura centinaia di funzioni, delle quali, molte, utilizzate poche volte se non, una volta soltanto.

Immaginiamo di avere questa procedura:

```
(defun fx1 ( / )
...
)
```

Immaginiamo poi che, al suo interno, si faccia uso di altre funzioni, 'fx1_a', 'fx1_b', 'fx1_c'. Se queste fossero utilizzate esclusivamente da 'fx1', sarebbe inutile renderle disponibili al resto del software, inoltre sarebbe uno spreco di risorse (memoria), in quanto ogni funzione definita occupa parte della memoria del PC. L'accumularsi di procedure non utilizzate porta al degrado delle prestazioni del nostro sistema. Per ovviare a questo inconveniente, bisogna comprendere, prima di tutto, che una funzione non è altro che una variabile di tipo particolare.

Infatti, invece di contenere semplici dati, essa contiene del codice interpretabile. Di conseguenza è possibile trattarle esattamente come le comuni variabili.

Ad esempio, se volessimo una funzione identica a 'max' ma con un nome differente, potremmo scrivere:

```
(setq massimo max)
```

e potremmo così usare la nuova funzione 'massimo':

```
(setq n (massimo 1 7 35 3))
```

Come si può constatare, la questione è estremamente semplice.

Tornando ora al nostro problema iniziale, come potremo definire funzioni limitate ad un contesto ristretto?

Abbiamo detto che una funzione si comporta come una variabile, quindi potremo scrivere:

```
(defun fx1 ( / fx1_a)
      (defun fx1_a ( / )
...
      )

)
```

Dichiarando ’fx1_a’ nell’elenco delle variabili locali a ’fx1’ e definendola poi all’interno del corpo di quest’ultima, otterremo che ’fx1_a’ venga creata all’avvio della funzione che la contiene e distrutta al suo termine, esattamente come una variabile locale.

Attenzione, si ci dimenticassimo di inserire ’fx1_a’ nell’elenco delle variabili locali, verrebbe considerata globale e tornerebbe ad essere una normale funzione, presente e usabile in qualsiasi punto del nostro programma.

Le variabili ’Locali’ che diventano ’Globali’

Continuiamo a parlare delle variabili e della loro visibilità. Poniamo di avere questo programma:

```
(setq y 5)
(defun fx ( / x)
      (setq x 10)
      (print x)
      (print y)
)
```

In questo caso la funzione ’fx’ stampa i valori delle variabili ’x’ e ’y’. Certamente il valore di ’x’ sarà pari a 10, infatti questa variabile è dichiarata come locale e impostata prima della sua stampa. Per quanto riguarda invece la variabile ’y’, questa non è stata assegnata e non è nemmeno dichiarata come variabile locale, quindi viene considerata come ’Globale’ e verrà stampato il valore 5, impostato all’esterno della funzione.

In realtà però non è affatto vero che se le variabili non vengono dichiarate locali sono per forza globali e viste da tutto il programma, infatti dipende dal contesto in cui ci si trova.

Il Lisp segue una regola molto particolare per individuare le variabili richieste. Quando utilizziamo una variabile, questa viene cercata nel contesto locale della funzione all’interno della quale si sta operando, se la variabile ricercata non è definita viene analizzato il contensto nel quale la funzione è stata richiamata, e così via fino a risalire al livello più alto, cioè al livello ’Globale’.

Facciamo subito un esempio:

```
(defun fx1 ( / x y)
      (setq x 1)
      (setq y 2)
      (printxy)
)
(defun printxy ( / )
      (print x)
```

```
      (print y)
)
(setq x "a")
(setq y "b")
(fx1)
(printxy)
```

Una volta eseguito, questo programma stamperà questa sequenza di valori:

```
1
2
"a"
"b"
```

Osservando attentamente il codice si può notare che la funzione 'printxy' stampa il contenuto delle variabili 'x' e 'y', queste ultime non essendo dichiarate come locali vengono considerate globali. La funzione 'printxy' viene poi utilizzata in due contesti differenti. Una prima volta all'interno della funzione 'fx1' che possiede due variabili locali 'x' e 'y', una seconda volta viene usata all'esterno di tutte le funzioni. Nel primo caso 'printxy' stampa il contenuto delle variabili presenti all'interno del contesto nel quale viene richiamata (nella funzione 'fx1'), nel secondo caso stampale variabili 'x' e 'y' presenti nel contesto globale del programma.

Da questo comportamento deriva il fatto che le variabili non dichiarate locali possono essere definite come globali a patto che, il contesto nel quale vengono utilizzate, non preveda ad un livello superiore, variabili con lo stesso nome. Di fatto, le variabili locali ad una funzione vengono sempre ereditate dalle procedure richiamate al suo interno.

Le funzioni sono variabili

Il linguaggio LISP utilizza i 'simboli' per poter accedere ai dati. In pratica un programma scritto in questo linguaggio consiste esclusivamente di 'simboli' e valori costanti come stringhe, numeri interi o reali.

Per fare alcuni esempi, sono 'simboli':

```
princ
defun
if
+
```

mentre sono costanti:

```
"stringa"
124
```

```
12.03
```

Anche le normali variabili utilizzate nei programmi non sono altro che 'simboli' legati ad un valore costante. Quindi quando utilizziamo:

```
(setq x 12.3)
```

Semplicemente creiamo un 'simbolo' x con un suo valore costante. Perfino le funzioni sono 'simboli' con valori costanti, in questo caso il valore è rappresentato dal codice di definizione della funzione stessa.

Partendo da questi presupposti possiamo creare funzioni come questa:

```
(defun test ( x y / result)
      (setq result ((if (> x y) - +) x y))
result
)
```

Questa funzione, dati due numeri, sottrae y da x se quest'ultimo ha un valore maggiore, viceversa li somma se x è minore o uguale a y.

La cosa interessante è che la funzione 'if' restituisce i simboli + e - esattamente come se fossero normali variabili.

Un modo diverso, ma più comprensibile, di quanto detto è questo:

```
(defun test2 ( x y / result operazione)
      (if (> x y)
            (setq operazione -)
            (setq operazione +)
      )
      (setq result (operazione x y))
result
)
```

Il risultato finale è lo stesso, in questo caso si può comprendere quanto le funzioni non siano altro che variabili e più in generale 'simboli'.

Eliminare con il valore predefinito

L'utilizzo delle variabili e funzioni, presenta una particolarità di grande importanza.

Se provassimo ad eseguire questa linea di codice:

```
(print NonEsisto)
```

ci accorgeremmo che, Lisp esegue il tutto senza problemi, anche se, 'NonEsisto' non è

mai stata impostata e quindi, in teoria non esiste!

Da questo comportamento se ne può dedurre una regola generale. Regola che sta alla base del concetto di variabile e simbolo in genere.

In Lisp, qualsiasi simbolo (inteso come variabile, nome di funzione, ecc...), se non impostato o definito, ha sempre valore pari a 'nil'.

Legato a questa regola, esiste un altro aspetto da non sottovalutare. Solitamente, sviluppando software applicativo, si tende a dimenticare che variabili, funzioni ecc... occupano memoria. Alle volte si può arrivare ad utilizzare molto più spazio di memoria di quanto, effettivamente, sia necessario, rallentando così il sistema e pregiudicando l'efficienza generale del programma.

La domanda che sorge, forse spontaneamente, è la seguente :

```
"Come fare per ridurre l'utilizzo di memoria?"
```

La risposta è quasi scontata :

```
"Riducendo il numero di variabili globali e di funzioni."
```

Le strade per fare ciò sono diverse. Si può ridurre il numero di variabili globali, riutilizzandole per più scopi, oppure è possibile costruire funzioni che compiano più operazioni, o ancora, si può distruggere ciò che non serve.

Quest'ultimo caso, rappresenta la soluzione più semplice, applicabile anche a software già esistenti (di cui magari si fa manutenzione), ed è possibile farlo proprio grazie al principio visto poco sopra.

Dato che tutti i simboli non esistenti hanno valore pari a 'nil', per eliminare un simbolo già utilizzato ed inizializzato, sarà sufficiente assegnargli tale valore.

Supponiamo di avere questa variabile:

```
(setq a "testo di prova")
```

per eliminarla, recuperando la memoria utilizzata:

```
(setq a nil)
```

Stesso discorso per le funzioni:

```
(defun miaFunzione ( / )
;...
)
```

per eliminarla:

```
(setq miaFunzione nil)
```

L'uso cauto e sapiente del valore di default dei simboli ('nil') e della possibilità di eliminarli, non solo consente di scrivere meno codice, evitando in molti casi

l'inizializzazione esplicita delle variabili, ma rende il software più efficiente.

Introduzione ai commenti

Con l'introduzione dei concetti e delle strutture espresse in questo capitolo, è ora possibile iniziare a scrivere procedure di notevole complessità. Ciò, alle volte, rende difficile la rilettura e la comprensione del codice scritto, soprattutto se i programmi fatti, vengono ripresi dopo lungo tempo per correzioni o modifiche.

Per poter rendere più chiaro e comprensibile un listato di programma, Lisp mette a disposizione la possibilità di inserire righe di commento per aiutare il programmatore nella descrizione delle operazioni svolte.

Riprendendo uno degli esempi fa in precedenza, è possibile scrivere:

```
;questa funzione readdoppia
;il numero 123.23
(defun PrimaFunzione ( / risultato)
      (setq risultato (raddoppia 123.23))
      (print (strcat "Valore = " (rtos risultato)))
)
; questa funzione, passato un valore
; lo restituisce raddoppiato
(defun raddoppia (valore / ris)
      (setq ris (* valore 2)) ;raddoppia il valore
ris ;lo restituisce
)
```

Come si può notare i commenti sono sempre inseriti dopo il carattere ';', infatti tutto ciò che, su una riga, viene digitato dopo il punto e virgola non viene considerato in fase di esecuzione del programma, ed è quindi considerato commento al codice. L'unica eccezione è quando il punto e virgola viene utilizzato all'interno di una coppia di doppi apici (es.: "esempio ; "), in questo caso ';' farà parte di una semplice stringa.

Capitolo 4
Input Utente

L’operazione di richiesta di informazioni all’utente è una delle parti essenziali durante la realizzazione di qualsiasi softwaare interattivo. In questo capitolo introdurremo le tecniche necessare per comunicare con l’operatore in modo semplice, efficace, e coerente con la normale interfaccia del CAD.

Richiesta di dati

Una delle caratteristiche di maggiore importanza di Lisp, è la sua capacità di comunicare con l’operatore. Infatti è possibile, tramite una serie di apposite funzioni, fare domande all’utente, richiedere la selezione di oggetti, e compiere tutte quelle operazioni di interazione con l’operatore indispensabili per la realizzazione di qualsiasi programma.

Di seguito sono elencate tutte le funzioni utilizzabili per l’input utente, ognuna delle quali richiede un particolare tipo di dato all’operatore

Funzione	Descrizione
getint	Richiede l’input di un valore intero.
getreal	Richiede l’input di un valore reale.
Getstring	Richiede un valore tipo stringa.
getdist	Richiede di specificare una distanza.
getpoint	Richiede un punto sul disegno.
getcorner	Specifica l’angolo di un rettangolo.
getangle	Richiede di specificare un angolo.
getorient	Richiede di specificare un angolo.
getkword	Richiede una parola chiave.

Ognuna di queste funzioni opera in maniera identica richiedendo un tipo particolare di dato a fronte di un messaggio di richiesta, ad esempio:

```
(getint "Inserire un numero intero: ")
```

questa riga fa sì che sulla riga di comando compaia il messaggio di richiesta al quale l’utente può rispondere esclusivamente inserendo un numero valido, premendo il tasto invio oppure il tasto ESC.

Una particolarità, molto importante delle funzioni di richiesta è quella di controllare il tipo di dato inserito e, nel caso non sia corretto, rispondere con un adeguato messaggio di errore in maniera automatica, in modo che l’utente possa ritentare l’inserimento del dato richiesto.

Le funzioni di richiesta tipo ’getstring’, ’getint’, ’getreal’, ’getdist’, ritornano

rispettivamente una stringa, un intero, un numero reale, un reale che rappresenta la distanza indicata tramite due punti o tramite la digitazione di un numero. Altre, come getpoint, ritornano una lista di tre valori reali che rappresenta il punto indicato espresso con le coordinate x,y,z. Infine la più particolare, 'getkword', ritorna una delle opzioni che l'utente ha la possibilità di digitare.

Per capire meglio l'uso delle varie funzioni esaminiamo i seguenti esempi:

```
(getstring "Inserire il nome del file: ")
```

Questa linea ritornerà il testo digitato dall'utente.

```
(getpoint "Indicare un punto:")
```

Questa linea ritornerà un punto specificato a video o indicato come coordinate (es.: (10.0 3.2 7.6)), si tenga inoltre presente che se a 'getpoint' si aggiunge un parametro prima del messaggio si vedrà una linea disegnata a video che parte dal punto indicato, ad esempio:

```
(getpoint '(0 0) "Indicare altro punto: ")
```

in questo caso si vedrà una linea che parte dal punto 0,0 e termina in corrispondenza del puntatore sul disegno.

Osserviamo ora quanto segue:

```
(initget "Linea Cerchio Arco")
(getkword "Disegna Linea/Cerchio/Arco: ")
```

L'utilizzo di questa coppia di linee di codice fa sì che l'utente possa rispondere alla richiesta solamente con le tre lettere L, C, A oppure con le parole linea, cerchio, arco (sia scritte maiuscole che minuscole). Infatti l'utilizzo di 'getkword' è subordinato ad una chiamata alla funzione 'initget' che imposta le opzioni disponibili con le relative abbreviazioni scritte in maiuscolo (es.: L per linea).

Tutte le funzioni di input, ad esclusione di 'getstring', 'getreal', 'getint' e 'getkword', permettono l'input sia tramite digitazione da tastiera sia tramite l'utilizzo del mouse sul disegno, infatti alla seguente riga di codice è possibile rispondere in due modi differenti:

```
(getdist "Indicare la distanza: ")
```

il primo sistema per rispondere a questa richiesta è la digitazione della distanza voluta, il secondo sistema è l'indicazione sul disegno di due punti, dai quali la funzione calcolerà la distanza utilizzandola come valore richiesto.

E' importante ricordare che la pressione del solo tasto invio causa la restituzione, da parte delle varie funzioni, del valore nil.

Controllo sull'immissione dati

Utilizzando le funzioni esposte nella sezione precedente ci si renderà presto conto della difficoltà nel gestire alcune situazioni come la semplice pressione del tasti invio o l'immissione di 0 o di cifre negative.

Esiste in Lisp la possibilità di controllare cosa l'utente può o non può rispondere alle funzioni viste in precedenza, facciamo un esempio. Nel caso in cui si desideri richiedere un numero intero all'operatore senza però consentirgli di digitare 0 oppure valori negativi si utilizzerà:

```
(initget 6)
(getint "Inserire valore: ")
```

utilizzando la funzione 'initget' è possibile controllare cosa l'operatore può inserire alla successiva richiesta di dati. La funzione 'initget' vale però solamente per una richiesta di dati.

Il numero digitato come parametro di 'initget' è la combinazione di questi valori:

1	Non consente l'input nullo (solo tasto invio)
2	Non consente input pari a 0
4	Non consente inserimento di valori negativi
8	Non controlla i limiti del disegno
16	-Non utilizzato-
32	Utilizza una linea di tipo tratteggiato
64	Non considera la coordinata Z (solo getdist)
128	Consente qualsiasi input

sommando questi valori si possono regolare gli input dell'operatore.

La regolazione dell'input vista ora, può essere, nel caso si utilizzi 'getkword' combinata con sue opzioni:

```
(initget 6 "Linea Cerchio Arco")
(getkword "Disegna Linea/Cerchio/Arco: ")
```

in questo caso l'utente potrà rispondere con una delle tre opzioni ma non con un solo invio o con un valori negativi. Le opzioni che l'utente può digitare possono indifferentemente essere scritte in maiuscolo, minuscolo o solamente con una singola lettera, quest'ultima sarà determinata delle maiuscole utilizzate con 'Initget' (nel nostro caso 'L', 'C', 'A').

Funzioni Speciali

Lisp fornisce, oltre alle funzione viste in precedenza, due importanti aggiunte che consentono la scelta di file e colori in maniera 'visuale' tramite finestre di dialogo standard.

La funzione "getfiled" permette la scelta di un file tramite la finestra di dialogo standard. La sua sintassi è la seguente:

```
(getfiled titolo nomefile ext flags)
```

Chiamando "getfiled" è necessario specificare nell'ordine, il titolo della finestra che comparirà, il nome del file di default, l'estensione dello stesso e un'opzione numerica data dalla somma dei seguenti valori:

1 Attiva la conferma della sovrascrittura di file esistenti

4 L'utente può modificare l'estensione

8 Attiva la ricerca del file nei percorsi del CAD.

La funzione restituirà il nome del file scelto comprensivo del percorso. Nel caso l'utente scelga il tasto annulla verrà restituito nil.

La funzione acad_colordlg permette la scelta di un colore tramite una finestra contenente i colori disponibili. La sua sintassi è la seguente:

```
(acad_colordlg colore)
```

La chiamata alla funzione avviene semplicemente indicando il colore da selezionare (da 0 a 256) all'apertura della maschera. Al termine della funzione sarà restituito il valore del colore selezionato oppure nil nel caso si sia premuto il tasto annulla. Occorre tenere presente che il colore 0 corrisponde al 'colore da blocco' mentre il colore 256 al 'colore da layer'

Esempio riassuntivo

Per comprendere meglio quanto esposto in questo capitolo, realizzeremo ora una nuova versione del comando 'cerchio' presente nei CAD. Questa procedura, oltre a disegnare un cerchio specificando centro e raggio, consentirà all'operatore di stabilirne il colore e il layer di appartenenza.

Di seguito viene mostrato il codice del comando completo di commenti descrittivi:

```
(defun c:CerchioPlus ( / punto1 raggio colore layer)
      (princ "\nComando CerchioPlus V.1.0.0")
      (setq colore nil)
```

```
      (initget 1);impedisce un input nullo [solo Invio]
      ;richiede un punto
      (setq punto1 (getpoint "\nIndicare centro cerchio:"))
      ;impedisce pressione del solo invio,l'inserimento di 0,
      ;disegna linee tratteggiate
      (initget 35)
      ;richiede una distanza disegnando una linea da punto1
      (setq raggio
      (getdist punto1 "\nIndicare raggio del cerchio: ")
      )
      ;finchè non si è scelto un colore valido
      (while (not colore)
            (initget 4);impedisce l'input di valori negativi
            ;richiede di specificare un valore intero
            (setq colore (getint "\nColore [Invio per lista]:
"))
            ;se viene premuto solo invio
            (if (not colore)
                  (setq colore (acad_colordlg 256)) ;scelta
colore
            );endif
      );endw
      (cond
            ((= colore 256);colore 256 corrisponde a "da
layer"
                  (setq colore "BYLAYER")
            )
            ((= colore 0);256 corrisponde al colore da blocco
                  (setq colore "BYBLOCK")
            )
            ((> colore 256)
                  (princ "\nIl colore indicato non è
valido!")
                  (setq colore nil)
            )
      );endc
      (setq layer (getstring "\nLayer di appartenenza: <0> "))
      (if (not layer);se si preme solo invio
            (setq layer "0")
```

```
      );endif
      ;disegna un cerchio specificando il centro e il raggio
      (command "_circle" punto1 raggio)
      (if colore ;se è stato specificato un colore
            ;modifica il colore dell'ultima entità inserita
            (command "_chprop" "_last" "" "_c" colore "")
      );endif
      ;modifica il layer dell'ultima entità inserita
      (command "_chprop" "_last" "" "_la" layer "")
      (prin1)
);enddef
```

Capitolo 5
Le Liste

L’organizzazione dei dati all’interno dei software Lisp passa, quasi sempre, attraverso l’utilizzo di elementi chiamati Liste, che non sono altro che sequenze non ordinate di dati generici. Questo capitolo spiegherà come sono fatte le liste e come queste possano essere manipolate dal linguaggio.

Introduzione alle liste

Come si è visto nell’introduzione a questo testo Lisp è un derivato del più famoso (e generico) linguaggio LISP. Il termine LISP non è altro che l’acronimo di ’LIST Processing’, da ciò si può dedurre che Lisp è un linguaggio che nasce e si sviluppa attorno al concetto di lista, facendone il principale elemento per la programmazione.

Una lista in Lisp non è altro che un insieme non ordinato di simboli, dove per simbolo si intende una variabile, il nome di una funzione, una stringa, un numero intero, un reale e qualsiasi altro elemento utilizzato all’interno di un programma.

Per capire meglio cosa sia effettivamente una lista si può paragonarla ad un array (o vettore) dinamico i cui elementi non abbiano un tipo specifico ma variabile a seconda delle esigenze, inoltre le liste hanno la possibilità di contenere altre liste al loro interno come elementi dell’insieme.

Ovviamente Lisp consente di creare liste di qualsiasi tipo, permette la ricerca di elementi al loro interno, consente la loro manipolazione, la cancellazione, l’ordinamento.

Si tenga sempre presente che le liste saranno la base per poter accedere al database del disegno per la sua modifica e per la sua consultazione. Con questo potente costrutto (la lista) sarà consentita la realizzazione di applicativi molto sofisticati ed estremamente rapidi utilizzando poche e semplici righe di codice.

Creazione di una lista

La creazione di liste avviene in tre differenti modi. Il primo, fa uso del simbolo ’ che permette di creare una lista formata da elementi costanti, ad esempio:

```
(setq lsta '(1 2 3 4 "test"))
```

in questo caso la variabile ’lsta’ conterrà una lista formata da 4 numeri ed una stringa. Questo modo di costruire liste è però limitato dal fatto che i membri della lista stessa non vengono valutati ma vengono inseriti così come sono, quindi, se usassimo come membro una variabile, nella lista verrebbe inserito il nome della stessa e non il suo valore, ad esempio:

```
(setq vr1 "test")
(setq lsta '(1 2 3 4 vr1))
```

al termine di queste istruzioni 'lsta' conterrà '(1 2 3 4 vr1)' e non '(1 2 3 4 "test")' come ci si potrebbe aspettare.

Per superare questo limite esiste la funzione 'LIST' che genera una lista valutandone i membri, ad esempio riprendendo il precedente esempio:

```
(setq vr1 "test")
(setq lsta (list 1 2 3 4 vr1))
```

ora la variabile 'lsta' conterrà '(1 2 3 4 "test")' visto che la funzione list valuta i suoi membri prima di inserirli nella lista. Ovviamente è possibile nidificare le liste:

```
(setq lsta (list (list "a" "b") (list 1 2)))
```

in questo caso 'lsta' vale '(("a" "b") (1 2))' quindi è una lista che contiene due altre liste formate ciascuna da due elementi.

Esiste poi il terzo sistema per la creazione delle liste. Questo si basa sull'utilizzo della funzione 'cons' che consente la creazione di liste dall'unione di due elementi, ad esempio:

```
(setq a "test")
(setq lsta (cons a (list "b" "c")))
```

ora 'lsta' vale '("test" "b" "c")', questo significa che 'cons' ha costruito una nuova lista partendo dall'unione di una variabile ('a') e di una lista esistente ('(list "b" "c")').

Sembrerebbe quindi che l'utilizzo di 'cons' permetta solo di aggiungere elementi ad una lista e non di crearne una partendo da semplici valori, questo non è vero in quanto:

```
(setq lsta (cons "test" '()))
```

'lsta', eseguita questa istruzione, conterrà '("test")', quindi sarà una lista formata da un solo elemento. Addirittura è possibile creare una lista partendo da una variabile il cui valore è 'nil':

```
(setq lsta nil)
(setq lsta (cons "test" lsta))
```

anche in questo caso 'lsta' conterrà una lista con un solo elemento di tipo stringa.

Nell'uso di 'cons' è sempre necessario ricordare che la funzione aggiunge un elemento (di qualsiasi tipo) ad una lista, questo significa che:

```
(setq lsta (cons '(1 2) '(3 4)))
```

al termine di questa istruzione 'lsta' conterrà '((1 2) 3 4)', quindi una lista formata da 3 elementi, una sottolista più due valori interi.

Un altro aspetto particolare di 'cons' è quello di permettere la creazione di 'coppie puntate', che non sono altro che liste formate da due valori separati da un punto, ad

esempio:

```
(A . B), (1 . 2), (1 . "test"), (2 . ("a" "b" "c"))
```

questi sono esempi di coppie puntate, liste formate da due elementi separati da un punto. Lo scopo delle coppie puntate è quello di permettere l'interrogazione e la modifica del database del disegno.

Di seguito sono mostrati alcuni esempi per la creazione di coppie puntate:

```
(cons "a" "b")          ->          ("a" . "b")
(cons 1 2)              ->          (1 . 2)
(cons 1 "test")         ->          (1 . "test")
```

Quelli visti finora sono i metodi principali per la creazione delle liste, nei paragrafi seguenti vedremo come manipolare le liste create.

Estrazione dati

Per accedere ai membri di una qualsiasi lista, Lisp mette a disposizione del programmatore una vasta schiera di funzioni:

```
assoc
car
cdr
cadr
last
member
nth
```

Ognuna di queste procedure permette di estrarre da una lista un determinato elemento in base alla sua posizione o al suo contenuto.

La funzione 'assoc' cerca all'interno di una lista formata da sottoliste di 2 elementi ciascuna, la prima che contiene l'elemento chiave indicato, si osservi quanto segue:

```
(setq lsta '(("a" 1) ("b" 2) ("c" 3) ("d" 4)))
```

questa lista è formata da quattro elementi, ciascuno composto da due valori (il primo viene chiamato chiave), applicando la funzione 'assoc':

```
(assoc "b" lsta) restituisce ("b" 2)
(assoc "t" lsta) restituisce nil
```

Come si può facilmente comprendere, 'assoc' cerca il primo componente della lista che contenga la chiave indicata e ne restituisce il valore, se la ricerca fallisce viene restituito 'nil'.

Al contrario di 'assoc', le due 'car' e 'cdr' si limitano a ritornare il primo o gli elementi, dal secondo in poi, presenti in una lista data, ad esempio:

```
(setq lsta '("a" "b" "c" "d"))
(car lsta) restituisce "a"
(cdr lsta) restituisce ("b" "c" "d")
```

E' poi consentito sommare le due funzioni ottenendo 'cadr' che restituisce il secondo elemento della lista:

```
(cadr lsta) restituisce "b"
```

Per ottenere invece l'ultimo elemento:

```
(last lsta) restituisce "d"
```

Se poi è necessario estrarre l'ennesimo elemento, è possibile farlo con l'aiuto di 'nth':

```
(nth 0 lsta) restituisce "a"
(nth 2 lsta) restituisce "c"
(nth 23 lsta) restituisce nil
```

E' bene tener presente che se viene indicato un elemento non presente nella lista verrà ritornato, come nell'esempio, il valore 'nil'.

Passiamo ora ad esaminare 'member' che permette di sapere se, un dato elemento , è presente o meno all'interno di una lista, per esempio:

```
(setq lsta '("a" "b" "c" "d"))
(member 1 lsta) restituisce nil
(member "a" lsta) restituisce ("a" "b" "c" "d")
(member '(1 2) lsta) restituisce nil
(member "c" lsta) restituisce ("c" "d")
```

Come si può notare 'member', quando individua l'elemento cercato, ritorna la parte di lista che inizia dall'elemento indicato, in caso contrario viene ritornato nil.

Foreach

Uno dei costrutti maggiormente utilizzati con le liste è 'foreach' che consente di scorrere, una lista, un elemento per volta, permettendone l'analisi, ad esempio:

```
(setq lsta '("a" "b" "c" "d"))
(foreach elemento lsta
      (print elemento)
)
```

Questo spezzone di codice stampa gli elementi della lista 'lsta' uno alla volta. Da ciò si può intuire che il ciclo in esame inserisce all'interno di una variabile (nel caso precedente 'elemento') un elemento alla volta, preso dalla lista, quindi all'interno del ciclo stesso sarà possibile intervenire sull'elemento estratto.

Altre funzioni

Oltre a quelle viste finora, esistono diverse altre funzioni importanti, per la gestione delle liste. Esamineremo ora, nell'ordine:

```
acad_strlsort
append
length
listp
reverse
subst
```

La prima funzione in analisi è 'acad_strlsort' che, data una lista composta da sole stringhe la ordina in maniera alfabetica, ad esempio:

```
(setq lsta '("b" "a" "d" "c"))
(acad_strlsort lsta) ritorna ("a" "b" "c" "d")
```

Una particolarità di tale funzione è data dal fatto che se gli venisse passata una lista contenente dati diversi dalle stringhe, verrebbe generato un errore e verrebbe ritornato nil.

Altra procedura, semplice ma molto utile è 'append' che permette di aggiungere in coda ad una lista un'altra lista:

```
(setq lsta '("b" "a" "d" "c"))
(setq lsta2 '("1"))
(append lsta lsta2) ritorna ("b" "a" "d" "c" "1")
```

Una volta generata una lista se fosse necessario ricavare il numero di elementi che la compongono:

```
(length lsta) ritorna 4
```

Se fosse poi necessario controllare se una data variabile sia una lista:

```
(listp lsta) ritorna T
(listp "23") ritorna nil
```

Come è facile intuire 'listp' ritorna T solo se il dato passato come parametro è una lista.

Nel caso fosse necessario invertire tutti gli elementi della lista:

```
(reverse '(1 2 3 4)) ritorna (4 3 2 1)
```

Esiste poi 'subst' che permette di sostituire uno degli elementi presenti nella lista con un nuovo elemento, ad esempio:

```
(setq esempio '("1" "2" "3" "4"))
(subst "test" "2" esempio) ritorna ("1" "test" "3" "4")
```

La sintassi di questa funzione è:

```
(subst [nuovo elemento] [vecchio elemento] [lista])
```

Ovviamente, come per tutte le liste, l'elemento sostituito può essere sia un singolo valore sia una lista.

Funzioni personalizzate con argomenti variabili

Il Lisp fornisce molte funzione dotate di una caratteristica particolare. Osserviamo ad esempio la riga seguente:

```
(rtos 12.34)
```

La funzione 'rtos' converte un numero reale nella corrispondente stringa, ciò significa che il risultato dell'espressione precedente sarà "12.3400".

Osserviamo ora la seguente:

```
(rtos 12.34 1)
```

In questo caso otterremo la stringa "1.2340E+01". Il secondo parametro passato ('1'), dice alla funzione di restituire il numero con un formato scientifico e non decimale.

Ora, poco importa il risultato dell'operazione. La cosa veramente interessante è il fatto che 'rtos' accetti, a seconda dei casi, un numero di parametri variabile. Nel primo esempio le veniva passato un solo parametro ('12.34'), nel secondo le venivano passati due parametri, il numero reale più un dato che indicava il formato per la conversione.

Addirittura, esistono funzioni che accettano un numero indefinito di valori passati come parametri, un esempio è il caso di 'max':

```
(max 1 34 8 22 0 23 4)
```

Questa funzione ritorna il valore maggiore tra quelli specificati. Importante è notare che non esiste un numero massimo di valore passabili alla funzione.

Purtroppo, il linguaggio Lisp trattato da questo testo non consente la realizzazione di funzioni con numero di parametri variabile, né tantomeno l'omissione di uno dei

parametri dichiarati.

Se ad esempio scrivessimo:

```
(defun miafun ( testo tipo / )
;...
)
(miafun "mio testo")
```

La chiamata a 'miafun', omettendo uno dei due parametri, causa la generazione di un errore, che in questo caso è 'error: incorrect number of arguments to a function' o qualcosa di similare (a seconda dell'ambiente utilizzato).

Ciò sta a significare che, qualsiasi funzione definita, deve essere richiamata specificando lo stesso numero di parametri in essa dichiarati.

L'unico vero sistema per superare questa limitazione è l'utilizzo del linguaggio C/C++ per la realizzazione di applicazioni SDS/ADS/ARX.

Nonostante ciò, la flessibilità di Lisp, unita ad un poco di fantasia, permette di superare, almeno parzialmente, il problema.

Come sappiamo una lista può contenere un numero variabile di valori, di qualsiasi tipo, inoltre da una lista è possibile estrarre, selettivamente, uno o più elementi.

Partendo da questi presupposti, è possibile 'simulare' una funzione con un numero variabile di argomenti, dotandola di un solo argomento di tipo lista.

Facciamo un esempio, scrivendo la funzione 'somma' che, dati un numero indefinito di valori numerici, ritorna la loro somma algebrica:

```
(defun somma ( listaValori / sommatoria valore )
      (setq sommatoria 0)
      (foreach valore listaValori
            (setq sommatoria (+ valore sommatoria))
      )
sommatoria
)
```

Questa funzione verrà poi utilizzata così:

```
(setq sommaValori (somma (list 1 3 23 23 92 384)))
```

Naturalmente, visto che l'unico parametro posseduto da 'somma' deve essere una lista, utilizzeremo la funzione 'list' per racchiudere i dati da elaborare. Ciò su cui bisogna porre attenzione è il fatto che, in questo modo, con la sola aggiunta di 'list' nella chiamata, possiamo ottenere esattamente quanto serve.

Facciamo un secondo esempio:

```
(defun esempioFx ( listaParametri / valore1 valore2)
      (setq valore1 (nth 0 listaParametri))
      (setq valore2 (nth 1 listaParametri))
      (if (/= (type valore1) 'STR)
            (alert "Tipo primo parametro: Stringa!")
      )
      (if (and valore2 (/= (type valore2) 'INT))
            (alert "Tipo secondo parametro: numero Intero!")
      )
      ;....elaborazione....
)
```

Questa funzione richiede due parametri, il primo di tipo stringa ed il secondo di tipo intero. Il secondo parametro potrà essere omesso ma, se presente, dovrà essere un valore di tipo intero.

'esempioFx' potrà essere utilizzata così:

```
(esempioFx (list "primo valore" 12))
(esempioFx (list "primo valore"))
```

Per comprendere completamente la procedura appena realizzata, è indispensabile esaminare l'utilizzo della funzione 'type'. Questa permette di stabilire che tipo di dato sia contenuto all'interno di una variabile. I tipo riconosciuti sono i seguenti:

Tipo Variabile	Descrizione
'ENAME	Nome di entità
'EXRXSUBR	Applicazione esterna (ObjectARX)
'FILE	Identificativo di file
'INT	Numero intero
'LIST	Lista o Lista puntata
'PAGETB	Funzione di 'paging table'
'PICKSET	Gruppo di selezione
'REAL	Numero Reale
'SAFEARRAY	Valore di tipo 'Safearray'
'STR	Stringa
'SUBR	Funzione interna
'SYM	Simbolo generico

'VRIANT	Valore di tipo 'Variant'
'USUBR da file sorgenti	Funzione Lisp personalizzata o caricata
'VLA-object	Oggetto ActiveX

Infine, un accenno va fatto alla funzione 'listp', la quale verifica se l'argomento passato è una lista valida, nel qual caso ritorna T, viceversa nil. Ad esempio, modificando la funzione precedente:

```
(defun esempioFx ( listaParametri / valore1 valore2)
      (if (not (listp listaParametri))
            (progn
                  (alert "Tipo primo parametro: lista!")
                  (exit)
            )
      )
      (setq valore1 (nth 0 listaParametri))
      (setq valore2 (nth 1 listaParametri))
      (if (/= (type valore1) 'STR)
            (alert "Il primo parametro deve essere una
Stringa!")
      )
      (if (and valore2 (/= (type valore2) 'INT))
            (alert "Tipo secondo parametro: numero Intero!")
      )
      ;....elaborazione....
)
```

In questo caso, 'listp' verifica se l'unico parametro di 'esempioFx' sia una lista, interrompendo il programma nel caso in cui non lo sia.

Esempio Riassuntivo

Per riassumere quanto visto in questo capitolo, di seguito, è presentato il codice che realizza un comando per il disegno di una semplice tabella divisa in righe e colonne più uno spazio per l'intestazione.

Nell'esame dell'esempio si tenga presente il fatto che, i comandi dell'ambiente CAD accettano, nell'indicazione dei punti, sia la dicitura 'x,y,z' sottoforma di stringa sia una lista (x y z) sottoforma di lista.

```
;*******************************************************
```

```
;Procedure: RTABGEN1
;Versione: 1.0.0
;Autore: Roberto Rossi
;Descrizione: permette di disegnare una tabella
; indicando alcuni semplici parametri
;*******************************************************
(defun c:RTABGEN1( / largTab nRighe nColonne pInitTab
AltIntesta AltRighe parametri)
(setq largTab (getdist "\nIndicare larghezza tabella: "))
(setq nRighe (getint "\nIndicare numero righe: "))
(setq nColonne (getint "\nIndicare numero colonne: "))
(setq AltIntesta
      (getdist "\nIndicare altezza Intestazione: ")
)
(setq AltRighe (getdist "\nIndicare altezza Righe: "))
;richiede un punto ritornando una lista
;con le coordinate (x y z)
(setq pInitTab (getpoint "\nIndicare inizio tabella (angolo
superiore sinistro): "))
;creazione lista contenente i dati
;necessari alla realizzazione della tabella
(setq parametri (list largTab nRighe nColonne pInitTab
AltIntesta AltRighe))
(gentab1 parametri)
(prin1)
);enddef

;genera una tabella in base ai parametri
(defun gentab1( dati / largT nR nC pnt AIntesta
ARighe x y ytemp i lColonna)
(setq largT (car dati) ;estrae il primo elemento
      nR (cadr dati) ;estrae il secondo elemento
      nC (nth 2 dati) ;estrae il terzo elemento
      pnt (nth 3 dati) ;...
      AIntesta (nth 4 dati) ;...
      ARighe (nth 5 dati) ;...
);endset
;estrae
```

```
(setq x (car pnt)) ;estrae dal punto la coordinata x
(setq y (cadr pnt)) ;estrae dal punto la coordinata y
(setq ytemp y)
(setq i 0)
;disegna una linea passando al comando due punti,
;il secondo dei quali costruito al momento
(command "_line" pnt (list (+ x largT) y) "")
(command "_copy" "_last" "" pnt (list x (- y AIntesta)))
(setq ytemp (- y AIntesta))
(while (< i nR)
      (setq ytemp (- ytemp ARighe))
      (command "_line"
      (list x ytemp)
      (list (+ x largT) ytemp) "")
      (setq i (1+ i))
);endw
(command "_line" pnt (list x ytemp) "")
(setq lColonna (/ largT nC))
(command "_array" "_last" "" "_R" 1 (1+ nC) lColonna)
(prin1)
);enddef
```

Capitolo 6
Gestione dei File

Così come per i capitoli precedenti, anche questo risulta di fondamentale importanza per la realizzazione di un qualsiasi programma. Le operazioni di creazione, lettura e scrittura di un file sono la base per poter gestire dati permanenti legati al nostro software.

Questo capitolo mostrerà come il linguaggio Lisp effettua tali operazioni.

Apertura e chiusura di un file

Come in qualsiasi altro linguaggio di programmazione, prima di poter leggere o scrivere un file, occorre aprirlo, ed al termine delle operazioni chiuderlo. Questi due, fondamentali, compiti sono svolti dalle funzioni ’open’ e ’close’ da utilizzare nel seguente modo:

```
(open [nome file] [modalità di apertura])
(close [id del file aperto])
```

Come si può notare, la funzione ’open’ richiede l’indicazione del nome di file da aprire e della modalità con la quale quest’ultimo sarà aperto, tale parametro influirà poi sulle possibili operazioni che il programma potrà compiere sul file in questione.

Lisp permette di specificare una tra le seguenti modalità di apertura:

- "r", apre il file in sola lettura;
- "w", apre il file in sola scrittura, sovrascrivendo i dati esistenti o creando un nuovo file nel caso non esista;
- "a", apre il file in sola scrittura, accodando i dati alla fine del file, o creando lo stesso qualora non esista.

La funzione ’open’, dopo aver aperto il file specificato ritorna il suo ID oppure NIL nel caso l’operazione non riesca, inoltre consente di specificare il percorso del file da aprire in due modi differenti:

```
"c:\\miacartella\\miofile.txt" oppure
"c:/miacartella/miofile.txt"
```

ovviamente tocca al programmatore utilizzare la forma per lui più comoda.

Una volta aperto un file, eseguite le necessarie operazioni di lettura/scrittura è sempre necessario chiuderlo attraverso l’utilizzo di ’close’ specificando la variabile contenente l’ID del file precedentemente aperto, come in questo esempio:

```
(setq idf (open "c:/miacartella/miofile.txt" "r"))
...
(close idf)
```

Questo codice apre un file in sola lettura e, successivamente, utilizzando il suo id (contenuto nella variabile idf) lo chiude.

Scrittura dati in un file

Sono due le funzioni per la scrittura in un file. La prima permette la scrittura di un singolo carattere, mentre la seconda scrive un'intera riga comprensiva dei caratteri di ritorno a capo finali.

Per l'inserimento di un semplice carattere in un file:

```
(setq idf (open "c:/miacartella/miofile.txt" "w"))
(write-char 67 idf)
(close idf)
```

Questo semplice esempio inserisce la lettera C (carattere ASCII 67) all'interno del file identificato dalla variabile idf. Si noti che la modalità di apertura "w" fa sì che, qualora il file indicato esista, questo viene sovrascritto, con conseguente perdita del suo contenuto.

Discorso simile per scrivere un'intera linea:

```
(setq idf (open "c:/miacartella/miofile.txt" "w"))
(write-line "testo della linea" idf)
(close idf)
```

La differenza in questo caso, stà nel fatto che la linea da scrivere nel file viene indicata passando alla funzione una normale stringa fissa o una qualsiasi variabile contenente lo stesso tipo di dato.

Lettura

La lettura di un file non differisce molto dalla scrittura ed è consentita tramite queste funzioni:

```
(read-char [id del file aperto])
(read-line [id del file aperto])
```

Nel primo caso viene restituito un carattere, nel secondo un'intera linea fino al ritorno a capo escluso.

Ovviamente la lettura avviene in maniera sequenziale, quindi, utilizzando due volte di seguito la funzione 'read-line' verranno lette le prime due righe del file indicato.

Vediamo ora come visualizzare il contenuto di un intero file di testo:

```
(setq idf (open "c:/miacartella/miofile.txt" "r"))
(setq riga (read-line idf))
(while riga
      (print riga)
      (setq riga (read-line idf))
)
(close idf)
```

In qeusto caso, dopo aver aperto il file in lettura, il programma legge una linea alla volta e ne stampa, con 'print', il contenuto. La lettura termina quando, la funzione 'read-line', ritorna NIL, ciò sta a significare che è stata raggiunta la fine del file (lo stesso comportamento lo ha anche 'read-char'). Alla fine della lettura viene chiuso il file utilizzato.

Funzioni di utilità

Finora abbiamo aperto i file indicando sempre il loro percorso, però è possibile farlo anche senza inserire questo dato, in tal caso il file verrà cercato all'interno dei percorsi di ricerca dell'ambiente CAD (i percorsi di ricerca sono una serie di cartelle nelle quali vengono cercati i file se non individuati nella cartella corrente), modificabili dalla maschera 'opzioni' o 'preferenze'.

Quindi se, all'apertura di un file non venisse indicato il percorso, il sistema lo cercherebbe nei suoi percorsi di ricerca, solo se non venisse trovato nemmeno qui verrebbe restituito il valore NIL.

Esiste però una funzione che ci permette di verificare l'esistenza di un file prima di tentare di aprirlo, o semplicemente prima di utilizzarlo:

```
(findfile [nome file da cercare])
```

Questa funzione restituisce il nome del file cercato completo di disco e percorso, nel caso in cui non venisse trovato verrebbe restituito NIL.

Anche per 'findfile' vale il discorso fatto sui percorsi di ricerca, infatti se indicassimo semplicemente il nome di file senza percorso, la sua ricerca avverrebbe esattamente come per la funzione 'open'.

Esempio Riassuntivo

Per dimostrare l'utilizzo dei file verrà realizzata una funzione che gestisce una rubrica di contatti, nella quale sarà possibile inserire per ogni contatto nome, cognome ed indirizzo di E-Mail. A questa rubrica sarà possibile aggiungere ed eliminare elementi, inoltre sarà possibile salvarla su disco o caricarne una già salvata. Tutto ciò verrà fatto

sfruttando la gestione dei file, le liste e molto di quanto visto finora.

```
;*******************************************
; Gestione Agenda Contatti
; Descrizione: questa funzione gestisce una
; semplice Rubrica contatti con Lisp,
; tale rubrica conterrà solo nomi, cognomi e
; indirizzi email.
;
; Lo scopo di questa procedura è quello di
; dimostrare l'utilizzo di liste e dei file
;
; Autore: Roberto Rossi
; Versione: 1.0.0
;*******************************************
;
; In questo programma vengono utilizzate le
; variabili pubbliche
;
; PubLstContatti
; PubFileContatti
;
; che conterranno rispettivamente l'elenco
; contatti e il nome del file della rubrica
;
;*******************************************
;gestisce la rubrica
;permette il caricamento, il salvataggio,
;l'aggiunta di un contatto, la rimozione di un
;contatto, la visualizzazione della rubrica
(defun c:Rubrica( / risposta)
(while (/= risposta "Fine")
      (initget 1
"Nuova Carica Salva Visualizza Aggiungi Elimina Fine")
      (if pubFileContatti
            (princ (strcat "\n File Rubrica: "
pubFileContatti))
            (princ "\n File Rubrica: NESSUNO ")
      );endif
```

```
      (Setq risposta
      (getkword
      "\n Rubrica Nuova/Carica/Salva/Vis./Agg./Elim./Fine: ")
      );endset
      (cond
            ((= risposta "Nuova")
                  (Rubrica_New)
            )
            ((= risposta "Carica")
                  (Rubrica_Load)
            )
            ((= risposta "Salva")
                  (Rubrica_Save)
            )
            ((= risposta "Visualizza")
                  (Visual_Rubrica)
            )
            ((= risposta "Aggiungi")
                  (add_Rubrica)
            )
            ((= risposta "Elimina")
                  (Remove_Rubrica)
            )
      );Endc
);Endw
);enddef

;inizializza una nuova rubrica
(defun Rubrica_New ( / nomeFile idf riga listaC)
(setq PubLstContatti nil)
(setq PubFileContatti (get_fRunbrica 1))
);enddef

;carica la rubrica
(defun Rubrica_Load ( / nomeFile idf nome cognome mail)
(setq PubLstContatti nil)
(setq PubFileContatti (get_fRunbrica 0))
(if PubFileContatti
```

```
      (progn
            (setq idf (open PubFileContatti "r"))
            (setq nome "")
            (while nome
                  (setq nome (read-line idf))
                  (if nome
                        (progn
                              (setq cognome (read-line
idf))
                              (setq mail (read-line idf))
                              (setq PubLstContatti
                                    (cons (list nome
cognome mail) PubLstContatti)
                              );endset
                        );endp
                  );endif
            );endw
            (close idf)
            (Alert "Rubrica Caricata con successo")
      );endp
);endif
(prin1)
);enddef

;salva la rubrica nel file indicato
(defun Rubrica_Save ( / idf elemento)
(if PubFileContatti
      (progn
            (setq idf (open PubFileContatti "w"))
            (foreach elemento PubLstContatti
                  (write-line (car elemento) idf)
                  (write-line (cadr elemento) idf)
                  (write-line (caddr elemento) idf)
            );endfor
            (close idf)
            (Alert "Rubrica Salvata con successo")
      );endp
      (progn
```

```
            (alert "Nessuna Rubrica Selezionata!")
      );endp
);endif
(prin1)
);enddef

;visualizza tutti gli elementi della rubrica
(defun Visual_Rubrica ( / elemento i)
      (setq i 0)
      (textscr)
      (foreach elemento PubLstContatti
            (princ (strcat "\n Contatto N." (itoa i)))
            (princ (strcat "\n Nome: " (car elemento)))
            (princ (strcat "\n Cognome: " (cadr elemento)))
            (princ (strcat "\n E-Mail: " (caddr elemento)))
            (princ "\n +----------")
            (setq i (1+ i))
      );endfor
(princ "\n\n\n")
(prin1)
);enddef

;visualizza un elemento della rubrica
(defun Visual_Rubrica_El ( nelemento / elemento)
(setq elemento (nth nelemento PubLstContatti))
(if elemento
      (progn
            (textscr)
            (princ (strcat "\n Contatto N." (itoa
nelemento)))
            (princ "\n +---------+")
            (princ (strcat "\n Nome: " (car elemento)))
            (princ (strcat "\n Cognome: " (cadr elemento)))
            (princ (strcat "\n E-Mail: " (caddr elemento)))
            (princ "\n +----------")
      );Endp
);endif
(prin1)
```

```
);enddef

;aggiunge un elemento alla rubrica
(defun add_Rubrica ( / elemento nome cognome mail)
      (setq nome (getstring "\nNome: "))
      (setq cognome (getstring "\nCognome: "))
      (setq mail (getstring "\nE-Mail: "))
      (setq PubLstContatti (cons (list nome cognome mail)
PubLstContatti))
      (prin1)
);enddef

;elimina l'elemento n dalla rubrica
(defun Remove_Rubrica ( / nelemento )
      (setq nelemento
      (getint
"\nIndicare il numero del contatto da rimuovere: "))
      (if nelemento
            (progn
                  (setq PubLstContatti
                  (RemoveEl_fromList nelemento
PubLstContatti))
            );endp
      );endif
(prin1)
);enddef

;scegli il file della rubrica e ne restituisce
;nome completo di percorso
(defun get_fRunbrica ( flag / )
      (getfiled "Scelta Rubrica" "rubrica.rbl" "rbl" flag)
);enddef

;rimuove l'elemento numero nelemento dalla lista
;(il primo elemento è 0)
;ritorna la lista senza l'elemento indicato
(defun RemoveEl_fromList ( nelemento lista / newl idf)
      (setq newl nil)
```

```
      (setq idx 0)
      (if lista
            (progn
                  (foreach elemento lista
                        (if (/= nelemento idx)
                              (progn
                                    (setq newl (cons
elemento newl))
                              );Endp
                        );endif
                              (setq idx (1+ idx))
                  );endfor
            );endp
      );Endif
newl
);enddef
```

Capitolo 7
Il Database del disegno

Introduzione al Database del disegno

In questo capitolo esamineremo la caratteristica più interessante, ed allo stesso tempo più complessa, del linguaggio Lisp, e cioè la sua capacità di intervenire sul disegno presente nell'ambiente CAD. Prima però di iniziare l'esame dei metodi per modificare i disegni, occorre introdurre alcuni concetti necessari per poter apprendere meglio quanto verrà esposto.

Prima di tutto è bene dimenticare la forma grafica del disegno e considerarlo come un grosso database, nel quale sono inserite tutte quelle informazioni che permettono al CAD una rappresentazione grafica. Si pensi ad esempio ad una linea, in realtà, all'interno del database del disegno questa è rappresentata come una lista (proprio una lista di LISP) contenente due punti, l'inizio e la fine della linea stessa.

Allo stesso modo tutti gli oggetti grafici utilizzabili nel disegno non sono altro che semplici liste contenenti i dati per la loro rappresentazione. L'insieme di tali liste forma quello che viene chiamato 'database del disegno'.

Per comprendere meglio si osservi quanto segue:

```
((-1 . <Entity name: 60000022>) (0 . "LINE") (8 . "PIANO1")
(10 0.0 0.0 0.0) (11 10.0 10.0 0.0) (21 0 0.0 0.0 1.0))
```

questa lista è la definizione, secondo il CAD di una semplice linea, posizionata sul piano 'PIANO1', il cui inizio è rappresentato dal punto '0,0,0' e la sua fine '10,10,0'. Si tenga sempre presente che la rappresentazione dei punto nel CAD è espressa con le coordinate X,Y,Z.

Una delle caratteristiche fondamentali delle liste di definizione degli oggetti è la loro formazione a coppie di valori. Ogni sottolista (chiamata gruppo), che rappresenta la definizione di un dato dell'oggetto in esame è, in molti casi, composta da una 'coppia puntata', cioè da due valori, di cui il primo rappresenta il codice del campo (chiamato codice di gruppo), mentre il secondo è il vero e proprio valore. In tutti gli altri casi vale comunque la regola per cui il primo dato presente determina il codice del gruppo e i successivi rappresentano i valori del gruppo stesso.

I nomi delle entità e il loro utilizzo

Ogni parte di un disegno, sia essa visibile o invisibile, è considerata come una entità. Quindi linee, cerchi, archi, polilinee, testi, blocchi, layer, tipi di linea, stili di testo, attributi e tutto ciò che si può utilizzare all'interno di un progetto è definito come entità.

Ovviamente ogni tipo di oggetto avrà delle sue caratteristiche, diverse da quelle possedute dagli altri tipi di elementi. L'unica proprietà posseduta da tutte le entità è il 'nome entità'. Osserviamo la lista seguente che definisce una linea:

```
((-1 . <Entity name: 60000022>) (0 . "LINE") (8 . "PIANO1")
(10 0.0 0.0 0.0) (11 10.0 10.0 0.0) (21 0 0.0 0.0 1.0))
```

Ogni sottolista che definisce la linea, contiene al suo interno il tipo di campo, stabilito dal primo elemento di ogni lista. L'elemento con codice di gruppo -1 è il nome di entità assegnato dal CAD all'oggetto.

Tramite il 'nome entità' è possibile accedere agli oggetti per la loro interrogazione o la loro modifica, inoltre è possibile utilizzarlo con i comandi standard del CAD per compiere le normali operazioni.

Esistono diversi modi per ottenere il 'nome entità' di un oggetto presente nel disegno. Il sistema più semplice è l'utilizzo della funzione 'entsel' in questo modo:

```
(setq elemento (car(entsel "Selezionare un elemento: ")))
```

una volta eseguita questa riga di codice la variabile 'elemento' conterrà l'identificativo dell'entità selezionata. Con questo dato sarà possibile, ad esempio, agire sull'entità con i comandi standard del CAD:

```
(command "_ERASE" elemento "")
```

in tal caso l'elemento precedentemente selezionato verrebbe eliminato dal disegno. In genere è permesso utilizzare le variabili contenenti i nomi di entità con tutti i comando che richiedono la selezione di elementi grafici.

Bisogna sempre tenere presente che 'entsel' restituisce una lista formata da due elementi, il primo rappresenta il nome dell'entità selezionata, il secondo un punto tridimensionale sull'elemento indicato, inoltre consente la selezione soltanto di una singola entità, nel caso non si selezioni nulla cliccando in una zona vuota del disegno, la funzione restituirà il valore nil.

Nell'esempio precedente è stato utilizzato il comando '_ERASE' per eliminare l'entità selezionata dal disegno, LISP fornisce un metodo alternativo per l'eliminazione di qualsiasi entità, sia grafica che non, attraverso l'uso di 'entdel' come segue:

```
(entdel elemento)
```

in questo caso verrà eliminata l'entità presente nella variabile 'elemento' ovunque essa si trovi.

All'interno di un disegno CAD, esistono, due tipologie di oggetti particolari, dette 'complesse' che hanno la caratteristica di possedere più entità al loro interno. Tali oggetti sono le polilinee e i blocchi.

Nel caso delle polilinee, queste sono formate da un numero N di spezzoni di linea, ognuno dei quali con proprietà proprie, allo stesso modo i blocchi contengono, al loro interno, oggetti grafici quali linee, cerchi, archi ecc, ed inoltre possono includere anche gli attributi, fondamentali per poter associare dati variabili agli elementi.

Per poter selezionare una sottoentità presente nelle polilinee o nei blocchi occorre

utilizzare le funzioni 'nentsel' e 'nentselp' in questo modo:

```
(setq elemento (car(nentsel "Selezionare entità")))
```

al termine di questa linea di codice la variabile 'elemento' conterrà il nome di entità corrispondente alla parte dell'oggetto complesso che l'operatore selezionerà, quindi nel caso in cui venga selezionata una parte di una polilinea, la variabile conterrà il nome di questa sottoentità, allo stesso modo cliccano su un attributo appartenete ad un blocco verrebbe restituito il suo identificativo.

La differenza che esiste tra 'nentsel' e 'nentselp' risiede nel fatto che, nell'utilizzo della seconda funzione è possibile specificare, come secondo parametro un punto che servirà per selezionare l'entità di cui si vuole conoscere il nome.

Un'altra importante funzione che permette di ottenere il nome delle entità è 'entlast', utilizzandola è possibile conoscere l'identificativo dell'ultimo oggetto inserito nel disegno, sia esso grafico o meno.

```
(setq ultimo (entlast))
```

questa linea inserisce in 'ultimo' il nome dell'ultima entità presente nel database del disegno.

Entget e i codici di gruppo

Una volta ottenuto il nome dell'entità sulla quale si vuole intervenire è possibile utilizzare la funzione 'entget' per ottenerne la lista di definizione. Il compito di 'entget' è infatti quello di restituire tale lista. Supponendo di avere una variabile 'elemento' contenente il nome di entità di una linea:

```
(print (entget elemento))
```

questa linea stamperà la lista che definisce 'elemento', e nel caso il nome di entità si riferisca ad una linea otterremo qualcosa del genere:

```
((-1 . <Entity name: 60000022>) (0 . "LINE") (8 . "PIANO1")
(10 0.0 0.0 0.0) (11 10.0 10.0 0.0) (21 0 0.0 0.0 1.0))
```

Da questa lista è possibile conoscere le caratteristiche dell'oggetto e, come vedremo successivamente, sarà possibile apportare modifiche allo stesso.

Come già detto precedentemente, il primo valore di ogni sottolista di definizione delle entità è chiamato 'codice di gruppo' poiché stabilisce il tipo di dato memorizzato, ad esempio il valore '-1' rappresenta il campo contenente il nome dell'entità, '0' invece rappresenta il campo in cui è memorizzato il tipo di oggetto (nel caso precedente "LINE", cioè una linea).

L'elenco dei 'codici di gruppo' è molto esteso, si pensi che si va dal codice -1 al codice 1071, per un totale di alcune centinaia di codici.

In questo testo non viene riportato questo elenco poichè ciò esula dagli scopi del testo stesso, la cosa importante è sapere che questi 'codici di gruppo' sono gli stessi, con lo stesso significato, di quelli utilizzati all'interno dei file DXF.

Per chi non ne fosse a conoscenza ricordo che il formato di file DXF permette lo scambio di file tra ambienti CAD diversi, la loro particolarità è quella di essere file di testo e di utilizzare gli stessi codici di gruppo ottenuti tramite 'entget'. Inoltre il formato DXF è di pubblico dominio, quindi è disponibile un'ampia documentazione per ogni versione di tale formato.

Modifiche delle entità

Una volta ottenuta la lista di definizione di un oggetto con 'entget', LISP consente la modifica di tale lista al fine di modificare l'oggetto definito direttamente sul disegno senza dover intervenire con i comandi standard del CAD.

Le funzione preposte a questo compito sono due, 'entmod' ed 'entupd', la prima modifica effettivamente l'entità, la seconda aggiorna la sua visualizzazione nel disegno.

Per mostrare come apportare modifiche agli oggetti tramite 'entmod' poniamo di voler cambiare il colore ad un oggetto. Si osservi il codice seguente, nel quale viene definito un nuovo comando ('CambiaCol') che, quando eseguito, permette la selezione di un oggetto grafico e quindi ne modifica il colore a seconda di quanto scelto dall'utente in una apposita maschera.

```
1.(defun c:CambiaCol( / elemento nomEnt OldColor NewColor)
2.
3.(setq nomEnt (car(entsel "\nSelezionare oggetto a cui
cambiare colore: ")))
4.(if nomEnt ;se è stato selezionato un oggetto procede
5. (progn
6. (setq NewColor (Acad_colordlg 0 T)) ;selezione colore
7. (setq elemento (entget nomEnt)) ;estrazione lista
8. (setq OldColor (assoc 62 elemento)) ;estrazione colore
9. (if OldColor
10. (progn ;se esiste un colore precedente
11. ;lo sostituisce con quello indicato
12. (setq elemento (subst (cons 62 NewColor) oldColor
elemento))
13. );endp
14. (progn ;se ha il colore di default
15. ;aggiunge il campo colore con il colore da assegnare
```

```
16. (setq elemento (cons (cons 62 NewColor) elemento))
17. );endp
18. );endif
19. (entmod elemento) ;modifica l'entità del disegno
20. (entupd nomEnt) ;aggiorna la visualizzazione corrente
21. );endp
22. (progn
23. (alert "Non è stato selezionato alcun oggetto !")
24. );endp
25.);endif
26. (prin1)
27.);enddef
```

N.B.: la numerazione inserita sulle righe serve solamente per poter spiegare con maggior chiarezza la procedura e non fa parte della stessa.

Prima di iniziare l'analisi della funzione è indispensabile sapere che, il codice di gruppo che identifica il colore di un oggetto, all'interno della lista di definizione dello stesso è il 62, si osservi la seguente lista che descrive una linea di colore rosso:

```
((-1 . <Entity name: 60000022>) (0 . "LINE") (8 . "0") (62 .
1) (10 0.0 0.0 0.0) (11 10.0 10.0 0.0) (21 0 0.0 0.0 1.0))
```

Come si vede il gruppo 62 contiene il valore 1 che identifica proprio il colore della linea (rosso). Nel caso in cui il gruppo 62 non sia presente l'oggetto assumerà il colore di default, solitamente 'daLayer', quindi dipenderà dal piano sul quale l'elemento è posizionato.

Fatte queste precisazioni passiamo ad esaminare la procedura per la modifica del colore degli oggetti grafici.

Alla riga 3 la funzione richiede all'utente la selezione dell'oggetto su cui agire (tramite 'entsel'), una volta verificata l'avvenuta selezione (riga 4) il programmo chiede il colore da assegnare tramite la maschera standard del CAD (riga 6). Fatte queste operazioni preliminari viene estratta la lista di definizione per l'entità selezionata (riga 7) e quindi estratto il colore dal gruppo 62 (riga 8).

Una volta verificata la presenza del campo colore (riga 9), questo viene sostituito con il nuovo valore (riga 12), viceversa se il colore non è presente nell'oggetto selezionato, l'apposito campo viene semplicemente aggiunto alla lista dell'entità (riga 16).

Infine per rendere effettiva la modifica effettuata si utilizza 'entmod' per inserire la nuova lista nel database del disegno e 'entupd' per aggiornare a video l'entità modificata.

Come si può facilmente notare, nella pratica, l'intervento su entità grafiche si riduce alla modifica di semplici liste LISP formate da sottoliste. Ovviamente questa strada non

è l'unica possibilità che il LISP mette a disposizione per modificare il disegno, infatti la procedura appena vista potrebbe essere riscritta come segue:

```
1. (defun c:CambiaColS ( / nomEnt NewColor)
2.
3. (setq nomEnt (car(entsel "\nSelezionare oggetto a cui
cambiare colore: ")))
4. (if nomEnt ;se è stato selezionato un oggetto procede
5. (progn
6. (setq NewColor (Acad_colordlg 0 T))
7. (command "_change" nomEnt "" "_p" "_c" NewColor "")
8. );endp
9. (progn
10. (alert "Non è stato selezionato alcun oggetto !")
11. );endp
12. );endif
13. (prin1)
14. );enddef
```

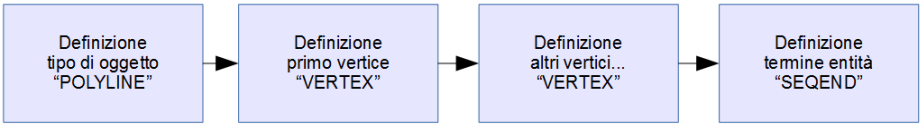
Come si può vedere questa procedura è circa la metà di quella precedente, nonostante le sue funzionalità siano identiche. Questo risultato, all'apparenza eccezionale, lo si deve all'utilizzo indiretto (tramite la funzione 'command') del comando standard 'CHANGE' (riga 7) dell'ambiente CAD utilizzato. Sarà proprio quest'ultimo che si farà carico di effettuare la modifica del colore sull'oggetto.

Purtroppo però, la scelta di intervenire sugli oggetti tramite l'utilizzo di 'command' penalizza fortemente il programma dal punto di vista velocistico, in quanto, l'accesso diretto al database del disegno utilizzato dalla prima procedura sviluppata è circa il 30% più rapido (questo dato può arrivare fino al 100% a seconda dell'operazione che si compie e del CAD utilizzato).

Ovviamente sta al programmatore trovare la soluzione migliore per le esigenze del software che si vuole sviluppare, quindi alle volte sarà certamente comodo l'utilizzo indiretto dei comandi del CAD, altre volte sarà migliore l'accesso diretto al database del disegno.

I programmi CAD moderni, prevedono l'utilizzo di due tipi particolari di entità, le polilinee e i blocchi. Entrambi questi oggetti vengono inseriti nel database del disegno non con una singola registrazione ma come una serie di entità separate.

Osserviamo la figura seguente che schematizza come una polilinea è memorizzata all'interno del database del disegno:



Ogni sezione dello schema rappresenta un’entità registrata nel disegno, nel caso della polilinea, questa è formata da una prima entità che rappresenta l’intestazione, in cui vengono inseriti i dati generali dell’oggetto (ad esempio il colore), successivamente, vengono create N entità, una per ogni vertice dell’oggetto, infine per identificare il termine della polilinea viene utilizzata una entità il cui codice di gruppo 0 contiene la dicitura maiuscola ’SEQEND’. L’insieme di queste entità forma la polilinea nel suo complesso.

E’ bene ricordare che in questo caso per polilinea si intende una polilinea 3D e non una polilinea 2D, che è strutturalmente differente.

Questa struttura consente, tramite LISP, di intervenire direttamente sui vertici dell’oggetto trattandoli come entità a sé stanti.

Per comprendere meglio la struttura della polilinea 3D, si osservi il codice seguente che definisce il comando ’PRINTDBPOLY’:

```
1. (defun c:PrintDBPoly ( / elemento nomEnt dato)
2. (setq nomEnt (car(entsel "\nSelezionare polilinea:")))
3. (if nomEnt ;se è stato selezionato un oggetto procede
4. (progn
5. (setq elemento (entget nomEnt)) ;estrazione lista
6. (if (= (cdr(assoc 0 elemento)) "POLYLINE")
7. (progn
8. (princ "\n\nDefinizione oggetto selezionato")
9. (textpage)
10. (while (/= (cdr(assoc 0 elemento)) "SEQEND")
11. (princ "\n\n")
12. (princ elemento)
13. (setq nomEnt (entnext nomEnt))
14. (setq elemento (entget nomEnt))
15. );endw
16. (princ "\n\n")
17. (princ elemento) ;stampa SEQEND
18. (princ "\n\nFINE Definizione oggetto\n")
19. );endp
```

```
20. (progn ;else
21. (alert "L'elemento non è una polilinea 3D!")
22. );endp
23. );Endif
24. );endp
25. (progn
26. (alert "Non è stato selezionato alcun oggetto !")
27. );endp
28. );endif
29. (prin1)
30. );enddef
```

Lo scopo di questo comando è quello di stampare (a video) il database del disegno relativo alla polilinea che l'operatore selezionerà.

Il suo funzionamento non presenta nessuna novità rispetto a quanto visto finora, salvo il fatto che, alla riga 9 viene utilizzata la funzione 'TEXTPAGE' che pulisce e visualizza la finestra di testo del CAD, in modo da consentire una miglior visione di quello che verrà visualizzato.

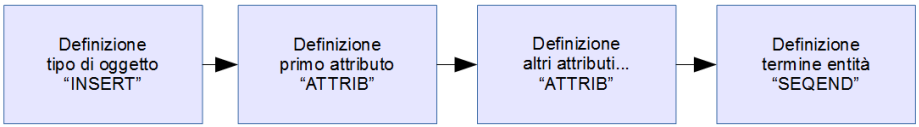
Esaminando il comando si vede che dopo aver richiesto la selezione di una polilinea (riga 2), il programma controlla che l'utente abbia effettivamente indicato un oggetto (riga 3) e quindi se l'oggetto selezionato è una polilinea (riga 6), quest'ultima operazione viene ovviamente eseguita dopo aver estratto la lista dell'entità scelta (riga 5).

Solo a questo punto viene pulita l'area di testo del CAD e visualizzata la sua finestra (con 'textpage'), si entra a questo punto in un ciclo che si concluderà solo quando verrà individuato l'elemento contenente nel codice di gruppo '0' il valore 'SEQEND' che identifica la fine della polilinea (riga 10).

All'interno del ciclo viene stampata la lista dell'entità in esame (riga 12), dopo di chè il programma passa all'entità successiva con l'utilizzo di 'ENTNEXT' (riga 13) e quindi estrae la sua lista (riga 14). Una volta terminato il ciclo viene stampata anche l'entità contenente il 'SEQEND'.

E' bene ricordare che, l'utilizzo della coppia '\n' con la funzione 'PRINC' ha lo scopo di generare dei ritorni a capo nella stampa a video.

Un altro oggetto grafico strutturalmente simile alla polilinea 3D è il blocco. Quest'ultimo quando dotato di attributi assume, all'interno del database del disegno la forma seguente:



Anche il blocco è formato da n entità, la prima delle quali (inserimento blocco) contiene i dati generici sull'elemento, come ad esempio, il punto in cui è stato inserito, il colore del blocco, il layer sul quale esso si posiziona ecc... Di seguito, come entità separate, si trovano gli attributi del blocco, infine la classica entità che determina la fine dell'oggetto blocco (SEQEND).

La struttura esposta è valida solamente per i blocchi dotati di attributi, in caso contrario esisterebbe, nel database del disegno, solamente la prima parte (intestazione blocco).

Come nel caso della polilinea, andremo ora a realizzare un comando che visualizza il database del disegno per un blocco che l'operatore selezionerà.

```
1. (defun c:PrintDBBlock ( / elemento nomEnt dato)
2. (setq nomEnt (car(entsel "\nSelezionare blocco: ")))
3. (if nomEnt ;se è stato selezionato un oggetto procede
4. (progn
5. (setq elemento (entget nomEnt)) ;estrazione lista
6. ;controlla se è un blocco
7. (if (= (cdr(assoc 0 elemento)) "INSERT")
8. (progn
9. (princ "\n\nDefinizione oggetto selezionato")
10. (textpage)
11. ;controlla se sono presenti attributi
12. (if (= (cdr(assoc 66 elemento)) 1)
13. (progn
14. ;stampa il DB finchè non viene trovata la fine
15. (while (/= (cdr(assoc 0 elemento)) "SEQEND")
16. (princ "\n\n")
17. (princ elemento)
18. (setq nomEnt (entnext nomEnt))
19. (setq elemento (entget nomEnt))
20. );endw
```

```
21. );endp
22. (progn
23. (princ "\n\n")
24. (princ "Il blocco indicato NON ha attributi.")
25. );endp
26. );endif
27. (princ "\n\n")
28. ;stampa il SEQEND
29. (princ elemento)
30. (princ "\n\nFINE Definizione oggetto\n")
31. );endp
32. (progn ;else
33. (alert "L'elemento indicato non è un blocco !")
34. );endp
35. );Endif
36. );endp
37. (progn
38. (alert "Non è stato selezionato alcun oggetto !")
39. );endp
40. );endif
41. (prin1)
42. );enddef
```

Esaminando il comando 'PRINTDBBLOCK' si noteranno le similitudini con il precedente 'PRINTDBPOLY', infatti la struttura base è la medesima. Alla riga 2 viene chiesto di selezionare un oggetto nel disegno, quindi viene controllata l'avvenuta selezione (riga 3), e viene estratta la lista dell'oggetto selezionato (riga 6).

Alla successiva riga (riga 7) viene controllato se l'oggetto indicato è effettivamente un blocco, infatti il codice di gruppo 0, nel caso dei blocchi, contiene sempre il valore 'INSERT'. Questo controllo deve essere però effettuato solo sull'entità di intestazione e non sugli attributi o sul SEQEND.

A questo punto viene controllato se il codice di gruppo 66 contiene il valore 1 (riga 12), questo poichè se ciò risultasse vero l'utente avrebbe selezionato un blocco con attributi, viceversa la mancanza del codice di gruppo 66 o il suo valore posto a 0 indicherebbe l'assenza di attributi.

Una volta verificata la presenza degli attributi si procede, con il solito ciclo, alla stampa degli attributi (riga 17), modificando di volta in volta con 'ENTNEXT' l'entità in esame, ovviamente il ciclo terminerà quando verrà incontrata l'entità contenente, nel codice di gruppo 0, il valore 'SEQEND' che determina la fine del blocco (riga 15).

Infine viene stampata anche l'entità contenente il terminatore 'SEQEND' (riga 29).

Anche nel caso di blocchi ed attributo è possibile intervenire separatamente sia sull'inserzione sia sugli attributi, modificando a piacimento le loro caratteristiche mediante le funzioni 'ENTMOD' ed 'ENTUPD'.

E' bene ricordare che in genere è sufficiente possedere il nome dell'entità di un qualsiasi elemento sia grafico che non per poterlo modificare tramite le due apposite funzioni, esattamente come, in questo capitolo, è stato fatto per la modifica del colore su oggetti grafici.

Creazione delle entità con entmake

Fino a questo punto, per creare una nuova entità, si è fatto ricorso all'utilizzo della funzione 'COMMAND' in combinazione con degli apposito comandi CAD per la generazione di nuovi oggetti (es.: i comandi LINE, PLINE, CIRCLE ecc...). Il linguaggio LISP dispone di una funzione apposita per generare nuove entità, senza essere costretti a ricorrere all'uso di 'COMMAND', il suo nome è 'ENTMAKE'.

Come si è già detto, lo scopo di 'ENTMAKE' è quello di aggiungere una nuova entità al disegno corrente. La sua sintassi è identica a quella di 'ENTMOD', che ricordo permette la modifica di entità già esistenti. Si osservi quanto segue:

```
(entmake [lista entità])
```

la funzione richiede un solo parametro, nel quale viene inserita la lista dell'entità da generare. Tale lista deve possedere la stessa struttura di quelle risultanti da 'ENTGET', ad esempio se si volesse creare un cerchio, di colore blu con raggio 10 sarebbe sufficiente:

```
(entmake '((0 . "CIRCLE") (62 . 5) (10 2.0 2.0 0.0) (40
10.0)))
```

In questo caso a 'ENTMAKE' viene passata una lista contenente:

1. codice di gruppo 0 che specifica il tipo di oggetto (CIRCLE);
2. codice di gruppo 62 per specificare il colore da assegnare all'oggetto;
3. codice di gruppo 10 specifica il centro del cerchio (punto 2,2,0);
4. codice di gruppo 40 specifica il raggio.

Si tenga sempre presente che il parametro passato a 'ENTMAKE' deve essere una semplice lista sia essa costante (come nell'esempio) sia contenuta in una variabile o generata dalla funzione 'LIST'. Risulta quindi possibile riscrivere la precedente linea di codice nel seguente modo:

```
(setq listaent (list (0 . "CIRCLE") (62 . 5) (10 2.0 2.0 0.0)
(40 10.0)))
(entmake listaent)
```

oppure

```
(entmake (list (0 . "CIRCLE") (62 . 5) (10 2.0 2.0 0.0) (40
10.0)))
```

L'utilizzo della funzione 'LIST' permette di inserire variabili all'interno della lista passata a 'ENTMAKE' così da poter variare dinamicamente uno o più valori della lista di costruzione. Se ad esempio si volesse chiedere all'operatore quale colore assegnare all'oggetto si potrebbe scrivere:

```
(setq colore (getint "Indicare colore oggetto: "))
(setq lstcolore (cons 62 colore))
(entmake (list (0 . "CIRCLE") lstcolore (10 2.0 2.0 0.0) (40
10.0)))
```

Al contrario di 'ENTMOD', la funzione di creazione 'ENTMAKE' non richiede l'inserimento del codice di gruppo (-1) che identifica il nome dell'entità, questo poiché tale elemento verrà automaticamente aggiunto dal CAD al termine della creazione.

Nel caso in cui 'ENTMAKE' non riesca a creare un oggetto, restituirà il valore nil, al contrario se l'operazione di creazione terminasse correttamente verrebbe restituita la sua lista di definizione.

In genere, per creare correttamente un oggetto utilizzando 'ENTMAKE', è sufficiente osservare, per quel tipo di elemento, ciò che viene restituito da 'ENTGET', replicando i codici di gruppo usati senza inserire il codice -1.

Le Tabelle dei Simboli

Il linguaggio Lisp implementato nei sistemi CAD consente, oltre all'accesso alle entità del disegno, di interagire con gli oggetti non grafici. In particolare è possibile lavorare con quelle che vengono chiamate 'tabelle dei simboli', che includono:

- I piani (LAYER);
- Gli stili di testo (STYLE);
- Le definizioni dei blocchi (BLOCK);
- I tipi di linea (LTYPE);
- Le viste (VIEW);

- Gli ucs (UCS);
- Le VPORT;
- Le quote (DIMSTYLE);
- Le applicazioni (APPID).

Le tabelle più utilizzate sono senza dubbio le prime tre, e sono proprio quelle che esamineremo in questa sezione.

Le funzioni utilizzabili per la modifica delle tabelle sono 'entdel', 'entget', 'entmake', 'entmod' e 'handent', mentre le funzioni per l'interrogazione sono solamente 'tblnext', 'tblsearch' e 'tblobjname'.

Per scorrere una tabella è sufficiente utilizzare 'tblnext', ad esempio per stampare l'elenco dei layer:

```
(setq nomel (tblnext "LAYER" t))
(while nomel
      (print nomel)
      (setq nomel (tblnext "LAYER" nil))
)
```

Ma come funziona, di preciso, 'tblnext'? Prima di tutto bisogna indicare, come primo parametro il nome della tabella sulla quale agire, dopo di che potremo specificare se ritornare il primo elemento(t), oppure il prossimo elemento della tabella specificando 'nil' o omettendo il secondo parametro. Come si può osservare, inizialmente viene letto il primo elemento, successivamente vengono letti gli elementi successivi della tabella, specificando 'nil' come secondo parametro.

Ogni elemento letto viene restituito come lista composta da codici DXF, ad esempio il layer 0 può essere definito come:

```
((0 . "LAYER") (2 . "0") (70 . 0) (62 . 7) (6 . "Continuous"))
```

Nel caso in cui non si desideri scorrere la tabella ma, più semplicemente, si voglia ottenere un elemento specifico come, ad esempio, la definizione del layer 0, si potrà utilizzare 'tblsearch' in questo modo:

```
(tblsearch "LAYER" "0")
```

In questo caso verrà restituita la stessa lista vista in precedenza.

Se volessimo modificare le caratteristiche del layer 0, ad esempio il colore, dovremmo scomodare la nuova funzione 'tblobjname'. Questa funzione consente di ottenere il nome di entità relativo ad un singolo elemento presente nelle tabelle dei simboli, grazie a questo nome di entità potremo effettuare modifiche utilizzando 'entmod'. Vediamo come assegnare, al layer 0, il colore rosso (1):

```
;lettura nome entità layer 0
(setq entlay (tblobjname "LAYER" "0"))
;lettura definizione
(setq deflay (entget entlay))
;lettura gruppo 62 (colore)
(setq color (assoc 62 deflay))
;sostituzione vecchio colore
(setq newlay (subst '(62 . 1) color deflay))
(entmod newlay) ;applicazione modifiche
```

In questa procedura, viene utilizza 'tblobjname' passandogli la tabella da interrogare (LAYER) e l'elemento di cui vogliamo ottenere il nome di entità, una volta fatto ciò, con le classiche procedure di modifica è stato cambiato il colore, infine abbiamo applicato il tutto usando 'entmod'. Si tenga presente che il gruppo 62 definisce il colore, come riportato nella documentazione ufficiale dei CAD relativamente al formato DXF.

Allo stesso modo è possibile interrogare le tabelle dei blocchi, ad esempio, per sapere se è presente o meno una definizione all'interno del disegno corrente. Supponiamo di voler controllare la presenza del blocco 'BL01':

```
(tblsearch "BLOCK" "BL01")
```

Ricercando, nella tabella dei blocchi l'elemento 'BL01', se 'tblsearch' ritornasse il valore 'nil' significherebbe l'assenza di tale blocco viceversa, otterremo la lista di definizione di 'BL01'.

Nell'utilizzo della tabella dei blocchi è necessario ricordare che, la presenza di un blocco in tale tabella NON significa che questo sia inserito nel disegno infatti, quando inseriamo un blocco in un disegno questo, prima viene salvato nella tabella dei blocchi poi, viene inserito un suo riferimento all'interno del disegno vero e proprio. Così facendo, il CAD memorizza una sola definizione di blocco, utilizzando poi un semplice riferimento per aggiungerlo al disegno visibile.

Una volta comprese le metodologie di interazione con le tabelle dei simboli, sarà possibile effettuare qualsiasi operazione, come ad esempio la copia di un layer. Il codice seguente, presa la definizione del layer '0', crea il nuovo piano 'TEST01' con le stesse caratteristiche:

```
;lettura nome entità layer 0
(setq entlay (tblobjname "LAYER" "0"))
;lettura definizione
(setq deflay (entget entlay))
;lettura gruppo 2 (nome layer)
(setq nomelay (assoc 2 deflay))
;sostituzione vecchio nome
```

```
(setq newlay (subst '(2 . "TEST01") nomelay deflay))
(entmake newlay) ;creazione nuovo layer
```

Ovviamente, la creazione del nuovo piano avviene sfruttando la funzione 'entmake' con le stesse modalità utilizzate per la creazione delle entità.

Allo stesso modo sarà possibile interrogare la tabella degli stili di testo, ad esempio per averne l'elenco, comprensivo di font associati:

```
(setq defstile (tblnext "STYLE" t))
(while defstile
      (setq nomestile (cdr (assoc 2 defstile)))
      (setq nomefont (cdr (assoc 3 defstile)))
      (print (strcat "Nome Stile: " nomestile
" -> Font associato: " nomefont))
      (setq defstile (tblnext "STYLE" nil))
)
```

Come per le tabelle già viste in precedenza, anche in questo caso è possibile scorrere l'elenco di stili, estraendo i dati dai singoli elementi. Di seguito ecco un esempio della definizione di uno stile di testo:

```
((0 . "STYLE") (2 . "STANDARD") (70 . 0) (40 . 0.0) (41 . 0.9)
(50 . 0.0) (71 . 0) (42 . 0.2) (3 . "romans.shx") (4 . ""))
```

Per approfondire ulteriormente le conoscenze relative alle tabelle dei simboli, consiglio di consultare la documentazione ufficiale relativa al formato DXF.

Capitolo 8
I Gruppi di Selezione

Nei capitoli precedenti, si è visto come poter agire sul disegno tramite i nomi delle entità. L'unica vera limitazione imposta dai sistemi fin qui utilizzati è l'impossibilità di memorizzare un gruppo di nomi di entità, infatti utilizzando la funzione 'ENTSEL' è possibile selezionare un solo elemento alla volta, inoltre questa selezione deve essere fatta dall'utente e non può prescindere da un'operazione manuale.

I gruppi di selezione permettono, sia l'indicazione da parte dell'operatore di più entità nel disegno, sia la selezione automatica di tutti gli oggetti aventi certe caratteristiche. E' quindi possibile risolvere i due problemi sopra indicati con l'uso di questo particolare meccanismo (i gruppi di selezione).

Creazione dei gruppi di selezione

La creazione di gruppi di selezione avviene, nella maggior parte delle situazioni, attraverso l'utilizzo della funzione 'SSGET', la quale consente sia l'input utente sia la selezione automatica tramite le caratteristiche degli oggetti.

Si osservi il codice seguente:

```
(setq gruppo (ssget))
```

In questo caso si utilizza 'SSGET' nella sua forma più semplice per richiedere all'operatore la selezione di uno o più oggetti, infatti l'esecuzione di questa linea di codice visualizzerà sulla linea di comando la richiesta standard per la selezione oggetti e questa continuerà finché non si premerà il tasto Invio o il tasto destro del mouse, proprio per confermare l'avvenuta selezione come è solito fare con i comandi standard del CAD. Nella variabile gruppo verrà memorizzato il nome del gruppo di selezione creato, contenente i nomi delle entità selezionate.

Cosa molto importante è che durante la selezione l'utente avrà la possibilità di utilizzare tutti i metodi standard per questo tipo di operazione, quindi potrà selezionare un oggetto per volta, più oggetti con una finestra di selezione ecc..., esattamente come per qualsiasi altro comando standard.

In realtà, quella vista finora, è la sintassi più semplice di 'SSGET', infatti questa potente funzione può accettare molti parametri, questo per consentire la selezione degli oggetti nel modo più flessibile possibile. La sintassi completa di 'SSGET' è la seguente:

```
(ssget [modo] [punto1 [punto2]] [lista-punti] [filtro])
```

Il parametro di 'SSGET', cioè il 'modo', determina come verranno selezionate le entità nel disegno, di seguito sono elencati i modo più comunemente utilizzati con le relative sintassi.

Modi comuni:

- "C", permette di specificare due punti e seleziona tutto ciò che sta all'interno (anche parzialmente) del rettangolo definito dai due, es.: (ssget "C" '(1 1) '(2

2));

- "CP", permette di specificare una lista contenente n punti e seleziona tutto ciò che si trova all'interno o interseca il poligono costruito con i punti specificati, es.: (ssget "CP" lstpunti);
- "F", permette di specificare una lista contenente n punti e seleziona tutto ciò che interseca la linea disegnata tramite i punti specificati, es.: (ssget "F" lstpunti);
- "L", costruisce un gruppo di selezione con l'ultimo oggetto inserito nel disegno;
- "P", Crea un gruppo di selezione con gli ultimi oggetti selezionati dall'utente. Es. (ssget "P")
- "W", permette di specificare due punti e seleziona tutto ciò che sta completamente all'interno del rettangolo definito dai due, es.: (ssget "W" '(1 1) '(2 2));
- "WP", permette di specificare una lista contenente n punti e seleziona tutto ciò che si trova completamente all'interno il poligono costruito con i punti specificati, es.: (ssget "WP" lstpunti);
- "X", seleziona l'intero disegno, es.: (ssget "X")

La modalità più utilizzata, che permette di agire su tutto il disegno, anche sulle entità non visibili o su piani congelati è il modo 'X', utilizzato come segue:

```
(ssget "X")
```

questa linea ritorna un gruppo di selezione contenente tutte le entità grafiche del disegno. In aggiunta a questa sintassi è possibile indicare le caratteristiche che, gli oggetti da includere nel gruppo, devono avere, ad esempio:

```
(ssget "X" '((0 . "CIRCLE") (62 . 1)))
```

questo codice seleziona, tutti i cerchi con colore rosso (valore 1), come si può intuire, utilizzando la modalità 'X' e passando alla funzione una lista (con lo stesso formato di quelle restituire da 'ENTGET' o usate con 'ENTMOD' ed 'ENTMAKE') che descrive le caratteristiche degli oggetti da selezionare, è possibile creare gruppi di selezione contenenti solo le entità da elaborare.

La lista utilizzata corrisponde, nella sintassi generale di 'SSGET', al parametro che definisce il filtro, infatti è possibile utilizzarlo anche in combinazione con gli altri modi permessi dalla funzione, sempre specificandola come ultimo parametro inserito.

Facciamo ora, alcuni esempi per comprendere meglio l'uso di 'SSGET'. Esaminiamo le

seguenti righe:

```
(setq gruppo (ssget "X" '((0 . "CIRCLE") (62 . 1))))
(command "_change" gruppo "" "_p" "_c" 2 "")
```

Queste due semplici righe, trasformano tutti i cerchi rossi contenuti nel disegno, assegnandogli il colore giallo (valore 2). La cosa fondamentale da notare, è la possibilità di passare le variabili contenenti gruppi di selezione direttamente ai comandi del CAD, in modo che questi agiscano sugli oggetti presenti nel gruppo.

Al contrario della modalità 'X', gli altri modi di selezione agiscono solo sugli oggetti visibili o su layer 'spenti' e non congelati, ad esempio:

```
(setq gruppo (ssget "C" '(0 0) '(10 15)))
(command "_change" gruppo "" "_p" "_c" 2 "")
```

In questo caso verra cambiato il colore a tutti gli oggetti che sono all'interno, anche parzialmente, dell'area rettangolare tra i punto 0,0 e 10,15. In tal caso la selezione è simile a quella fatta manualmente indicando i due punti, con la differenza che la modalità di selezione è fissa, nell'esempio il modo 'C' stabilisce che vengano selezionati gli oggetti che si trovano, sia completamente, sia parzialmente all'interno dell'area indicata.

Come detto in precedenza è sempre possibile aggiungere un filtro:

```
(setq gruppo (ssget "C" '(0 0) '(10 15) '((0 . "CIRCLE")) ))
(command "_change" gruppo "" "_p" "_c" 2 "")
```

Come nel caso precedente vengono selezionati gli oggetti presente all'interno di un'area, con la differenza che in questo caso vengono inseriti nel gruppo di selezione solo ed esclusivamente i cerchi presenti.

Come si può intuire la funzione 'SSGET' è estremamente potente e flessibile, in grado di permettere qualsiasi tipo di selezione all'interno di un disegno. Per contro è indispensabile, data la sua complessità, una certa attenzione nel suo uso, infatti gli oggetti che vengono inclusi nei gruppi di selezione non vengono mai evidenziati a video, quindi non si ha una conferma visiva di ciò che la funzione prende in considerazione.

In aggiunta a ciò che si è visto fino ad ora 'SSGET' dispone di molte altre caratteristiche, che purtroppo esulano degli scopi che questo testo si prefigge, quindi si rimanda alla documentazione ufficiale dell'ambiente CAD utilizzato. Si tenga inoltre presente che, a seconda del programma posseduto, le modalità di selezione possono variare, sia come indicazione, sia come numero, sia nelle modalità di impiego.

Estrazione dei dati da un gruppo

L'unica operazione, consentita da LISP, per l'interrogazione dei gruppi di selezione, è l'estrazione di uno dei nomi di entità in esso contenuti.

Per svolgere questo compito si utilizza la funzione 'SSNAME', la quale estrae l'ennesimo nome di entità presente nel gruppo indicato, vediamone di seguito la sintassi:

```
(ssname [gruppo] [indice])
```

Come si può facilmente intuire passando a 'SSNAME' un gruppo di selezione e un indice numerico, che corrisponde al numero dell'elemento da estrarre, la funzione ritorna il nome di entità scelto. E' importante sapere che il primo elemento di un gruppo ha come indice 0, il secondo 1 e così via fino all'ultimo elemento. Nel caso si specifichi un indice non valido (non presente nel gruppo), 'SSNAME' restituirà nil.

Osservando il seguente esempio si capirà meglio l'uso della funzione in esame:

```
;creazione gruppo di selezione con selezione manuale
(setq gruppo (ssget))
;estrae il primo nome di entità
(setq elemento1 (ssname gruppo 0))
;estrae il secondo nome di entità
(setq elemento2 (ssname gruppo 1))
```

Queste linee si limitano a chiedere una selezione all'operatore, quindi inseriscono nelle apposite variabili, il primo e il secondo elemento presenti all'interno del gruppo creato.

Oltre a quanto visto esistono due funzioni di controllo, le quali nonostante non siano particolarmente potenti svolgono un ruolo importante nella gestione dei gruppi di selezione. La prima è 'SSLENGTH' che ritorna il numero di elementi di un gruppo, la seconda 'SSMEMB' determina se un nome di entità è o meno presente in un gruppo, vediamole nel dettaglio.

La funzione 'SSLENGTH' è estremamente semplice da utilizzare, in quanto richiede semplicemente l'indicazione del gruppo di cui si vuole conosce la dimensione, ad esempio:

```
(setq gruppo (ssget))
(setq nelementi (sslength gruppo))
```

Al termine di queste rige di codice, la variabile 'nelementi' conterrà il numero di entità presenti in 'gruppo'. L'uso di questa funzione è determinante nel momento in cui si vogliono esaminare, una alla volta, tutte le entità presenti in un gruppo, di seguito è possibile constatare come, con 'SSLENGTH' questo sia possibile:

```
(setq gruppo (ssget))
```

```
(setq nelementi (1- (sslength gruppo)))
(while (> nelementi -1)
      ;estrae nome intità
      (setq elemento (ssname gruppo nelementi))
      ;stampa a video il nome dell'entità esaminata
      (print elemento)
      (setq nelementi (1- nelementi)) ;decrementa nelementi
)
```

il codice presentato si limita a stampare i nomi delle entità presenti in un gruppo di selezione (variabile 'gruppo'), come si può notare per la scansione degli elementi si utilizza un ciclo che, decrementando progressivamente la variabile 'nelemento' consente di estrarre, una alla volta, i nomi delle entità presenti nella selezione fatta dall'utente.

Un fatto che può apparire strano è l'uso di 'nelementi' variabile che, prima del ciclo, viene impostata a [numero di elementi - 1], questo poichè l'ultimo elemento che la funzione 'SSNAME' può estrarre da un gruppo è equivalente proprio al numero di elementi meno uno, inoltre il ciclo terminerà solo quando è avvenuta l'estrazione dell'elemento con indice 0 che per 'SSNAME' è il primo oggetto del gruppo di selezione.

Esaminiamo ora la funzione 'SSMEMB' che permette di sapere se una certa entità è presente all'interno di un gruppo. Osserviamo il codice che segue:

```
(if (ssmemb NomeEnt Gruppo)
      (alert "Entità Presente")
)
```

la variabile 'NomeEnt' contiene il nome dell'entità da cercare, mentre 'Gruppo' contiene il gruppo di selezione, qualora 'SSMEMB' trovi l'entità nel gruppo restituisce il suo nome, quindi il programma visualizzerà una piccola finestra di messaggio che conferma l'individuazione (utilizzando la funzione 'ALERT'), viceversa 'SSMEMB' ritorna nil.

La funzione in esame non ha un utilizzo massiccio come 'SSNAME' o 'SSLENGTH' però è l'unico sistema rapido per sapere se un'entità è presente o meno all'interno di un gruppo.

Modifica dei gruppi di selezione

La modifica di un gruppo di selezione può avvenire aggiungendo elementi allo stesso oppure togliendoli.

Per aggiungere un elemento ad un gruppo preesistente si utilizza la funzione 'SSADD':

```
(ssadd [nome di entità] [gruppo di selezione])
```

dopo aver visto la sua definizione osserviamo l'esempio:

```
;chiede all'operatore la selezione di oggetti
(setq gruppo (ssget))
;selezione oggetto da aggiungere al gruppo precedente
(setq elemento
(car (entsel "Selezionare un solo elemento")))
(setq gruppo (ssadd elemento gruppo))
```

Il codice esposto richiede due selezioni, la prima genera un gruppo, mentre la seconda memorizza un nome di entità all'interno della variabile 'elemento' la quale, viene inserito tramite 'SSADD' nel gruppo di selezione 'gruppo'. Importante è notare che 'SSADD' non effettua direttamente l'aggiunta ma si limita a ritornare tutto il nuovo gruppo di selezione comprensivo dell'entità aggiunta.

Un altro uso di 'SSADD' consente la creazione di gruppi di selezione vuoti, utili quando si desidera generarli senza l'uso di 'SSGET', si osservi il codice:

```
(setq grp (ssadd))
```

Al termine di questa riga la variabile 'grp' sarà a tutti gli effetti un gruppo di selezione vuoto, solo a questo punto sarà possibile aggiungere elementi a 'grp', viceversa se l'operazione appena vista non fosse compiuta non sarebbe possibile aggiungere alcuna entità alla variabile in quanto, 'SSADD' lavora solo ed esclusivamente sui gruppi di selezione e mai sulle variabili di altro tipo.

Un'altra funzione che permette di agire sui gruppi di selezione è 'SSDEL', la quale si limita a ritornare un gruppo dal quale viene tolta l'entità indacata come parametro della funzione, vediamo di seguito la sua sintassi:

```
(ssdel [nome di entità] [gruppo di selezione])
```

Il suo funzionamento è del tutto simile a 'SSADD' con la differenza che, in questo caso, l'entità indicata verrà eliminata dal gruppo. Ovviamente questa cancellazione avviene solo all'interno del gruppo di selezione e non si rifletterà mai sul disegno che contiene l'entità.

Esempio Riassuntivo

Per dimostrare cosa è possibile fare con i gruppi di selezione verrà realizzato un piccolo comando che, selezionati alcuni oggetti, permette l'editazione consecutiva di tutti i testi presenti nella selezione effettuata dall'operatore.

Vediamo di seguito il codice completo:

```
1. (defun c:EDMText( / gruppo elName elemento lunghezza tipo)
2. (princ "\nComando EDMText V.1.0.0")
3. (princ "\n Selezionare i testi da editare:\n")
4. (setq gruppo (ssget))
5. (if gruppo
6. (progn
7. (setq lunghezza (1- (sslength gruppo)))
8. (while (> lunghezza -1)
9. (setq elName (ssname gruppo lunghezza))
10. (setq elemento (entget elName))
11. (setq tipo (cdr (assoc 0 elemento)))
12. (if (= tipo "TEXT")
13. (progn
14. (command "_DDEDIT" elName "")
15. )
16. )
17. (setq lunghezza (1- lunghezza))
18. )
19. )
20. )
21. (prin1)
22. );enddef
```

Il comando qui realizzato si chiama 'EDMText', il suo funzionamento è basato sull'utilizzo di un gruppo di selezione che, una volta esaminato consentirà al software, l'editazione dei singoli testi presenti nel gruppo stesso. Esaminiamone ora il funzionamento.

Alla riga 2 viene stampata la richiesta per l'utente, infatti non avendo la possibilità di modificare il messaggio che compare con 'SSGET' è necessario indicarlo in questo modo, di seguito viene chiesto all'operatore di selezionare gli oggetti (riga 3) e quindi si passa a controllare se il gruppo creato è valido, se l'operatore non effettua alcuna selezione il gruppo varrà nil e quindi la procedura terminerà.

Alla riga 7 viene calcolato l'ultimo elemento valido (numero elementi meno 1) del gruppo di selezione creato, quindi si entra nel ciclo 'while' che analizzerà tutti gli elementi del gruppo. Per ogni elemento analizzato viene estratto il suo nome di entità (riga 9) e la sua lista di definizione (riga 10), dalla quale viene ricavato il tipo di oggetto (riga 11) che verrà utilizzato per sapere se l'elemento in analisi è un testo (riga 12), in questo caso il programma procede all'applicazione del comando standard 'DDEDIT' per permettere all'operatore di editare manualmente l'elemento.

Ovviamente l'ultima operazione compiuta nel ciclo 'while' è il decremento della

variabile 'lunghezza', questo per poter consentire una corretta scansione di tutto il gruppo di selezione.

Infine si ricorda che la funzione inserita alla riga 21 ha il compito di impedire la visualizzazione, sulla linea di comando, dell'ultimo valore ritornato dalla procedura, permettendo di terminare correttamente il comando senza visualizzare strani messaggi che l'utente potrebbe non essere in grado di interpretare.

Capitolo 9
Eseguire e ridefinire i comandi

Una parte fondamentale di un CAD sono i comandi richiamabili dalla “linea di comando” presente nel software. Quasi tutte le funzioni del software sono disponibili attraverso l’uso di parole chiave digitabili. Questo capitolo mostrerà come avvantaggiarsi di tale caratteristica, impartendo comandi dall’interno dei propri sofware Lisp.

Utilizzare un comando

All’interno di AutoCAD, progeCAD o IntelliCAD è possibile richiamare un comando solitamente in quattro modi differenti:

1. Utilizzando una voce del menu presente nella parte alta della finestra del programma;
2. Utilizzando una delle innumerevoli icone visibili;
3. Digitandolo sulla linea di comando;
4. Digitando sulla linea di comando il nome della funzione da eseguire, in lingua inglese, facendola precedere da un ’_’. Ad esempio ’_copy’, ’_line’, ’_circle’ ecc...

Si deve sempre tener presente che, all’interno dei menu e delle icone, i comandi vengono richiamati utilizzando la forma esposta al punto 4, quindi quando si utilizza l’icona per disegnare una linea, il CAD eseguirà il comando ’_line’, questo permette di avere un menu unico indipendentemente dalla lingua del CAD usato. Infatti utilizzando il comando ’_line’ in un CAD in lingua italiana e in uno in lingua inglese otterremo il medesimo risultato e cioè l’esecuzione del comando di disegno di una linea.

Tale caratteristica consente, a chi scrive software in LISP, VBA, o in qualsiasi altro linguaggio che interagisce con l’ambiente CAD, di scrivere procedure indipendenti dalla lingua dell’ambiente. Ad esempio, se volessimo disegnare, in Lisp, una linea tra i punti 0,0 e 10,10 scriveremo:

```
(command "_line" "0,0" "10,10" "")
```

Questo codice funzionerà indipendentemente dalla lingua usata dal motore di disegno.

Le cose dette finora sono corrette nella maggior parte dei casi, esiste però una situazione in cui l’utilizzo dei comandi in inglese preceduti da ’_’ non permette di ottenere il risultato voluto.

Il problema si verifica se un qualsiasi applicativo ridefinisce o elimina i comandi standard del CAD. Supponiamo che, un applicativo installato (sempre caricato) abbia ridefinito il comando per il disegno di una linea o peggio l’abbia eliminato, in questo caso la precedente linea di LISP non funzionerebbe ed il nostro software si interromperebbe immediatamente.

Per ovviare a tale problema i CAD hanno introdotto un sistema alternativo e sempre funzionante per richiamare i comandi standard dell'ambiente. La soluzione è l'utilizzo del comando in lingua inglese preceduto da '_.' (sottolineo + punto).

Con questo sistema la nostra linea di LISP diventa:

```
(command "_.line" "0,0" "10,10" "")
```

Questo codice funzionerà in qualsiasi versione di CAD (AutoCAD/progeCAD/IntelliCAD) e in qualsiasi situazione.

Una cosa interessante è notare che l'utilizzo dei comandi in Inglese preceduti da '_' è valida anche per le opzione del comando stesso, ciò significa che per indicare, ad esempio, l'opzione estensione del comando zoom potremo scrivere:

```
_zoom [invio]
_e [invio]
```

Per creare applicazioni funzionanti su qualsiasi versione nazionalizzata dell'ambiente CAD è necessario unire l'utilizzo dei comandi internazionali (in inglese preceduti da '_.') con quello delle proprie opzioni (in inglese precedute da '_').

Utilizzando le recenti versioni di AutoCAD si può incorrere in un'ultimo problema. Se volessimo inserire un blocco manualmente utilizzeremmo il comando '_.insert'. Se però volessimo, da un menu o da un'icona, inserire automaticamente un determinato blocco nel punto 0,0, senza che compaia la maschera di inserimento, potremmo pensare di scrivere:

```
_.insert;nomeblocco;0,0;;;
```

Tutto ciò purtroppo non funziona, in quanto dopo l'esecuzione del comando '_.insert' verrà visualizzata la maschera di inserimento che bloccherebbe irrimediabilmente la procedura.

La soluzione del problema è molto semplice, è sufficiente aggiungere il segno '-' (meno) prima del nome del comando, ciò elimina la finestra di dialogo e attiva la versione a linea di comando della funzione, quindi la linea corretta per inserire un blocco nel punto 0,0 diventa:

```
_.-insert;nomeblocco;0,0;;;
```

Questa 'macro', utilizzabile sia nelle voci di menu che all'interno delle icone di AutoCAD, funzionerà correttamente e con qualsiasi versione del CAD.

Eliminare o ridefinire un comando standard

Gli ambienti CAD permettono sia l'eliminazione che la ridefinizione dei comandi standard.

Per l'eliminazione di un qualsiasi comando è possibile utilizzare il comando '_UNDEFINE'. Dopo la sue esecuzione verrà richiesto di indicare il comando da eleminare, il quale non sarà più disponibile, ad esempio per disattivare il comando 'COPIA', dovremo digitare sulla linea di comando:

```
_undefine [invio]
copia [invio]
```

Si tenga sempre presente che, eliminando la versione nazionalizzata di un comando (copia) si disattiva anche la versione internazionale (_copy), ma non verrà in alcun modo modificato il comportamento della versione standard del comando originale (_.copy).

Una volta disattivato un comando è possibile definirne una nuova versione, magari modificata e più adatta alle nostre esigenze. Prendiamo ad esempio il caso in cui volessimo realizzare un comando 'COPIA' che esegua una copia multipla senza che venga richiesta l'imissione dell'opzione 'M'. Nella versione originale del comando, dovremmo digitare, per eseguire una copia multipla:

```
copia [invio]
[selezione degli oggetti da copiare]
m [invio]
[indicazione della posizione delle copie]
```

Nella versione modificata del comando non dovrà più essere specificata l'opzione 'M', la quale sarà automatica.

Come sempre succede in LISP definiamo il nostro nuovo comando 'COPIA'in questo modo:

```
(defun c:copia (/ selezione)
      (setq selezione (ssget)); seleziona oggetti da copiare
      (if selezione ; se è stato selezionato qualcosa
            (progn ; eseguo una copia multipla
                  (command "_.copy" selezione "" "_m")
            );endp
      );endif
(prin1)
);fine comando
```

Eliminata la definizione del comando 'copia' (con "_UNDEFINE") e caricata questa procedura, che definisce il nuovo comando copia nella versione italiana dell'ambiente CAD, tutti i menù e le icone che utilizzano tale comando useranno la nostra versione e non quella originale. Definendo il comando in lingua (nel nostro caso l'italiano) verrà automaticamente ridefinita anche la versione internazionale (nel nostro caso '_copy').

Ripristinare un comando

Se per qualche motivo volessimo ripristinare la versione originale di un comando disattivato o ridefinito, dovremmo semplicemente utilizzare '_REDEFINE' in questo modo:

```
_redefine [invio]
copia [invio]
```

questo riporterà il comando 'copia' nello stato iniziale. Anche in questo caso ridefinendo la versione nazionale (nel nostro caso copia) verrà automaticamente ripristinate anche quella internazionale (_copy).

Capitolo 10
Capire ed utilizzare le DCL

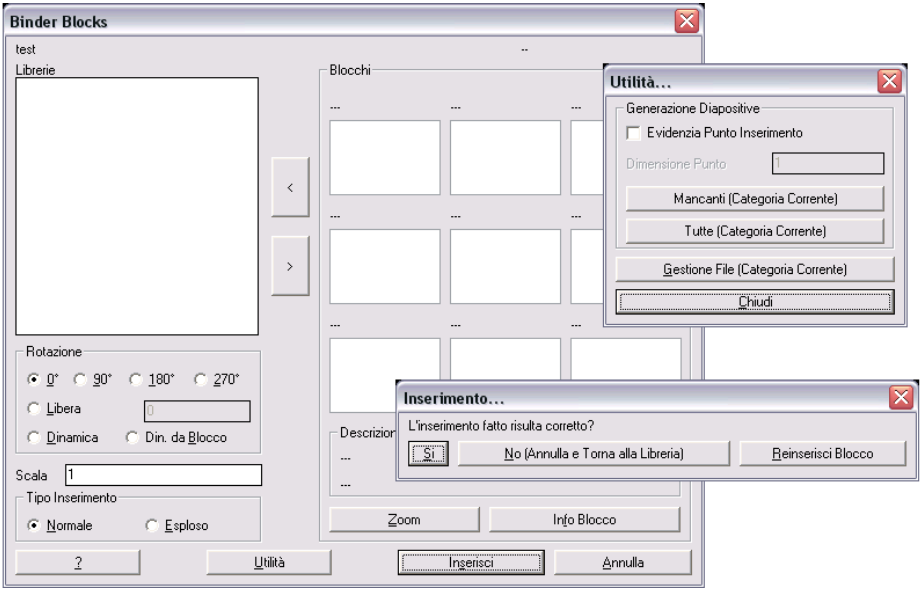
Una volta appreso il linguaggio Lisp e iniziato lo sviluppo dei primi applicativi, ci si renderà presto conto delle sue limitazione nell'interazione con l'utente. Infatti l'unica possibilità che il linguaggio fornisce è rappresentata dai messaggi visualizzati sulla linea di comando o da una semplice finestra di dialogo con un solo tasto per la sua chiusura (richiamata dalla funzione 'alert').

Tutti gli applicativi ed in particolare i moderni software, hanno sempre un'interfaccia a finestre, il più possibile intuitiva e semplice. Questa tendenza rende i programmi che comunicano tramite linea di comando, poco attraenti e di difficile utilizzo, proprio per la mancanza di abitudine che l'utente ha verso questo tipo di interfaccia.

Anche lo stesso ambiente CAD è indirizzato verso l'interfaccia a finestre, infatti molte delle sue funzionalità sono implementate all'interno di semplici dialog. Ciò è particolarmente gradito agli utenti alle prime armi e semplifica la vita a tutti gli altri (programmatori compresi).

Questa sezione ci mostrerà come il linguaggio Lisp, consente al programmatore la creazione e l'utilizzo di maschere di dialogo piacevoli e intuitive.

Di seguito ecco alcuni esempi di finestre di dialogo generate da programmi Lisp:



Queste maschere sono state generate grazie all'utilizzo delle DCL. Una DCL non è altro che una finestra di dialogo pilotata da Lisp o da C/C++.

Cosa sono le DCL e qual è la loro struttura

L'acronimo DCL significa 'Dialog Control Language' ed esprime in maniera abbastanza chiara la natura di questo sistema.

Al contrario di molti altri linguaggi come Visual Basic (VBA), le DCL non vengono disegnate in maniera visuale ma vengono scritte. Infatti una DCL, altro non è che un file di testo scritto utilizzando un apposito linguaggio (solitamente i file scritti con tale linguaggi hanno estensione .DCL).

Prima di poter scrivere una finestra di dialogo, è bene comprendere come questa sia strutturata. In realtà, al contrario di quanto si possa pensare, una finestra non è altro che un contenitore, all'interno del quale vengono posizionati, in maniera libera, i vari controlli (pulsanti, caselle di testo ecc), nel nostro caso esistono delle regole precise per il posizionamento di questi ultimi.

La prima regola che si deve tenere sempre presente è che i controlli inseribili all'interno di una finestra di dialogo sono di due tipi:

1. Controlli contenitore. Sono utilizzati per raggruppare altri controlli, e nella maggioranza dei casi servono per stabilirne il posizionamento;
2. Controlli standard. Rappresentano i normali controlli su cui l'utente agirà. Sono di questo tipo, ad esempio, le caselle di testo, i pulsanti, le liste, le immagini ecc...
3. Vediamo immediatamente un esempio pratico di una semplice finestra di dialogo scritta utilizzando le DCL:

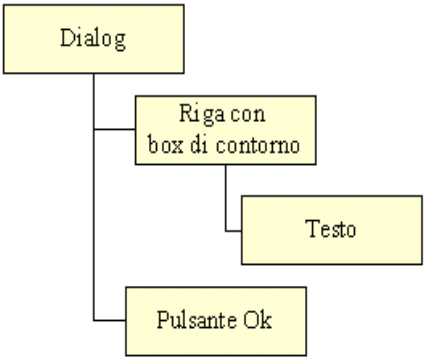
In questo caso siamo di fronte ad una finestra, nella quale sono inseriti tre oggetti:

1. Una cornice che racchiude il testo;
2. Un'etichetta che visualizza una semplice linea di testo;
3. Un pulsante.

Teniamo presente che la cornice non è un normale controllo, bensì un controllo detto contenitore, infatti al suo interno è stata posizionata l'etichetta per visualizzare il

messaggio.

Rappresentata gerarchicamente la precedente finestra di dialogo assume questo aspetto:



Scrivere una DCL. Controlli, Attributi, Valori

Vediamo ora come scrivere la DCL presentata nella precedente sezione.

Prima di tutto ecco la definizione di una finestra di dialogo:

```
CiaoMondo: dialog {
label = "Salve Mondo";
...
}
```

Queste tre semplici righe definiscono una finestra di dialogo che, all'interno del programma Lisp, sarà chiamata 'CiaoMondo' e, una volta aperta, avrà titolo 'Salve Mondo'. Ovviamente al posto dei ... andremo ad inserire le definizioni degli oggetti presenti all'interno della finestra.

Aggiungiamo ora la cornice:

```
CiaoMondo: dialog {
label = "Salve Mondo";
      : boxed_row {
      ...
      }
...
```

```
}
```

La nostra cornice è rappresentata dall'oggetto 'boxed_row', il quale, conterrà altri controlli (nel nostro caso del testo). Esistono due tipi di cornice rappresentati da 'boxed_row' e da 'boxed_column', l'unica differenza fra i due è rappresentata dal fatto che il primo allinea i controlli al suo interno uno di fianco all'altro, mentre il secondo li posiziona uno sotto l'altro.

Per concludere la nostra dialog inseriamo la linea di testo e il pulsante:

```
CiaoMondo: dialog {
label = "Salve Mondo";
      : boxed_row {
            : text {
                  label = "Buon giorno, sono una DCL.:) ";
            }
      }
ok_only;
}
```

Come si può notare, il testo visualizzato all'interno della finestra è rappresentato dall'oggetto 'text' e il suo contenuto da 'label', infine il pulsante che chiuderà la finestra è rappresentato dall'oggetto 'ok_only'.

Analizzando la dialog scritta, possiamo individuare diversi tipi di elementi:

1. Gli oggetti che verranno visualizzati nella finestra (es.: text, boxed_row, ecc...);
2. Le caratteristiche degli oggetti visualizzati (es.: label);
3. Le parentesi graffe che delimitano gli oggetti;
4. Il carattere ; utilizzato come terminatore di linea.

Per identificare gli elementi 1) e 2), il linguaggio DCL utilizza nomi specifici. Nel primo caso vengono chiamati 'tile', nel secondo 'attributi'. Da ora per riferirci agli uni o agli altri utilizzeremo anche noi questa terminologia. Per quanto invece riguarda gli oggetti indicati al punto 3) questi servono per delimitare la definizione di un qualsiasi tile.

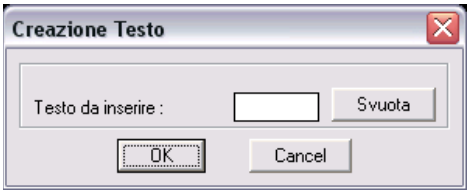
Infine non vanno dimenticati i 'valori'. Un valore, in pratica, non è altro che il contenuto di un attributo. Ad esempio il valore dell'attributo 'label' del tile 'text', presente nella DCL appena realizzata, è "Buon giorno, sono una DCL.:) ".

Un'ultima nota riguarda il tile principale che realizza la finestra, 'dialog'. Questo tile è l'unico a disporre di un nome, utilizzabile poi all'interno dei programmi Lisp e specificato prima dei due punti (:) .

Utilizzare le DCL. Visualizzazione, modifica attributi, eventi

Scopo di questa sezione è quello di apprendere come utilizzare, tramite il linguaggio Lisp, le finestre di dialogo, al fine di semplificare l'interazione con l'utente.

Prima di tutto diamo un'occhiata a come dovrà essere la nostra finestra di dialogo.



Quello che si vuole realizzare è un semplice comando che permetta all'utente di specificare un testo, il quale, alla pressione di un apposito tasto ('Ok') verrà inserito nel disegno corrente come entità di tipo testo su singola linea assegnandogli il colore rosso.

Nella finestra di dialogo, sarà presente anche un pulsante per la cancellazione del testo digitato.

Iniziamo col creare il file 'esempio1.dcl', che conterrà la definizione della nostra finestra di dialogo (per crearlo è sufficiente utilizzare 'notepad').

Apriamo il file in questione e iniziamo ad inserire la definizione della finestra, il cui nome sarà 'testorosso':

```
testorosso: dialog {
}
```

E' importante notare che il nome assegnato ('testorosso') sarà poi utilizzato all'interno del programma di gestione Lisp per accedere alla finestra, senza il quale, la nostra dialog sarebbe inutilizzabile.

Aggiungiamo ora la proprietà 'label' per assegnare il titolo alla nostra finestra:

```
testorosso: dialog {
      label = "Creazione Testo";
}
```

Passiamo ad inserire il bordo che conterrà poi la casella di testo ('edit_box') ed il pulsante per svuotarla ('button'):

```
testorosso: dialog {
```

```
      label = "Creazione Testo";
      : boxed_row {
      }
}
```

All'interno della nostra 'boxed_row' andiamo a posizionare la nostra 'edit_box':

```
testorosso: dialog {
      label = "Creazione Testo";
      : boxed_row {
            : edit_box {
            }
      }
}
```

Dato che all'interno della procedura Lisp di gestione bisognerà accedere alla 'edit_box', gli assegneremo un nome utilizzando l'apposita proprietà 'key', la quale può essere utilizzata in tutti i 'tile' che devono essere controllati dal programma Lisp. A questa aggiungeremo anche la proprietà 'label' in modo che, alla sinistra della 'edit_box', compaia la sua descrizione:

```
testorosso: dialog {
label = "Creazione Testo";
      : boxed_row {
            : edit_box {
                  key = "txt";
                  label = "Testo da inserire: ";
            }
      }
}
```

Passiamo ora ad inserire il pulsante accanto alla 'edit_box'. Questo tasto avrà la semplice funzione di cancellare il contenuto della stessa. Tenendo presente che tale elemento ('button') viene inserito all'interno di una 'row' (in questo caso una 'boxed_row'), il programma lo visualizzerà sulla stessa riga della 'edit_box', subito alla sua destra:

```
testorosso: dialog {
label = "Creazione Testo";
      : boxed_row {
            : edit_box {
                  key = "txt";
                  label = "Testo da inserire: ";
```

```
            }
            : button {
                  key = "svuotatxt";
                  label = "Svuota";
            }
      }
}
```

Anche in questo caso abbiamo utilizzato la proprietà ’key’ per assegnare un nome al pulsante, in modo da consentire al programma che gestirà la finestra di accedervi, inoltre viene utilizzata la proprietà ’label’ per inserire un testo descrittivo all’interno del ’button’.

Per terminare la nostra dialog manca l’inserimento dei tasti ’Ok’ ed ’Annulla’. Questi pulsanti, come diversi altri (vedi manuali ufficiali), sono definiti da specifici tile. I gruppi di pulsanti più importanti ed utilizzati sono:

ok_only, rappresenta il tasto ’Ok’, la cui ’key’ è preassegnata e vale ’Accept’;

ok_cancel, aggiunge al tasto ’Ok’ il tasto ’Annulla’, il quale ha una ’key’ che vale ’Cancel’.

Una cosa fondamentale da sapere, è che una dialog deve avere, al suo interno, almeno un pulsante con ’key’ uguale a ’Accept’ oppure ’Cancel’. All’interno dei software Lisp di gestione per fare riferimento a tali pulsanti, è sufficiente utilizzare tali ’key’.

Ecco infine la nostra DCL completa:

```
testorosso: dialog {
label = "Creazione Testo";
      : boxed_row {
            : edit_box {
                  key = "txt";
                  label = "Testo da inserire: ";
            }
            : button {
                  key = "svuotatxt";
                  label = "Svuota";
            }
      }
ok_cancel;
}
```

L’inserimento dei pulsanti ’Ok’ e ’Annulla’ all’esterno della ’boxed_row’ posizionerà questi ultimi nella parte inferiore della finestra. Ciò avviene poichè l’allineamento di

default all'interno di una dialog è per 'colonna', di conseguenza gli oggetti al suo interno vengono disposti, normalmente, uno sotto l'altro.

Passiamo ora alla scrittura del programma Lisp di gestione. Prima di tutto definiamo il comando che gestirà la nostra finestra:

```
(defun c:InserisciTesto ( / testo dcl_id)
)
```

Come si nota il comando definito è 'InserisciTesto' e sono già state predisposte alcune variabili locali che utilizzeremo.

La gestione di una DCL avviene solitamente in 5 passaggi:

1. caricamento del file .DCL contenente la definizione, della o delle dialog da visualizzare. Questa operazione avviene utilizzando la funzione 'load_dialog';
2. utilizzo della funzione 'new_dialog' per inizializzare la finestra richiesta. Tale operazione è indispensabile, in quanto all'interno di un unico file .DCL possono essere definite più finestre di dialogo (dialog);
3. inizializzazione dei tile presenti all'interno della finestra;
4. visualizzazione della dialog. Questa operazione avviene tramite la funzione 'start_dialog' che rende la finestra disponibile all'operatore;
5. scaricamento del file .DCL aperto. Tale operazione è richiesta al termine del programma, una volta che tutte le finestre utilizzate sono state chiuse.

Inseriamo ora il codice necessario affinché la nostra finestra venga visualizzata:

```
(defun c:InserisciTesto ( / testo dcl_id)
      (setq dcl_id (load_dialog "esempio1.dcl"))
      (if (not (new_dialog "testorosso" dcl_id))
            (exit)
      )
(start_dialog)
(unload_dialog dcl_id)
(prin1)
)
```

Esaminando il listato si noterà come, prima viene caricato il file .DCL (load_dialog), poi viene inizializzata la dialog (new_dialog), quindi viene visualizzata (start_dialog), ed infine viene scaricato il file .DCL caricato (unload_dialog) tramite la variabile dcl_id che ne contiene l'identificativo.

Si tenga presente che, una volta visualizzata la finestra, il programma si ferma alla riga contenente la funzione 'start_dialog' e non avanza finchè la finestra non viene chiusa dall'operatore. La chiusura di una dialog può avvenire in due modi. Il primo è

rappresentato dalla pressione del tasto 'X' presente nell'angolo in alto a destra, che corrisponde alla pressione del tasto con key pari a 'Cancel', in alternativa la finestra può essere chiusa premendo un tasto appositamente adibito dal programma a compiere tale operazione.

Nel codice presentato non viene definito cosa accadrà quando l'utente cliccherà sul tasto 'Ok' (key pari a 'Accept') o su 'Annulla' (key uguale a 'Cancel'). In questo caso il comportamento di default prevede che automaticamente, alla pressione di uno dei due, la finestra si chiuda, permettendo al programma di continuare.

Per completare la nostra procedura manca il codice per inserire il testo e, cosa più importante, bisogna capire quale pulsante l'operatore premerà. Infatti premendo 'Ok' il programma disegnerà l'entità richiesta, premendo 'Annulla' oppure cliccando sulla 'X' della finestra, il programma non compirà alcuna operazione.

Vediamo subito la soluzione:

```
(defun c:InserisciTesto ( / testo dcl_id tastoPremuto)
(setq dcl_id (load_dialog "esempio1.dcl"))
      (if (not (new_dialog "testorosso" dcl_id))
            (exit)
      )
(setq tastoPremuto (start_dialog))
(if (= tastoPremuto 1)
      (progn
      ...
      )
)
(unload_dialog dcl_id)
(prin1)
)
```

Prima di tutto viene memorizzato, nella variabile 'tastoPremuto' il valore che identifica il pulsante che l'operatore ha premuto. La funzione 'start_dialog' oltre a visualizzare la finestra di dialogo restituisce un valore 0 nel caso in cui l'utente prema 'Cancel' e 1 nel caso prema 'Accept'.

A questo punto manca solamente il codice per il disegno del testo che posizioneremo al posto di '...' . Per poterlo fare, occorre sapere cosa l'utente ha digitato all'interno della 'edit_box'.

La funzione 'get_tile' consente di conoscere lo stato di un qualsiasi tile, ad esempio il contenuto di una 'edit_box', l'elemento selezionato in una 'list_box' e così via. Questa funzione deve però essere utilizzata mentre la finestra è ancora aperta, quindi non sarà possibile inserirla dopo l'esecuzione di 'start_dialog', in quanto, la finestra è, in

quel punto, già chiusa.

Nel nostro caso l'operazione di lettura deve avvenire quando l'operatore preme il tasto 'Accept'. La funzione 'action_tile' permette di associare, ad un evento, una serie di operazioni. Nel caso di un pulsante, questa funzione, ci consente di intercettare la sua pressione e quindi, ci permette di compiere una qualsiasi operazione, prima che la finestra venga chiusa.

Si tenga sempre presente che, l'uso di 'action_tile' per definire le operazioni da compiere alla pressione dei tasti predefiniti (es.: accept, cancel ecc..), sostituisce il comportamento standard. Nel caso venga premuto il nostro 'accept', il valore ritornato dalla funzione 'start_dialog' non sarà 1 (come solitamente accade) bensì sempre 0, questo comportamento ci costringe a chiudere la finestra indicando il valore di ritorno per 'start_dialog'. Tale valore deve essere specificato come parametro a 'done_dialog' (vedi esempio).

Utilizziamo subito la funzione 'action_tile' per leggere i valori dei tile presenti, prima che la finestra si chiuda:

```
(defun c:InserisciTesto ( / testo dcl_id tastoPremuto)
(setq dcl_id (load_dialog "esempio1.dcl"))
      (if (not (new_dialog "testorosso" dcl_id))
            (exit)
      )
(action_tile
      "accept"
      "(setq testo (get_tile \"txt\")) (done_dialog 1)")
      (setq tastoPremuto (start_dialog))
      (if (= tastoPremuto 1)
            (progn
            ...
            )
      )
(unload_dialog dcl_id)
(prin1)
)
```

Si ricordi sempre che, la funzione 'action_tile' richiede due parametri, nel primo viene specificato il nome del tile su cui agire, il secondo contiene il codice Lisp da eseguire. Proprio parlando delle istruzioni Lisp utilizzate con questa funzione è fondamentale utilizzare la coppia \" al posto del semplice ".

Giunti a questo punto, manca solamente l'inserimento del codice per la cancellazione della casella di testo tramite il tasto 'Svuota', il disegno del testo indicato e

l'impostazione del suo colore:

```
(defun c:InserisciTesto ( / testo dcl_id tastoPremuto)
(setq dcl_id (load_dialog "esempio1.dcl"))
      (if (not (new_dialog "testorosso" dcl_id))
            (exit)
      )
(action_tile "svuotatxt" "(set_tile \"txt\" \"\")")
(action_tile
"accept"
"(setq testo (get_tile \"txt\")) (done_dialog 1)")
(setq tastoPremuto (start_dialog))
      (if (= tastoPremuto 1)
            (progn
            (command "_text" pause "2" "0" testo)
            (command "_change" "_last" "" "_p" "_c" "_red"
"")
            )
      )
(unload_dialog dcl_id)
(prin1)
)
```

Per svuotare la casella di testo 'txt', è sufficiente utilizzare la funzione 'set_tile', la quale permette di impostare la proprietà 'value' di un qualsiasi tile. Nel caso della nostra 'edit_box' ciò si traduce nella scrittura di un testo al suo interno, ovviamente, per svuotarla, imposteremo il testo pari ad una stringa vuota (set_tile "txt" "").

Per quanto riguarda l'inserimento del testo nel disegno, viene dato per scontato che lo stile di testo corrente abbia altezza 0, inoltre l'operatore dovrà indicare il punto di inserimento.

Come si può notare dall'esempio fatto in questa sezione, l'utilizzo delle DCL non è complesso, l'unico vero limite è dato dal numero ridotto di controlli (tile) disponibili che non prevedono, ad esempio, l'uso di viste ad albero (come quelle utilizzate in Windows da 'gestione risorse') o l'utilizzo di altri controlli avanzati previsti normalmente dal sistema operativo.

Nella successiva sezione troveremo l'elenco dei tile disponibili. Ciò ci permetterà di verificare le potenzialità e i limiti del linguaggio DCL.

Utilizzare le DCL. Realizzare una progress bar

Iniziamo definendo, con precisione, cosa si intende quando si parla di “progress bar”. Questo oggetto, di tipo grafico, è quella che in italiano viene chiamata “barra di progresso” ed ha in compito di mostrare, all’utente, lo stato di avanzamento di una procedura automatica particolarmente lunga.

Ecco un esempio :

In particolare la barra di avanzamento è quella rossa che indica lo stato della procedura correntemente in funzione.

Nel linguaggio DCL non esiste uno specifico componente in grado di assolvere al compito di una “progress bar”, ciò significa che, per ottenerla, dovremo simularla, utilizzando un altro tipo di oggetto.

Nel nostro esempio utilizzeremo un controllo di tipo “image”. Questo tipo di oggetto, normalmente, è studiato per accogliere una immagine ma noi, sfruttando la capacità del linguaggio di riempire l’area di una “image” con un colore uniforme, andremo a simulare il comportamento di una barra di avanzamento.

Iniziamo osservando il codice della nostra DCL :

```
progress : dialog {
      label = "DCL Progress Bar ";
      : row {
         : button {
        key = "val100";
        label = "100";
         }
         : button {
        key = "val1k";
        label = "1.000";
         }
         : button {
        key = "val10k";
```

```
        label = "10.000";
         }
         : button {
        key = "val100k";
        label = "100.000";
         }
         : button {
        key = "val1m";
        label = "1.000.000";
         }
      }
      : image {
         key = "progressbar";
         fixed_width  = 100;
         height = 1;
      }
      : row {
         : button {
        label = "OK";
        key = "ok";
        is_cancel = true;
         }
      }
} //enddialog
```

In questa maschera vengono definiti solo alcuni tasti e un oggetto di tipo “image” che sarà il fulcro di tutto il sistema.

I primo quattro “button” avranno il compito di stabilire la velocità di avanzamento della barra, impostandone la “dimensione” virtuale, da 0 a X.

L'ultimo tasto avrà il compito di chiudere la finestra.

Ma veniamo al codice sorgente che sta dietro il funzionamento della nostra barra di avanzamento :

```
;comando PROGB che mostra il funzionamento
;di una progress bar utilizzata ll'interno di una
;DCL
(defun c:progb ( / dcl_id x y)
      (setq dcl_id (load_dialog "progress.dcl"))
```

```
      (new_dialog "progress" dcl_id)
      (setq x (dimx_tile "progressbar"))
      (setq y (dimy_tile "progressbar"))
      (start_image "progressbar")
      (fill_image 0 0 x y -15)
      (end_image)

      (action_tile "val100" "(execute 100)")
      (action_tile "val1k" "(execute 1000)")
      (action_tile "val10k" "(execute 10000)")
      (action_tile "val100k" "(execute 100000)")
      (action_tile "val1m" "(execute 1000000)")
      (action_tile "cancel" "(done_dialog)")
      (start_dialog)
      (unload_dialog dcl_id)
);Enddef

;mostra la progressbar in funzione
; maxVal è il numero di incrementi che subirà
; la progress bar
(defun execute ( maxVal / x y counter col)
     ;ricava le dimensioni della zona che ospiterà
l'avanzamento
      ;della progress bar
      (setq x (dimx_tile "progressbar"))
      (setq y (dimy_tile "progressbar"))
      (start_image "progressbar")
      ;riempie la progress bar con il colore di fondo
      ;delle finestre del sistema
      (fill_image 0 0 x y -15)
      (end_image)
      (setq counter 0)
      (while (<= counter maxVal)
             (start_image "progressbar")
             ;colora la zona interna della progress bar
               (fill_image 0 0 (/ (* counter x) maxVal) y 1)
             (end_image)
             (setq counter (1+ counter))
```

```
      )
);Enddef
```

Il programma è composto solamente da due funzioni. La prima è il comando vero e proprio che l'utente utilizzerà, “PROGB”, mentre la seconda è la funzione che, realmente, gestisce l'avanzamento della barra di progresso, “EXECUTE”.

All'interno del comando “PROGB” è presente solo il codice che gestisce le operazioni che l'operatore può compiere sulla dcl, in particolare bisogna soffermarci sui 5 tasti che azionano la barra di avanzamento. Prendiamo, ad esempio, questa linea di codice .

```
(action_tile "val10k" "(execute 10000)")
```

questa istruzione dice al programma di eseguire la funzione “EXECUTE” passandole un valore “10000”, quando l'utente cliccherà sul tasto “VAL10K”.

“EXECUTE” infatti, è la responsabile della gestione della barra di avanzamento. Osserviamo meglio cosa contiene :

```
(defun execute ( maxVal / x y counter col)
      (setq x (dimx_tile "progressbar"))
      (setq y (dimy_tile "progressbar"))
      (start_image "progressbar")
      (fill_image 0 0 x y -15)
      (end_image)
      (setq counter 0)
      (while (<= counter maxVal)
            (start_image "progressbar")
            (fill_image 0 0 (/ (* counter x) maxVal) y 1)
            (end_image)
            (setq counter (1+ counter))
      )
);Enddef
```

Inizialmente vengono ricavate le dimensioni X e Y della progress bar presente nella finestra, dopo di che, questa viene inizializzata e riempita con il colore dello sfondo della finestra corrente (valore -15).

Infine, abbiamo il fulcro della procedura, il ciclo “WHILE”, nel quale viene riempita progressivamente la barra di avanzamento incrementando, via via, l'area colorata (in rosso, con valore 1). Naturalmente, il ciclo farà un numero di interazioni pari al parametro passato alla funzione, “MAXVAL”.

Volendo applicare questa tecnica ad una procedura reale, sarà sufficiente inserire le operazioni da svolgere all'interno di un ciclo che riempie progressivamente l'immagine adibita a barra di avanzamento.

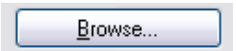
L'unico requisito indispensabile a questo sistema è quello di conoscere il numero esatto di interazioni necessarie al ciclo per terminare.

I Tile previsti dal linguaggio DCL

Di seguito vengono riportati tutti i tile di cui dispone il linguaggio DCL. Si tenga presente che ognuno di questi ha specifiche proprietà e modalità di utilizzo. Visto lo scopo di questo testo, per i dettagli su proprietà e modalità di utilizzo si rimanda alla documentazione ufficiale dell'ambiente CAD utilizzato.

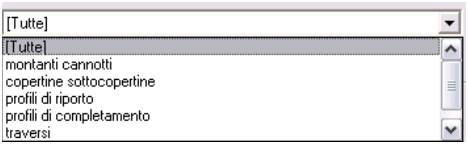
Iniziamo col visionare i tile principali per la definizione delle DCL:

- **button**



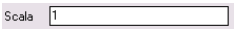
  Il classico pulsante, che è il tile base per consentire all'operatore l'esecuzione di comandi all'interno delle dialog.

- **popup_list**



  Una semplice lista a comparsa, che permette la selezione di una delle voci presenti nella stessa.

- **edit_box**



  Casella di testo nella quale l'operatore potrà digitare testi a linea singola.

- **radio_button**

  Elemento base per la scelta di un'opzione esclusiva all'interno di un gruppo.

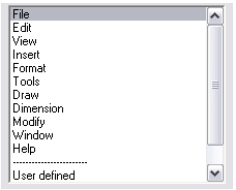
- **image_button**

  Questo tile consente l'utilizzo di immagini (diapositive .sld) sulle quali l'operatore potrà cliccare come fosse un pulsante.

- **slider**

  Lo slider consente l'utilizzo di una barra di scorrimento utilizzabile per la scelta di dati numerici interni ad un gruppo di valori.

- **list_box**



  Una semplice lista, che permette sia la selezione di una singola voce, sia la selezione multipla di più elementi.

- **toggle**

  La classica casella di scelta, utilizzata normalmente per attivare o disattivare le opzioni proposte dalle dialog.

Il secondo gruppo di tile è rappresentato da quelli che consentono il raggruppamento, al loro interno di altri tile, in modo da controllare sia il raggruppamento logico delle opzioni sia la struttura della dialog:

- **column, boxed_column**
  Permettono l'incolonnamento di più tile, con la possibilità di racchiuderli all'interno di un bordo con titolo.

- **dialog**
  In questo caso si tratta del tile primario, utilizzato per racchiudere un'intera finestra di dialogo.

- **radio_column, boxed_radio_column**
  Permettono l'incolonnamento di più tile di tipo 'radio_button', con la possibilità di racchiuderli all'interno di un bordo con titolo, che permette la creazione di un gruppo ad esclusione automatica.

- **radio_row, boxed_radio_row**
  Permette l'affiancamento di più tile di tipo 'radio_button', con la possibilità di racchiuderli all'interno di un bordo con titolo, che permette la creazione di un gruppo ad esclusione automatica.

- **row, boxed_row**

Permette l'affiancamento di più tile, con la possibilità di racchiuderli all'interno di un bordo con titolo.

Di seguito ecco i tile di tipo informativo e decorativo:

- **image**
  Semplice area rettangolare, all'interno della quale visualizzare una immagine.

- **spacer_0**
  Permette l'inserimento di uno spazio che si espande orizzontalmente o verticalmente a seconda del contesto nel quale si trova.

- **text**
  Semplice area di testo per la visualizzazione di informazioni sulle dialog.

- **spacer_1**
  Permette l'inserimento di uno spazio con altezza e larghezza proporzionali.

- **spacer**
  Rappresenta un semplice spazio, modificabile a piacere, inseribile tra due tile.

Esistono poi i tile adibiti alla concatenazione di parti testuali:

- **concatenation**
  Permette la concatenazione di più tile contenenti testo, i quali vengono dinamicamente modificati da programma.

- **text_part**
  Consente l'inserimento di grossi testi all'interno delle dialog.

- **paragraph**
  Dispone su singola colonna più elementi di tipo testuale ('text_part' o 'concatenation').

Infine i tile relativi ai pulsanti predefiniti e ai messaggi di errore:

- **errtile**
  Inserisce un'area testuale, adibita a contenere gli eventuali avvertimenti o messaggi di errore.

- **ok_cancel_help**
  Rappresenta l'unione dei pulsanti 'Ok', 'Annulla', 'Aiuto'.

- **ok_only**
  Rappresenta il solo pulsante 'Ok'.

- **ok_cancel_help_errtile**

Rappresenta l’unione dei pulsanti ’Ok’, ’Annulla’, ’Aiuto’ con l’aggiunta dell’area testuare per i messaggi di errore.

- **ok_cancel**
  Rappresenta l’unione dei pulsanti ’Ok’ e ’Annulla’.

- **ok_cancel_help_info**
  Rappresenta l’unione dei pulsanti ’Ok’, ’Annulla’, ’Aiuto’, ’Informazioni’.

Capitolo 11
Gestione degli Errori

Nonostante il CAD abbia una propria gestione automatica per gli errori che possono verificarsi durante le procedure Lisp, in software di media e grande complessità risulterà molto utile poter introdurre una gestione alternativa di eventi non previsti.

Le funzioni di gestione

Il linguaggio Lisp, sia in AutoCAD quanto in progeCAD o IntelliCAD, prevede alcune funzioni per l'intercettazione degli errori che possono avvenire durante l'esecuzione di un programma.

Esistono due categorie di funzioni per la gestione dell'errore:

1. funzioni informative, per comunicare eventuali avvertimenti all'operatore;
2. funzioni per l'intercettazione degli stati di errore, in modo da poter intervenire sul comportamento standard dell'ambiente CAD.

Nella prima categoria troviamo la sola funzione 'alert' che, come visto nei capitoli precedenti, consente di visualizzare una piccola finestrella, nella quale è possibile inserire un messaggio:

```
(alert "messaggio di testo")
```

Con 'alert' è possibile utilizzare i caratteri '\n' per inserire dei testi su più linee:

```
(alert "messaggio di testo\nsu più righe")
```

L'unica limitazione alla quantità di testo utilizzabile con questa funzione è data dalle impostazioni della modalità video.

Alla seconda categoria appartengono invece:

- **(*error* stringa)**. Funzione per l'intercettazione dei messaggi di errore
- **(exit)**. Interrompe immediatamente l'applicazione corrente
- **(quit)**. Interrompe immediatamente l'applicazione corrente
- **(vl-catch-all-apply 'funzione lista)**. Evita il verificarsi di qualsiasi errore, ritornando un 'oggetto errore'
- **(vl-catch-all-error-message oggetto-errore)**. Ritorna l'adeguato messaggio da un 'oggetto errore'
- **(vl-catch-all-error-p arg)**. Determina se l'argomento passato è un 'oggetto errore' ritornato da 'vl-catch-all-apply'

Attenzione: le funzioni 'vl-catch-all-apply', 'vl-catch-all-error-message' e 'vl-catch-all-

error-p' sono presenti solo a partire dalla versione 2000 di AutoCAD, mentre progeCAD e IntelliCAD NON le supportano.

La funzione *error*

error è il principale sistema di cui Lisp dispone per intercettare e gestire gli errori. Per utilizzarla è necessario definirla, infatti di default vale nil.

Quando *error* non è definita, ad ogni errore di programma, il CAD si interrompe visualizzando l'appropriato messaggio di avvertimento, ad esempio eseguendo:

```
(itoa "a")
```

viene restituito il messaggio:

```
(in AutoCAD)
;errore: tipo di argomento errato: fixnump: "a"
(in progeCAD/IntelliCAD)
error: bad argument type
```

Tutti gli errori gestiti possiedono un codice numerico. Ogni qual volta si verifica un errore, viene stampato il messaggio di avvertimento e viene memorizzato il suo valore numerico nella variabile di sistema ERRNO (per l'elenco dei possibili valori si veda più avanti in questo capitolo).

Definendo la funzione *error* ci si deve ricordare che questa accetta un parametro, il quale conterrà il messaggio standard di errore. Osserviamo questo spezzone di codice:

```
(defun *error* ( msg / noerr)
      (setq noerr (getvar "ERRNO"))
      (alert (strcat "\n" msg ": " (itoa noerr)))
)
```

Una volta caricata questa funzione, per ogni errore che il CAD incontrerà, il messaggio di avvertimento verrà visualizzato in una piccola finestra e non sulla riga di comando (come normalmente avviene). Ad esempio digitando:

```
(itoa "a")
```

verrà mostrato il messaggio:

In pratica, la nostra funzione riceve i messaggi di errore, li filtra, e gestisce il modo in cui questi vengono trattati.

E' ovvio ricordare che all'interno di un programma esiste una sola *error*. Per evitare problemi è sempre bene, prima di ridefinirla, memorizzare la precedente *error*, in modo da poterla riattivare o richiamare al temine delle nostre procedure. Facciamo un esempio:

```
(setq error_originale *error*) ;salva la funzione corrente
(defun *error* ( msg / noerr) ;ridefinisce *error*
      (setq noerr (getvar "ERRNO"))
      (alert (strcat "\n" msg ": " (itoa noerr)))
)
```

Per poter ripristinare il gestore originale:

```
(setq *error* error_originale)
```

Inoltre si tenga presente il fatto che, è possibile richiamare il gestore originale, utilizzando la sua copia memorizzata:

```
(error_originale msg)
```

Intercettare gli errori senza interruzioni

Al verificarsi di un qualsiasi errore, l'ambiente CAD, interrompe l'esecuzione del programma. Anche utilizzando *error*, questo comportamento non può essere evitato.

A partire però dalla release 2000 di AutoCAD, è possibile utilizzare le funzioni 'vl-catch-all-apply', 'vl-catch-all-error-message' e 'vl-catch-all-error-p' per consentire la prosecuzione delle routine Lisp, anche in presenza di errori.

Prendiamo ad esempio questa linea:

```
(setq res (/ 100 a))
```

Nel caso in cui la variabile 'a' valga 0, il programma sarebbe interrotto e verrebbe segnalato l'errore di 'divisione per 0'. Per impedire che avvenga l'interruzione è possibile richiamare la funzione '/' tramite 'vl-catch-all-error-message'. Quest'ultima

accetta due argomenti, il primo è la funzione da eseguire, il secondo è la lista degli argomenti da passare alla stessa. Riscrivendo la riga precedente:

```
(setq res (vl-catch-all-apply '/ '(100 a)))
```

Se la divisione va a buon fine in 'res' verrà inserito il risultato, in caso contrario verrà restituito uno pseudo codice di errore che potrà essere utilizzato per conoscerne la causa.

In questo modo, nel caso in cui 'a' valga 0, il programma non verrà interrotto.

'vl-catch-all-error-message' non restituisce un codice immediatamente utilizzabile. Prima di tutto bisogna stabilire se 'res' contenga o meno la segnalazione di errore, in questo modo:

```
(if (vl-catch-all-error-p res)
      (progn
            (alert "Errore Rilevato!")
      );Endp
);Endif
```

Naturalmente è anche possibile trasformare questo pseudo codice nel relativo messaggio di errore, utilizzabile nei messaggi. Questo è consentito dalla funzione 'vl-catch-all-error-message' che, dato un codice di errore, lo trasforma nel relativo messaggio. In questo modo:

```
(if (vl-catch-all-error-p res)
      (progn
            (alert (vl-catch-all-error-message res))
      );Endp
);Endif
```

Per capire meglio, di seguito è riportata una semplice funzione che esegue la divisione tra due numeri. In questo caso, sfruttando le tecniche appena apprese, se la divisione non risultasse possibile restituirebbe nil, visualizzando un appropriato messaggio di errore:

```
(defun dividi (valore1 valore2 / res)
(setq res (vl-catch-all-apply '/ (list valore1 valore2)))
(if (vl-catch-all-error-p res)
      (progn
            (alert (vl-catch-all-error-message res))
            (setq res nil)
      );Endp
);Endif
res
```

```
)
```

I codici di errore

L'interprete Lisp genera, quando incontra un errore una serie di valori utili alla precisa identificazione del problema. La variabile di sistema ERRNO contiene il codice dell'ultimo errore generato durante l'esecuzione delle procedure.

E' possibile controllare questa variabile utilizzando questa espressione:

```
(getvar "errno")
```

ERRNO viene posta a 0 solamente all'apertura dei disegni. Ciò significa che, per verificare con certezza, quale errore avvenga all'interno di una procedura, è bene porre a 0 ERRNO prima di eseguire la funzione da controllare.

Ogni CAD (AutoCAD/IntelliCAD/progeCAD/ecc...) possiede una proprio elenco di errori. Nonostante molti siano comuni, è molto importante riferirsi alla documentazione ufficiale del software che si sta utilizzando per comprendere l'esatto significato dei valori contenuti in ERRNO.

La valutazione degli argomenti

Il modo con cui il Lisp valuta gli argomenti passati ad una funzione, è legato direttamente alla gestione degli errori, in quanto può modificare il comportamento del software in base alla versione dell'ambiente CAD.

Lisp valuta tutti gli argomenti passati alle funzioni prima di controllare la correttezza dei tipi. Osserviamo questo codice:

```
(defun f1 ()
      (print "Funzione 1")
      "a"
)
(defun f2 ()
      (print "Funzione 2")
      23
)
(defun f3 ()
      (print "Funzione 3")
      2
)
```

Le tre funzioni ’f1’, ’f2’, ’f3’, ritornano rispettivamente, una stringa, un intero e un altro numero intero. Una volta caricate, proviamo ad eseguire:

```
(gcd (f1) (f2) (f3))
```

Lisp ci avviserà del fatto che sono stati specificato troppi argomenti, infatti la funzione ’gcd’ che ritorna il massimo comune denominatore tra due interi, richiede solo due parametri di tipo intero.

Considerando che ’f1’, non ritorna un intero ma una stringa, si può dedurre che, Lisp prima controlla la correttezza del numero di parametri e solo successivamente valuta i tipi di dato passati alle funzioni. Infatti eseguendo:

```
(gcd (f1) (f2))
```

Ci verrà segnalato il fatto che il tipo di dato passato a ’gcd’ non è corretto (in quanto ’f1’ ritorna una stringa e non un numero intero).

Altro aspetto interessante è che, comunque, sia nel primo che nel secondo caso, tutti gli argomenti verranno valutati, e ciò lo si può verificare osservando le informazioni stampate sulla linea di comando.

In pratica la riga:

```
(gcd (f1) (f2) (f3))
```

Stamperà:

```
"Funzione 1"
"Funzione 2"
"Funzione 3"
```

E subito dopo il messaggio di errore. Quindi si può concludere che, Lisp valuta tutti gli argomenti, successivamente controlla la coerenza del numero degli stessi ed infine verifica la correttezza dei tipi di dato.

Questo comportamento, una volta compreso, non crea particolari problemi. Il vero inconveniente può arrivare se si scrive software per versioni di AutoCAD precedenti alla V.2000. Infatti tutte le release, fino alla V.14 compresa, hanno un comportamento diverso.

In queste versioni, Lisp valuta uno alla volta gli argomenti passati alle funzioni, al termine di ogni singola valutazione viene controllato il tipo di dato restituito, scrivendo:

```
(gcd (f1) (f2) (f3))
```

Verrà stampato solo:

```
"Funzione 1"
```

Subito dopo verrà generato un errore, dato che ’f1’ restituisce una stringa e non un

numero intero (necessario a ’gcd’).

Questa diversità dipende proprio dal fatto che, subito dopo aver valutato il primo argomento (’f1’), viene controllato il tipo di dato restituito.

Per quanto riguarda progeCAD e IntelliCAD, il loro comportamento è, ed è sempre stato, uguale a quello delle recenti versioni di AutoCAD (dalla V.2000 in poi).

Capitolo 12
Avvio Automatico degli Applicativi

Una volta preparato il nostro programma, non rimane che caricarlo ed utilizzarlo.

La soluzione più immediata è quella di caricare il nostro software tramite il comando '_appload', oppure tramite l'utilizzo della funzione 'load' del Lisp.

Purtroppo, questi sistemi richiedono l'intervento manuale dell'operatore. Nel caso di 'load', si potrebbe pensare di utilizzarlo inserito all'interno di un'icona o in una voce di menù, in modo da richiamarlo automaticamente. Ad esempio:

```
^C^C(load "filediprogramma.lsp")[COMANDO]
```

Così facendo, prima di eseguire il comando voluto, viene caricato il file Lisp necessario al suo funzionamento.

Questo è un buon sistema per piccole applicazioni, non certo per software multifunzione che richiedono il caricamento e l'inizializzazione di molti dati, in quando il continuo utilizzo di 'load' darebbe luogo ad un degrado eccessivo delle prestazioni e ad un'esposizione eccessiva dei dettagli dell'applicazione.

Un ulteriore aspetto da non sottovalutare, legato all'uso di 'load', è l'impossibilità di digitare il comando a tastiera, prima di aver premuto, almeno una volta, la sua icona/voce di menù.

Per ovviare a questi problemi, sia AutoCAD, sia progeCAD/IntelliCAD, dispongono di meccanismi di caricamento automatico di procedure Lisp.

Nelle sezioni successive verrà spiegato come funzionano e quali vantaggi portano. Vista la notevole diversità che esiste, tra i metodi di autocaricamento di progeCAD/IntelliCAD e di AutoCAD, verranno esaminati separatamente.

Caricamento Automatico con AutoCAD

Il sistema di caricamento più semplice, e alla portata di tutti, passa attraverso l'utilizzo del comando '_APPLOAD'. Nella sua maschera è possibile modificare il 'Gruppo di Avvio', che include la lista dei file caricati all'avviamento di AutoCAD.

In aggiunta a questo meccanismo troviamo altri sistemi, più sofisticati, che fanno uso di file speciali, 'ACAD.LSP' e 'ACADDOC.LSP'. Nell'installazione standard di AutoCAD, nessuno dei due è presente, e la loro creazione è a carico esclusivo del programmatore.

Ognuno possiede una funzione specifica. 'ACAD.LSP' viene eseguito una sola volta all'avvio del CAD, e risulta molto utile quando si desidera caricare elementi 'fissi' come menù, barre di icone, file .ARX o per impostare variabili d'ambiente che agiscono indipendentemente dal disegno (Es.: ATTDIA, ATTREQ ecc...).

Diversamente 'ACADDOC.LSP' viene eseguito ogni qual volta un disegno viene aperto/creato. In tal modo è possibile inizializzare eventuali variabili relative al singolo

progetto e caricare i necessari comandi personalizzati.

Bisogna sempre ricordare che, ogni disegno, possiede un proprio spazio Lisp. Ciò significa che, definendo una variabile all'interno di un progetto, questa non è disponibile negli altri. Se ad esempio ci trovassimo in 'DISEGNO1' ed eseguissimo:

```
(setq a "test")
```

e quindi aprissimo 'DISEGNO2', ci accorgeremmo che l'istruzione:

```
(print a)
```

stampa il valore 'nil' e non, come ci potremmo aspettare, "test". Solamente ritornando in 'DISEGNO1' la variabile 'a' riassumerebbe il suo valore iniziale.

In pratica, ogni disegno possiede proprie funzioni e variabili Lisp, indipendenti e separate da quelle degli altri file aperti.

Ma torniamo ora a parlare di 'ACAD.LSP' e 'ACADDOC.LSP'. Prima di poterli utilizzare, manca ancora un particolare. Dove posizionarli in modo che AutoCAD li carichi ?

Come spesso accade, la risposta è semplice, ma non scontata.

Per utilizzare i due file è necessario posizionarli in una delle cartelle elencate nei 'Percorsi di ricerca dei file di supporto'. Per conoscere e modificare l'elenco di queste cartelle possiamo aprire la maschera delle opzioni (con il comando '_OPTIONS'), andando poi nella sezione 'file'. A questo elenco sono da aggiungere, la cartella corrente e quella del disegno aperto, sempre automaticamente e in modo trasparente, incluse durante le ricerche.

Ciò che abbiamo visto finora è il comportamento standard di AutoCAD. E' tuttavia possibile intervenire su tale comportamento per modificarlo e adattarlo alle nostre esigenze. Esistono due variabili d'ambiente molto particolari:

- ACADLSPASDOC;
- LISPINIT.

La prima, solitamente, è impostata a '0', consentendo così il caricamento di 'ACAD.LSP' solo nel primo disegno aperto in una sessione. Impostata invece a '1', indica al CAD di caricare 'ACAD.LSP' in ogni disegno che viene aperto, in questo caso lo stesso comportamento tenuto da 'ACADDOC.LSP'.

Principalmente, si deve l'esistenza di questa variabile alla necessaria compatibilità con le precedenti versioni di AutoCAD. Infatti fino alla release 14 (compresa), non era supportato 'ACADDOC.LSP' e quindi, 'ACAD.LSP' veniva caricato una volta per ogni disegno. Solo dalla versione 2000 è stato modificato il comportamento di default del CAD, supportando il nuovo file.

Per quanto invece riguarda 'LISPINIT', questa ha il compito di stabilire se debbano

esistere diverse sessioni dell'ambiente Lisp, oppure se ne debba esistere una soltanto, condividendo così variabili e funzioni, fra tutti i disegni aperti.

La sua impostazione standard è '1', ed equivale a sessioni separate, se invece la impostassimo a '0', otterremmo la singola sessione con la relativa condivisione di funzioni e variabili.

Caricamento Automatico con progeCAD/IntelliCAD

Nel caso si lavori in progeCAD o con una versione di IntelliCAD, la situazione rispetto ad AutoCAD risulta semplificata.

Qui esiste un solo file caricato automaticamente, 'ICAD.LSP'. Questo avviene all'apertura di ogni disegno e non è possibile intervenire per modificare tale comportamento.

Come per AutoCAD, anche in questo caso il file viene cercato nei 'percorsi di ricerca' impostati nella maschera delle 'opzioni' (comando '_CONFIG') dalla sezione 'Percorsi/File'.

IntelliCAD/progeCAD possiede una caratteristica importante, che non deve mai essere sottovalutata. La sessione Lisp è una soltanto, quindi variabili e funzioni sono sempre condivise e comuni a tutti i disegni aperti. Non solo, al caricamento di ogni file, sia le variabili che le funzioni vengono completamente eliminate, ed è quindi indispensabile ricaricarle tramite 'ICAD.LSP'.

La funzione S::StartUp

I file 'ACAD.LSP', 'ICAD.LSP' e 'ACADDOC.LSP' vengono caricati in memoria, ed eseguiti, prima del termine del caricamento dei disegni. Ciò comporta l'impossibilità di utilizzare la funzione 'command', il cui funzionamento non è garantito prima del completo caricamento dei file.

Per ovviare a questo inconveniente si può ricorrere alla funzione speciale 'S::StartUp'.

Se all'interno di uno dei tre file menzionati, è presente la definizione di 'S::StartUp', questa viene automaticamente eseguita non appena il disegno è completamente caricato.

Ad esempio, provando ad inserire in 'ICAD.LSP' (o 'ACADDOC.LSP'), questo codice:

```
(defun S:StartUp ( )
      (alert "Esecuzione S::StartUp!")
)
```

Si noterà la comparsa del messaggio ogni qual volta un disegno viene creato o aperto.

Caricamento intelligente in AutoCAD

AutoCAD possiede una comoda caratteristica, che consente la selettività dei caricamenti. Se digitassimo:

```
(autoload "CMDLISP" '("CMD1" "CMD2" "CMD3"))
```

AutoCAD caricherà, in maniera autonoma, il file 'CMDLISP.LSP' la prima volta che l'operatore tenterà di utilizzare i comandi 'CMD1', 'CMD2', 'CMD3'.

E' chiaro che, questo sistema, consente di non definire tutti i comandi necessari in una sola volta, risparmiando memoria e migliorando le prestazioni generali.

Capitolo 13
Le Funzioni Avanzate

Come tutti i linguaggi di programmazione, anche Lisp possiede diverse funzioni, molto potenti, ma poco utilizzate. Nonostante ciò, il loro uso può agevolare notevolmente il lavoro, contribuendo a semplificare il codice sorgente, riducendone le dimensioni ed ottimizzandone le prestazioni.

In questo capitolo vedremo alcune di queste funzioni, esaminandone la sintassi e mostrando alcuni esempi per il loro utilizzo.

La funzione Apply

Questa funzione, disponibile nel Lisp sia di AutoCAD, sia di progeCAD/IntelliCAD, si rivela di grande aiuto quando si ha a che fare con procedure che necessitano dell'elaborazione di liste di argomenti.

La sua sintassi è questa:

```
(apply 'funzione lista)
```

dove per 'lista' si intente una variabile in formato lista, la quale conterrà i valori da passare alla 'funzione' che si trova dopo l'apice.

In pratica, 'apply' esegue la funzione indicata come primo argomento, passando a quest'ultima gli elementi della lista come fossero argomenti separati.

La sua più utile applicazione si ha quando i valori da passare alla funzione non sono costanti nel tempo, ma vengono creati in maniera dinamica dal codice durante la sua esecuzione.

Facciamo un esempio concreto, utilizzando la funzione 'max' (già vista nei capitoli precedenti).

Come sappiamo, questa funzione accetta come parametri una serie di valori e restituisce il maggiore tra quelli passati.

```
(max 3 2) restituisce 3
(max 2.5 1) restituisce 2.5
```

e così via ...

Esaminiamo ora il codice seguente:

```
(defun c:c1 ()
      (max 3 2)
)
```

Se carichiamo questa funzione, e la eseguiamo digitando 'c1', otterremo come risultato il valore 3. Infatti, la funzione 'max' opera sui due valori numerici (3 e 2) che le sono stati passati come argomenti.

Questo modo di procedere presenta un inconveniente, richiede infatti che i valori da passare alla funzione siano ben definiti all'interno del sorgente Lisp cioè, il programmatore li deve conoscere in anticipo.

Capita però, che i valori non siano sempre noti al programmatore nel momento in cui il codice viene scritto. Ad esempio, quando si vuole calcolare il maggiore tra due valori immessi da tastiera.

E' chiaro che, in casi come questo, il programmatore non conosce quali valori immetterà l'utente e quindi, la funzione max, per come è strutturata, non ci è di grande aiuto dal momento che non possiamo passarle dei valori che non conosciamo (max ? ? ?).

Ecco che ci aiuta la funzione 'apply'.

Esaminiamo quanto segue:

```
(defun c:c1 ()
      (princ "\Inseriamo 2 numeri e calcoliamo il maggiore.")

      ; assegnamo i due valori con la funzione "GETINT"
      ; andrebbe bene anche la funzione "GETREAL"
      (setq primo_valore (getint "\nInserisci 1 valore...")
                  secondo_valore (getint "\nInserisci 2
valore...")
      )

      ; creiamo la lista dei valori immessi
      (setq lista_valori (list primo_valore secondo_valore))
      (setq valore_massimo (apply 'max lista_valori))

      ; ricava il valore massimo dalla funzione "APPLY"
      ; che si "applica" alla funzione "MAX" alla quale
      ; è stata passata una variabile sotto forma di lista
      ; il cui contenuto non è inizialmente conosciuto
      ; N.B. non dimenticare l'apice ' prima di max!!!
      (princ "\nIl valore maggiore è ")
      (princ valore_massimo); lo stampa a video
      (princ)
)
```

In questo caso otterremo in risposta il numero maggiore inserito dopo le richieste. Indispensabile è la presenza della lista da passare come argomento per la funzione 'max'.

Chiaramente questo è un esempio sull'uso di 'apply', e ovviamente ci si può sbizzarrire sul come utilizzarla al meglio nelle varie situazioni e con le funzioni che accettano parametri.

La Funzione Mapcar

Questa interessante funzione, permette l'esecuzione automatica di una qualsiasi funzione, una volta per ogni elemento presente all'interno di una lista data.

Ecco la sua sintassi:

```
(mapcar funzione lista1... listan)
```

In questa espressione possiamo specificare la funzione da utilizzare e una o più liste, i cui elementi saranno passati alla procedura specificata. Il valore di ritorno di 'mapcar' sarà una lista contenente i dati elaborati.

Supponiamo di avere le variabili 'a', 'b' e 'c' di cui incrementare il valore e quindi inserire i risultati in una lista:

```
(setq a (1+ a))
(setq b (1+ b))
(setq c (1+ c))
(setq lista (list a b c))
```

Usando 'mapcar' il tutto può ridursi a:

```
(setq lista (list a b c))
(setq lista (mapcar '1+ lista))
```

Al termine della procedura, la variabile 'lista' conterrà i valori iniziali incrementati di 1. Come si può vedere, i valori contenuti nella lista passata a 'mapcar' vengono elaborati uno ad uno, infine il valore delle singole elaborazioni viene restituito.

Esistono anche utilizzi particolari, uno dei quali è l'assegnazione delle variabili. Pensiamo, ad esempio, di avere una lista contenente 3 valori, e immaginiamo di volerli associare alle variabili 'a', 'b' e 'c'. Normalmente faremmo così:

```
(setq a (nth 0 listaValori))
(setq b (nth 1 listaValori))
(setq c (nth 2 listaValori))
```

Con 'mapcar' il tutto si può ridurre a:

```
(mapcar 'set '(a b c) listaValori)
```

Come si può vedere la sintesi possibile tramite questa funzione è estrema.

Un uso ancora più interessante è quello che si ha in congiunzione con la funzione 'lambda' che verrà dettagliato più avanti in questo capitolo.

Per concludere l'argomento 'mapcar', ecco un significativo esempio della sua duttilità:

```
(setq listaStringhe '("1" "10" "23" "44" "5"))
(setq listaValori (mapcar '(atoi) listaStringhe))
```

In questo caso abbiamo convertito una lista di stringhe, nella corrispondente lista di valori numerici.

La Funzione Lambda

Con 'lambda' siamo di fronte ad una funzione simile alla famosa 'defun' (già vista nei precedenti capitoli). Infatti, il compito principale di 'lambda' è quello di definire ed eseguire una funzione. Cosa, allora, la differenzia da 'defun'? La vera differenza sta nel fatto che 'lambda' è l'unica in grado di lavorare con funzioni anonime.

In sostanza, è possibile lavorare con funzioni la cui esecuzione è temporanea, create, eseguite, senza che sia necessario conservarne in memoria la definizione.

Questo consente l'utilizzo di quelle, che in altri linguaggi, vengono chiamate funzioni 'in-line', da utilizzare quando non è necessario definire una funzione indipendente, ma è comodo utilizzarne una.

Iniziamo col chiarirne la sintassi:

```
(lambda argomenti espressioni...)
```

Per capirne l'utilizzo, vediamo subito un semplice esempio. Consideriamo la conversione di un angolo da gradi in radianti. Se dovessimo scrivere una funzione per questa trasformazione, avremmo:

```
(defun GradiRadianti ( valore )
      (* pi (/ valore 180.0))
)
```

Si tenga presente che la costante 'pi' è già definita in tutti gli interpreti Lisp e contiene il valore di Pigreco.

Ora, poniamo di avere una lista contenente parecchi valori da convertire, potremmo scrivere:

```
(foreach elemento listaValoriRadianti
      (setq listaGr (cons (GradiRadianti elemento) listaGr))
)
```

Così facendo, in 'listaGr' otterremo i nostri valori convertiti in gradi. Fino a qui tutto

bene.

Il problema nasce quando il software inizia a diventare di dimensioni consistenti e la creazione di tante piccole funzioni (come 'GradiRadianti') inizia a degradare le prestazione e ad aumentare, eccessivamente, la memoria utilizzata dal programma. Nonostante questi problemi possano sembrare trascurabili, su sistemi non particolarmente prestanti o con procedure che fanno uso massiccio di grandi quantità di dati, possono divenire significativi.

La soluzione è molto semplice e si chiama 'Lambda'. Osserviamo la funzione 'GradiRadianti' modificata in modo da utilizzarla:

```
(lambda (v) (* pi (/ v 180.0)))
```

Applichiamo ora questa espressione al programma, in modo da convertire la nostra lista di valori, facendoci aiutare anche dalla funzione 'mapcar', vista in precedenza:

```
(setq listaGr
      (mapcar '(lambda (v) (* pi (/ v 180.0)))
      listaValoriRadianti)
) ;fine di setq
```

Come si può vedere, il ciclo è stato sostituito da 'mapcar', mentre la funzione di conversione è stata rimpiazzata direttamente da una 'lambda'.

Questo meccanismo ci consente di ridurre il numero delle funzioni usate nel programma, e permette di scrivere codice più sintetico ed efficiente.

La funzione Gc

Molti programmatori pensano che la gestione della memoria utilizzata dai software scritti in Lisp sia, gestita e ottimizzata, interamente ed esclusivamente dall'interprete Lisp.

Se è certamente vero che la creazione delle variabili e delle funzioni è completamente dinamica, non è affatto vero che la loro eliminazione dalla memoria sia altrettanto automatica.

Solitamente, per eliminare una variabile o una funzione dalla memoria, si utilizzano espressioni come:

```
(setq nomeVariabile nil)
(setq nomeFunzione nil)
```

Questo, apparentemente, elimina i simboli rilasciando la memoria occupata, o almeno così dovrebbe accadere. Nella realtà, non sempre l'interprete effettua questa operazione, limitandosi a rendere inaccessibili i simboli (variabili e funzioni), lasciando

così parte della memoria occupata ma non utilizzata.

Le ragioni che accompagnano questo comportamento sono diverse, e tutte da ricercare nelle procedure di ottimizzazione delle prestazioni. Inoltre, a seconda dell'interprete utilizzato (AutoCAD, progeCAD o IntelliCAD), l'inconveniente si può manifestare in modo più o meno evidente.

Dopo aver fatto questa premessa, vediamo la sintassi della funzione 'gc':

```
(gc)
```

Può sembrare strano ma, 'gc', non ha parametri e non restituisce alcun valore significativo.

Il suo scopo è unico e molto importante, infatti consente di forzare l'interprete Lisp a rilasciare la memoria utilizzata, ma non assegnata a variabili e funzioni.

In pratica, risolve l'inconveniente sopra esposto chiedendo, esplicitamente, di effettuare il rilascio della memoria occupata, ma non utilizzata da parti del programma.

Su applicazioni che utilizzano grandi quantità di dati o un elevato numero di procedure e variabili, l'uso di questa funzione riduce la memoria occupata dal software, migliorandone l'efficienza e le prestazioni generali.

Ovviamente, non occorre ricorrere costantemente alla funzione 'gc', è sufficiente richiamarla dopo un numero consistente di operazioni.

La funzione wcmatch

La funzione "wcmatch" ha il compito di verificare la presenza di un certo modello all'interno di una qualsiasi stringa data. L'utilità di questa funzione diventa evidente quando, data una stringa si vuole verificare la presenza, al suo interno, di determinate combinazioni di caratteri.

Pensiamo ad esempio il caso in cui volessimo sapere se la stringa inizia con il carattere R maiuscolo, oppure se un numero è reale, oppure quando siamo interessati a sapere se siamo in presenza di dati che rispettano più di una condizione.

La funzione "wcmatch" è cosi definita:

```
(wcmatch stringa pattern)
```

Ovviamente, "stringa" sarà il dato da verificare mentre "pattern" rappresenta il modello da utilizzare per la verifica. La funzione ritornerà poi vero (t) nel caso in cui il modello viene verificato, oppure falso (nil) in caso contrario.

Iniziamo ipotizzando di voler verificare se la nostra stringa inizia con il carattere R maiuscolo:

```
(wcmatch "Roberto" "R*")
```

In questo caso il risultato sarà un valore “t”, infatti il modello viene rispettato perfettamente.

Come secondo esempio possiamo considerare la rudimentale verifica di un numero reale che dovrà essere composto da 2 cifre, il separatore e altre due cifre per i decimali:

```
(wcmatch "12.03" "##.##")
```

In questo caso, la verifica fatta consiste nell’identificare la presenza, nella stringa, di un punto che separa due parti distinte, ognuna delle quali di tipo numerico. Anche in questo caso l’istruzione ritornerà un valore vero.

Allo stesso modo è possibile comporre modelli multipli, nei quali sono presenti più condizioni che vanno a sommarsi. Supponiamo di voler verificare che una stringa :

- Contenga tre caratteri;
- Non contenga il carattere “q”;
- Inizi con la lettera “R”.

Ecco l’istruzione :

```
(wcmatch "Roberto" "???,~*q*,R*")
```

Come si può vedere le singole condizioni sono separate da una virgola e devono essere tutte verificate affinchè “wcmatch” ritorni un valore vero.

Il modello, come si capisce, consente l’utilizzo di alcuni caratteri speciali :

Carattere	Definizione
#	Identifica un singolo carattere numerico
@	Identifica un singolo carattere alfabetico
.	Identifica un singolo carattere non alfanumerico
* (asterisco)	Identifica qualsiasi sequenza di caratteri
? (punto di domanda)	Identifica un singolo carattere, qualunque esso sia
~ (tilde)	Usato per primo identifica tutto eccetto il modello
[...]	Identifica uno dei caratteri racchiusi

[~...]	Identifica uno dei caratteri non racchiusi
-	Con le parentesi specifica un range di singoli caratteri
, (virgola)	Separatore tra due modelli
‘ (apice inverso)	Permette l’inserimento di un carattere che, altrimenti, sarebbe interpretato come speciale

La funzione boole

Lisp permette di effettuare le operazioni booleane bit a bit. Tramite questa funzione “boole” saremo in grado di effettuare le quattro fondamentali operazioni booleane tra due, o più, numeri interi.

Nella pratica, dati due numeri interi, la funzione convetirà ciascuno nel corrispondente valore binario, occupandosi poi di applicare l’operazione booleana ad ogni singola coppia di bit. La funzione ritornerà poi un valore intero con il numero generato dall’operazione.

Le operazioni supportate sono:

Operatore	Operazione	I bit risultanti valgono 1 quando
1	AND	Entrambe i bit sono a 1
6	XOR	Uno o entrambe i bit sono a 1
7	OR	Entrambe o nessun bit è a 1
8 (complemento a 1)	NOR	Entrambe i bit sono a 0

Per capire il funzionamento, ma soprattutto l’utilità, della funzione “boole” è necessario fare un passo indietro e spiegare meglio la matematica booleana. Iniziamo quindi a vedere come i numeri, sui quali effettueremo le nostre operazioni, vengono gestiti. La prima che “boole” fa è quella di convertire i valori interi, nei corrispondenti binari, che ricordo essere un modo per rappresentare qualsiasi numero attraverso l’utilizzo di soli 1 e 0. Ecco alcuni esempi :

Valore intero	Valore binario
8	1000
50	110010
300	100101100

456 111001000

Per approfondimenti sulla natura dei numeri binari consiglio di leggere

https://it.wikipedia.org/wiki/Sistema_numerico_binario.

Una volta convertiti i numeri interi in binari, la funzione “boole” procede all’esecuzione dell’operazione richiesta, agendo un bit alla volta. Ad esempio, volendo fare un “AND” tra i numeri 11 e 8 dovremo scrivere :

```
(boole 1 11 8)
```

In questo caso 1 è l’operatore, mentre i valori seguenti sono quelli che andranno processati. Questa espressione ritorna il numero “8”. Tale risultato è così generato :

Intero 11 = 1011	Intero 8 = 1000	Risultato AND
1	1	1
0	0	0
1	0	0
1	0	0

Per valutare il risultato ottenuto occorre rammentare che, l’operatore AND produce 1 solamente quando entrambi i bit confrontati risultano pari a 1. Ovviamente, il risultato binario ottenuto (1000), trasformato in decimale, equivale al numero intero “8”.

Giunti a questo punto, qualcuno potrebbe chiedersi a cosa possa servire tutto ciò. Effettivamente la funzione “boole” non è poi cosi usata, in realtà la si incontra solamente in casi particolari oppure all’interno di software particolarmente sofisticati, inoltre la si può trovare nel caso si valutino alcune speciali variabili del CAD.

L’utilizzo più frequente di “boole” lo troviamo appunto nell’utilizzo delle variabili del CAD, alcune delle quali, contengono al loro interno, la possibilità di impostare più opzioni, tutte contenute in un singolo valore numerico. Ciò avviene proprio grazie all’utilizzo della matematica booleana e alla funzione “boole”.

Facciamo qualche esempio. Iniziamo dalla variabile “OSMODE” che permette di stabilire quali modalità di snap ad oggetto sono attive. Nella pratica, questa variabile permette di settare, in modo indipendente gli uni dagli altri, almeno una dozzina di valori, ognuno dei quali determina se la modalità di snap ad oggetto relativa debba, o meno, essere attiva.

La variabile “OSMODE” è la somma di alcuni valori interi, ognuno dei quali, identifica una modalità differente:

- 0. Nessuno snap attivo
- 1. FINe;
- 2. MEDio;
- 4. CENtro;
- 8. NODo;
- 16. QUAdrante;
- 32. INTersezione;
- 64. INSerimento;
- 128. PERpendicolare;
- 256. TANgente;
- 512. VICino;
- 2048. Intersezione APParente;
- 4096. ESTensione;
- 8192. PARallelo.

Naturalmente, quando un disegno viene salvato, il valore di tale variabile torna ad essere quello iniziale, cioè 0.

Supponiamo ora di voler sapere se la vista è stata modificata. Questo implica controllare se è presente, in DBMOD, il valore 4. Ecco come fare sfruttando “boole” :

```
(boole 1 (getvar "DBMOD") 16)
```

nel caso in cui 16 sia effettivamente presente in DBMOD la funzione ritornerà 16, in caso negativo il valore restituito sarà sempre 0. Così facendo sarà possibile verificare anche gli altri valori.

Un secondo esempio è rappresentato dalla variabile “DBMOD” con la quale è possibile sapere se il disegno corrente è stato modificato o meno. In questo caso, i valori utilizzati sono :

- 1. Database di oggetti modificato;
- 4. Variabile di database modificato;
- 8 .Finestra modificata;

- 16. Vista modificata;
- 32. Campo modificato.

Esattamente come fatto per DBMOD, se volessimo sapere se lo snap tangente è attivo:

```
(boole 1 (getvar "DBMOD") 256)
```

Nel caso in cui ottenessimo dall'istruzione 256 significherebbe che la modalità tangente è attiva, in caso contrario otterremo sempre 0.

Questa tipologia di variabile, nella documentazione dei CAD, viene chiamata bitflag oppure bitcode.

Per quanto riguarda invece gli utilizzi personali della funzione "boole", qui il limite è dato solamente dalla fantasia del programmatore. Non solo, come per le variabili, può essere utilizzata per raggruppare più dati all'interno dello stesso valore ma può essere usata, ad esempio, per creare rudimentali sistemi di criptazione. Comunque sia, il suo limite rimanga l'inventiva del programmatore.

Le funzioni ricorsive

Iniziamo col definire cos'è la ricorsione. Termine con il quale si identifica un algoritmo che, dato un insieme di dati, li analizza suddividendoli e applicando lo stesso algoritmo iniziale ai dati così semplificati.

Più che speciali funzioni, le funzioni ricorsive, e la ricorsione in genere, sono tecniche in grado di semplificare, o alle volte complicare, un programma. Utili nel caso si vogliano analizzare dei dati strutturati su più livelli, l'uso della ricorsione è un sistema efficace ed in grado di ridurre la quantità di codice.

Nella pratica, una funzione ricorsiva non è altro che una funzione che richiama se stessa.

Un primo esempio, semplice, può essere il calcolo del fattoriale di un numero. Fattoriale che ricordo essere il prodotto dei primi n numeri naturali inclusi nel numero dato. Possiamo vedere il fattoriale di n come:

```
1 * 2 * 3 * .... * n-1 * n
```

Proviamo ora a realizzare due distinte funzioni, la prima che calcola il fattoriale usando tecniche classiche, e la seconda ricorsiva:

```
;versione sviluppata con tecniche classiche (iterativo)
(defun std_fatt( n / fatt)
      (setq fatt 1)
      (while (<= 1 n)
```

```
            (setq fatt (* fatt n))
            (setq n (- n 1))
      )
fatt
)

;versione ricorsiva
(defun ric_fatt( n / result)
      (if (<= n 1)
            (setq result 1)
            (setq result (* n (ric_fatt (- n 1))))
      )
result
)
```

Come si può osservare la procedura ricorsiva è tale, in quanto richiama se stessa. Evidente è il vantaggio relativo alla riduzione del numero di linee di codice necessarie. Purtroppo, solitamente, le procedure ricorsive hanno alcuni svantaggi. Prima di tutto, può accadere che siano più lente delle corrispondenti iterative, inoltre risultano essere più complesse da comprendere, cosa che può influire negativamente sulle fasi di modifica e debug di vecchi programmi.

La tecnica ricorsiva si adatta molto bene all'elaborazione di dati in genere, potendo scorrerne in modo semplice la struttura.

In senso generale, l'uso di questo tipo di procedura deve essere ponderato tenendo presente i fattori già menzionati, che possono essere riassunti così:

- Numero di linee di codice necessarie. Le funzioni ricorsive sono, solitamente, più brevi;
- Velocità di esecuzione necessaria. Le funzioni ricorsive sono, solitamente, più lente ed occupano una maggiore quantità di memoria;
- Semplicità nella rilettura e modifica del codice. Le funzioni ricorsive sono, solitamente, più complesse da rileggere, interpretare e modificare.

Capitolo 14
I sorgenti del software

Il codice sorgente di un programma non è altro che un testo. Naturalmente avendo accesso a tale testo è possibile capire e copiare ciò che è stato fatto. Se questo non risulta un problema con programmi piccoli e limitati, nel caso in cui si realizzino procedure complesse, magari da rivendere, il problema della riservatezza del codice sorgente può essere determinante.

Nonostante l'autore di questo testo sia un sostenitore del Software Libero, è chiaro che in moltissime realtà di oggi, la protezione del sorgente da occhi indiscreti risulti essenziale.

Questo capitolo mostrerà come poter impedire, più o meno efficacemente, l'accesso al codice sorgente di un programma, quando questo viene distribuito a terzi.

Proteggere il sorgente da occhi indiscreti

Se si creano utilità LISP e si decide di distribuirle con licenza 'Open Source' o più semplicemente 'Freeware' non sussistono problemi, se però si decide di produrre software 'Shareware' o più generalmente commerciale, uno dei maggiori problemi che si incontrano è quello di proteggere il lavoro svolto in modo da rendere inaccessibile il codice sorgente.

Attualmente esistono diversi sistemi gratuiti per la protezione dei file LISP. Il primo, ed il più semplice, passa attraverso l'utilizzo di una piccola utilità chiamata 'Kelvinator', che si occupa di rendere illeggibile un sorgente, lasciandolo comunque perfettamente funzionante.

Vediamo ora come agisce il 'Kelvinator' . Il software si limita a rendere il codice sorgente difficilmente leggibile, eliminando i commenti e disponendo tutto il sorgente su una sola linea, inoltre provvede a sostituire i nomi delle variabili, con identificativi casuali.

Osserviamo questo codice:

```
;linea di commento
;linea di commento
(defun c:prova ( / obj ent elemento)
  ;linea di commento
  (princ "\nComando di prova")
  (setq obj 23)
  (setq ent "nome oggetto")
);Enddef
```

ora passiamo ad esaminare lo stesso sorgente dopo averlo elaborato con il 'Kelvinator':

```
(DEFUN C:PROVA(/ Qj Q@ QQ)(PRINC"\nComando di prova")
(SETQ Qj 23)(SETQ Q@"nome oggetto"))
```

come si può notare sono stati eliminati i commenti, tutte le variabili sono state sostituite

con sequenze di caratteri casuali, ed infine tutto il sorgente è ora disposto su una singola riga.

Questo tipo di trasformazione impedisce una lettura comoda del sorgente, ma purtroppo non ne garantisce la riservatezza, in quanto il codice è ancora modificabile. Esistono inoltre degli editor di testo che permettono la riformattazione del codice, eliminando di fatto una delle protezioni fornite (la disposizione del codice su linea singola).

Per aumentare il grado di protezione è possibile unire all'utilizzo di 'Kelvinator' quello di 'Protect'. Questo software, presente oramai da anni, permette di criptare il codice sorgente, rendendolo di fatto illeggibile. Di seguito è riportato uno spezzone del sorgente utilizzato per l'esempio precedente, elaborato con 'Protect':

```
AutoCAD PROTECTED LISP file
▮rZš▮KØ'`ú¥▮}¬▮▮▮▮m°AÓçï▮L±KÆßö£▮*▮~¿▮Mú▮WÁ£#/~▮k¹▮k‹
```

Questo sistema purtroppo non garantisce l'inviolabilità del codice in quanto è possibile trovare una utilità che permette la sua decriptazione. Garantisce comunque, unito a 'kelvinator' un discreto livello di protezione.

Per superare i limiti dei sistemi precedenti, e per risolvere in maniera definitiva il problema della protezione dei sorgenti AutoDESK, la creatrice di 'AutoCAD' ha introdotto dalla versione 14 'Visual LISP'. Questo ambiente di programmazione permette di produrre due tipologie di file criptati, i .FAS e i .VLX.

I file .FAS sono semplicemente i nostri sorgenti LISP criptati con uno speciale algoritmo, molto più sicuro di quello usato da Protect. I .VLX sono l'insieme di tutte le parti di un progetto (file lisp, file dcl ecc...), codificati e riuniti in un singolo file.

L'utilizzo di 'Visual Lisp' è attualmente l'unico sistema di protezione gratuita (infatti è incluso in AutoCAD) ed efficace per i nostri sorgenti.

Purtroppo però tale ambiente è disponibile ed utilizzabile sono in congiunzione ad 'AutoCAD' 14 o successivo. Chi utilizzasse invece IntelliCAD o uno dei suoi derivati, non potrebbe usufruire di questo strumento.

IntelliCAD infatti non supporta nè i file .FAS nè tantomeno i .VLX. L'unico sistema gratuito disponibile è quello fornito da 'Kelvinator' + 'Protect'.

Per quanto riguarda la compatibilità con le varie versioni dei due Cad, possiamo dire che 'Kelvinator' e 'Protect' sono accettati da tutte le versioni di AutoCAD (partendo dalla release 10) e da ogni versione di IntelliCAD che supporti il linguaggi LISP.

Kelvinator e Protect sono disponibili a questo indirizzo:

http://www.angelfire.com/pa3/autocad/autocad_autolisp.htm

Capitolo 15
Scriviamo del buon codice?

Questo capitolo sarà interamente dedicato alle metodologia per poter scrivere un buon codice sorgente, in grado di essere manutenuto e sviluppato per un lungo periodo di tempo.

Nel modo più semplice possibile, alle volte anche estremizzando la semplificazione, verranno presentati i concetti necessari a qualsiasi programmatore, in grado di ridurre la complessità di molte fasi di sviluppo, rendendo la stesura e la modifica dei programmi Lisp più immediata, rapida e intuitiva.

Primo approccio

Prima di spiegare come scrivere un buon programma, cerchiamo di capire cos'è un buon programma e quali sono le sue caratteristiche.

Sicuramente, per poter dire che un software è un buon software, questo deve fare ciò per cui è stato creato. Ad esempio, immaginando di avere un programma che verifica se un numero sia primo o meno, la sua caratteristica fondamentale sarebbe quella di calcolare correttamente i numeri primi, in modo da poter determinare se il dato fornito in input sia o meno tale, il tutto senza errori.

Bene, il fatto che un programma produca un risultato corretto è sufficiente per affermare che sia un buon programma? La risposta è **no** !

Ancora meglio sarebbe, se rispondessimo con un **ni** !

Se il nostro programma fosse composto da poche linee, allora sicuramente la risposta potrebbe essere "**si"** ma, nel caso in cui ci trovassimo di fronte a un software più complesso, composto magari da centinaia, se non da migliaia di linee di codice, in questo caso il fatto che venga prodotto un risultato corretto non è condizione sufficiente per giudicarlo un buon programma.

A questo punto sorge spontanea la domanda, perché?

Il software, nella maggior parte dei casi, non è una entità stabile e costante. Il software nel tempo viene cambiato, viene modificato, viene espanso, si "evolve", viene riadattato, trasformato, alle volte finisce per svolgere compiti per i quali non era stato pensato. Ciò significa che, il codice sorgente di un programma dovrà essere, riletto, modificato, ristrutturato, magari da persone diverse rispetto a quelle che lo hanno pensato e scritto inizialmente.

Da queste considerazioni nascono le caratteristiche di un buon programma, che possono essere cosi riassunte :

- **Correttezza**. La capacità di svolgere il proprio lavoro in modo corretto, senza errori;
- **Robustezza**. Cioè la propensione del programma a gestione e, in un certo

senso, a sopportare, le situazioni impreviste, magari proseguendo il proprio funzionamento senza apparenti problemi.

- **Leggibilità** del codice sorgente. Che garantisce la comprensione del programma a chi ne legge il codice;
- **Efficienza**. Nella pratica la velocità di esecuzione;
- **Portabilità**. Che consiste nella possibilità di usare lo stesso software su sistemi e/o software CAD differenti;
- **Riusabilità**. Che è la propensione del codice sorgente ad essere riutilizzato in altri programmi, magari in modi e condizioni differenti.

Correttezza

Ovviamente la prima caratteristica necessaria ad ogni software è la capacità di svolgere in modo corretto le operazioni per le quali è stato progettato.

Quindi, un programma studiato per verificare se un numero è primo oppure no, deve svolgere il suo compito senza errori, indicando sempre quali numeri siano primi e quali non lo siano.

Al contrario di qualsiasi altro aspetto o regola, l'assenza della "Correttezza" fa si che un software sia completamente, o parzialmente, inutile e ciò non è certo sintomo di un buon programma.

Quando si parla di correttezza occorre tener presente anche un aspetto secondario che, tanto secondario non è. Un software è un qualcosa che, forniti dei dati, li elabora e produce un risultato. Prendendo come esempio il nostro software di verifica dei numeri primi, i dati in ingresso sono rappresentati dal numero che qualcuno fornisce al software per il controllo.

Cosa succederebbe se al programma non venisse fornito un numero ma, ad esempio, delle lettere? Andrebbe in errore? Riuscirebbe ugualmente a gestire il dato errato?

Un buon programma è in grado di gestire la maggior parte degli eventi inattesi che possono verificarsi. Questa caratteristica, strettamente legata alla correttezza di un software viene, spesso, definita "Robustezza".

Robustezza

Questa caratteristica non sempre risulta facilmente comprensibile, e spesso nemmeno facilmente realizzabile.

Con robustezza, solitamente si intende la capacità di un programma di gestire eventi imprevisti, riuscendo comunque a gestirne gli effetti, senza generare errori visibili dall'utente o in grado di danneggiare il lavoro svolto.

Una domanda che ci si dovrebbe fare è : "è possibile che al nostro programma arrivino dati errati?"

Ad esempio, immaginiamo una finestrella nella quale l'utente debba inserire un numero. L'utente potrebbe, per errore, fornire lettere, spazi, tabulazioni, numeri con la virgola o con il punto come separatore decimale, addirittura potrebbe non scrivere nulla. Tutti casi che, anche se improbabili, sono certamente possibili e, come l'esperienza insegna, accadranno.

Pensiamo poi ai file. Un software che tenta di leggere un file che non esiste, oppure che esiste ma non è leggibile in quanto l'utente corrente non possiede i permessi necessari per aprirlo.

Come si può facilmente intuire, i casi possibili sono molti, e spesso non è possibile verificarli tutti, infatti alcuni inconvenienti vengono scoperti durante l'utilizzo quotidiano e sovente alcuni casi rimangono irrisolti in quanto di difficile riproducibilità, un buon software dovrebbe cercare di prevenire errori del genere, al fine di ottenere il tanto desiderato dato corretto.

Ma facciamo un esempio pratico.

Uno degli esempi più immediati che permettono di capire quale sia l'importanza di non produrre errori inaspettati è la divisione per 0 :

```
(setq val1 145.34)
(setq val2 0)
(setq risultato (/ val1 val2))
```

Ipotizziamo di avere due valori numerici, magari inseriti dall'utente, o derivanti da qualche operazione matematica. Ipotizziamo, come nell'esempio, che i due vengano usati in un'operazione di divisione, cosa accadrebbe se uno dei due fosse 0?

Il risultato sarebbe questo :

```
error: divide by zero
```

Immaginiamo un programma complesso che a un certo punto si interrompe in questo modo. L'utente sarebbe felice? Io penso proprio di no. E' indispensabile evitare questo tipo di situazione.

Ovviamente i casi sono quasi infiniti, l'esempio mostra un errore su un'operazione matematica, ma potremo generarne anche cercando di scrivere in un file a sola lettura ecc...

Non ci sono regole particolari da seguire per ottenere un software robusto, tranne mantenere una concentrazione adeguata e domandarsi sempre :

Quello che sto facendo potrebbe non funzionare in determinate situazioni?

Esiste poi una soluzione “estrema” per la gestione degli errori imprevisti. Soluzione già descritta nel capitolo riguardante la “Gestione degli Errori”.

In senso generale è sempre bene poter gestire i vari errori in modo specifico, indicando di conseguenza all’utente, cosa c’è che non va e come intervenire per risolvere l'inconveniente. Spesso e volentieri messaggi generici e senza indicazioni particolari risultano essere completamente inutili, se non irritanti.

Leggibilità

Cos’è la leggibilità di un codice sorgente? In poche parole un listato di programma è leggibile quando, se io lo osservo, capisco subito cosa fa e dove posso intervenire per effettuare le modifiche necessarie.

Molto spesso i programmatori, ed in genere chi sviluppa software, si concentra relativamente su questo aspetto, al contrario dovrebbe essere un’attività al centro delle proprie preoccupazioni.

Un codice sorgente confuso, mal scritto, mal formattato, privo di commenti, troppo ramificato, eccessivamente sofisticato, ecc..., può essere un enorme impedimento per tutti quelli che, in tempi successivo dovranno mettere mano ai listati per eliminare eventuali errori o per introdurre nuove funzionalità.

Facciamo un esempio, prendiamo uno spezzone di codice di esempio, questo :

```
(defun c:CambiaCol ( / elemento nomEnt OldColor NewColor)
(setq nomEnt (car(entsel "\nSelezionare oggetto a cui cambiare
colore : ")))
(if nomEnt  ;se è stato selezionato un oggetto procede
      (progn
            (setq NewColor (acad_colordlg 0 T)) ;selezione
colore
            (setq elemento (entget nomEnt)) ;estrazione
dell’entità
            (setq OldColor (assoc 62 elemento)) ;estrazione
colore
            (if OldColor
                  (progn ;se esiste un colore precedente
                        ;lo sostituisce con quello indicato
                        (setq elemento (subst (cons 62
NewColor) oldColor elemento))
                  );endp
```

```
                    (progn ;se ha il colore di default
                           ;aggiunge il campo colore con il
colore voluto
                           (setq elemento (cons (cons 62
NewColor) elemento))
                    );endp
             );endif
             (entmod elemento) ;modifica l'entitè
             (entupd nomeEnt) ;aggiorna la visualizzazione
      );endp
      (progn
             (alert "Non è stato selezionato alcun oggetto !")
      );endp
);endif
(prin1)
);enddef
```

Lo stesso codice lo potremo scrivere cosi :

```
(defun c:CambiaCol ( / elemento nomEnt OldColor NewColor)
(setq nomEnt (car(entsel "\nSelezionare oggetto a cui cambiare
colore : "))) (if nomEnt (progn (setq NewColor (Acad_colordlg
0 T)) (setq elemento (entget nomEnt)) (setq OldColor (assoc 62
elemento)) (if OldColor (progn (setq elemento (subst (cons 62
NewColor) oldColor elemento)) )(progn (setq elemento (cons
(cons 62 NewColor) elemento)) )) (entmod elemento) (entupd
nomeEnt) )(progn (alert "Non ? stato selezionato alcun oggetto
!") ))(prin1))
```

In questo modo, il programma sarà ancora perfettamente funzionante ma... quanto tempo ci vorrà per capire cosa fa e come la fa? Di più o di meno rispetto alla versione iniziale? La risposa mi pare troppo ovvia, la versione scritta e formattata meglio, sarà più leggibile, ed infinitamente più semplice da modifiare.

A questo punto possiamo stilare un piccolo elenco di regole minime per rendere un programma leggibile:

- Prima di tutto l'indentazione del codice. Come si può osservare dall'esempio proposto, usare l'indentazione delle linee per identificare le strutture di controllo usate (defun, if, while, ecc...), può rendere più chiaro il codice e facile la sua comprensione;
- Commenti. Commentare il codice è sicuramente uno dei sistemi migliori per spiegare cosa fa. Naturalmente non bisogna eccedere, ma commenti mirati

nei punti complessi aiuta moltissimo la comprensione di ciò che viene fatto;

- I nomi delle variabili e delle funzioni. Quando creiamo una variabile è fondamentale assegnargli un nome significativo, ad esempio “NomeCliente” è molto meglio di “xadsge1”. Stesso discorso per i nomi delle funzioni. Naturalmente anche l’utilizzo di maiuscole e minuscole è rilevante, meglio “NomeCliente” di “nomecliente” o di “NoMeClIeNtE”;

- Se il programma è composto da più file, può essere determinante utilizzare uno stile comune per la scrittura del codice, evitando di usare regole di scrittura diverse per ogni file;

- Sempre pensando ai commenti, è utile mettere dei commenti subito prima delle funzioni per dichiararne l’utilità e gli eventuali dati di input e di output, così come può essere utile un commento all’inizio di un file per dichiararne la funzione;

- Un fattore alle volte sottovalutato è la lunghezza di una procedura o di livello di astrazione. E’ molto più complesso lavorare con una funzione lunga 1000 linee piuttosto che con 10 funzioni da 100 linee ciascuna. Questo consente di isolare determinati processi logici al fine di rendere tutto più chiaro;

- Ovviamente, sempre parlando di livelli di astrazione, è anche importante non esagerare. Dividere una procedura di 1000 istruzioni in 1000 funzioni interconnesse, non è affatto una buona idea.

Rispettando queste poche regole si otterrà un codice sorgente più leggibile, in grado di affrontare in modo più semplice un’eventuale evoluzione, ed in senso genera, maggiormente pronto per il futuro.

Efficienza

Quello dell’efficienza è un tema delicato. Al contrario di altri, questo argomento deve essere ben valutato, di caso in caso.

Prima di tutto cerchiamo di capire cosa sia l’efficienza di un software. Nella pratica, quando un programma è rapido nell’esecuzione e, allo stesso tempo, utilizza una quantità di risorse limitata (memoria, disco, ecc...), allora possiamo affermare che questo abbia un ottimo grado di efficienza.

Se, sulla carta, lo sviluppo di un software efficiente è sempre preferibile, nella realtà ciò non è sempre necessario, nè auspicabile.

Per rendere un software efficiente esistono una moltitudine di tecniche, tutti aventi lo scopo di rendere le procedure più rapide nell’esecuzione. Questo può avvenire in due modi diversi, ottimizzando le cose fatte, oppure eliminando le operazioni superflue,

oppure entrambe le cose.

Il processo di ottimizzazione, spesso, porta un grandissimo svantaggio, i software risultanti risultano più complessi e criptici da leggere e comprendere. E qui entra in gioco la capacità di valutazione da parte dello sviluppatore o del team di sviluppatori.

Facciamo un esempio. Supponiamo di realizzare la procedura di configurazione del nostro software, una funzione che verrà usata raramente dall'utente. In questo caso, per quale motivo concentrare i nostri sforzi sul rendere efficiente questa parte? Infatti non vi è nessun motivo valido, essendo di fronte ad una funzionalità che non richiede particolari requisiti di velocità, sarà molto meglio concentrarci su altre caratteristiche, come la leggibilità, la portabilità, ecc...

Allo stesso modo, se ipotizzassimo la realizzazione di un procedura per la creazione di molte entità grafiche, le cui prestazioni, intese come velocità di creazione, risultassero determinanti per la soddisfazione del cliente, a quel punto si potrebbero sacrificare altre caratteristiche per dare all'utente ciò che gli serve.

E' importante distinguere poi tra i due tipi di efficienza :

- **Efficienza reale**. Una procedura particolarmente rapida nell'esecuzione e in grado di utilizzare pochissime risorse è, sicuramente, un'applicazione realmente efficiente;

- **Efficienza apparente**. In questo caso, invece che concentrarsi sulle reali prestazioni del software, lo sviluppatore si concentra sulla percezione che l'utente ha delle varie operazioni. Supponiamo di avere una procedura di stampa, operazione che può avvenire dopo che l'utente ha compilato un'apposita maschera. Un software molto efficiente stamperebbe molto rapidamente. Volendo invece sfruttare l'efficienza apparente, l'utente potrebbe confermare l'operazione di stampa, che procederebbe in modo autonomo, lasciando che l'operatore posso continuare con il suo lavoro. In questo modo, nonostante la stampa vera e propria non sia particolarmente ottimizzata, l'utente avrebbe la sensazione di un software molto rapido e reattivo, per il semplice motivo che non lo costringe all'attesa ma lo lascia proseguire in altre attività.

Utilizzando queste tecniche in modo attento sarà possibile produrre un codice sorgente in grado di migliorare la soddisfazione dei clienti e degli utilizzatori in genere.

Portabilità

Per portabilità di un software, comunemente si intende la possibilità di farlo funzionare su sistemi differenti. In questo caso, data la natura stessa del linguaggio Lisp e del sistema sul quale questo viene eseguito, l'operazione potrebbe essere più complessa.

Se fossimo di fronte ad un normale programma, questo si limiterebbe a modificare il

proprio comportamento in base al sistema operativo sul quale si trova. Nel nostro caso, non solo sarà necessario tener presente il sistema operativo presente, ma dovremo anche variare il nostro software in base al CAD sul quale andrà a funzionare.

Se certamente è vero che tutti i sistemi CAD che supportano Lisp hanno comportamenti simili, queste similitudini non sono quasi mai sufficienti per poter utilizzare un codice sorgente senza prevedere delle differenze.

Proprio queste diversità, alle volte anche molto importanti, fanno si che le operazioni per rendere un software Lisp portabile, richiedano un certo sforzo, che aumenta all'aumentare del numero di CAD e di sistemi operativo sui quali dovrà funzionare.

Ci sono diversi aspetti da considerare per rendere un software portabile :

- Quali sono i CAD sui quali il programma dovrà funzionare;
- Quali sono i sistemi operativi sui quali il programma opererà;
- Evitare di utilizzare funzioni specifiche di un solo CAD o di un sistema operativo, soprattutto se queste non sono replicabili sugli altri sistemi;
- Parametrizzare il codice sorgente cercando di effettuare il minor numero di distinzioni possibile. Se il codice risultasse molto diversificato a seconda del sistema, questo genererebbe notevoli problemi di manutenzione e sviluppo.

Quando andremo a valutare i costi e i tempi necessari per lo sviluppo di un software portabile, potremo renderci conto che questa è una caratteristica particolarmente onerosa, e come tale, richiede molta attenzione.

Usabilità

Anche se questa sezione è posta per ultima, in realtà è una delle più importanti, essendo quella che riguarda l'utente finale e il suo modo di operare con il software. L'usabilità rappresenta la facilità con cui l'utilizzatore lavora con il nostro programma. Più il programma è semplice da utilizzare, più l'usabilità è alta.

Il massimo grado di usabilità si ottiene quanto, un utente che mai ha visto il nostro software, è in grado di farlo operare al 100% senza alcuna forma di documentazione o formazione. Naturalmente questa è solo un'ipotesi teorica, in quanto, è quasi impossibile da ottenere.

Quando si pensa all'interfaccia per il proprio software occorre pensare che :

- Non è fatta per noi sviluppatori ma è fatta per un generico utente finale, con competenze e abitudini diverse dalle nostre;
- Gli utenti utilizzano i software per essere più produttivi, non perchè trovano

piacere nel farlo;

- Gli utenti solitamente vogliono eseguire i propri compiti nel modo più efficiente possibile, ed è per questo che usano i nostri programmi;
- Sono gli utilizzatori a decidere se un'interfaccia è facile da usare oppure no. Se gli utenti affermano che "non si capisce nulla", significa che abbiamo sbagliato, non significa che "gli utenti sono utonti".

Partendo da queste considerazioni, appare chiaro come, lo studio dell'usabilità vada fatto fino dalle prime fasi di sviluppo del software e non possa essere destinato ad un "momento successivo". Alle volte l'interfaccia verso l'utente condiziona pesantemente anche la struttura del proprio codice sorgente.

Nelle fasi di creazione dell'interfaccia sarebbe ottima cosa poter coinvolgere le persone che useranno veramente il programma, questo al fine di capire le loro opinioni, consentendogli di pilotare il processo di creazione. Per lo stesso motivo, è importante sottoporre le varie bozze alla loro valutazione.

Creare software usabili, significa avere dei benefici immediati che interessano diverse aree:

- Maggiore soddisfazione del cliente e/o dell'utilizzatore in genere;
- Minori costi di formazione del personale;
- Minori costi di supporto per il software;
- Minor necessità di aggiornamenti per affinare l'interfaccia;
- Documentazione più ridotta;
- Utenti più produttivi e meno "irritati".

Nel caso in cui si realizzino software non solo per se stessi, ma anche per altri, occorre capire come la progettazione dell'interfaccia sia, probabilmente, l'aspetto che più determina la soddisfazione degli utilizzatori/clienti, infatti il loro giudizio sul prodotto è determinato proprio da quanto vedono e da quanto percepiscono nell'utilizzo quotidiano.

Molto significativa ed istruttiva, può essere questa spassosa rappresentazione di come l'interfaccia utente, ed in genere il proprio software, venga visto durante le varie fasi di analisi e di realizzazione.

Come viene visto lo stesso programma da persone diverse, in fasi diverse ?



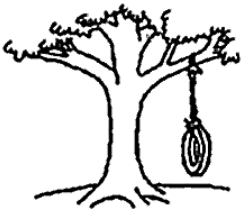
Visto dal Cliente

Visto dai programmatori

Visto nelle specifiche

Installato

Visto dall'analisi



Cosa serviva all'utente

Capitolo 16
Tecniche e Procedure

In questo capitolo esamineremo alcune tecniche disponibili con il linguaggio Lisp, ma non sempre immediate da utilizzare.

Funzioni Matematiche. Il Lisp interpreta se stesso.

Una delle più intriganti caratteristiche del linguaggio Lisp è la possibilità di interpretare ed eseguire sè stesso. In pratica, un programma che crea o riceve espressioni Lisp che, "magicamente", vengono poi eseguite dal CAD.

Per comprendere questa caratteristica, vedremo ora come rappresentare graficamente delle semplici funzioni matematiche. Ovviamente, la loro rappresentazione avverrà all'interno di un disegno.

Sia il Lisp di AutoCAD sia quello di IntelliCAD o progeCAD mettono a disposizione del programmatore una serie di funzioni matematiche:

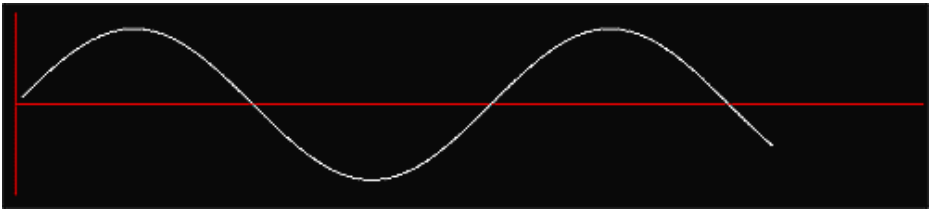
- +. Addizione.
- -. Sottrazione.
- *. Moltiplicazione.
- /. Divisione.
- ~. Compie un'operazione NOT sui bit di un numero intero.
- 1+. Incremento.
- 1-. Decremento.
- abs. Valore assoluto.
- atan. Arcotangente.
- cos. Coseno.
- exp. Elevamento a potenza di e (2.71828).
- expt. Elevamento a potenza di un numero qualsiasi.
- fix. Approssima un numero reale all'intero inferiore.
- float. Converte un numero in un reale.

- gcd, massimo comun denominatore.
- log. Logaritmo naturale.
- logand. Opera un AND tra i bit di una serie di numeri interi.
- logior. Opera un OR tra i bit di una serie di numeri interi.
- Lsh, compie un'operazione di shift sui bit di un numero intero
- max. Trova il numero maggiore.
- min. Trova il numero minore.
- rem. Calcola il resto.
- sin. Seno
- sqrt. Radice quadrata.

Iniziamo con l'esaminare una semplice funzione matematica:

```
y = sin(x)
```

La sua rappresentazione grafica è:



Per poter disegnare con il CAD questa funzione, occorre prima di tutto conoscere come è possibile rappresentare la stessa in modo che il LISP la possa interpretare al meglio. La soluzione migliore è quella di riscrivere la nostra funzione utilizzando una diversa notazione, in questo caso quella "polacca prefissa":

```
y = sin(x) diventa y = (sin x)
```

Questo diverso sistema di rappresentazione della funzione permette di costruire un software che si limiti ad utilizzare direttamente l'espressione per il calcolo dei punti che rappresentano la funzione stessa.

Infatti, come già detto, il linguaggio LISP permette di interpretare direttamente le espressioni nella forma appena vista, che è la sintassi propria di questo linguaggio

(Lisp). Ad esempio supponiamo di avere una riga di codice che imposti una variabile in questo modo:

```
(setq a "(+ (sin x) x)")
```

la nostra variabile "a" è una stringa in cui abbiamo inserito la funzione da rappresentare. Poniamo ora di avere un'altra variabile di nome "x" contenente un numero reale:

```
(setq x 12.00)
```

Se volessimo calcolare il risultato dell'espressione contenuta nella variabile "a" ci basterebbe trasformare lo stesso in un'istruzione LISP e quindi farla valutare all'interprete:

```
(setq risultato (eval (read a)))
```

Al termine di questa riga la variabile risultato conterrà il numero "11.4634" che è il risultato dell'espressione "(+ (sin x) x)", che potremo anche scrivere come "sin(x) + x".

Basando il programma su questa possibilità è semplice scrivere la procedura per la rappresentazione grafica delle funzioni matematiche, le uniche cose necessarie, e richieste all'operatore saranno, l'indicazione del range all'interno del quale calcolare i punti e il livello di dettaglio che la curva dovrà avere.

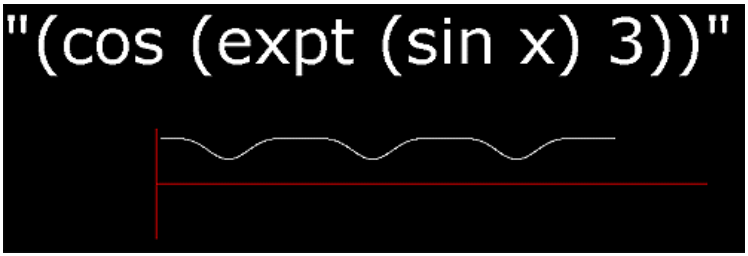
Ecco il programma completo per la rappresentazione grafica delle funzioni matematiche:

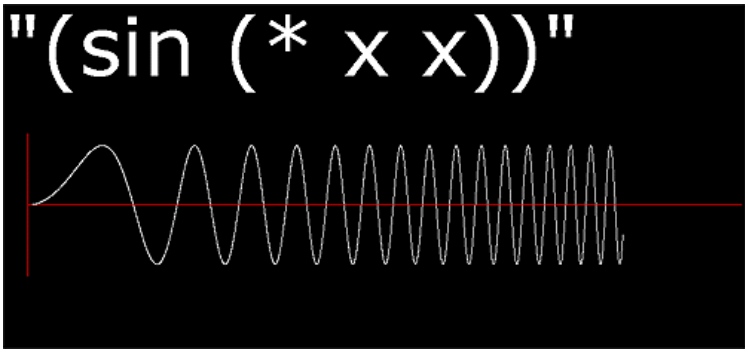
```
;disegnare funzioni matematiche con il linguaggio Lisp
; Autore: Roberto Rossi
; Versione: 1.0.0
(defun c:DrawFx ( / funzione mini mass passo)
(setq funzione (getstring
"\nFunzione da disegnare(racchiusa tra doppi apici): "
));endset
(setq mini (getreal "\nInizio calcolo funzione > 0: "))
(if (or (not mini) (<= mini 0))
      (setq mini 0.01)
);endif
(setq mass (getreal "\nFine calcolo funzione > 0: "))
(if (not mass)
      (setq mass 10)
);endif
(setq passo (getreal "\nIntervallo (es.: 0.1): "))
(if (not passo)
```

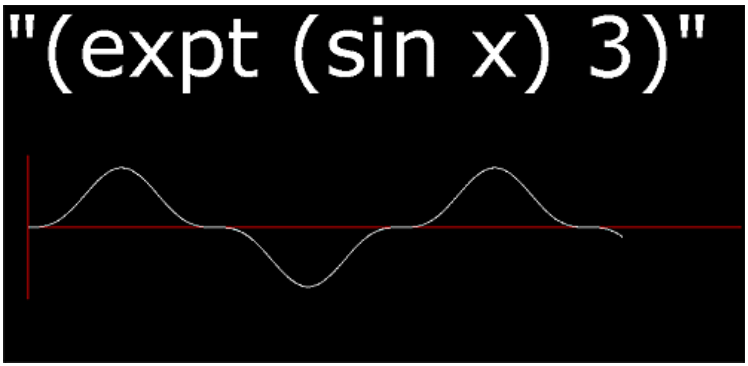
```
      (setq passo 0.1)
);endif
(if (/= funzione "")
      (progn
            (resetVar)
            (rfx funzione mini mass passo)
            (princ "\nDisegno funzione concluso con
successo.")
      );endp
);endif
(prin1)
);enddef

;imposta le variabili di sistema in modo che non
interferiscano
;con la procedura
(defun resetVar ( )
(setvar "OSMODE" 0)
(setvar "UCSICON" 0)
(setvar "CMDECHO" 0)
)

;disegna gli assi cartesiani
(defun drawAxis ( ext dmaxx dmaxy / x xmin y ymin)
(setq x (+ (* (/ dmaxx 100) ext) dmaxx))
(setq y (+ (* (/ dmaxy 100) ext) dmaxy))
(command "_line" (list 0 0 0) (list x 0 0)
""
"_change" (entlast) "" "_p" "_C" "1" "" "_line"
(list 0 (* -1 y) 0) (list 0 y 0)
"" "_change" (entlast) "" "_p" "_C" "1" ""
)
);endd

;disegna una funzione matematica
;
;Parametri:
; funzione = funzione da rappresentare es.: "(sin x")
```

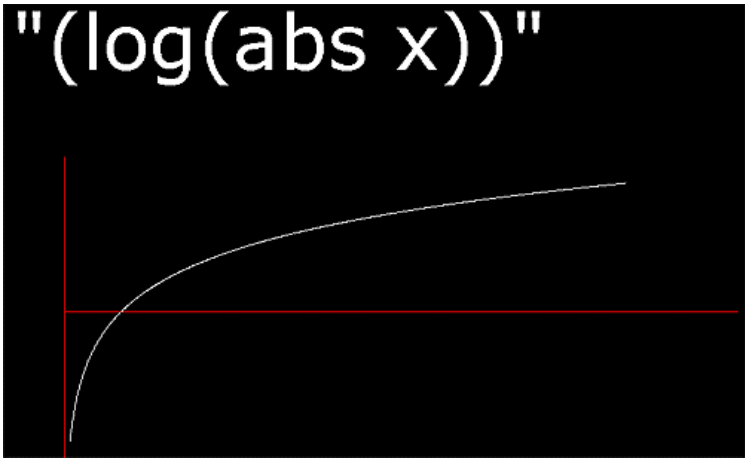
```
; xmin = valore di x per l'inizio del calcolo
; xmax = valore di x per la fine del calcolo
; passo = passo tra un calcolo e il successivo
(defun rfx ( funzione xmin xmax passo / x y fxtmp ymax)
;trasforma la funzione da stringa a simboli LISP
(setq fxtmp (read funzione))
(setq x xmin)
(setq ymax 0)
;inizia la rappresentazione
(command "_line")
;continua il calcolo da xmin a xmax
(while (> xmax xmin)
      (setq x xmin)
      ;valuta la funzione
      (setq y (eval fxtmp))
      (if (> y ymax)
            (setq ymax y)
      );endif
      ;disegna il punto
      (command (list x y 0))
      (setq xmin (+ xmin passo))
);endw
(command "");termina il disegno
;disegna gli assi cartesiani
(drawAxis 20 xmax ymax)
;modifica la visualizzazione
(command "_zoom" "_e" "_zoom" ".5x")
(prin1)
);enddef
```

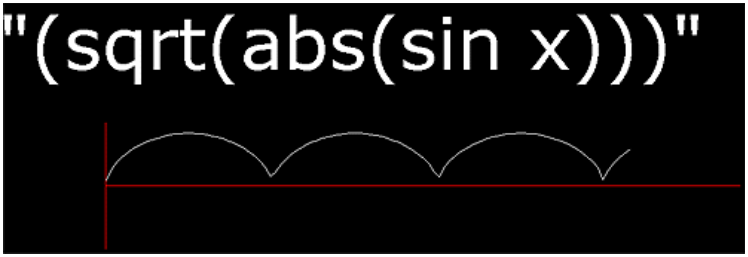
Una volta caricato, per avviare il programma sarà necessario digitare il nome del comando definito, "DrawFx", il quale potrà disegnare funzioni come queste:











Il debug del software

Prima di tutto cerchiamo di capire cos'è il Debug.

Quando si utilizza un software, può capitare di incontrare un comportamento anomalo, magari incomprensibile, imputabile al programma stesso, in pratica un errore. Il debug (chiamato anche debugging) è, semplicemente, l'attività necessaria all'individuazione e alla rimozione di questo errore.

Anche scrivendo piccole procedure può capitare di inserire involontariamente errori nel codice che, una volta riscontrati dagli utilizzatori, dovranno essere trovati e rimossi.

Sviluppando software con il linguaggio Lisp, si possono commettere vari errori, tutti in grado di compromettere il buon esito delle operazioni svolte. Naturalmente, questi si

manifestano in modo differente gli uni dagli altri, alcuni con un messaggio preciso, altri con uno generico, altri ancora sono in grado di bloccare completamente il CAD, costringendoci ad un riavvio.

Prima di addentrarci nel complesso mondo del debug, analizziamo sinteticamente ciò che differenzia AutoCAD da IntelliCAD (e dai suoi derivati) per quanto riguarda l'argomento. AutoCAD, come i suoi utenti avranno notato, possiede un ambiente integrato (Visual Lisp) che consente l'editazione del codice Lisp, inoltre lo stesso strumento fornisce al programmatore la possibilità di seguire in maniera interattiva il flusso di un programma, implementando quello che normalmente viene chiamato debugger, cioè in software che consente l'analisi dei programmi durante il loro funzionamento. Al contrario IntelliCAD e i suoi derivati, non forniscono nulla che faciliti le operazioni di debug.

Considerando che questo testo non vuole privilegiare una piattaforma rispetto all'altra, analizzeremo le tecniche per facilitare l'analisi dei software e la risoluzione dei più comuni errori, senza appoggiarci a strumenti specifici ma, sfruttando solamente le caratteristiche del linguaggio in modo da poter applicare i concetti esposti sia in AutoCAD sia nel mondo IntelliCAD.

Le operazioni di correzione di software possono richiedere molto tempo, soprattutto nel caso si sia in presenza di un programma complesso. Bisogna inoltre tener presente che, l'esperienza del programmatore incide notevolmente, infatti una persona esperta riuscirà, quasi sempre, a risolvere un problema molto prima di un neofita. Non bisogna quindi scoraggiarsi nel caso in cui non si riesca subito a trovare una soluzione ai problemi che si incontrano, tenendo sempre a mente che, più codice si scrive, più sarà facile scriverlo e debuggarlo.

Per poter affrontare il debug di qualsiasi programma, e in genere per poter scrivere qualsiasi software, è necessario essere consci di due fatti. Prima di tutto è indispensabile capire che i programmi per computer non fanno ciò che lo sviluppatore vuole che facciano ma, più semplicemente, fanno ciò il programmatore 'dice' loro di fare. In pratica, se in uno dei vostri software Lisp venisse riscontrato un errore, la maggior parte delle volte, la causa sarebbe da ricercare a metà strada tra la vostra tastiera e il vostro monitor! Esatto! E' proprio così. La causa della maggior parte degli errori nel software è il programmatore!

Questo però non deve far pensare che, chi inserisce un errore all'interno di un programma sia un cattivo programmatore, anche perché non è solo la disattenzione che genera errori. Spesso e volentieri, ci sono altre componenti che influiscono sul modo di sviluppare i programmi, primo fra tutti il tempo, infatti è pratica comune cercare di ridurre al minimo i tempi necessari per lo sviluppo dei software, soprattutto commerciali, cosa che costringe a ritmi più elevati con conseguente aumento degli errori.

Capito questo si può affrontare la ricerca dell'errore in modo più efficace.

Solitamente gli errori che si possono commettere programmando con il linguaggio Lisp,

si possono dividere in tre categorie:

- Errori Sintattici;
- Errori Logici;
- Errori Operativi.

Della prima fanno parte tutti i problemi generati, da esempio, dall'errata digitazione del nome di una variabile o di una funzione:

```
(setq a (distanc p1 p2))
```

dove il nome di funzione corretto è 'distance'.

Un altro errore sintattico è rappresentato dalla dichiarazione di variabili locali con lo stesso nome di funzioni:

```
(defun identificativo1 ( / )
...
)
(defun funzione1 ( / identificativo1 valore)
...
(setq valore (identificativo1))
...
)
```

così facendo, quando si tenta di richiamare la funzione 'identificativo1' in realtà verrà utilizzata l'omonima variabile locale, generando un "simpatico" errore.

L'ennesimo errore sintattico è riscontrabile quando, inavvertitamente, inseriamo troppe parentesi o dimentichiamo parametri necessari durante l'utilizzo delle funzioni.

La seconda tipologia di errore, identifica tutti quei problemi causati dalla scarsa attenzione o dalla fretta, durante il lavoro. Ad esempio:

```
(setq i 100)
(while (> i 0)
      (if (= i 5)
            (setq i (+ i 1))
      )
      (setq i (- i 1))
)
```

Questo spezzone di codice impedirà, di fatto, che il ciclo termini causando l'esecuzione del software per un tempo infinito.

Altro tipico esempio di errore logico è il passaggio di parametri di tipo insensato alle funzioni.

Eseguendo queste linee:

```
(setq p1 nil)
(setq p2 (list 0 0 0))
(setq d (distance p1 p2))
```

otterremo sicuramente un errore. La funzione 'distance', per poter calcolare una distanza, necessita di due punti, al contrario gli viene passato un punto (p2) e un valore nullo (p1), da ciò l'illogicità dell'operazione, causa del problema. In questo caso, per prevenire questa tipologia di malfunzionamento è bene effettuare controlli sui tipi, prima di eseguire le funzioni critiche nel software. Tali verifiche sfrutteranno le funzioni 'Type', 'Listp', 'Numberp' e 'Minusp' per assicurarsi che, le variabili passate alle funzioni, contengano il giusto valore.

Infine abbiamo, quelli che io chiamo "Errori Operativi". In questo caso stiamo parlando di tutte quelle operazioni che, se in teoria sono possibili, nella pratica vengono impedite dal sistema di sviluppo.

Ad esempio, utilizzando la funzione 'exit', all'interno di un 'action_tile' di una dcl, otterremo il bloccaggio del CAD, esattamente come succederebbe se tentassimo di utilizzare la funzione 'command'. Anche se queste operazioni dovrebbero, per logica, essere possibili, il fatto che una finestra (DCL) sia aperta ne impedisce il corretto funzionamento.

Chiariti questi "dettagli", andremo ad esaminare ora alcune interessanti tecniche che, in futuro, ci aiuteranno nell'analisi del software, alla ricerca dell'errore perduto.

Iniziamo con questo piccolo programma:

```
(defun test ( / ent definizione selezione vecchiotxt nuovotxt
pianoOrigine)
(setq pianoOrigine (getvar "CLAYER"))
(setvar "CLAYER" "0")
(setq selezione (entsel "\nSeleziona un testo: "))
(setq testo (getstring "\nNuovo valore testo: "))
(setq ent (car selezione))
(setq definizione (entget ent))
(setq vecchiotxt (assoc 1 definizione))
(setq nuovotxt (cons 1 testo))
(setq definizione (subst nuovotxt
vecchiotxt definizione))
(entmod definizione)
(entupd ent)
```

```
(setvar "CLAYER" pianoOrigine)
(prin1)
)
(test) ;avvia la procedura al caricamento
```

Questa procedura richiede all'utente la selezione di un'entità testo (TEXT) modificandone poi il valore, quest'ultimo immesso dall'operatore attraverso l'apposita richiesta.

Qual è il problema?

Anche se, all'apparenza, il programma non presenta evidenti errori, se l'operatore non selezionasse un'entità testo ma, ad esempio, cliccasse in una zona vuota del disegno, otterrebbe questo "simpatico" messaggio:

```
"errore: tipo di argomento errato: lentityp nil"
```

Come se non bastasse, l'interruzione del programma causerebbe il mancato ripristino del layer corrente, che rimarrebbe impostato su 0. Nell'esempio, questa impostazione è, ai fini del programma, chiaramente superflua ma, dimostra che il bloccaggio di un software può portare a conseguenze più gravi rispetto al semplice messaggio visualizzato.

Per trovare una soluzione, la prima cosa da fare è quella di esaminare con attenzione il messaggio che il CAD fornisce, dal quale si può capire che:

Il messaggio si riferisce ad un argomento, intendendo chiaramente indicare l'argomento di una funzione Lisp;

Una funzione Lisp si aspettava di ricevere un valore valido ma, al suo posto gli è stato passato un valore 'nil', evidentemente inaccettabile.

A ciò dobbiamo aggiungere il fatto che, l'errore si verifica dopo che l'operatore ha immesso i dati richiesti, cioè dopo aver cliccato sull'entità e dopo aver specificato il nuovo testo. Con questi dati, esaminiamo il sorgente, concentrando la nostra attenzione su ciò che accade dopo le istruzioni che richiedono l'interazione dell'utente:

```
(setq ent (car selezione))
(setq definizione (entget ent))
(setq vecchiotxt (assoc 1 definizione))
(setq nuovotxt (cons 1 testo))
(setq definizione (subst nuovotxt
vecchiotxt definizione))
(entmod definizione)
(entupd ent)
(setvar "CLAYER" pianoOrigine)
```

Sicuramente la causa del problema non può essere la funzione ’car’, visto che l’esecuzione sulla linea di comando di ’(car nil)’ non produce alcun errore, possiamo allora incolpare la successiva ’entget’, trovando certamente il colpevole, in quanto l’esecuzione sulla linea di comando di ’(entget nil)’, produce esattamente lo stesso errore generato dal software. Per poter giungere più facilmente a questa conclusione, almeno inizialmente, sarà necessario consultare la documentazione del CAD che si utilizza, infatti il manuale cartaceo e/o in forma elettronica deve essere considerato come la "bibbia" del buon programmatore, e come tale deve essere il punto di partenza per poter rispondere alla proprie domande.

Proprio in merito alla documentazione, si può tranquillamente affermare che, qualsiasi ambiente o linguaggio di programmazione dotato di una scarsa e/o confusa documentazione è, certamente, un "cattivo strumento".

Bene, trovato il colpevole non ci resta che impedire che l’errore si verifichi introducendo qualche tipo di controllo e, magari, un messaggio di errore più appropriato in modo che l’operato comprenda dove ha sbagliato. Ci sono diversi modi per implementare una correzione del genere, il primo e più semplice sistema è quello di inserire un controllo usando la struttura ’if’, in modo da evitare che a ’entget’ venga passato un valore ’nil’, in questo modo:

```
(defun test ( / ent definizione
selezione vecchiotxt
nuovotxt pianoOrigine)
(setq pianoOrigine (getvar "CLAYER"))
(setvar "CLAYER" "0")
(setq selezione (entsel "\nSeleziona un testo: "))
(setq testo (getstring "\nNuovo valore testo: "))
(if selezione
      (progn
            (setq ent (car selezione))
            (setq definizione (entget ent))
            (setq vecchiotxt (assoc 1 definizione))
            (setq nuovotxt (cons 1 testo))
            (setq definizione (subst nuovotxt
vecchiotxt definizione))
            (entmod definizione)
            (entupd ent)
      )
      (progn
            (alert
"Attenzione, non è stata selezionata nessuna entità!")
```

```
      )
)
(setvar "CLAYER" pianoOrigine)
(prin1)
)
(test) ;avvia la procedura al caricamento
```

Come si può vedere, solamente se la variabile 'selezione' è vera (quindi diversa da 'nil') il programma procederà nelle operazioni di modifica, in caso contrario segnalerà la mancata selezione.

Bene, problema risolto.

Purtroppo, molte volte, per ogni problema che si risolve, un altro problema viene trovato. Infatti, il nostro programma soffre ancora di una "incompatibilità" con i layer (piani) bloccati. Nella pratica, se l'entità selezionata si trovasse su un piano bloccato, il software non riuscirebbe ad eseguire le modifiche, nonostante ciò nulla verrebbe segnalato lasciando l'operatore in uno stato di incertezza in quanto il programma non dice nulla, i dati sono stati inseriti correttamente ma non viene modificato nulla!

Anche se qualcuno potrebbe pensare che questo non sia un vero e proprio errore, in realtà lo è, ed è anche uno dei più fastidiosi e subdoli in quanto si nasconde, senza generare alcun messaggio. Così facendo, se l'operatore non si accorgesse della mancata modifica del testo, potrebbe concludere un disegno in modo errato, proprio a causa del nostro software.

Bene, ora saremo costretti a sistemare il tutto, come? Come al solito esistono diverse strade, la più immediata è quella di controllare il valore restituito dalla funzione 'entmod', la quale ritorna 'nil' nel caso in cui non riesca a modificare una entità:

```
(defun test ( / ent definizione selezione
vecchiotxt nuovotxt pianoOrigine
risultato)
(setq pianoOrigine (getvar "CLAYER"))
(setvar "CLAYER" "0")
(setq selezione (entsel "\nSeleziona un testo: "))
(setq testo (getstring "\nNuovo valore testo: "))
(if selezione
      (progn
            (setq ent (car selezione))
            (setq definizione (entget ent))
            (setq vecchiotxt (assoc 1 definizione))
            (setq nuovotxt (cons 1 testo))
            (setq definizione (subst nuovotxt
```

```
vecchiotxt definizione))
            (setq risultato (entmod definizione))
            (if risultato
                  (progn
                        (entupd ent)
                  )
                  (progn
                        (alert
            "Errore nella selezione, verificare stato
layer.")
                  )
            );endif
      )
      (progn
            (alert "Attenzione, selezionare entità!")
      )
)
(setvar "CLAYER" pianoOrigine)
(prin1)
)
(test) ;avvia la procedura al caricamento
```

Su programmi molto semplici come quello visto in questo esempio, gli errori possono essere identificati e risolti con una certa semplicità, purtroppo in presenza di software complessi l'operazione potrebbe non essere tanto immediata.

Per agevolare il lavoro del programmatore possiamo sfruttare alcune caratteristiche avanzate del linguaggio Lisp. Nell'analisi del sorgente, una delle più importanti funzionalità è quella di poter ispezionare il contenuto delle variabili in un determinato punto del listato, ciò consente di comprendere meglio il procedere del software e permette una diagnosi più rapida degli eventuali problemi.

Normalmente, se si desidera verificare il valore di una variabile ci si affida alla funzione 'print', ad esempio, nel nostro precedente esempio, per verificare il valore di "definizione" prima che venga effettuata la modifica con 'entmod', potremmo scrivere:

```
...
(setq definizione (subst nuovotxt
vecchiotxt definizione))
(print definizione)
(setq risultato (entmod definizione))
...
```

Ovviamente, questo tipo di controllo, ci costringe a modificare continuamente il sorgente per posizionare i vari 'print' dove servono. Questo, alla lunga, intralcia l'attività di sviluppo. Non dobbiamo poi dimenticare che, una volta inseriti i 'print', gli stessi dovranno essere eliminati prima di emettere la versione definitiva del programma.

Un'idea interessante è quella di predisporre un'apposita funzione di stampa che possa essere inserita nel codice ma, al contrario di 'print', possa essere attivata o disattivata senza necessariamente modificare i sorgenti. Potremmo costruire:

```
(defun debug_print ( msg / )
(if __DEBUG__
      (progn
            (print msg)
      )
)
)
```

Utilizzeremo la nuova 'debug_print' esattamente come nell'esempio precedente:

```
...
(setq definizione (subst nuovotxt
vecchiotxt definizione))
(debug_print definizione)
(setq risultato (entmod definizione))
...
```

Il vantaggio sta nel fatto che, solamente quando la variabile globale '__DEBUG__' è impostata a 't', la funzione verrà eseguita. Questo sistema permetterà di attivare o disattivare i messaggi per il debug in modo semplice ed immediato, impostando semplicemente la variabile '__DEBUG__', quindi potremo inserire tutti i 'debug_print' che occorrono, senza doversi più preoccupare di rimuoverli.

Anche questo metodo ha però i suoi limiti, il più grande è la mancanza assoluta di interattività, limitandosi alla sola visualizzazione dei dati senza poter intervenire in alcun modo sul flusso del software. Per aggiungere questa importante caratteristica possiamo sfruttare le caratteristiche più avanzate del linguaggio, cioè la sua capacità di autointerpretarsi e l'ereditarietà delle variabili.

Nella pratica, proveremo a realizzare una funzione che consente al programmatore di inserire una interruzione interattiva all'interno del codice sorgente, in modo che il programma si interrompa in un punto preciso, consentendo di verificare il valore delle variabili, di poterne cambiare il contenuto e permettendo l'esecuzione di un qualsiasi codice Lisp.

Implementando la nuova funzione in modo che questa possa essere disattivata utilizzando la variabile "__DEBUG__", potremo usare una procedura del genere:

```
(defun debug_break ( / debug_break_msg debug_break_user)
(if __DEBUG__ ;se debug è attivo
      (progn
             (setq debug_break_msg "\ndebug break: ")
             (while (/= (setq debug_break_user
                    (getstring debug_break_msg)) "")
                    ;per proseguire premere INVIO
                    (cond
                           ((= (strcase debug_break_user)
"Q") ;termine
                           (exit)
                    )
                    ((/= "") ;interpreta input
                           (print (eval
                                  (read debug_break_user)))
                                  ;interpretazione input utente
                    )
                    );endcond
             );endw
      );endp
)
(prin1)
);ednddef
```

Potremo utilizzare la nuova procedura in questo modo:

```
...
(setq definizione (subst nuovotxt
vecchiotxt definizione))
(debug_break)
(setq risultato (entmod definizione))
...
```

Quando il programma arriverà a 'debug_break', il software visualizzerà la richiesta:

```
debug break:
```

alla quale si potrà rispondere in tre modi. Premendo invio il programma continuerà normalmente, come se la funzione non esistesse, inserendo "Q" che terminerà immediatamente l'esecuzione del software infine, sarà possibile eseguire qualsiasi istruzione lisp, cosa che ci consentirà di ispezionare e modificare il valore delle variabili.

Per comprendere meglio, vediamo alcuni esempi applicati al nostro sorgente:

```
debug break: (print definizione)
```

Questa richiesta stamperà il contenuto della variabile "definizione".

```
debug break: Q
```

L'utilizzo di "Q" interromperà il programma.

```
debug break: (setq definizione "prova")
```

In questo caso verrà modificato il valore di "definizione".

In aggiunta, grazie all'ereditarietà delle variabili, saremo in grado di interrogare e modificare non solo le variabili locali alla funzione nella quale inseriremo 'debug_break', ma anche tutte quelle utilizzate nelle procedure che la richiamano. Ad esempio:

```
(defun fx2 ( / varA varB)
...
(debug break)
...
)
(defun fx1 ( / var1 var2 var3)
...
(fx2)
...
)
```

In questo caso, durante l'interruzione causata da 'debug_break', saremo in grado di agire sia su 'varA' e 'varB', sia su 'var1', 'var2' e 'var3'.

Infine, potremo disattivare i nostri punti di interruzioni impostando a 'nil' la variabile '__DEBUG__'.

Utilizzando queste metodologie, l'analisi del sorgente per l'individuazione dei bug sarà, certamente, più confortevole e produttiva, riducendo drasticamente la frustrazione derivante dall'uso dei semplici messaggi fatti con 'print' o 'alert'.

Tradurre i messaggi di un programma

Quando si realizzano software destinati a molti utenti, o più semplicemente quando i nostri software devo essere utilizzati da personale non italiano, la traduzione dei messaggi e delle maschere risulta determinante.

Con i programmi scritti con Lisp ci sono solitamente due tipi di file che deveno essere

modificati per poter fornire un software multilingua. Prima di tutti i file DCL che, come sappiamo, sono le maschere di dialogo. In questo caso tali file sono di difficile parametrizzazione e, normalmente, ne esiste uno per ogni lingua. Purtroppo questo fatto complica la manutenzione ma è, solitamente, la strada più semplice. Ovviamente, i file Lisp che li utilizzeranno non faranno altro che caricare la DCL nella lingua corrente, magari basandosi sul nome del file (es.: miofile-it.dcl, miofile-en.dcl, ecc).

Discorso molto diverso per il programma vero e proprio, contenuto nei file .LSP. In questo caso la parametrizzazione è d'obbligo e può essere fatta con una miriade di tecniche diverse. In questo caso esamineremo una delle piu semplici.

Iniziamo subito analizzando la procedura che andremo a parametrizzare per supportare la gestione multilingua :

```
;disegnare funzioni matematiche con il linguaggio Lisp
; Autore: Roberto Rossi
; Versione: 1.0.0
(defun c:DrawFx ( / funzione mini mass passo)
(setq funzione (getstring "\nFunzione da disegnare(racchiusa
tra doppi apici): "));endset
(setq mini (getreal "\nInizio calcolo funzione > 0: "))
(if (or (not mini) (<= mini 0))
      (setq mini 0.01)
);endif
(setq mass (getreal "\nFine calcolo funzione > 0: "))
(if (not mass)
      (setq mass 10)
);endif
(setq passo (getreal "\nIntervallo (es.: 0.1): "))
(if (not passo)
      (setq passo 0.1)
);endif
(if (/= funzione "")
      (progn
            (resetVar)
            (rfx funzione mini mass passo)
            (princ "\nDisegno funzione concluso con
successo.")
      );endp
);endif
(prin1)
);enddef
```

```
;imposta le variabili di sistema in modo che non
interferiscano
;con la procedura
(defun resetVar ( )
(setvar "OSMODE" 0)
(setvar "UCSICON" 0)
(setvar "CMDECHO" 0)
)

;disegna gli assi cartesiani
(defun drawAxis ( ext dmaxx dmaxy / x xmin y ymin)
(setq x (+ (* (/ dmaxx 100) ext) dmaxx))
(setq y (+ (* (/ dmaxy 100) ext) dmaxy))
(command "_line" (list 0 0 0) (list x 0 0) ""
"_change" (entlast) "" "_p" "_C" "1" "" "_line"
(list 0 (* -1 y) 0) (list 0 y 0)
"" "_change" (entlast) "" "_p" "_C" "1" ""
)
);endd

;disegna una funzione matematica
;
;Parametri:
; funzione = funzione da rappresentare es.: "(sin x")
; xmin = valore di x per l'inizio del calcolo
; xmax = valore di x per la fine del calcolo
; passo = passo tra un calcolo e il successivo
(defun rfx ( funzione xmin xmax passo / x y fxtmp ymax)
;trasforma la funzione da stringa a simboli LISP
(setq fxtmp (read funzione))
(setq x xmin)
(setq ymax 0)
;inizia la rappresentazione
(command "_line")
;continua il calcolo da xmin a xmax
(while (> xmax xmin)
      (setq x xmin)
```

```
      ;valuta la funzione
      (setq y (eval fxtmp))
      (if (> y ymax)
            (setq ymax y)
      );endif
      ;disegna il punto
      (command (list x y 0))
      (setq xmin (+ xmin passo))
);endw
(command "");termina il disegno
;disegna gli assi cartesiani
(drawAxis 20 xmax ymax)
;modifica la visualizzazione
(command "_zoom" "_e" "_zoom" ".5x")
(prin1)
);enddef
```

Questo programma lo avrete sicuramente riconosciuto, è stato presentato in un'altro capitolo di questo testo e permette di disegnare una funzione matematica semplice, sfruttando la potenza del nostro CAD.

Anche se le frasi che il software è in grado di mostrare non sono molte, sono più che sufficienti per spiegare il concetto.

Iniziamo stilando la lista di come dovrà essere la nostra procedura una volta terminata :

- Dovrà poter gestire un numero infinito di lingue;
- Il programma imposterà la propria lingua in base alla lingua del sistema nel quale viene eseguito;
- Nel caso in cui la lingua del sistema non sia prevista, verrà impostata automaticamente la lingua italiana. Nel caso in cui anche quest'ultima non sia disponibile il software dovrà avvisare l'utente ed interrompere il proprio funzionamento;
- Le frasi di ogni lingua verranno salvate all'interno di un file;
- I file contenenti le frasi dovranno essere facilmente modificabili anche dai non programmatori;
- Ogni frase sarà identificata di un numero intero in modo univoco;

- Nessun limite al numero di frasi possibili.

Anche se le specifiche presentate possono sembrare impegnative, nella realtà è tutto molto più semplici di quanto appaia. La gestione multilingua verrà affidata ad un apposito gruppo di funzioni specifiche che ora andremo ad esaminare nel dettaglio.

Iniziamo vedendo come le frasi verranno memorizzate nei file di lingua. Esaminiamo il file di definizione per le frasi in italiano (disponibile nel file “locale-ita.lst”):

```
(1 "\nFunzione da disegnare(racchiusa tra doppi apici): ")
(2 "\nInizio calcolo funzione > 0: ")
(3 "\nFine calcolo funzione > 0: ")
(4 "\nIntervallo (es.: 0.1): ")
(5 "\nDisegno funzione concluso con successo.")
```

Le particolarità di questo formato sono cosi riassumibili:

- Ogni linea inizia e termina con una parentesi tonda;
- Dopo aver aperto la parentesi viene inserito l’indice numerico univoco che identifica la frase, seguito da uno spazio;
- Tutte le frasi sono specificate tra doppi apici e separate dall’indice attraverso l’uso di uno spazio.

Vediamo ora il codice completo della libreria di funzioni che gestirà interamente il caricamento e l’utilizzo dei file di frasi (disponibile nel file “ml_lib.lsp”):

```
;questa variabile globale conterrà l’intero elenco di frasi
definite per la lingua corrente
(setq ml_languageDatabaseList nil)


;ritorna la lingua corrente, se non è possibile identificarla
automaticamente
;questa verrà impostata su italiano
(defun ml_loadGetLocale ( / locale)
  (setq locale (getvar "locale"))
  (if (not locale)
    (progn
      (princ "\nNon è stato possibile deterinare la lingua del
sistema, verrà impostata la lingua italiana.\n")
      (setq locale "ITA")
    );endp
  );endiif
```

```
  locale
);enddef

;questa funzione carica il file di lingua in base al cad
utilizzato al momento
;E' opzionalmente possibile indicare il nome del file di
lingua da caricare
(defun ml_loadLanguageFile ( fixFileName forceReload /
languageDatabaseName locale idfile
                                    filename line lstline)
  ;il caricamento avviene solo quando non è stato gia caricato
un database oppure quando
  ;espressamente richiesto dal parametro forceReload
  (if (or (not ml_languageDatabaseList) forceReload)
    (progn
      (setq ml_languageDatabaseList nil) ;reset database
generale frasi

      (setq locale (ml_loadGetLocale));lettura lingua corrente

      ;questo è il nome standard del file dal quale verranno
lette le frasi
      ;al posto di [] verrà inserito l'identificativo della
lingua
      ;(es.:locale-ita.lst, locale-eng.lst, ecc...)
      (setq languageDatabaseName (strcat "locale-" locale
".lst"))

      (if fixFileName ;utilizzo file specificato
      (setq languageDatabaseName fixFileName)
      );endif

      ;caricamento file
      (print languageDatabaseName)
      (setq filename (findfile languageDatabaseName))
      (print filename)
      (if filename
      (progn
        (setq idfile (open filename "r"))
```

```
        (if idfile
          (progn
            (setq line (read-line idfile))
            (while line
             (setq lstline (read line))
             (if (= (type lstline) 'LIST);minimo controllo
sintassi linea
               (setq ml_languageDatabaseList (cons lstline
ml_languageDatabaseList))
             );endif
             (setq line (read-line idfile))
            );endw
            (close idfile)
          );endp
          (progn
            (princ (strcat "\nImpossibile caricare il file di
lingua : " filename))
            (exit)
          );endp
        );endif
      );endp
      (progn
        (princ (strcat "\nImpossibile trovare il file di
lingua : " languageDatabaseName))
        (exit)
      );endp
      );endif
    );endp
  );endif

);enddef

;questa funzione, ritorna la frase corrispondente all'indice
dato
(defun ml_get (idx / result el)
  (setq result "")
  (ml_loadLanguageFile nil nil)

  ;se lista database valida, ricerca frase e restituzione
```

```
  (if ml_languageDatabaseList
    (progn
      (setq el (assoc idx ml_languageDatabaseList))
      (if el
     (setq result (cadr el));frase trovata
      );endif
    );endp
  );endif

  result
);enddef
```

Della nostra libreria di gestione multilingua utilizzeremo esclusivamente una funzione, “ml_get”, alla quale passeremo semplicemente l’indice della frase che vogliamo ottenere. La stessa funzione si incaricherà di stabilire quale lista di frasi utilizzare e si occuperà di caricarla nel caso la cosa non sia ancora stata fatta. Affinchè il tutto funzioni correttamente, i file “.lst” relativi alle varie lingue e i vari file “.lsp”, devono risiedere in uno dei percorsi di ricerca del CAD.

Ora guardiamo il codice del programma iniziale, modificato con l’aggiunta dell’uso della nostra libreria di traduzione. Per semplicità esamineremo solo la funzione dove vengono utilizzate le frasi tradotte :

```
(load "ml_lib.lsp") ;caricamento libreria di traduzione

(defun c:DrawFx ( / funzione mini mass passo)
;"\nFunzione da disegnare(racchiusa tra doppi apici): "
(setq funzione (getstring (ml_get 1)));endset
;"\nInizio calcolo funzione > 0: "
(setq mini (getreal (ml_get 2)))
(if (or (not mini) (<= mini 0))
     (setq mini 0.01)
);endif
;"\nFine calcolo funzione > 0: "
(setq mass (getreal (ml_get 3))
(if (not mass)
     (setq mass 10)
);endif
;"\nIntervallo (es.: 0.1): "
(setq passo (getreal (ml_get 4))
(if (not passo)
```

```
      (setq passo 0.1)
);endif
(if (/= funzione "")
      (progn
            (resetVar)
            (rfx funzione mini mass passo)
            ;"\nDisegno funzione concluso con successo."
            (princ (ml_get 5))
      );endp
);endif
(prin1)
);enddef

; il sorgente continua ...
```

Come si può osservare l'inserimento del supporto multilingua, in questo caso, è particolarmente semplice. Per rendere più comprensibile il programma sono state inserite le frasi come commenti, questo aumenta la leggibilità del sorgente.

Nella pratica, sarà sufficiente sostituire, ad ogni stringa fissa, la chiamata ad una funzione specificando il corrispondente indice numerico univoco.

Processare più disegni automaticamente

Prima o poi, chi sviluppa applicazioni Lisp, si trova a dover lavorare su più file, di conseguenza si trova nella necessità di aprire un documento, elaborarlo, quindi salvarlo e chiuderlo. Chi per la prima volta si cimentasse in queste operazioni si accorgerebbe che, qualcosa non va.

Se si provasse ad eseguire il comando "apri" (_OPEN) da un file lisp ci si renderebbe conto che, terminata l'operazione di apertura, la procedura Lisp viene sempre interrotta.

Questo potrebbe far pensare che il linguaggio Lisp non sia in grado di elaborare più file in sequenza. In realtà sfruttando la capacità del CAD di eseguire, oltre a Lisp, anche dei semplici script di comandi è possibile aprire e chiudere più file eseguendo all'interno di ognuno una qualsiasi procedura.

Il processo può essere cosi riassunto :

- Viene lanciato un comando Lisp;
- Lisp chiede all'utente quali file processare;

- Lisp crea uno script che, apre un file alla volta e dopo averlo aperto carica ed esegue una procedura Lisp;
- Lisp esegue lo script che ha appena creato.

In pratica il nostro programma sarà diviso in due parti, la prima che si occuperà di gestione lo script e la seconda che agirà sui singoli file.

A questo punto facciamo un esempio pratico. Ipotizziamo di voler modificare un riferimento esterno all'interno di N file cambiando il percorso del dwg al quale punta, quindi crea un file zip usando il comando "_etransmit".

Iniziamo approntando il file Lisp che esegue la modifica del riferimento esterno. In particolare scriviamo un semplice programma Lisp che modifica il percorso del riferimento esterno di nome "test", collegandolo al file "c:\test\drawext.dwg":

```
(command "_-xref" "_p" "TEST" "c:/test/new_drawext.dwg")
(command "_-etransmit" "_c" "[nome file]")
(command "_qsave")
```

Il nostro script è molto piccolo, poche istruzioni. Ovviamente è possibile estendere il tutto inserendo un programma più complesso. Naturalmente, al posto di "[nome file]" verrà inserito il percorso completo del file .zip che verrà generato.

Passiamo ora ad analizzare il file lisp che si occuperà della scansione dei file, della creazione dello script e della sua esecuzione.

In questo caso il programma è decisamente più complesso, eccolo :

```
; Author : Roberto Rossi
; Web : http://www.redchar.net
; Version : 1.0.2
;
; Questa procedura si occupa di sostituire il file collegato a
riferimento esterno (XREF / XRIF).
; Il programma data una cartella con i file dwg da elaborare,
il nome del
; riferimento esterno da modificare e il nuovo file da
collegare ai riferimenti,
; modificherà tutti i dwg presenti nella cartella del file
indicato.


;main procedure
(defun main ( / filesList fileName folderName scriptName
scriptPath lispName lispPath newpath xrefname)
```

```
      (setq scriptName "gest.scr") ;nome file script che verrà
generato
      (setq lispName "gest.lsp") ;nome file lisp che verrà
generato
      (setq xrefname (getstring "\nNome riferimento esterno da
sostituire o [TEST]: "))
      (setq fileName (getfiled "Selezionare uno dei file dwg
da trattare" "" "dwg" 0))
      (setq newpath (getfiled "Selezionare nuovo file per
riferimento esterno" "" "dwg" 0))
      (if (and fileName newpath)
      (progn
            (setq folderName (vl-filename-directory
fileName)) ;estrazione percorso
            (setq folderName (reverseBar folderName))
;cartella dwg, conversione barre \ in /
            (setq newpath (reverseBar newpath)) ;percorso
nuovo file per xref, conversione barre \ in /
            (if (= xrefname "")
                  (setq xrefname "TEST")
            )
            (setq filesList (vl-directory-files folderName
"*.dwg")) ;estra elenco dwg da elaborare
            (setq scriptPath (strcat folderName "/"
scriptName)) ;percorso script da creare
            (setq lispPath (strcat folderName "/"
lispName)) ;percorso lisp da creare
            (if filesList
            (progn
                  (vl-file-delete scriptPath) ;elimina
precedente script
                  (vl-file-delete lispPath) ;elimina
precedente lisp
                  ;crea script e lisp
                  (createScript scriptPath lispPath nil
folderName filesList newpath xrefname)
            );endp
            );endif filesList
            (createScript scriptPath nil t nil nil nil
nil) ;esecuzione script creato
      );endp
```

```
      );endif
(prin1)
);Enddef

;create script and/or execute it
(defun createScript (scriptName lispName exec folderName
filesList newpath xrefname / line fileName idfileL idfileS)
      (if exec
      (progn
            ;esecuzione script
            (if (findfile scriptName)
                  (command "_script" scriptName)
            );endif
      );endp
      (progn
            ;creazione script
            (foreach fileName filesList
                  (if (not idfileS)
                        (setq idfileS (open scriptName "w"))
                  )
                  ;scrittura script
                  (write-line "_open" idfileS) ;apertura file
                  (write-line (strcat folderName "/"
fileName) idfileS)
                  (write-line (strcat "(load \"" lispName
"\")") idfileS) caricamento procedura lisp
                  (write-line "_close" idfileS)
            );endfor
            (if idfileS
                  (close idfileS)
            );endif
            (setq idfileL (open lispName "w"))
            (if idfileL
            (progn
                  ;scrittura file lisp
                  (write-line (strcat "(command \"_-
xref\" \"_p\" \"" xrefname "\" \"" newpath "\")") idfileL)
                  (write-line "(command \"_qsave\")" idfileL)
```

```
                    (write-line (strcat "(command \"_-
etransmit\" \"_c\" (strcat (getvar \"dwgprefix\")
(getvar \"dwgname\") \".zip\") )") idfileL)
                    (close idfileL)
             );Endp
             )
      );endp
      );Endif
);enddef

;inverti le barre \ > /
(defun reverseBar (string / ch i ls result)
      (setq result "")
      (setq ls (strlen string))
      (setq i 0)
      (while (< i ls) ;scansiona stringa un carattere alla
volta
             (setq ch (substr string (1+ i) 1))
             (if (= ch "\\")
                    (setq ch "/")
             )
             (setq result (strcat result ch))
             (setq i (1+ i))
      )
result
)

ù;avvio procedura al caricamento
(main)
```

Il programma non fa altro che replicare la struttura proposta all'inizio. Si occupa di creare lo script che apre i file ed esegue, su ognuno, una specifica procedura lisp ("gest.lsp"), dopo di che crea il file lisp che verrà eseguito su ogni documento, "gest.lsp".

Come si può vedere, al termine del caricamento del file lisp, viene eseguita immediatamente la funzione "main" che si occupa di avviare il programma.

Come salvare tutti i blocchi presenti in un disegno

Questa sezione si pone come obbiettivo quello di spiegare come, dato un disegno contenente dei blocchi, sia possibile salvarli tutti su disco in modo da ottenere una cartella con tanti dwg quanti sono i blocchi presenti.

Quello che andremo a realizzare è un buon esempio di applicazione delle tecniche di base utilizzate dal linguaggio Lisp nei CAD, in particolare tratteremo la scansione delle tabelle dei blocchi, l'utilizzo delle funzioni di gestione file e l'uso della funzione "command".

Il programma agirà in questo modo:

- Prima di tutto verrà chiesto all'utente in quale cartella desidera salvare i blocchi del disegno corrente;
- Una volta inserito, verificherà che il percorso specificato sia effettivamente valido e che sia possibile scrivere al suo interno;
- Analizzerà il disegno corrente per ottenere l'elenco dei blocchi da salvare;
- Utilizzando il comando "_wblock" procederà al salvataggio di tutti i blocchi nella cartella specificata dall'utente.

Il funzionamento del software sarà particolarmente semplice, una volta caricato il file Lisp ("save_blocks.lsp"), basterà digitare il comando "SaveAllBlocks".

Ecco il sorgente completo :

```
;Questa procedura salva, all'interno di una cartella, tutti i
blocchi presenti
;nel disegno corrente

;ritorna l'elenco dei blocchi presenti nel disegno corrente
(defun getBlocksList ( / result blk)
  ;ottiene il primo blocco presente nel disegno
  (setq blk (tblnext "BLOCK" t))
  (while blk
      ;scorre i restanti blocchi presenti e ne ricava il nome
    (setq blk (cdr (assoc 2 blk)))
    (setq result (cons blk result))
    (setq blk (tblnext "BLOCK" nil))
  );endwhile
```

```
result
);enddef

;chiede all'utente di selezionare una cartella e ne
restituisce il percorso
(defun getFolder ( / result idf path ch chpath continue)
  (setq continue t)
  (while continue
    (setq path (getstring "\nSelezionare percorso di
destinazione : "))
    (if (and path (/= path ""))
      (progn
            ;verifica la presenza del carattere finale \ o /
      (setq chpath "")
      (setq ch (substr path (strlen path) 1))
      (if (and (/= ch "\\") (/= ch "/"))
        (setq chpath "\\")
      )

            ;verifica se la cartella è scrivibile
      (setq idf (open (strcat path chpath "temp.tmp") "w"))
      (if idf
        (progn
          (close idf)
          (setq result (strcat path chpath))
          (setq continue nil)
        );endp
        (progn
          (princ "\nIl percorso inserito non è valido!
Riprovare.")
        )
      );endif
      );endp
      (progn
      (if (= path "")
        (setq continue nil)
      )
      )
```

```
    );endif
  );endw
result
);enddef

;questo comanda salva tutti i blocchi presenti su disco, nella
cartella
;specificata dall'utente
(defun c:saveAllBlocks ( / blk blklist folder blkname filename
i)
  (setq folder (getFolder)) ;chiede percorso a utente
  (if folder ;solo se percorso è valido
    (progn
            ;legge lista dei blocchi da salvare
      (setq blklist (getBlocksList))
            ;calcola numero di blocchi esistenti
      (setq i (length blklist))
            ;sopprime i messaggi superflui generati dai
commandi del cad
      (setvar "cmdecho" 0)
            ;scorre un blocco alla volta
      (foreach blkname blklist
            ;crea percorso per salvataggio blocco
      (setq filename (strcat folder blkname ".dwg"))
            ;verifica presenza copia precedente
      (if (findfile filename)
                  ;ellimina vecchio file
        (vl-file-delete filename)
      );endif
            ;salva blocco
      (command "_wblock" filename blkname)
            ;mostra numero blocchi mancanti al termine
      (princ (strcat "\nBlocchi da salvare : " (itoa i)))
      (setq i (1- i))
      );endfor
      (princ "\nProcedura conclusa.")
    );endp
  );endif
```

```
(prin1)
);enddef
```

Come si può vedere il listato non è particolarmente complesso. Durante la sua stesura sono stati inseriti alcuni commenti nei punti chiave in modo da rendere tutto più comprensibile. Lo script è immediatamente utilizzabile senza alcuna modifica.

Distinguere un blocco da un Xrif

Una delle caratteristiche più interessanti dei moderni cad è quella di poter inserire, nei disegni, altri disegni senza includerli realmente all'interno del file dwg. Al contrario dei blocchi, che risiedono completamente nel disegno, i cad dispongono di quelli che vengono chiamati "riferimenti esterni" (xrif o xref). In questo caso, potremo inserire un disegni, all'interno del documento corrente, senza che questo venga incluso realmente, ottenendo comunque di visualizzarlo come parte del progetto corrente.

Quando, da programma, si tenta di agire su un riferimento esterno, la prima cosa che salta all'occhio è la sua "somiglianza" con le normali entità "blocco". Questo può inizialmente disorientare, anche se poi la questione è più immediata di quanto possa apparire.

Uno dei primi problemi che si incontra è come distinguere un normale blocco da un xrif. In questo caso tutto si risolve con la tabella dei blocchi.

E' importante ricordare che, ogni blocco o xrif inserito nel disegno, non contiene tutti i dati necessari alla sua visualizzazione ma solamente l'indicazione di quale blocco sia inserito. I dati necessari alla rappresentazione sono contenuti in quella che viene denominata "tabella dei blocchi", che altro non è se non una parte di dwg nella quale vengono memorizzati i dati sui blocchi e sugli xrif.

Facciamo un esempio.

Iniziamo con l'osservare la definizione di due entità, un blocco e un riferimento esterno, inseriti in un disegno, usando l'espressione lisp:

```
(entget (car (entsel)))
```

Iniziamo da in ipotetico blocco di nome "TEST":

```
((-1 . <Entity name: 7ffffb05cf0>) (0 . "INSERT") (330 .
<Entity name:7ffffb039f0>) (5 . "247") (100 . "AcDbEntity")
(67 . 0) (410 . "Model") (8 . "0") (100 .
"AcDbBlockReference") (2 . "test") (10 0.0 0.0 0.0) (41 . 1.0)
(42 . 1.0) (43 . 1.0) (50 . 0.0) (70 . 0) (71 . 0) (44 . 0.0)
(45 . 0.0) (210 0.0 0.0 1.0))
```

Confrontiamolo con un ipotetico riferimento esterno (xrif) "TEST1":

```
((-1 . <Entity name: 7ffffb05d40>) (0 . "INSERT") (330 .
<Entity name:7ffffb039f0>) (5 . "24C") (100 . "AcDbEntity")
(67 . 0) (410 . "Model") (8 . "0") (100 .
"AcDbBlockReference") (2 . "test1") (10 2827.36 1366.49 0.0)
(41 . 1.0) (42 . 1.0) (43 . 1.0) (50 . 0.0) (70 . 0) (71 . 0)
(44 . 0.0) (45 . 0.0) (210 0.0 0.0 1.0))
```

Come si può osservare i dati sono molto simili e non consentono di distinguere il blocco dal riferimento esterno.

Proviamo ora ad osservare le definizioni dei due, prese dalla tabella dei blocchi, sfruttando la seguente istruzione lisp:

```
(tblsearch "block" [nome blocco])
```

Iniziamo con "TEST":

```
(tblsearch "block" "test")
((0 . "BLOCK") (2 . "test") (70 . 0) (10 0.0 0.0 0.0) (-2 .
<Entity name:7ffffb05c70>))
Passiamo ora a "TEST1" (il riferimento esterno):
(tblsearch "block" "test1")
((0 . "BLOCK") (2 . "test1") (70 . 36) (10 0.0 0.0 0.0) (1 .
"C:\\cartella\\file.dwg") (-2 . <Entity name: 7ffffb09a40>))
```

Ciò che si nota è che il riferimento esterno presenta alcuni dati aggiuntivi rispetto ad un comune blocco. Consultando la documentazione relativa al formato DXF è possibile comprendere meglio :

http://usa.autodesk.com/adsk/servlet/item?siteID=3D123112&id=3D12272454&lin=kID=3D10809853

Il codice DXF 70 permette infatti di stabilire la natura dell'elemento, in particolare consente di capire se è un xrif (riferimento esterno), che tipo di xrif è, ed in genere consente di stabilire la tipologia di blocco, inoltre il codice DXF 1 ci indica il percorso del riferimento esterno collegato.

Capitolo 17
Entità non grafiche, xrecord e dizionari

Nella programmazione di un software LISP può capitare di avere la necessità di immagazzinare delle informazioni a prescindere dalla presenza di entità grafiche nel disegno.

Esistono due sistemi per compiere tale operazione. Il primo coinvolge l'uso delle entità estese che sono dei dati non visibili associabili ad oggetti grafici già presenti nel disegno.

Il secondo sistema prevede l'utilizzo di xrecord e dizionari e non richiede alcun tipo di oggetto grafico per poter essere utilizzato. Quello che approfondiremo in queste pagine è proprio quest'ultimo metodo.

La specifica entità non-grafica che viene utilizzata per questo scopo è il dizionario.

Il dizionario è un'entità particolare che non si differenzia dalle altre entità salvo per il fatto di essere non-grafica ed è strutturata per contenere informazioni di vario genere.

Nella figura è rappresentata la struttura operativa dei dizionari.

Si nota in maniera chiara come funziona il meccanismo, infatti i vari dizionari vengono costruiti ed attaccati a partire da un'entità root (o entità principale) che sta all'inizio della catena e che viene creata automaticamente dal cad.

Le informazioni vengono poi immagazzinate all'interno dei vari elementi sotto forma di entità chiamate xrecord.

Un xrecord può contenere qualunque tipo di informazione possa servire al programma e, a differenza dei dati estesi (associati a entità grafiche) che hanno una lunghezza

fissata in 1000 caratteri (byte), non ha limite di lunghezza.

La gestione dei dizionari è affidata all'utilizzo di 6 funzioni che andremo ad esaminare in dettaglio:

- namedobjdict
- dictadd
- dictsearch
- dictnext
- dictrename
- dictremove

Lo scopo del testo sarà quindi quello di costruire un dizionario nuovo (in rosso nella figura) che verrà creato accanto a quelli neri (dizionari esistenti) utilizzando le funzioni appena citate.

Le varie funzioni verranno trattate in dettaglio anche se alcune di esse sono veramente intuitive e richiedono solo un minimo di pratica.

La funzione namedobjdict

E' questa, forse, la funzione più importante nella gestione dei dizionari in quanto permette di accedere all'entità root e al suo contenuto; infatti ritorna, come risultato, il nome dell'entità root.

Sulla riga di comando si digiti:

```
(namedobjdict)
```

Si otterrà qualcosa del genere:

```
<Entity name: 7ef51c60>
```

Piccola precisazione sui nomi di entità...

Il nome di entità ritornato da (namedobjdict) sarà quasi certamente diverso da quello scritto nel testo, in quanto AutoCAD assegna i nomi alle entità all'apertura del disegno, anzi può capitare spesso di aprire e chiudere un disegno senza salvare e senza fare modifiche e ritrovare nomi di entità diversi. Per cui i nomi delle entità valgono solo per la sessione di disegno corrente. Questa precisazione vale per tutti i nomi di entità contenuti in questo testo.

Assegnamo ad una variabile il valore ritornato da (namedobjdict), si digiti:

```
(setq noment (namedobjdict))
```

e vediamo cosa contiene digitando:

```
(entget noment)
```

Si otterrà qualcosa del genere:

```
((-1.<Entity name: 7ef51c60>)(0."DICTIONARY")(330.<Entity
name: 0>)(5."C")
(100."AcDbDictionary")(280 . 0)(281 . 1)
(3."ACAD_COLOR")(350.<Entity name: 7ef51dd8>)
(3."ACAD_GROUP")(350.<Entity name: 7ef51c68>)
(3."ACAD_LAYOUT")(350.<Entity name: 7ef51cd0>)
(3."ACAD_MATERIAL")(350.<Entity name: 7ef51dd0>)
(3."ACAD_MLINESTYLE")(350.<Entity name: 7ef51cb8>)
(3."ACAD_PLOTSETTINGS")(350.<Entity name: 7ef51cc8>)
(3."ACAD_PLOTSTYLENAME")(350.<Entity name: 7ef51c70>)
(3."ACAD_TABLESTYLE")(350.<Entity name: 7ef51e30>)
(3."AcDbVariableDictionary")(350.<Entity name: 7ef51d70>))
```

Dando uno sguardo attento a questa lista di coppie puntate notiamo subito che l'entità root è un dizionario (0 . "DICTIONARY") e che al suo interno sono contenuti i nomi dei dizionari che fanno parte dell'identità root.

I nomi di questi dizionari si ricavano dalle coppie puntate di codice dxf 3 (3 . "Nome dizionario"), ad es. (3 . "ACAD_COLOR") significa che all'interno dell'entità root esiste il dizionario chiamato "ACAD_COLOR".

Nell'esempio precedente si nota la presenza di nove dizionari:

- ACAD_COLOR
- ACAD_GROUP
- ACAD_LAYOUT
- ACAD_MATERIAL
- ACAD_ MLINESTYLE
- ACAD_ PLOTSETTINGS
- ACAD_ PLOTSTYLENAME
- ACAD_ TABLESTYLE

- AcDbVariableDictionary

Quando avremo aggiunto il nostro dizionario, eseguendo le istruzioni che vedremo tra poco, troveremo anche questo nell'entità root.

La funzione dictadd

Dopo aver esaminato la funzione "namedobjdict" è giunto il momento di aggiungere il nostro dizionario all'entità root.

Questo scopo si raggiunge utilizzando la funzione dictadd la cui sintassi è:

(dictadd entità_dizionario_root nome_dizionario entità_dizionario)

dove:

entità_dizionario_root sarebbe l'entità restituita da "namedobjdict";

nome_dizionario è un testo con il nome da dare al dizionario;

entità_dizionario è l'entità associata al nuovo dizionario.

Per prima cosa creiamo l'entità per il nuovo dizionario, che altro non è che una lista di coppie puntate.

Digitiamo:

```
(setq dizionario (list '(0."DICTIONARY") '(100.
"AcDbDictionary")))
```

oppure se si preferisce l'uso della funzione "cons":

```
(setq dizionario (list (cons 0 "DICTIONARY") (cons 100
"AcDbDictionary")))
```

in entrambi i casi la variabile dizionario sarà una lista di coppie puntate:

```
((0 . "DICTIONARY")(100 . "AcDbDictionary"))
```

Per passare dalla lista all'entità basterà utilizzare la funzione "entmakex"; possiamo tranquillamente riutilizzare la stessa variabile:

```
(setq dizionario (entmakex dizionario))
```

otterremo qualcosa del genere:

```
<Entity name: 7ef53700>
```

La nuova entità-dizionario è stata ottenuta.

Non ci resta che aggiungere la nuova entità, a cui daremo il nome "Nuovo_dizionario_1", alla root tramite la funzione "dictadd".

Basterà quindi digitare:

```
(dictadd (namedobjdict) "Nuovo_dizionario_1" dizionario)
```

e se tutto va a buon fine la funzione ritorna il nome dell'entità del nuovo dizionario , che è lo stesso ottenuto da entmakex:

```
<Entity name: 7ef53700>
```

Per accertarsi se veramente questo dizionario è stato aggiunto alla root digitiamo:

```
(entget (namedobjdict))
```

e noteremo in fondo il nostro nuovo dizionario:

```
((-1.<Entity name: 7ef51c60>)(0."DICTIONARY")(330.<Entity
name: 0>) (5 . "C") (100."AcDbDictionary")(280.0)(281.1)
(3."ACAD_COLOR")(350.<Entity name: 7ef51dd8>)
---
---
(3."ACAD_TABLESTYLE")(350.<Entity name: 7ef51e30>)
(3."AcDbVariableDictionary")(350.<Entity name: 7ef51d70>))
(3."Nuovo_dizionario_1")(350.<Entity name: 7ef53700>))
```

Il dizionario è stato creato, dobbiamo adesso aggiungere i dati che, come già accennato prima, sono costituiti da entità chiamate **Xrecord**.

Seguendo un procedimento analogo a quello usato per creare il dizionario scriviamo la lista di coppie puntate tipica di un **Xrecord**.

Digitiamo:

```
(setq xrecord (list '(0."XRECORD") '(100."AcDbXrecord")))
```

oppure se si preferisce l'uso della funzione "cons":

```
(setq xrecord (list (cons 0 "XRECORD") (cons 100
"AcDbXrecord")))
```

in entrambi i casi la variabile xrecord sarà una lista di coppie puntate:

```
((0 . "XRECORD")(100 . "AcDbXrecord"))
```

La lista per l'entità Xrecord è pronta, ma mancano ancora i suoi dati.

Prepariamo una lista anche per questi...

Digitiamo:

```
(setq lista_dati (list '(1."Dato_1") '(2."Dato_2")))
```

oppure se si preferisce l'uso della funzione cons:

```
(setq lista_dati (list (cons 1 "Dato_1") (cons 2 "Dato_2")))
```

in entrambi i casi la variabile “lista_dati” sarà una lista di coppie puntate:

```
((1 . "Dato_1")(2 . "Dato_2"))
```

Aggiungiamo la lista_dati alla lista dell’entità Xrecord. Digitiamo:

```
(setq xrecord (append xrecord lista_dati))
```

e quindi... xrecord sarà uguale alla lista...

```
((0."XRECORD")(100."AcDbXrecord") (1."Dato_1")(2."Dato_2"))
```

Per passare dalla lista all’entità basterà utilizzare la funzione “entmakex”; possiamo tranquillamente riutilizzare la stessa variabile:

```
(setq xrecord (entmakex xrecord))
```

otterremo qualcosa del genere:

```
<Entity name: 7ef52ea0>
```

La nuova entità-xrecord è stata ottenuta.

Non ci resta che aggiungere la nuova entità, a cui daremo il nome "XRECORD_1", all’entità-dizionario creata precedentemente. Basterà quindi digitare:

```
(dictadd dizionario "XRECORD_1" xrecord)
```

Se tutto va a buon fine la funzione ritorna il nome dell’entità del nuovo xrecord , che è lo stesso ottenuto da “entmakex” :

```
<Entity name: 7ef52ea0>
```

Tanto per esercizio proviamo ad aggiungerne un altro.....

Digitiamo...

```
(setq xrecord (list '(0."XRECORD") '(100."AcDbXrecord")))
(setq lista_dati (list '(1."Dato_1") '(2."Dato_2")))
(setq xrecord (append xrecord lista_dati))
(setq xrecord (entmakex xrecord))
(dictadd dizionario "XRECORD_2" xrecord)
```

Il dizionario, a questo punto, contiene 2 XRECORD.

Per concludere questa importante funzione è necessario fare un paio di precisazioni:

1. Abbiamo costruito la lista per l’entità XRECORD in due passaggi, ossia prima abbiamo scritto la lista tipica dell’entità e poi le abbiamo aggiunto la lista dei dati, ovviamente avremmo potuto fare tutto in un solo passaggio:

```
(setq xrecord (list '(0 . "XRECORD") '(100 . "AcDbXrecord") (1
. "Dato_1") '(2. "Dato_2")))
```

ma è preferibile tenere separati i dati dall'entità soprattutto quando la lista dati è molto lunga in modo da migliorare la leggibilità del sorgente.

2. Nella costruzione della lista per l'entità XRECORD abbiamo utilizzato, per immagazzinare i dati, coppie puntate costituite da un numero e un testo (1 . "Dato_1) e (2 . "Dato_2"), fermandoci a sole due coppie perché era sufficiente per fare un esempio. Ovviamente si possono aggiungere altri codici dxf , ma quali e quanti possiamo utilizzare? Da quanto si evince dalla guida Visual Lisp di Autocad i codici dxf utilizzabili vanno da 1 a 369 con l'esclusione di **5** e **105**, quindi coppie del tipo (5 . "dato...") e (105 . "dato...") non vanno bene. Ad ogni modo ce ne sono a sufficienza per tutte le esigenze.

Concludiamo dicendo che "Nuovo_dizionario_1" può essere attaccato alla root seguendo lo schema fin qui proposto ed evidenziato nella figura di poco fa, ma può anche essere attaccato ad un dizionario esistente.

In questo caso si dovrà prima ricavare il nome dell'entità di quest'ultimo con la funzione "dictsearch" di cui non abbiamo ancora parlato e passare questa entità alla funzione "dictadd".

Ciò porterà ad un'ulteriore ramificazione della struttura dei dizionari e a qualche difficoltà nella loro gestione a seconda della complessità della struttura.

Per motivi di chiarezza in questo tutorial prenderemo in considerazione solo dizionari attaccati alla root.

La funzione dictsearch

La parte di creazione di un dizionario e dell'aggiunta di un Xrecord è già stata affrontata, ci occupiamo adesso di come accedere ai dati contenuti in un dizionario. Per prima cosa è necessario sapere se un certo dizionario è contenuto nell'entità root.

Questo compito è affidato alla funzione "dictsearch" .

Supponiamo di cercare il dizionario chiamato "Dizionario_1" nella root.

La sintassi della funzione dictsearch è:

```
(dictsearch entità_dizionario_root nome_dizionario
[parametro_opzionale])
```

dove:

- entità_dizionario_root sarebbe l'entità restituita da (namedobjdict);
- nome_dizionario è un testo con il nome del dizionario da cercare;

- [parametro opzionale]...vedi dopo.

Per sapere se "Dizionario_1" è contenuto nella root digitiamo:

```
(setq dizionario (dictsearch (namedobjdict) "Dizionario_1"))
```

Se il dizionario esiste la funzione ritorna nella variabile dizionario qualcosa del genere:

```
((-1.<Entity name: 7ef68e48>) (0."DICTIONARY") (5."81")
(102."{ACAD_REACTORS")(330.<Entity name: 7ef68c60>)(102."}")
(330.<Entity name: 7ef68c60>)(100."AcDbDictionary")(280.0)
(281.1) (3."XRECORD_1")(350.<Entity name: 7ef68e50>)
(3."XRECORD_2") (350.<Entity name: 7ef68e68>))
```

ossia una lista dove si nota il nome dell'entità e, se ci sono, gli eventuali Xrecords.

Se siamo interessato a ricavare solo il nome dell'entità associata al dizionario che stiamo cercando possiamo digitare:

```
(setq dizionario (cdr (car dizionario )))
```

e otterremo nella variabile dizionario

```
<Entity name: 7ef68e48>
```

Se il dizionario non esiste la funzione ritorna "nil".

Concludiamo la spiegazione di questa funzione con un chiarimento sul parametro opzionale. Questo parametro opzionale, se è presente, magari è impostato su "T" (true), serve a spostare il puntatore della funzione "dictcnext" (di cui non abbiamo ancora parlato) sull'entità successiva a quella ritornata da "dictsearch".

In pratica, se il dizionario contenesse uno o più Xrecord, la funzione "dictsearch" col parametro impostato a "T" farebbe ritornare alla funzione "dictcnext" non più la lista dell'entità dizionario bensì la lista della prima entità Xrecord in esso contenuta.

Comunque l'uso di questo parametro è sostanzialmente inutile, per cui possiamo tranquillamente ometterne la presenza.

La funzione dictnext

L'uso della funzione "dictnext" è strettamente legata alla funzione "dictsearch".

Infatti, così come la funzione "dictsearch" serve a cercare un dizionario, la funzione "dictnext" serve a scorrerne il contenuto, fornendo via via il nome delle entità in esso contenute fino a raggiugere la fine.

La sintassi della funzione "dictnext" è:

```
(dictnext entità_dizionario [parametro_opzionale])
```

dove:

- entità_dizionario è il nome dell'entità dizionario da scorrere.

Diciamo subito che il parametro opzionale impostato a "T" riporta il puntatore della funzione all'inizio del dizionario; sarebbe come riavvolgere il nastro di una cassetta.

Se la funzione non trova elementi nel dizionario ritorna "nil".

Immaginiamo,ad esempio, di avere il nostro solito dizionario "Dizionario_1" che contiene due Xrecord. Per prima cosa cerchiamo il dizionario e digitiamo:

```
(setq dizionario (dictsearch (namedobjdict) "Dizionario_1"))
```

E otterremo come visto prima la lista:

```
((-1.<Entity name: 7ef68e48>)(0."DICTIONARY")(5."81")
(102."{ACAD_REACTORS")(330.<Entity name: 7ef68c60>)(102."}")
(330.<Entity name: 7ef68c60>)(100."AcDbDictionary")(280.0)
(281.1)(3."XRECORD_1")(350.<Entity name: 7ef68e50>)
(3."XRECORD_2")(350.<Entity name: 7ef68e68>))
```

Adesso, siccome dobbiamo passare alla funzione "dictnext" il nome dell'entità, ricaviamolo digitando:

```
(setq dizionario (cdr (car dizionario )))
```

e otterremo per la variabile dizionario:

```
<Entity name: 7ef68e48>
```

A questo punto basterà digitare:

```
(setq xrecord (dictnext dizionario))
```

e otterremo per la variabile "xrecord" la lista

```
((-1.<Entity name: 7ef68e50>)(0."XRECORD")(5."82")
(102."{ACAD_REACTORS")(330.<Entity name: 7ef68e48>)(102."}")
(330.<Entity name: 7ef68e48>)(100."AcDbXrecord")(280.1)
(1."Dato_1")(2."Dato_2"))
```

dove è possibile distinguere il tipo di entità (XRECORD) e i dati in essa contenuti, che sono i valori che abbiamo immesso prima quando abbiamo costruito il dizionario.

Digitiamo ancora:

```
(setq xrecord (dictnext dizionario))
```

e otterremo per la variabile xrecord la lista

```
((-1.<Entity name: 7ef68e68>)(0."XRECORD")(5."82")
(102."{ACAD_REACTORS")(330.<Entity name: 7ef68e48>)(102."}")
```

```
(330.<Entity name: 7ef68e48>)(100."AcDbXrecord")(280.1)
(1."Dato_3")(2."Dato_4"))
```

ossia la definizione dell'altro Xrecord. Digitiamo ancora:

```
(setq xrecord (dictnext dizionario))
```

A questo punto siamo arrivati alla fine del dizionario e se digitassiamo ancora:

```
(setq xrecord (dictnext dizionario))
```

otterremo "nil".

Proviamo per esercizio a riportare il puntatore di dictnext all'inizio del dizionario perché vogliamo vedere cosa contengono gli Xrecord e perchè vogliamo ricavare i dati in essi contenuti.

Digitiamo dunque:

```
(setq xrecord (dictnext dizionario T))
```

riotterremo per la variabile "xrecord" la lista

```
((-1 . <Entity name: 7ef68e50>) (0 . "XRECORD") (5 . "82")
(102 . "{ACAD_REACTORS") (330 . <Entity name: 7ef68e48>)
(102 . "}") (330 . <Entity name: 7ef68e48>) (100 .
"AcDbXrecord") (280 . 1) (1 . "Dato_1") (2 . "Dato_2"))
```

e siccome siamo interessato ad utilizzare i dati contenuti in questo Xrecord potremmo ad esempio digitare:

```
(setq dato1 (cdr (assoc 1 xrecord)))
```

Ottenendo così per la variabile dato1 il valore "Dato_1" che è quello contenuto nella coppia puntata (1 . "Dato_1")

Abbiamo visto quindi come la funzione "dictnext" scorre un dizionario e come possiamo accedere ai dati contenuti negli Xrecord.

La funzione dictremove

Dopo aver visto come si creano e cercano dizionari e xrecord vediamo come rimuoverli dal disegno.

Per rimuovere dal disegno un dizionario o un xrecord si fa uso della funzione dictremove la cui sintassi è:

```
(dictremove entità_dizionario simbolo)
```

dove:

- **entità_dizionario** è il nome dell'entità dove c'è il dizionario o l'Xrecord da eliminare;
- **simbolo** è il testo con cui abbiamo nominato il dizionario o l'Xrecord.

Faremo vedere l'uso di "dictremove" sia per rimuovere un dizionario che per rimuovere un Xrecord.

Prendiamo come esempio il solito dizionario che abbiamo creato precedentemente e inserito nella root e a cui abbiamo dato il nome "Nuovo_dizionario_1".

Questo dizionario contiene due Xrecord "XRECORD_1" e "XRECORD_2".

Caso 1: La rimozione di un dizionario dalla root

Se esaminiamo l'entità root prima della rimozione avremmo la situazione già vista prima:

```
(entget (namedobjdict))
((-1.<Entity name: 7ef51c60>)(0."DICTIONARY")(330.<Entity
name: 0>)(5."C") (100."AcDbDictionary")(280.0)(281.1)
(3."ACAD_COLOR")(350.<Entity name: 7ef51dd8>)
......
......
(3."AcDbVariableDictionary")(350.<Entity name: 7ef51d70>))
(3."Nuovo_dizionario_1")(350.<Entity name: 7ef68e48>))
```

basterà quindi scrivere:

```
(dictremove (namedobjdict) "Nuovo_dizionario_1")
```

e se tutto va a buon fine la funzione ritorna il nome dell'entità del dizionario rimosso:

```
<Entity name: 7ef68e48>
```

Proviamo a dare uno sguardo alla root adesso:

```
(entget (namedobjdict))
((-1.<Entity name: 7ef51c60>)(0."DICTIONARY")(330.<Entity
name: 0>)(5."C")(100."AcDbDictionary")(280.0)(281.1)
(3."ACAD_COLOR")(350.<Entity name: 7ef51dd8>)
......
......
(3."AcDbVariableDictionary")(350.<Entity name: 7ef51d70>))
```

Come si vede il dizionario "Nuovo_dizionario_1" è sparito.

Caso 2: La rimozione di un Xrecord da un dizionario

In questo caso dobbiamo, per prima cosa, ricavare il nome dell'entità da cui vogliamo rimuovere l'Xrecord, e poi utilizzare "dictremove". Nel nostro caso abbiamo inserito due Xrecord nel dizionario ("XRECORD_1" e "XRECORD_2") e proviamo ad eliminare "XRECORD_1".

Basterà semplicemente ricavare la lista di definizione del dizionario dalla root digitando:

```
(setq dizionario (dictsearch (namedobjdict) "Dizionario_1"))
((-1.<Entity name: 7ef68e48>)(0."DICTIONARY")(5."81")
(102."{ACAD_REACTORS")(330.<Entity name: 7ef68c60>)(102."}")
(330.<Entity name: 7ef68c60>)(100."AcDbDictionary")(280.0)
(281.1) (3."XRECORD_1")(350.<Entity name: 7ef68e50>)
(3."XRECORD_2")(350.<Entity name: 7ef68e68>))
```

e poi passare all'entità

```
(setq dizionario (cdr (car dizionario ))
```

oppure in un unico colpo...

```
(setq dizionario (cdr (car (dictsearch (namedobjdict)
"Dizionario_1"))))
```

Otterremo..

```
<Entity name: 7ef68e48>
```

Il nome dell'entità è stato ottenuto, l'Xrecord da eliminare si chiama "XRECORD_1", basterà scrivere allora:

```
(dictremove dizionario "XRECORD_1")
```

e se tutto va a buon fine la funzione ritorna il nome dell'entità dell'Xrecord rimosso

```
<Entity name: 7ef68e50>
```

Proviamo a dare uno sguardo alla lista del dizionario:

```
(entget dizionario)
((-1.<Entity name: 7ef53e48>)(0."DICTIONARY")(5."81")
(102."{ACAD_REACTORS")(330.<Entity name: 7ef53c60>)(102."}")
(330.<Entity name: 7ef53c60>)(100."AcDbDictionary")(280.0)
(281.1)(3."XRECORD_2")(350.<Entity name: 7ef53e68>))
```

Come si vede l'Xrecord "XRECORD_1" è sparito.

Concludiamo l'argomento "dictremove" dicendo che la funzione ritorna nil se qualcosa non va a buon fine e, cosa molto importante, ricordando che la funzione **non elimina** le varie entità non grafiche dal disegno, ma solo dalla struttura del dizionario.

Per cui se chiudessimo il disegno, anche salvando, alla sua riapertura ritroveremmo tutte le nostre entità non grafiche al loro posto.

Se vogliamo eliminare definitivamente le entità dobbiamo usare la funzione “entdel”, per cui le azioni che abbiamo fatto prima:

```
;per eliminare il dizionario
(dictremove (namedobjdict) "Nuovo_dizionario_1")
;per eliminare l’Xrecord
(dictremove dizionario "XRECORD_1" )
```

andrebbero scritte così:

```
(entdel (dictremove (namedobjdict) "Nuovo_dizionario_1"))
(entdel (dictremove dizionario "XRECORD_1" ))
```

La funzione dictrename

Questa funzione è veramente molto intuitiva, server per rinominare un dizionario o un Xrecord.

La sua sintassi è:

```
(dictrename entità vecchio_nome nuovo_nome)
```

dove:

- **entità** è il nome dell’entità che vogliamo rinominare;
- **vecchio_nome** è il testo con cui abbiamo nominato il dizionario o l’Xrecord;
- **nuovo_nome** è il testo con cui vogliamo rinominare.

Dal momento che l’uso di “dictrename” è praticamente uguale a quello di di “dictremove” ci limiteremo solo a fare degli esempi senza entrare nel dettaglio.

Per rinominare il nostro dizionario,contenuto nella root, da "Dizionario_1" a "Dizionario_2" basterà dunque scrivere:

```
(dictrename (namedobjdict) "Dizionario_1" "Dizionario_2")
```

e se tutto va a buon fine la funzione ritorna il nuovo nome del dizionario.

```
"Dizionario_2"
```

Se vogliamo rinominare l’Xrecord del dizionario da "XRECORD_1" a "NUOVO_XRECORD_1" si dovrà, ovviamente, prima ricavare il nome dell’entità dizionario e poi rinominare l’Xrecord.

Scriveremo dunque:

```
(setq dizionario (cdr (car (dictsearch (namedobjdict)
"Dizionario_1"))))
```

otterremo..

```
<Entity name: 7ef68e48>
```

e infine scrivere:

```
(dictrename dizionario "XRECORD_1" "NUOVO_XRECORD_1")
```

e se tutto va a buon fine la funzione ritorna il nuovo nome dell'Xrecord.

```
"NUOVO_XRECORD_1"
```

Concludiamo dicendo che la funzione ritorna nil se qualcosa va storto, ad es.:

- se il vecchio nome non esiste o è sbagliato;
- se il nuovo nome è già utilizzato;
- se il dizionario non esiste;
- ... e via discorrendo...

E per finire... un piccolo esercizio...

Concludiamo questo tutorial con un piccolo programma che riassume le cose che abbiamo imparato in queste pagine.

Il software è composto da due moduli; il listato lisp (dizionario.lsp) e l'interfaccia grafica (dizionario.dcl).

Create una cartella in C: (C:\DIZIONARIO) e posizionatevi dentro i due files.

Volendo potete creare la cartella dove volete, ma ciò comporta che si dovrà cambiare il percorso di ricerca della finestra di dialogo che è stato preimpostato in "C:\DIZIONARIO".

Con questo piccolo esempio si vuole creare un dizionario che immagazzini i dati di una serie di circuiti di conduzione dell'aria.

Immaginiamo una conduttura in lamiera rettangolare dove, all'interno, scorra dell'aria e immaginiamo di dover impostare, per ogni conduttura:

- una sigla di circuito;
- la tipologia del circuito;

- lo spessore dell'isolamento attorno alla lamiera;
- una breve descrizione che identifichi meglio il circuito.

Immaginiamo, ad esempio, che il circuito sia così impostato:

- sigla **M1**;
- tipologia **Mandata**;
- spessore **20 mm**;
- descrizione **Mandata aria camere**.

Bene, carichiamo il file dizionario.lsp e digitiamo sulla riga di comando CIRCUITI … apparirà questa finestra ...

La finestra è molto semplice da gestire.

Il focus iniziale è impostato per posizionarsi sulla edit box in alto a destra.

La Tipologia, per default, è impostata sulla Mandata. Notiamo subito che la popup list in alto a sinistra è vuota, infatti ancora non abbiamo inserito alcun circuito.

Il pulsante Elimina circuito è disattivato perché non ci sono circuiti da eliminare. Inseriamo i dati che caratterizzano il circuito e la dialog diventerà...

Abbiamo inserito i dati che caratterizzano il circuito:

- Sigla circuito;
- Descrizione;
- Spessore isolamento.

La tipologia era stata impostata di default.

Come si nota il pulsante Elimina circuito è ancora disattivato.

Per aggiungere il circuito M1 alla lista dei circuiti basta ora cliccare su Aggiungi circuito.

... la dialog diventerà...

Il focus iniziale è tornato sulla edit box in attesa di nuovi inserimenti.

Come si nota la popup list mostra il nuovo circuito e si vedono anche i dati di descrizione e spessore isolamento.

I pulsanti di azione sono adesso tutti attivi perché è possibile utilizzarli.

Se si tentasse di inserire un nuovo circuito (ad es. **M2**), all'uscita dalla edit box, i dati di descrizione e spessore isolamento precedenti sparirebbero in attesa dei nuovi dati per il nuovo circuito.

Si può provare ad aggiungere altri circuiti con altre caratteristiche per vedere come si comporta la finestra e magari provare a cancellarne qualcuno,

Tutto questo è lasciato al lettore come esercizio.

Si consiglia di leggere attentamente il listato sorgente per impadronirsi bene di questa efficace tecnica di immagazzinamento di informazioni e provare a fare delle modifiche per rendere il programma ancora più efficace.

Il tutorial finisce qui con la speranza di aver consegnato al lettore un interessante mezzo per ampliare i propri programmi.

Capitolo 18
I blocchi

Questo capitolo descrive i blocchi e le funzioni che li riguardano. I blocchi sono una parte fondamentale ed estremamente utile dei disegni CAD, semplici, potenti, funzionali ed estremamente efficienti, sono un potente strumento nelle mani di chi disegna.

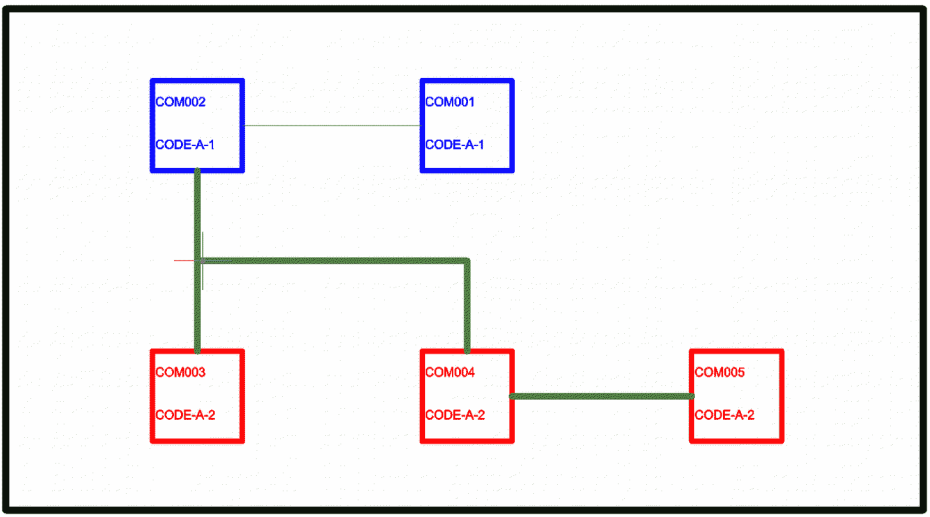
Vedremo ora come sono fatti i blocchi, cosa sono e come sono fatti gli attributi, vedremo come modificarli e come interagire con essi.

I blocchi cosa?

Cosa sono i blocchi?

Come spiega qualsiasi guida di **AutoCAD**, **progeCAD** o simili, un blocco è un **gruppo statico di oggetti con un nome** che include una o più entità grafiche ed è dotato di un punto base da utilizzare per il suo inserimento infine, facoltativamente, possono essere inclusi degli **oggetti chiamati attributi** per la memorizzazione di dati generici in formato testo.

La forma più semplice di blocco è rappresentata dall'insieme di entità grafiche, al quale viene assegnato un nome. Vediamo un piccolo esempio:



Questo disegno è contenuto nel file **schema.dwg**, incluso negli esempi.

Quello qui rappresentato è un generico schema che include una serie di elementi connessi tra loro, ed è composto quasi esclusivamente da blocchi, ognuno dei quali dotato di un significato ben preciso, in questo caso, ogni blocco rappresenta un

componente dotato di una sigla identificativa univoca e un codice, che potrebbe benissimo indicare in componente realmente acquistabile.

Quindi? Un **blocco non è altro che una "scatola grafica" con nome** che viene inserita una o più volte all'interno di un disegno. Ma non solo, un blocco è un elemento capace di **ospitare informazioni** decise da chi disegna o da chi progetta, al fine poi di riutilizzarle per, ad esempio, rapporti su ciò che è presente nel progetto.

La potenza dei blocchi risiede nella loro duttilità e nella capacità di chi disegna di utilizzarli al meglio. Potenzialmente, i modi di utilizzare i blocchi sono infiniti. Tutto dipende dalla fantasia dei progettisti e dai disegnatori.

Inoltre, il loro utilizzo ha anche un **impatto significativo e positivo sulle dimensioni e prestazione dei disegni** infatti, grazie alla loro natura, inducono le dimensioni su disco dei disegni e ne migliora sensibilmente le prestazioni, aumentando la velocità di disegno, soprattutto in presenza di disegni particolarmente grossi e complessi.

E ora?

E ora iniziamo a vedere, dal punto di vista della programmazione, come sono costruiti e come si utilizzano questi **fantastici blocchi**.

Prima di tutto occorre comprendere come sono memorizzati all'interno del nostro disegno (dwg o dxf).

Come abbiamo detto, i blocchi sono "scatole" che contengono una serie di entità, ed ogni scatola ha un nome univoco. Ci deve essere quindi un posto nel quale viene memorizzato il loro nome e il loro contenuto. Questo posto si chiama **Tabella di definizione dei blocchi**.

In questo spazio del nostro file di disegno (dwg o dxf) sono memorizzati tutti i dati relativi al come sono fatti i blocchi che abbiamo nel nostro progetto.

Un blocco, normalmente, viene definito mediante i comandi standard del CAD, in particolare attraverso l'utilizzo di **BLOCCO/BLOCK**.

A questo punto sorge spontanea una domanda, se in questa tabella sono descritti i blocchi, come si fa ad inserirne uno nel disegno vero e proprio in modo che venga mostrato? Come faccio ad usare questi blocchi?

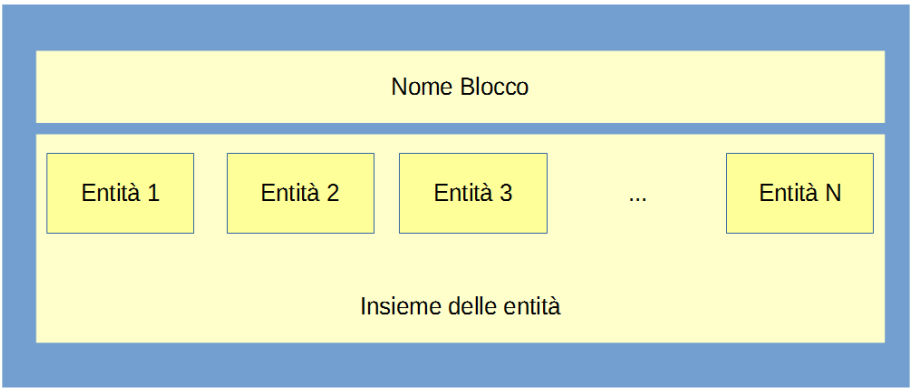
Come sappiamo il disegno è composto da tante entità grafiche, linee, cerchi, archi, testi, ecc... Bene, esiste un tipo di entità particolare che rappresenta un blocchi inserito nel nostro disegno, questo tipo di oggetto si chiama **INSERT** e viene comunemente chiamato **Riferimento di blocco**.

Nel nostro progetto, insieme a tutte le altre entità grafiche, sarà presente un **riferimento di blocco** per ogni elemento presente.

Non ci resta che tuffarci nel codice per vedere, più da vicino, **i blocchi**.

La definizione di un blocco

Prima di tutto vediamo, graficamente, **come è strutturata la definizione di un blocco**:



Come detto, i blocchi sono definiti all'interno di un'apposita zone del nostro file dwg/dxf, la famosa **Tabella di definizione dei blocchi**. Per accedere a questa zona possiamo utilizzare le funzioni Lisp **tblobjname** e **tblsearch**.

Questa definizione contiene tutti i dati per rappresentare il singolo blocco una volta che verrà inserito nel disegno. Ovviamente, nel nostro progetto verrà inserito solo il riferimento, un segnaposto che dice "qui c'è il blocco X".

Prendiamo il nostro disegno di esempio, con lo schema al suo interno, ed osserviamo la definizione del blocco di nome "COMPONENT". Per compiere questa operazione utilizziamo la funzione **tblsearch** per recuperarla:

```
(tblsearch "BLOCK" "COMPONENT")

((0 . "BLOCK")
 (330 . <Entity name: 3d8cd950>)
 (100 . "AcDbEntity")
 (67 . 0)
 (8 . "0")
 (100 . "AcDbBlockBegin")
 (2 . "component")
 (70 . 2)
 (10 0 0 0)
```

```
  (-2 . <Entity name: 3d8cdd90>)
)
```

Questo è l'inizio delle **definizione del nostro blocco**. Come abbiamo visto poco sopra, una definizione è composta dalle singole parti del blocco stesso, quindi ci sarà un elemento per il nome, come in questo caso, uno per ogni entità grafica e uno per ogni attributo (di cui parleremo più avanti).

Nella tabella che contiene tutte le definizioni, è possibile scorrere tutte le entità di un blocco utilizzando la normale funzione **entnext**. Una volta ottenuto il nome di entità del primo elemento (codice -2) è poi possibile proseguire ed ottenere tutte le entità successive, facenti parte del blocco considerato.

Vediamo ora una funzione che **stampa a video** tutti gli elementi presenti in una **definizione di blocco**:

```
;stampa la definzione di un blocco
(defun printBlockDef ( nameBl / ent defent nextOk)
(setq ent (tblsearch "BLOCK" nameBl))
(if ent
        (progn
            (princ "\nDefinizione blocco \"")
            (princ nameBl)
            (princ "\"")
            (setq nextOk t)
            (setq ent (cdr (assoc -2 ent)))
            (setq defent (entget ent))
            (print defent)
            (while nextOk
                (setq ent (entnext ent))
                (if ent
                    (setq defent (entget ent))
                    (setq nextOk nil)
                )
                (print defent)
            )
        )
    )
(prin1)
)
```

Per utilizzare **printBlockDef** è sufficiente specificare il nome di blocco da analizzare:

```
(printBlockDef "COMPONENT")
```

Ecco un esempio di più che potremo vedere:

```
Definizione blocco "COMPONENT"
((-1 . <Entity name: 3d8cdd90>) (0 . "LWPOLYLINE") (5 . "1D7")
(330 . <Entity name: 3d8cd950>) (100 . "AcDbEntity") (67 . 0)
(8 . "0") (100 . "AcDbPolyline") (90 . 4) (70 . 1) (43 . 0)
(38 . 0) (39 . 0) (10 0 0 0) (40 . 0) (41 . 0) (42 . 0) (91 .
0) (10 5 0 0) (40 . 0) (41 . 0) (42 . 0) (91 . 0) (10 5 5 0)
(40 . 0) (41 . 0) (42 . 0) (91 . 0) (10 0 5 0) (40 . 0) (41 .
0) (42 . 0) (91 . 0) (210 0 0 1))
((-1 . <Entity name: 3d8cded0>) (0 . "ATTDEF") (5 . "1D8")
(330 . <Entity name: 3d8cd950>) (100 . "AcDbEntity") (67 . 0)
(8 . "0") (100 . "AcDbText") (10 0.1571 3.5898 0) (40 . 0.5)
(1 . "?") (50 . 0) (41 . 1) (51 . 0) (7 . "Standard") (71 . 0)
(72 . 0) (11 0 0 0) (210 0 0 1) (100 .
"AcDbAttributeDefinition") (3 . "Sigla") (2 . "DESIGNATION")
(70 . 0) (73 . 0) (74 . 0) (280 . 1))
((-1 . <Entity name: 3d8cd450>) (0 . "ATTDEF") (5 . "1D9")
(330 . <Entity name: 3d8cd950>) (100 . "AcDbEntity") (67 . 0)
(8 . "0") (100 . "AcDbText") (10 0.1571 1.2232 0) (40 . 0.5)
(1 . "???") (50 . 0) (41 . 1) (51 . 0) (7 . "Standard") (71 .
0) (72 . 0) (11 0 0 0) (210 0 0 1) (100 .
"AcDbAttributeDefinition") (3 . "Codice") (2 . "CODE") (70 .
0) (73 . 0) (74 . 0) (280 . 1))
```

Il blocco "COMPONENT" contiene 4 entità in tutto, una polilinea e 2 attributi, ognuno dotato di una moltitudine di proprietà.

Come sempre è possibile capire il significato di ogni proprietà consultando una qualsiasi guida che descrive il formato DXF.

Ovviamente, utilizzando le funzioni di modifica **entmod** è possibile cambiare ogni aspetto delle entità presenti.

Un blocco inserito

Finora abbiamo parlato di come è memorizzata la struttura dei blocchi. Vediamo adesso come vengono **usati e come vengono inseriti** all'interno del nostro disegno.

Un blocco definito, ma non inserito, apparentemente "non esiste" nel nostro disegno.

Prendiamo il nostro esempio ed esaminiamo uno dei blocchi di nome "COMPONENT" presenti. Prima di tutto prendiamo il nome di entità selezionandola dal disegno:

```
(setq ent (car(entsel)))
```

A questo punto vediamo come è fatta internamente:

```
(entget ent)
```

Ecco cosa potremo osservare:

```
((-1 . <Entity name: 7658bb0>) (0 . "INSERT") (5 . "1E6") (330
. <Entity name: 7ae8030>) (100 . "AcDbEntity") (67 . 0) (410 .
"Model") (8 . "Level1") (100 . "AcDbBlockReference") (66 . 1)
(2 . "component") (10 0 15 0) (41 . 1) (42 . 1) (43 . 1) (50 .
0) (70 . 1) (71 . 1) (44 . 0) (45 . 0) (210 0 0 1))
```

Qui vediamo come il tipo di entità non sia, come in precedenza, "BLOCK" ma sia "**INSERT**", cosa che indica la presenza di un **inserimento di blocco** all'interno del disegno, quindi **visibile e accessibile dal disegnatore**. Inoltre abbiamo le solite proprietà come il layer (codice 8), il nome di blocco (codice 2), e molte altre, compreso il **codice 66**. Quest'ultimo elemento, quando impostato a 1 indica una cosa importantissima, dice che **il blocco è dotato di attributi**, cioè di testi (visibili o invisibili) modificabili, in grado di associare ad ogni singola istanza di blocco inserito delle informazioni, ma di questo parleremo più avanti.

Vediamo subito come **inserire un blocco** nel disegno, ecco un esempio:

```
(command "_insert" "component" "0,0" "" "" "")
```

Questa semplice istruzione inserisce il blocco di nome "COMPONENT" nel punto 0,0 e lo fa utilizzando il normale comando "_INSERT" del CAD.

Quando si effettua l'inserimento di un blocco con questo sistema, per prevenire errori, occorre sapere **dove il CAD va a cercare la definizione del blocco** indicato.

Abbiamo detto che la **definizione di un blocco è memorizzata all'interno del proprio progetto** ma, se si tenta di inserirne uno la cui definizione non è presente?

In questo caso il CAD cercherà il blocco **su disco**, e lo cercherà in quelli che, spesso, vengono chiamati **percorsi di ricerca dei file di supporto** e possono essere modificati mediante il comando "OPTIONS"/"OPZIONI".

Naturalmente, se il blocco non venisse trovato nemmeno su disco, il CAD mostrerebbe un avvertimento appropriato.

E' bene ricordare che **ogni blocco può essere salvato su disco** e questo avviene mediante l'uso del comando "WBLOCK"/"MBLOCCO". Nella pratica ciò che salveremo non sarà altro che un piccolo file DWG/DXF contenente esclusivamente il blocco indicato.

Ciò significa che **qualsiasi file di disegno può**, potenzialmente, **essere utilizzato come fosse un blocco** e inserito in un altro disegno.

Ora, una delle caratteristiche più rilevanti e utilizzate dei blocchi sono gli attributi, informazioni di tipo testuale, visibili o invisibile associabili ad ogni blocco inserito nel disegno. Questi elementi sono fondamentali quando si vogliono inserire dati utili all'interno degli elementi del disegno, ad esempio pensiamo ai codici di eventuali materiali da acquistare in fase di realizzazione fisica del progetto.

Nella sezione che segue vedremo proprio gli **attributi**.

Cosa sono gli attributi?

Come già detto, **un attributo è un forma di testo**. Può essere un testo **visibile**, può essere **invisibile**, può essere grande o piccolo, a una linea o multilinea ma, fondamentalmente, sono sempre accessibili e modificabili dall'utente, inoltre ogni inserimento possiede i suoi attributi, completamente indipendenti da quelli degli altri elementi inseriti, anche a parità di nome di blocco.

Se è vero che all'atto dell'inserimento di un blocco questo verrà dotato degli attributi presenti nella sua definizione, è altrettanto vero che poi, dopo essere stato inserito, sarè possibile modificare gli attributi presenti, uno per uno, indipendentemente da ciò che era quando il blocco è stato definito.

Comportamento opposto a quello delle **entità grafiche** interne, che invece **saranno uguali per tutte le istanze** dello stesso blocco.

Il primo, e più evidente effetto di questo comportamente, è il fatto che ogni attributo presente in un blocco inserito, possieda un proprio valore, un proprio testo, potenzialmente diverso da tutti gli altri.

Prendiamo il disegno di esempio mostrato in precedenza, in quel caso ogni blocco è dotato di due attributi, il primo contenente la sigla, cioè il numero univoco che distingue gli elementi, il secondo contenente il codice del materiale che, quando il progetto verrà realizzato, sarà fisicamente acquistato.

Ovviamente, questo è solo uno dei possibili utilizzi.

Blocchi inseriti con attributi

Per comprendere meglio cosa sono e come vengono gestiti gli attributi vediamone uno nel dettaglio.

Supponiamo di avere un **blocco dotato di attributi**, **inserito** nel nostro disegno. Prendiamo uno di quelli presenti nel nostro disegno di esempio.

Osserviamo nuovamente un blocco "component":

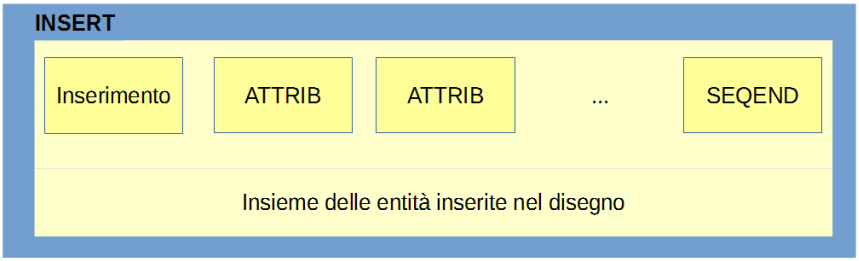
```
((-1 . <Entity name: 7658bb0>) (0 . "INSERT") (5 . "1E6") (330
. <Entity name: 7ae8030>) (100 . "AcDbEntity") (67 . 0) (410 .
"Model") (8 . "Level1") (100 . "AcDbBlockReference") (66 . 1)
(2 . "component") (10 0 15 0) (41 . 1) (42 . 1) (43 . 1) (50 .
0) (70 . 1) (71 . 1) (44 . 0) (45 . 0) (210 0 0 1))
```

Prima di tutto poniamo la nostra attenzione sull'elemento **66**.

La presenza del codice **66 impostato a 1** è molto importante perchè indica che **le entità successiva a "INSERT" saranno gli attributi** con le informazioni relative al singolo blocco inserito. La sequenza di attributi sarà interrotta quando troveremo un'entità di tipo **SEQEND**.

Si, gli attributi sono entità separate dal blocco inserito, ed è proprio questa loro caratteristica che li rende indipendenti e modificabili.

Per capire meglio, vediamo la struttura delle entità relative al nostro inserimento:



Bene... per iniziare ad interagire con le nuove entità **ATTRIB** costruiamo una funzione che, dato il nome, ci mostra tutte le parti del blocco inserito:

```
;stampa tutta ciò che riguarda un blocco isnerito
;partendo dall'entità INSERT
(defun blk_printInsertEntity ( blEnt / ent sixsix att blk)
(setq blk (entget blEnt))
```

```
;stampa elemento "INSERT"
(print blk)

(setq ent blEnt)
(setq sixsix (cdr(assoc 66 blk)))
  (if (= sixsix 1)
    (progn ;solo se ci sono attributi
      (setq ent (entnext ent));primo attributo
      (setq att (entget ent))

      ;primo attributo
      (print att)
      (while (/= (cdr(assoc 0 att)) "SEQEND")
        (setq ent (entnext ent));primo attributo
        (setq att (entget ent))

        ;definizione attributo
        (print att)
      )
    );endp
  );endif
(prin1)
);enddef
```

Per utilizzarla:

```
(blk_printInsertEntity (car(entsel)))
```

Una volta eseguita l'istruzione, basterà selezionare un blocco presente nel nostro disegno, per ottenere qualcosa di simile a questo:

```
((-1 . <Entity name: 3d8cdf10>) (0 . "INSERT") (5 . "1DA")
(330 . <Entity name: 3d8cb590>) (100 . "AcDbEntity") (67 . 0)
(410 . "Model") (8 . "Level2") (100 . "AcDbBlockReference")
(66 . 1) (2 . "component") (10 0 0 0) (41 . 1) (42 . 1) (43 .
1) (50 . 0) (70 . 1) (71 . 1) (44 . 0) (45 . 0) (210 0 0 1))
((-1 . <Entity name: 3d8cdad0>) (0 . "ATTRIB") (5 . "1DB")
(330 . <Entity name: 3d8cdf10>) (100 . "AcDbEntity") (67 . 0)
(410 . "Model") (8 . "0") (100 . "AcDbText") (10 0.1571 3.5898
0) (40 . 0.5) (1 . "COM003") (50 . 0) (41 . 1) (51 . 0) (7 .
"Standard") (71 . 0) (72 . 0) (11 0 0 0) (210 0 0 1) (100 .
```

```
"AcDbAttribute") (2 . "DESIGNATION") (70 . 0) (73 . 0) (280 .
1))
((-1 . <Entity name: 3d8ccfd0>) (0 . "ATTRIB") (5 . "1DC")
(330 . <Entity name: 3d8cdf10>) (100 . "AcDbEntity") (67 . 0)
(410 . "Model") (8 . "0") (100 . "AcDbText") (10 0.1571 1.2232
0) (40 . 0.5) (1 . "CODE-A-2") (50 . 0) (41 . 1) (51 . 0) (7 .
"Standard") (71 . 0) (72 . 0) (11 0 0 0) (210 0 0 1) (100 .
"AcDbAttribute") (2 . "CODE") (70 . 0) (73 . 0) (280 . 1))
((-1 . <Entity name: 3d8cd390>) (0 . "SEQEND") (5 . "1DD")
(330 . <Entity name: 3d8cdf10>) (100 . "AcDbEntity") (67 . 0)
(410 . "Model") (8 . "Level2") (-2 . <Entity name: 3d8cdf10>))
```

Come si può vedere, nel disegno, non è presente il semplice **INSERT** che indica la presenza di un blocco ma, per ogni inserimento, possono essere presenti eventuali attributi.

Le entità di tipo **ATTRIB** sono proprio gli attributi associati al nostro blocco.

Come già detto, la cosa fondamentale da comprendere è che, nonostante gli attributi siano inseriti con il blocco, sono effettivamente entità indipendenti e, come tali, possono essere modificate singolarmente. Quindi? Quindi a parità di nome di blocco, potremo avere un inserimento con attributi di un certo colore, un inserimento con lo stesso nome di blocco con attributi di colore diverso, potremo avere anche un numero diverso di attributi.

Naturalmente è possibile modificare e creare le entità **ATTRIB** con le solite funzioni **entmod** ed **entmake**.

Listare gli attributi

Dato un blocco presente in un disegno, a livello di programmazione, una delle funzioni che sono certamente indispensabili è quella che consente di **ottenere l'elenco degli attributi presenti** o meglio, l'elenco delle entità attributo, con il quale poi poter compiere le operazioni più comuni, come lettura, controllo e scrittura.

La funzione che costruiremo ora ritorna la lista delle entità attributo associati ad un **INSERT**.

Ecco la funzione che, strutturamente, è molto simile a quella che stampa la definizione del blocco (**blk_printInsertEntity**):

```
;ritorna la lista con i nomi delle entita'
;relative a tutti gli attributi trovati
;partendo dall'entità blocco inserito
(defun blk_getAttList ( blEnt / ent sixsix att attList blk)
(setq blk (entget blEnt))
```

```
(setq ent blEnt)
(setq sixsix (cdr(assoc 66 blk)))
  (if (= sixsix 1)
    (progn ;ci sono attributi
      (setq ent (entnext ent));primo attributo
      (setq att (entget ent))
      (while (/= (cdr(assoc 0 att)) "SEQEND")
        (setq attList (cons (cdr (assoc -1 att)) attList))
        (setq ent (entnext ent));primo attributo
        (setq att (entget ent))
      )
      (setq attList (reverse attList))
    );endp
  );endif
attList
);enddef
```

Per utilizzarla:

```
(blk_getAttList (car(entsel)))
```

Una volta eseguita l'istruzione basterà selezionare un blocco presente nel nostro disegno per ottenere qualcosa di simile a questo:

```
(<Entity name: 37df86a0> <Entity name: 37df8b60>)
```

In questo caso possiamo vedere come, il blocco selezionato abbia solo due attributi, identificati ognuno dal relativo nome di entità.

Ottenere un attributo

Prima di poter leggere o scrivere il valore di un attributo, quindi prima di poter salvare o recuperare informazioni, è necessario sapere **qual'è l'entità che lo rappresenta**.

Ogni attributo possiede un nome, alle volte chiamato tag, che lo identifica ed è specificato dal codice **2**.

Quindi, vediamo ora una funzione che ritorna l'entità dell'attributo con nome specificato, nel blocco indicato:

```
;ritorna l'entità attributo che corrisponde al nome
specificato o indice
```

```
(defun blk_getAttEName ( blEnt attName / ent entDef att i name
continue result selectByIndex)
  (if blEnt
    (progn
      (setq attLst (blk_getAttList blEnt))
      (if attLst
        (progn

          (if (= (type attName) 'STR)
            (progn
              (setq selectByIndex nil)
            )
            (progn
              (if (= (type attName) 'INT)
                (setq selectByIndex t)
                (progn
                  (print "blk_getAttEName : Require integer
value!")
                  (exit)
                )
              );endif
            )
          )

          (if selectByIndex
            (progn
            ;selezione attributo per indice (da 0 a n-1)
              (setq result (nth attName attLst))
            )
            (progn ;selezione per nome attributo
              (setq i 0)
              (setq continue t)
              (while (and
                       (setq att (nth i attLst))
                       continue
                     )
                (setq name (strcase (cdr (assoc 2 (entget
att)))))
```

```
                (if (= (strcase attName) name)
                  (progn
                    (setq result (nth i attLst))
                    (setq continue nil)
                  )
                );endif
                (setq i (1+ i))
              );endw
            );endp
          );endif
        );endp
      );endif
    );endp
  );endif
result
);enddef
```

Il suo utilizzo è semplice. Ad esempio, possiamo leggere l'attributo di nome "CODE" da un blocco selezionato:

```
(setq ent (car(entsel)))
(blk_getAttEName ent "CODE")
```

Ecco cosa potrebbe ritornare la funzione:

```
<Entity name: 8baa0b0>
```

Con questo dato sarà poi possibile agire sull'attributo trovato, sia in lettura che in scrittura.

Nel caso in cui l'attributo non venga trovato la funzione ritornerà **nil**.

Leggere gli attributi

Con il nome di entità relativo ad un determinato attributo possiamo ora vedere come **leggerne il valore**. Il valore di un attributo non è altro che il testo visualizzato o meno, il dato memorizzato.

Il valore degli attributi è associato al codice **DXF 1**.

Vediamo la funzione di lettura:

```
;ritorna il valore associato ad un attributo
;in base al suo nome o tramite indice
```

```
(defun blk_getAttValue ( blEnt attName / ent result)
  (if blEnt
    (progn
      (setq ent (blk_getAttEName blEnt attName))
      (if ent
        (setq result (cdr (assoc 1 (entget ent))))
      );endif
    );endp
  );endif
result
);enddef
```

Potremo usarla cosi:

```
(setq ent (car(entsel)))
(blk_getAttValue ent "CODE")
```

Ecco cosa potrebbe ritornare la funzione:

```
"CODE-A-1"
```

In pratica verrà ritornata la stringa con il testo presente oppure un **nil** nel caso l'attributo specificato non esista.

Scrivere gli attributi

A questo punto manca solo una cosa, la funzione per la **scrittura del valore di un attributo**. Questa operazione, una volta ottenuta l'entità sulla quale agire, è molto semplice e viene effettuata tramite la normale funzione **entmod**.

Ecco la **funzione che permette di scrivere il valore di un attributo data l'entità blocco e il suo nome**:

```
;ritorna il valore associato ad un attributo
;in base al suo nome o usando un indice
(defun blk_setAttValue ( blEnt attName attValue /
                         ent def oldValue newValue result)
  (if blEnt
    (progn
      (setq ent (blk_getAttEName blEnt attName))
      (if ent
        (progn
          (setq def (entget ent))
```

```
            (setq oldValue (assoc 1 def))
            (setq newValue (cons 1 attValue))
            (setq def (subst newValue oldValue def))
            (entmod def)
            (entupd ent)
            (setq result t)
          );endp
        );endif
      );endp
    );endif
  result
  );enddef
```

Potremo usarla cosi:

```
(setq ent (car(entsel)))
(blk_setAttValue ent "CODE" "NuovoValore")
```

La funzione ritornerà **t** o **nil** a seconda che l'operazione vada a buon fine o meno.

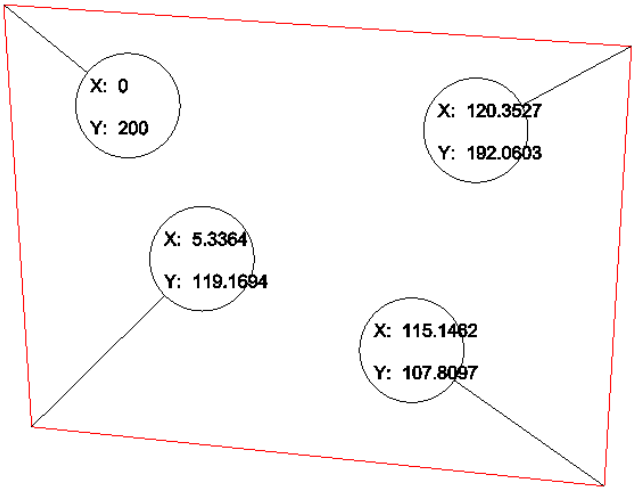
Dalla teoria alla pratica

Trasformiamo ora la teoria imparata fino a questo punto in pratica e vediamo un **esempio concreto di utilizzo dei blocchi**. Scriveremo una semplicissima procedura che risolve, attraverso l'utilizzo dei blocchi un piccolo problema.

Il problema da risolvere è facile da spiegare. **Come posso inserire, nel disegno, le coordinate X/Y di un punto specificato?**

Quello che farà il nuovo comando che stiamo per costruire è chiedere all'utente di specificare un punto sul disegno, chiedere dove posizionare le coordinate, inserendo poi un blocco nel quale verranno compilati due attributi, uno per X e uno per Y.

Per capire meglio quale sarà il risultato finale, osserviamo questo esempio:



Qui sono stati inseriti **quattro blocchi contenenti le coordinate dei punti indicati** dal disegnatore.

Per realizzare la nuova procedura, utilizzeremo quanto gia visto in questo capitolo. Il comando si chiamerà **blkInsPoint**:

```
;nuovo comando per inserimento blocco con punto
(defun c:blkInsPoint ( / pt pt2 ptBlk
                         atq eBlk entBlk
                          x y xStr yStr
                          ang)

  (setq atq (getvar "ATTREQ"))
  (setvar "ATTREQ" 0)
  (setq pt (getpoint "\nSpecifica il punto:"))
  (if pt
    (progn
      (setq ptBlk (getpoint pt
          "\nSpecifica il punto dove inserire X e Y:"))
      (if ptBlk
        (progn
          ;inserimento blocco
          (command "_insert"
```

```
                    "blk-point-signal" ptBlk "" "" "")
            (setq eBlk (entlast))
            ;estrazione coordinate
            (setq entBlk (entget eBlk))
            (setq x (car pt))
            (setq y (cadr pt))
            ;settaggio attributi
            (blk_setAttValue eBlk "X" (rtos x))
            (blk_setAttValue eBlk "Y" (rtos y))

            ;calcolo linea di collegamento
            (setq ang (angle pt ptBlk))
            (setq pt2 (polar ptBlk ang -10))
            (command "_line" pt pt2 "")
          )
        )
      )
    )
    (setvar "ATTREQ" atq)
    (prin1)
)
```

La prima cosa da dire è che, una volta caricata, avremo a disposizione il nuovo comando **blkInsPoint**.

Esaminando il codice è necessario soffermarsi su diversi particolari.

Prima di tutto parliamo della variabile **ATTREQ**. All'inizio della procedura questa viene impostata a 0, mentre viene ripristinata al termine.

ATTREQ permette di stabilire se, all'atto dell'inserimento di un blocco, gli attributi devono o meno mantenere i valori di default.

Per impostazione predefinita **ATTREQ** è impostata a 1, questo fa si che i valori degli attributi vengano chiesti all'utente durante l'operazione di inserimento. Impostandola a 0, queste **richieste vengono soppresse**.

Quindi, la procedura elimina la richiesta dei valori degli attributi prima di effettuare l'inserimento del blocco "blk-point-signal".

Proseguendo vedremo che, una volta chiesti all'utente i due punti necessari, viene effettuato l'inserimento del blocco che conterrà le coordinate del primo punto scelto.

Nel blocco appena inserito verranno settati poi i due attributi con le appropriate coordinate X e Y infine, verrà inserita una semplice linea che collega il primo punto con

il nostro blocco.

Per disegnare la linea sarà sufficiente calcolare la posizione dei suoi due punto tenendo presente che il cerchio all'interno del blocco inserito ha raggio 10.

La procedura completa può essere trovata nel file **blksLib.lsp** e può essere testata sul file **blk-points.dwg**, antrabe presenti negli allegati di questo testo.

Ancora due parole

Si potrebbero dire moltissime altre cose sui blocchi e sul loro utilizzo. Si potrebbe parlare della **creazione dei blocchi**, delle **proprietà speciali** degli attributi, di **definizione e ridefinizione**, di **blocchi dinamici**, **testi**, **attributi multilinea**, **layer associati** alle parti del blocco, e chi più ne ha più ne metta.

Quello che abbiamo visto finora permetterà una gestione di base di blocchi e informazioni correlate, consentendo la realizzazione di procedure per la memorizzazione e l'utilizzo di dati generici associati ai blocchi inseriti.

Molte sono le possibilità a disposizione e **l'unico reale limite è la fantasia di disegnatori, progettisti e programmatori**.

Capitolo 19
Gestore di blocchi
“Binder Blocks”

Questo capitolo nasce con l'intento di mostrare le fasi e il codice di un software completo. Non solo verrà mostrato il codice sorgente del software, ma verranno anche spiegate le varie fasi di lavorazione che hanno portato alla sua realizzazione.

In questo modo sarà possibile rendersi conto, in maniera più chiara, delle problematiche da affrontare per la realizzazione di una procedura personalizzata. Si potranno comprendere meglio le problematiche che si incontrano durante la pianificazione, la realizzazione e l'utilizzo di un software scritto da zero.

La necessità

Ogni software nasce per risolvere un problema o migliorare delle procedure operative non ottimizzate.

In questo caso, la necessità non è altro che il bisogno di gestire un insieme di simboli / blocchi da inserire poi nel disegno. Ovviamente il tutto dovrà essere fatto in modo che l'operatore aumenti la sua produttività, inserendo rapidamente i blocchi, trovando quanto cerca in modo rapido e veloce, cercando inoltre di semplificare la creazione e modifica della libreria stessa.

A fronte di tutto ciò, l'operatore dovrà trovarsi di fronte un programma intuitivo, con una curva di apprendimento particolarmente favorevole.

L'idea

Partendo dalle necessità, andiamo ora a chiarire l'idea che sta alla base del nuovo programma.

Prima di tutto la gestione dei blocchi. Blocchi che, nella pratica, sono dei semplici file dwg memorizzati su disco.

Proprio pensando a tali file, bisogna riflettere sul fatto che, comunemente, gli utenti organizzano i loro file in alberi di cartelle organizzati logicamente. Essendo questa una pratica consolidata si è scelto di riutilizzarla anche all'interno del nuovo software, consentendo agli operatori di gestire una libreria organizzata in cartelle, sottocartelle e semplici blocchi (file) dwg.

Purtroppo l'uso di una struttura simile ha delle limitazioni, la prima delle quali è la mancanza di informazioni aggiuntive associate ai singoli blocchi, ad esempio delle descrizioni estese. Per superare questi inconvenienti il sistema gestirà dei dati aggiuntivi associabili ai singoli blocchi/file.

Altre importanti funzionalità sicuramente da gestire sono quelle di scalatura e rotazione dei blocchi durante l'inserimento nel CAD.

Infine, ma non ultima, la funzione di ricerca che dovrà consentire di trovare un blocco

cercandolo, oltre che per nome, anche per uno qualsiasi dei suoi dati aggiuntivi.

Naturalmente, non se ne è parlato, ma la funzione principale sarà quella che consente di selezionare un blocco per poi inserirlo all'interno del disegno.

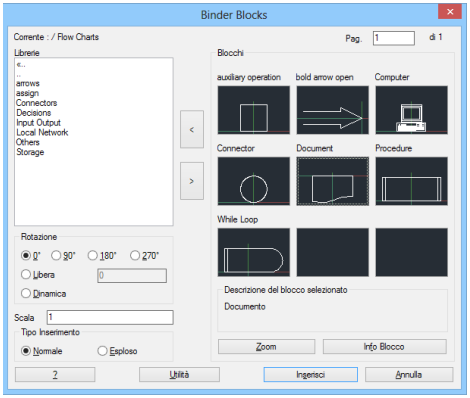
Tutto il programma prevederà solamente due comandi:

1. Il primo permetterà l'apertura della finestra principale dalla quale effettuare la scelta dell'elemento da inserire;
2. Il secondo permetterà l'accesso alle funzioni di ricerca, che consentiranno all'utente di trovare un elemento all'interno dell'archivio della libreria.

L'interfaccia grafica

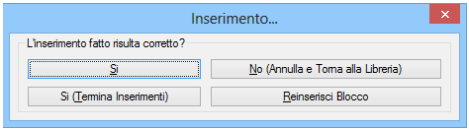
Tenendo sempre presente che, l'intuitività e la facilità di utilizzo devono essere al centro delle preoccupazioni, l'interfaccia grafica (GUI) è stata sviluppata pensando, prima di tutto ai blocchi. Purtroppo dovendo realizzare il tutto sfruttando Lisp e DCL ci si scontra con una serie, importante, di limitazioni. Questo ha condizionato il disegno dell'interfaccia.

Si è scelto un aspetto classico, nel quale i blocchi vengono presentati a gruppi di N, con la possibilità di scorrere le paginate di elementi con appositi tasti:



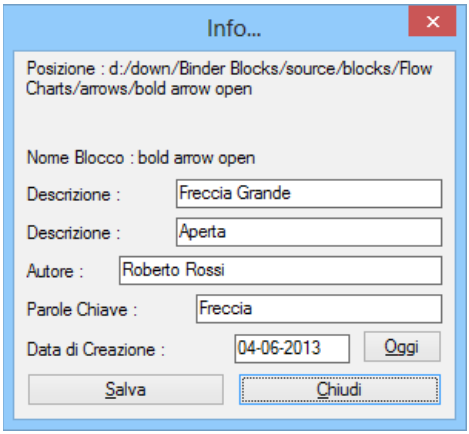
Come si può osservare questa, che è la maschera principale, è strutturata in tre zone distinte. La prima, a sinistra, consente di navigare tra le cartelle/categorie, la seconda a destra, contiene le anteprime dei blocchi presenti, mentre la parte bassa della maschera contiene tutte le funzioni disponibili, comprese le loro opzioni.

Legata alla maschera principale, ne troviamo una secondaria, visibile al termine di ogni inserimento che permette di scegliere come procedere:

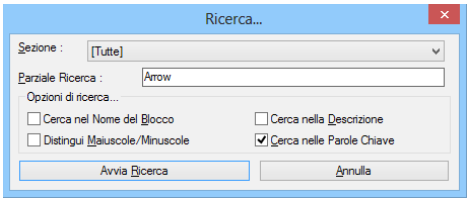


In questo caso, nulla di particolarmente complesso, una piccola finestra con poche opzioni, autoesplicative.

Abbiamo poi la maschera di gestione delle informazioni aggiuntive legate ai singoli blocchi :



La terza maschera di interfaccia è quella che consente la ricerca di un blocco:



Tutte le opzioni sono chiaramente visibili nella parte bassa e ruotano attorno alla casella principale nella quale inserire il testo da trovare. Per rendere più coerente l'interfaccia, i blocchi trovati verranno presentati all'utente tramite la stessa maschera utilizzata per i normali inserimenti. In tal modo, le operazioni di selezione finale del blocco e di inserimento rimarranno inalterate, sia che l'utente esegua una ricerca, sia che acceda direttamente alla finestra principale.

Storicamente, l'idea iniziale era quella di unificare le funzioni di ricerca, di visualizzazione e di inserimento, all'interno della stessa finestra. Questo approccio, purtroppo, si è dimostrato controproducente. L'integrazione delle funzioni di ricerca all'interno della maschera principale di inserimento, avrebbero richiesto lo sviluppo di un'interfaccia dotata di un numero molto maggiore di elementi cosa che, unita alle limitazioni delle DCL, avrebbe prodotto una finestra molto grande e sovraffollata.

Per questi motivi si è scelto di dividere le due funzioni in due maschere separate.

La sua struttura del codice sorgente

Oltre a come dovrà apparire all'utente, il programma deve essere pensato anche dal punto di vista della sua struttura interna, di come le parti di codice si relazioneranno tra loro.

Naturalmente, non sempre è possibile avere un'idea estremamente precisa di cosa serva e di come farlo.

"Binder Blocks" è un software libero e non è particolarmente complesso e, nella versione 0.9.9, è composto da:

- 2 file LSP;
- 1 file DCL;
- 1 file di configurazione;
- 1446 linee di cui 191 linee di commento.

La struttura del sorgente è particolarmente semplice. Il file principale, da caricare per poter utilizzare la libreria, è "binderbl.lsp". Al suo interno trovano posto tutte le funzioni principali, sia di gestione, sia di ricerca. Il secondo file, "bbsins.lsp", contiene le procedure per l'inserimento dinamico dei blocchi.

I comandi definiti e utilizzabile sulla linea di comando del cad, come gia detto, sono solamente due :

- BinderBlocks. Avvia la procedura principale di gestione ed inserimento blocchi;
- BinderBlocks_S. Avvia il modulo di ricerca.

Escludendo le procedure che definiscono i due comandi, tutte le funzioni utilizzate nel software hanno la particolarità di avere, come prefisso per il loro nome, i caratteri "bbs_". In questo modo è possibile distinguerle da qualsiasi altra procedura e, con più difficoltà, potranno interferire con altri programmi Lisp caricati.

Il file principale, “binderbl.lsp”, contiene un totale di 39 funzioni, mentre “bbsins.lsp” ne contiene 6. Da un punto di vista di codice sorgente, il software fa uso di alcune variabili globali, “tmpBlockCollectorOpz”, “tmpBlockCollectorSOpz” e “bbs_Var_getListAllFolderTemp”, che hano lo scopo di memorizzare le scelte fatte dall’utente tra due aperture della libreria. Voglio ricordare che, l’uso di variabili globali, è da limitare il più possibile in quanto, nel caso in cui due software facciano, per caso, uso della medesima variabile, i problemi che andranno a generarsi saranno di difficile identificazione e di altrettanto difficile risoluzione.

Come per tutti i software, è di fondamentale importanza commentare il codice sorgente. Questo garantisce che la rilettura, magari a distanza di tempo, possa essere più semplice, così come più facili saranno eventuali correzioni o modifiche. Come si può osservare, “Binder Blocks” è commentato in modo limitato ma, nonostante cio, è possibile identificare con relativa facilità l’utilità di ogni singola funzione.

Questo software è stato sviluppato nel corso di diversi mesi e, come tutti i programmi, è stato ripreso, modificato, corretto, potenziato. Questa evoluzione ha portato alcune piccole incoerenze all’interno del codice sorgente, come ad esempio alcune parti non più usate, oppure alcuni abbozzi di funzioni ideate per potenziamenti futuri, ma non terminate. Tutto questo è assolutamente normale ed è comune a tutti i programmi che si evolvono e maturano nel tempo.

Installazione ed utilizzo

Per vedere all’opera il software le operazioni necessarie sono veramente poche :

- Copiare i file di Binder Block in una cartella;
- Nell’ambiente CAD, inserire la cartella che contiene Binder Blocks nei percorsi di ricerca, solitamente presenti nella maschera delle opzioni richiamabile con il comando “_options”;
- Modificare il file “libfolder.cfg” inserendo, sulla prima linea, il percorso della cartella che contiene i nostri blocchi. A tale scopo è possibile indicare il percorso della cartella “Blocks” presente insieme ai file del programma, nella quale sono presenti una serie di blocchi organizzati da utilizzare come esempio di libreria;
- Sulla linea di comando digitare :

```
(load "binderbl.lsp")
```

- 5. A questo punto è possibile utilizzare i due nuovi comandi, “BinderBlocks” e “BinderBlocks_S”.

Licenza e Autore di Binder Blocks

Una delle parti più sottovalutate, soprattutto dagli sviluppatori alle prime armi, è l'assegnazione di una licenza di distribuzione al proprio lavoro, e l'inserimento delle note relative all'autore.

Anche se può apparire un'attività superflua, quando si realizza un software si deve sempre tener presente il fatto che, per loro natura, la copia, la distribuzione e il loro riutilizzo è un'operazione particolarmente semplice. E' quindi consigliabile inserire all'interno dei sorgenti, come commenti, i dettagli relativi ad una eventuale licenza di distribuzione e, ovviamente, all'autore.

Come si può osservare in "Binder Blocks" la cosa è stata fatta e, all'inizio di ogni file sorgente è possibile trovare quanto detto.

Capitolo 20
Funzioni di Visual Lisp

Questo nuovo capitolo descrive **Visual Lisp** e le sue funzioni. Nato come estensione del classico Lisp implementato da AutoCAD, **Visual Lisp** è ormai presente in tutti i maggiori CAD.

Ciò che verrà spiegato permetterà di compiere operazioni, spesso molto utili, ma precluse al normale Lisp di base implementato nei CAD.

Due parole di introduzione

Visual Lisp venne introdotto, per la prima volta, in AutoCAD 14 come estensione indipendente. Successivamente venne integrato definitivamente all'interno del CAD.

Il Visual Lisp che troviamo in AutoCAD è composto, prima di tutto, da **una potente IDE** per la scrittura e per il debug del codice. Questo strumento semplifica molto lo sviluppo delle applicazioni fornendo anche un sistema di compilazione e criptazione delle applicazioni. Non tutti i CAD forniscono uno strumento assimilabile all'IDE di AutoCAD ma, comunque, la maggior parte implementa le modifiche al linguaggio.

Dal punto di vista del linguaggio Lisp, **Visual Lisp aggiunge una nutrita serie di funzioni**, tutte contraddistinte dalle iniziali "**vl**". Queste espandono e potenziano molto il linguaggio di base.

In questo capitolo esamineremo proprio le funzioni proprie di Visual Lisp, implementate nella maggior parte dei CAD simili ad AutoCAD e a progeCAD.

File e Cartelle

Visual Lisp offre diverse funzioni che consentono al linguaggio di agire su file e cartelle del sistema, in modo semplice ed immediato.

Iniziamo da **vl-directory-files**. Questa, dato un percorso e un filtro, consente di ottenere la lista dei file al suo interno. Possiamo definirla cosi:

```
(vl-directory-files percorso filtro cosa)
```

Il primo parametro è il **percorso** di una qualsiasi cartella, il secondo parametro rappresenta un **filtro**, ad esempio con ***.lsp** vengono presi in considerazione solo i file con estensione **.lsp**, infine abbiamo un parametro numerico (cosa) che può valere -1, 0 o 1 e che, rispettivamente, permette di ottenere solo le **cartelle**, **file e cartelle** oppure solo **file**.

La funzione **vl-directory-files** ritorna sempre la lista di ciò che è stato trovato, oppure **nil**.

Vediamo un paio di esempi:

```
(vl-directory-files "c:/temp" "*.*" 1)
```

```
("." ".." "Cartella1" "Cartella2" "file1.dwg" "FileA.txt")

(vl-directory-files "c:/temp" "*.*" -1)
("." ".." "Cartella1" "Cartella2")

(vl-directory-files "c:/temp" "*.*" 0)
("file1.dwg" "FileA.txt")
```

Rimanendo in tema di cartelle, parliamo di una funzione dedicata esclusivamente a loro, **vl-mkdir**. Questa permette di creare una cartella sul nostro sistema. Il suo utilizzo è particolarmente intuitivo :

```
(vl-mkdir nomecartella)
```

Dato il nome della cartella da creare verrà ritornato **t** nel caso l'operazione sia riuscita e **nil** nel caso sia fallita.

Facciamo subito un esempio:

```
(vl-mkdir "c:/temp/cartella1")
```

Usando **vl-mkdir** bisogna sempre tener presente che è in grado di creare cartelle solo all'interno di percorsi che già esistono quindi, nel nostro esempio, "cartella1" verrà effettivamente creata solo se "temp" esiste già.

Abbandoniamo le cartelle e concentriamoci sulla **gestione dei file**.

Parliamo della funzione **vl-file-copy** che consente di copiare un singolo file. In realtà va anche oltre, consentendo di accodare un file ad un altro ottenendone la somma. Questa è la sua definizione:

```
(vl-file-copy filesorgente filedestinazione aggiungi)
```

I primi due parametri sono intuitivi, il file **da copiare** e il file **di destinazione**, il terzo parametro invece può essere **t** o **nil**, nel primo caso il file sorgente viene accodato a destinazione (se esiste), mentre nel secondo caso il file sorgente sovrascrive la destinazione se questa è già presente.

La funzione ritorna **t** se la copia è riuscita, **nil** nel caso di fallimento. Vediamo qualche esempio:

```
(vl-file-copy "c:/temp/test/origin/myfile.txt"
"c:/temp/test/destination/myfile.txt" nil)
(vl-file-copy "c:/temp/test/origin/myfile.txt"
"c:/temp/test/destination/myfile.txt" t)
```

Nel primo caso il file viene copiato, nel secondo viene accodato ad un eventuale file **myfile.txt** già esistente. L'ultimo parametro è opzionale e, se non specificato, è pari a **nil**.

Una volta copiati i file, passiamo alla cancellazione. Per l'operazione di eliminazione di un singolo file possiamo utilizzare **vl-file-delete**, che è definita cosi:

```
(vl-file-delete nomefile)
```

Semplice, basta chiamare **vl-file-delete** specificando il file da eliminare completo di percorso, verrà restituito **nil** se l'operazione fallisce e **t** se il file viene eliminato.

L'eliminazione effettuata tramite **vl-file-delete** solitamente non sposta il file indicato nel cestino del sistema.

Per quanto riguarda invece la ridenominazione dei file, esiste l'apposita funzione **vl-file-rename**, così definita:

```
(vl-file-rename vecchionome nuovonome)
```

Anche qui, la semplicità è estrema, è sufficiente specificare **il vecchio nome** del file completo di percorso e il **nuovo nome** sempre con percorso completo. Se la funzione va a buon fine verrà restituito **t**, altrimenti **nil**.

Esempio:

```
(vl-file-rename "c:/temp/myfilename.txt"
"c:/temp/mynewfilename.txt")
```

Tutto qui.

Passiamo ora a vedere quelle chi io chiamo **funzioni informative**, quelle che non agiscono direttamente ma che forniscono informazioni specifiche.

Prima di tutto **vl-file-directory-p**, così definita:

```
(vl-file-directory-p percorso)
```

Inserendo come parametro un percorso, **vl-file-directory-p** ritornerà **t** nel caso quello specificato sia una cartella e **nil** nel caso non lo sia.

Ad esempio:

```
(vl-file-directory-p "c:/temp")
```

Se **temp** esiste ed è una cartella, la funzione ritornerà **t**. Semplice, immediata, facile.

Altra funzione di informazione è **vl-file-size** che accetta il percorso completo di un file e restituisce la sua dimensione in byte.

Questa è la sua definizione:

```
(vl-file-size percorso)
```

Ad esempio:

```
(setq dimensione (vl-file-size
"c:/lamiacartella/ilmiofile.dwg"))
(setq dimensione (vl-file-size "simplex.shx"))
```

Come si può osservare, la funzione accetta anche nomi di file **senza il percorso**. In questo caso il file indicato verrà cercato all'interno delle cartelle specificate nei **percorsi di ricerca** del CAD.

Nel caso in cui il file indicato non venga trovato, il risultato della funzione sarà un classico **nil**.

Compagna di **vl-file-size** è la **vl-file-systime** che consente di ottenere la data dell'ultima modifica del file specificato.

La definizione:

```
(vl-file-systime percorso)
```

La **vl-file-systime** è particolare in quanto restituisce i dati come lista e non come valore singolo. In questa lista, nell'ordine, troviamo **anno, mese, giorno della settimana, giorno del mese, ora, minuti e secondi**.

Vediamo in esempio dell'utilizzo di **vl-file-systime** e di ciò che potrebbe ritornare:

```
(vl-file-systime "C:/lamiacartella/ilmiofile.txt")
(2015 3 6 7 8 41 12 0)
```

Come si può osservare il nostro file è stato modificato nel 2015, nel mese 3, giorno 6, ecc...

Passiamo ora a vedere alcune funzioni dedicate al **trattamento dei percorsi dei file**.

Una delle cose, alle volte fastidiose, è la **separazione dei percorsi** dei file nelle loro parti. Come fare, date un percorso completo a separare il nome del file, da quello della cartella? Oppure, come ottenere l'estensione del file?

Visual Lisp possiede alcune funzioni, proprio pensate per risolvere questo problema.

Partiamo da **vl-filename-base**. In questo caso, la funzione estrae il solo nome di un file da un percorso.

La sua definizione:

```
(vl-filename-base percorso)
```

Esempio:

```
(vl-filename-base "c:/lamiacartella/ilmiofile.txt")
"ilmiofile"
```

Dal percorso completo, la funzione elimina le cartelle e l'estensione del file, ritornando solamente il nome del file specificato.

La prossima funzione che analizzeremo è **vl-filename-directory**. In questo caso, quello restituito è il percorso dell'eventuale file specificato, senza nome di file ed estensione.

La definizione:

```
(vl-filename-directory percorso)
```

Un esempio:

```
(vl-filename-directory "c:/lamiacartella/ilmiofile.txt")
"c:\lamiacartella"
```

Ovviamente, la cartella è comprensiva di tutto il percorso, compreso il nome del disco.

Sempre parlando di cartelle troviamo **vl-filename-mktemp** che permette di ottenere il percorso di un nuovo file nella cartella temporanea del sistema operativo.

La sua definizione è semplicissima, nessun parametro:

```
(vl-filename-mktemp)
```

Esempio:

```
(vl-filename-mktemp)
"C:\\Users\\ilmioutente\\AppData\\Local\\Temp\\$VL~~3A55"
```

Quello che la funzione ritorna, come detto, è sostanzialmente un nome di file comprensivo di percorso. Un file che sicuramente non esiste, e che potremo usare per scrivere eventuali dati temporanei.

Le liste

In questa sezione vedremo le funzioni Visual Lisp dedicate espressamente al **trattamento delle liste**.

Cominciamo da una funziona ideata per creare nuove liste, la **vl-list***. Vediamone la definizione:

```
(vl-list* elemento elemento ...)
```

Dati una serie di elementi, specificati come parametri, **vl-list*** ritorna una lista composta dagli elementi stessi. Apparentemente il funzionamento è lo stesso della più classica funzione **list**, in realtà ci sono molte differenze.

Ci sono regole precise che **vl-list*** utilizza per creare la lista. Vediamole:

- Se è specificato **un singolo e unico elemento** di base (es.: stringa, numero interno, numero reale, ecc...), viene ritornato il solo elemento.

- Se vengono specificati **più elementi di base**, viene ritornata una lista puntata con gli elementi di base, nella quale l'ultimo è separato dagli altri.
- Viene ritornata una lista puntata se **l'ultimo elemento è di base** e le condizioni precedenti sono vere.
- Infine viene ritornata una normale lista se **nessuna delle condizioni precedenti** è vera.

Vediamo il tutto nella pratica:

```
(vl-list* "a")
ritorna -> "a"

(vl-list* "a" 2)
ritorna -> ("a" . 2)

(vl-list* "a" "b" "c")
ritorna -> ("a" "b" . "c")

(vl-list* "a" "b" '("c" 1))
ritorna -> ("a" "b" "c" 1)
```

In realtà, questa funzione non è particolarmente utilizzata, in quanto non è altro che una variante, più complessa, della classica funzione **list**.

Esaminiamo ora una funzione di conversione, la **vl-list->string**, che converte una lista di interi che rappresentano i caratteri della tabella ASCII, in una normale stringa.

Se poi servisse l'operazione di conversione inversa, esiste la **vl-string->list**, che converte una stringa nella corrispondente lista di codice numerici.

Vediamo la definizione delle due:

```
(vl-string->list stringa)
(vl-list->string listacaratteri)
```

Ed ecco qualche esempio, iniziamo dalla conversione da stringa a lista:

```
(vl-string->list "abc")
ritorna -> (97 98 99)
```

Poi, la conversione **da lista a stringa**:

```
(vl-list->string '(97 98 99))
ritorna -> "abc"
```

Sempre in tema di liste, vediamo ora la semplice, ed intuitiva, funzione **vl-list-length**.

La definizione:

```
(vl-list-length lista)
```

E subito un esempio:

```
(vl-list-length '("a" "b" 1))
ritorna -> 3
```

Ovviamente, la funzione non fa altro che ritornare il numero di elementi presenti nella lista passata come parametro.

Rimanendo sul semplice parliamo di **vl-position** che, data una lista, cerca un elemento e ne ritorna l'indice (da 0 a N).

La definizione:

```
(vl-position elemento lista)
```

Un esempio:

```
(vl-position 2 (list 1 2 "a" "b"))
ritorna -> 1
```

Il valore 1 rappresenta **l'indice dell'elemento** trovato, partendo da 0. Nella pratica, il primo elemento della lista ha indice 0.

Ora affrontiamo un argomento spesso ostico, **la rimozione di elementi da una lista**. Solitamente questo problema si risolve separando in due la lista, eliminando l'elemento superfluo e ricomponendola.

Visual Lisp dispone di apposite funzioni per rimuovere uno o più elementi da una lista in modo efficiente.

Quella principale è **vl-remove** che elimina uno o più elementi da una lista, restituendo la lista risultante. Eccone la definizione:

```
(vl-remove elemento lista)
```

Vediamola in azione:

```
(setq miaLista (list 1 "a" "b" "c" 1 "essere" pi "c"))
(setq listaModificata (vl-remove 1 miaLista))
(print miaLista)
(print listaModificata)
```

Il risultato di queste poche righe è la stampa di due liste, quella originale e quella modificata dopo aver rimosso l'elemento 1 da miaLista, cosi:

```
;originale
```

```
(1 "a" "b" "c" 1 "essere" 3.1416 "c")
;modificata
("a" "b" "c" "essere" 3.1416 "c")
```

Ciò che è interessante notare è che **vl-remove** elimina tutte le occorrenze trovate dell'elemento specificato.

Compagne di **vl-remove** sono la **vl-remove-if** e la **vl-remove-if-not**.

Vediamo le loro definizioni:

```
(vl-remove-if funzioneTest lista)
(vl-remove-if-not funzioneTest lista)
```

Entrambe basano il loro funzionamento su **funzioneTest** che valuta ogni elemento di **lista** per identificare quelli da eliminare o da mantenere.

Nel primo caso, **vl-remove-if**, ritorna una lista formata da tutti gli elementi che hanno fallito il test di **funzioneTest**, mentre nel secondo caso, **vl-remove-if-not** ritorna una lista formata da tutti gli elementi che hanno superato il test fatto da **funzioneTest**.

Facciamo un esempio:

```
(defun verifica (val / )
    (if (= (type val) 'STR)
        (setq val 0)
    )
    (if (> val 10)
        t
        nil
    )
)

(setq miaLista (list 2 10 34 23 8 11 20 "test1" "test2"))

(setq res1 (vl-remove-if verifica miaLista))
(setq res2 (vl-remove-if-not verifica miaLista))
```

In questo breve codice viene definita una funzione di verifica che ritorna t se il valore esaminato è maggiore di 10, mentre nil nel caso questo sia uguale o minore. Per le stringhe, queste vengono conteggiate come se valessero 0.

La funzione verifica viene poi utilizzata da **vl-remove-if** e **vl-remove-if-not** per valutare due liste e restituire il risultato in **res1** e **res2**.

Le due funzioni passano, uno alla volta, tutti gli elementi della lista in esame alla

funzione verifica e, in base al valore ritornato da quest'ultima, stabiliscono quali elementi della lista mantenere e quali scartare.

Una volta eseguito, il codice di esempio, imposterà le due variabili **res1** e **res2** a questi valori:

```
res1 => (2 10 8 "test1" "test2")
res2 => (34 23 11 20)
```

vl-remove-if rimuove tutti gli elementi per i quali la funzione verifica ritorna t mentre, **vl-remove-if-not** fa esattamente il contrario, rimuovendo tutti gli elementi per i quali la funzione verifica ritorna nil.

Naturalmente, il risultato è una lista dei valori rimanenti.

A questo punto credo sia arrivato il momento di vedere due funzioni molto, molto utili. Sto parlando delle funzioni per l'ordinamento di una lista.

Ordinare una lista è una di quelle operazioni fondamentali in molti casi, pensiamo ad esempio se volessimo mostrare una lista di layer o di nomi di blocco, non sarebbe affatto professionale mostrarle non ordinate.

Le funzioni di ordinamento sono due, **vl-sort** e **vl-sort-i**.

Vediamo le definizioni:

```
(vl-sort lista funzioneConfronto)
(vl-sort-i lista funzioneConfronto)
```

Il funzionamento di entrambe è identico, serve una lista da passare alla funzione e una funzione di confronto che riceverà due elementi da confrontare e restituirà t nel caso il primo precede il secondo nell'ordinamento voluto.

L'unica differenza tra le due è rappresentata dalla lista ritornata dalle funzioni. Nel primo caso **vl-sort** ritorna la lista ordinata, nel secondo caso, **vl-sort-i** ritorna la lista degli indici degli elementi ordinati (da 0 a N-1).

Vediamo un esempio per capire meglio:

```
(setq laLista (list 1 20 3 11 89 8 5 2))
(setq nuovaLista (vl-sort laLista '<))
```

Come funzione di confronto è stata usata **<**, ciò significa che il risultato dell'ordinamento, inserito nella variabile "nuovaLista", sarà:

```
(1 2 3 5 8 11 20 89)
```

Naturalmente, al posto di **<**, potremo usare una nostra funzione. In questo caso potremo scrivere:

```
(defun mioConfronto ( a b )
```

```
    (if (< a b)
        t
        nil
    )
)

(setq laLista (list 1 20 3 11 89 8 5 2))

 ;non funziona con funzioni personalizzate
(setq nuovaLista (vl-sort laLista 'mioConfronto))
```

Questo codice produce lo stesso risultato visto in precedenze cioè, nuovaLista verrà impostata a:

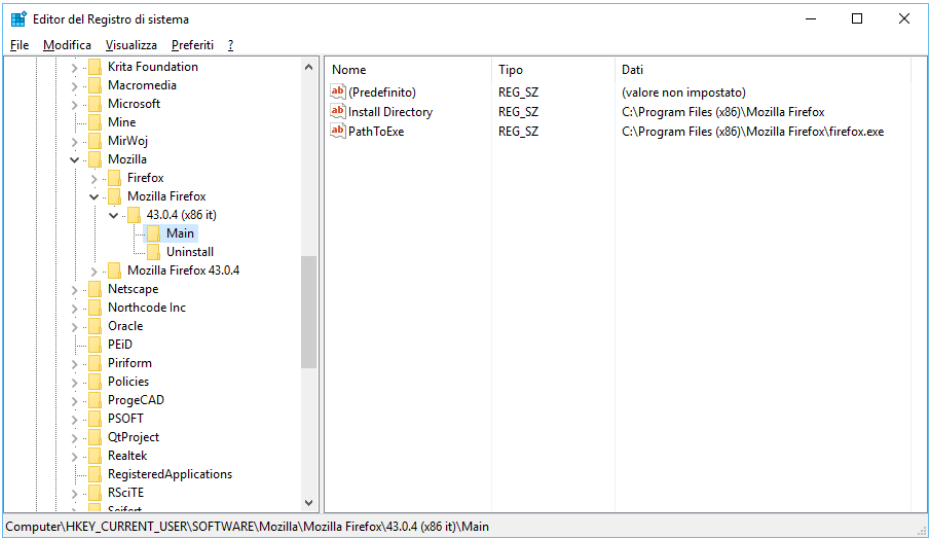
```
(1 2 3 5 8 11 20 89)
```

Come la lista viene ordinata dipende dalla funzione di confronto, che in questo caso si chiama **mioConfronto**. Questa routine non fa altro che accettare due valori, a e b, confrontarli e determinare quale dei due è maggiore dell'altro, ritornando poi **t** o **nil**. Nell'esempio, internamente a **mioConfronto**, viene fatto una comparazione tramite **<** ma si può usare qualsiasi altro modo per stabilire chi sia maggiore o minore. Se volessimo ordinare una lista di nomi di entità? In questo caso la funzione di confronto potrebbe fare considerazioni ben più complesse di quelle fatte per confrontare due numeri interi.

Il Registro di sistema

Cos'è il **registro di sistema**? In Windows il Registro di sistema è un archivio, gerarchicamente organizzato, che serve per memorizzare informazioni di vario genere. Ogni Windows ha un unico e solo Registro di sistema. Solitamente viene utilizzato dai programmi, e dallo stesso sistema operativo, per memorizzare le proprie impostazioni.

Per vederlo, possiamo accedervi tramite l'apposita utilità:



Quelle che vediamo qui sono alcuni valori salvati dal browser web Firefox. La cosa fondamentale da comprendere è come sia composto il registro. Ci sono solamente tre elementi, le chiavi, i valori e i contenuti dei valori(i dati). Le chiavi sono quelle presenti sulla sinistra con l'aspetto di semplici "cartelle", i valori sono quelli presenti nella parte destra della maschera, ognuno con un proprio nome di valore e un proprio contenuto.

Visual Lisp dispone di tutto quello che serve per elencare, verificare, leggere, scrivere e cancellare le chiavi di registro.

Le funzioni che analizzeremo sono le seguenti:

- **vl-registry-descendents**. Permette di avere l'elenco delle chiavi e dei valori presenti nel registro.
- **vl-registry-read**. Legge un valore.

- **vl-registry-write**. Scrive un valore.
- **vl-registry-delete**. Consente di eliminare un singolo valore o una singola chiave.

Vediamone le definizioni ed analizziamole una alla volta. In realtà, vista la loro semplicità, non servirebbe nulla di più della definizione ma, per completezza, aggiungeremo anche un paio di esempi.

Iniziamo con **vl-registry-descendents**:

```
(vl-registry-descendents chiave valore)
```

Questa funzione ritorna una lista contenente le sotto-chiavi (cartelle) presenti nella chiave (cartella) specificata. Ad esempio osserviamo questa istruzione :

```
(vl-registry-descendents
"HKEY_CURRENT_USER\\SOFTWARE\\Mozilla\\")
ritorna -> ("Mozilla Firefox 43.0.4" "Mozilla Firefox"
"Firefox")
```

Come si può vedere dalla figura più sopra, la chiave "Mozilla" contiene tre sotto-chiavi, ritornate dalla funzione come lista.

Un altro esempio:

```
(vl-registry-descendents
"HKEY_CURRENT_USER\\SOFTWARE\\Mozilla\\Mozilla Firefox\\43.0.4
(x86 it)\\")
ritorna -> ("Uninstall" "Main")
```

Stesso discorso, la chiave "43.0.4 (x86 it)" contiene due elementi "figli".

Nel caso in cui la chiave specificata non sia presente, la funzione ritornerà un semplice **nil**. Infine, specificando una stringa vuota come secondo parametro verrà ritornato l'elenco dei valori (non delle sotto-chiavi) presenti nella chiave specificata, ad esempio:

```
(vl-registry-descendents
"HKEY_CURRENT_USER\\SOFTWARE\\Mozilla\\Mozilla Firefox\\43.0.4
(x86 it)\\Main" "")
ritorna -> ("PathToExe" "Install Directory")
```

Nella pratica **vl-registry-descendents** permette di navigare il registro del sistema e di verifica la presenza delle chiavi analizzate.

Passiamo ora a **vl-registry-read** che legge un valore presente in una qualsiasi chiave accessibile del registro. Ecco la sua definizione:

```
(vl-registry-read chiave nomevalore)
```

L'uso è estremamente intuitivo. Leggiamo ora il percorso completo dell'eseguibile di Firefox:

```
(vl-registry-read
"HKEY_CURRENT_USER\\SOFTWARE\\Mozilla\\Mozilla Firefox\\43.0.4
(x86 it)\\Main" "PathToExe")
ritorna -> "C:\\Program Files (x86)\\Mozilla
Firefox\\firefox.exe"
```

Nulla di più semplice.

Visual Lisp dispone anche della funzione complementare alla lettura, la scrittura. Per scrivere un valore nel registro potremo utilizzare **vl-registry-write**. Eccone la definizione:

```
(vl-registry-write reg-key nomevalore valore)
```

Esattamente come per la lettura specificheremo la chiave di registro da scrivere e il nome del valore sul cui agire, in più specificheremo il dato da inserire.

Esempio:

```
(vl-registry-write
"HKEY_CURRENT_USER\\SOFTWARE\\Mozilla\\Mozilla Firefox\\43.0.4
(x86 it)\\Main" "MiaChiave" "MioValore")
ritorna -> "MioValore"
```

Molto interessante il fatto che, la funzione si occuperà di creare l'intera gerarchia della chiave nel caso questa non esista.

Al termine dell'operazione verrà restituito il valore scritto o, nel caso di fallimento, un semplice **nil**.

Altra funzione importante nella gestione dei registri è quella relativa all'eliminazione di una chiave o di un valore, il suo nome è **vl-registry-delete** ed è cosi definita:

```
(vl-registry-delete reg-key nomevalore)
```

Specificando solo il primo parametro sarà possibile eliminare un'intera chiave, mentre aggiungendo anche il secondo sarà possibile cancellare solo un certo valore. Immediata e intuitiva.

Le Stringhe

Entriamo ora nel mondo delle stringe e analizziamo le funzioni di Visual Lisp che le trattano.

Parliamo di codici ASCII. Cos'è un codice ASCII? Questi codici sono dei valori interi che rappresentano, in modo univoco, un carattere alfanumerico. Il linguaggio Lisp dispone della funzione **ascii** in grado di ritornare il codice ASCII di un qualsiasi carattere, ad esempio:

```
(ascii "a")
ritorna -> 97
```

Per compiere l'operazione inversa, cioè trasformare un codice ASCII nel corrispondente carattere abbiamo invece **chr**:

```
(chr 97)
ritorna -> "a"
```

Partendo da queste basi, Visual Lisp aggiunge una serie di funzioni di utilità per agevolare ulteriormente il trattamento dei codici ASCII.

Partiamo da **vl-string->list** che, dato un elenco di caratteri, ritorna una lista contenente i relativi codici ASCII:

```
(vl-string->list "abcd")
ritorna -> (97 98 99 100)
```

Abbiamo poi **vl-string-translate** che permette di sostituire, all'interno di una stringa, uno o più caratteri. La sua definizione:

```
(vl-string-translate cerca sostituzione stringa)
```

Supponiamo di voler sostituire tutti i caratteri "e" presenti in un testo con "f":

```
(vl-string-translate "e" "f" "Essere o non essere")
ritorna -> "Essfrf o non fssfrf"
```

La stessa funzione permette anche sostituzioni multiple. Sostituiamo ora tutte le "e" e le "o" minuscole con le corrispondenti maiuscole "E" ed "O":

```
(vl-string-translate "eo" "EO" "Essere o non essere")
ritorna -> "EssErE O nOn EssErE"
```

Vediamo ora **vl-string-elt** che ritorna il codice ASCII del carattere alla posizione N nella stringa specificata. Eccone la definizione:

```
(vl-string-elt stringa N)
```

Ad esempio:

```
(vl-string-elt "Essere o non essere" 2)
ritorna -> 115
```

Un'altra funzione che utilizza i codici ASCII è la **vl-string-position**. In questo caso possiamo ottenere la posizione della prima occorrenza di un dato codice ASCII all'interno di una stringa:

```
(vl-string-position codice stringa inizio direzione)
```

Ovviamente nel primo parametro **codice** verrà specificato il codice ASCII da cercare, in **stringa** avremo il testo nel quale trovare il carattere, con **inizio** potremo indicare la posizione per l'inizio della ricerca ed infine, con **direzione**, potremo decidere se iniziare la ricerca dall'inizio del testo o dalla fine.

Facciamo qualche esempio:

```
(vl-string-position (ascii "o") "Essere o non essere")
ritorna -> 7
```

Il primo carattere del testo ha sempre posizione 0 quindi con 7 viene identificato l'ottavo carattere della stringa. Ma andiamo oltre, provando la ricerca facendola iniziare dalla posizione 8:

```
(vl-string-position (ascii "o") "Essere o non essere" 8)
ritorna -> 10
```

Infine, proviamo una ricerca partendo dalla fine del testo:

```
(vl-string-position (ascii "o") "Essere o non essere" 0 t)
ritorna -> 10
```

Questa ricerca avviene partendo dall'ultimo carattere e procede verso il primo quindi, l'espressione ritorna la posizione dell'ultima "o" presente nel testo.

Passiamo ora a parlare di **vl-string-trim** che consente l'eliminazione un set di caratteri dall'inizio e della fine di un testo. Funzioni molto simili sono **vl-string-left-trim** e **vl-string-right-trim** che operano, rispettivamente, sulla parte iniziale o finale del testo. Ecco le definizioni :

```
(vl-string-trim caratteri stringa)
(vl-string-left-trim caratteri stringa)
(vl-string-right-trim caratteri stringa)
```

Facciamo qualche esempio, iniziando da qualcosa di molto comune, l'eliminazione di tutti gli spazzi prima e dopo un testo:

```
(vl-string-trim " " "   Essere o non essere  ")
ritorna -> "Essere o non essere"
```

Proviamo ora ad eliminare, oltre gli spazi, anche le lettere "e" minuscole e le "E" maiuscole:

```
(vl-string-trim " eE" "   Essere o non essere  ")
ritorna -> "ssere o non esser"
```

Non c'è altro da dire, se non il fatto che il primo parametro, come si può vedere, è semplicemente l'elenco dei caratteri da eliminare e distingue tra maiuscole e minuscole.

Passiamo ora ad esaminare alcune funzioni di ricerca. Prima di tutto **vl-string-search** che cerca un parziale all'interno di una stringa e, nel caso venga individuato, ritorna l'indice del primo carattere trovato.

La definizione:

```
(vl-string-search ricerca stringa inizio)
```

Il parametro **ricerca** rappresenta il parziale da trovare, **stringa** è il testo nel quale effettuare la ricerca mentre l'ultimo parametro, opzionale, permette di stabilire da quale carattere iniziare la ricerca (il primo carattere è il numero 0).

Facciamo subito un esempio:

```
(vl-string-search "essere" "essere o non essere")
ritorna -> 0
(vl-string-search "essere" "essere o non essere" 5)
ritorna -> 13
```

Nel primo caso viene identificata la parola "essere" all'inizio del testo (carattere 0), nel secondo caso la parola viene trovata in posizione 13 in quanto la ricerca inizia dal sesto carattere della stringa (5 partendo da 0). Ovviamente, se il testo cercato non è presente, la funzione ritornerà nil.

Sempre in tema di stringhe esiste una funzione, simile alla precedente, ma che una volta trovata il testo indicato lo sostituisce. La funzione di ricerca e sostituzione si chiama **vl-string-subst** ed è cosi definita:

```
(vl-string-subst nuovaStr ricerca stringa inizio)
```

Anche in questo caso vale quanto detto per **vl-string-search**, compreso il fatto che l'ultimo parametro, **inizio**, è facoltativo. La differenza risiede nella possibilità di specificare un testo che verrà inserito al posto della stringa trovata, **nuovaStr**.

Quando si utilizza **vl-string-subst** bisogna sempre tenere a mente che, solamente il primo testo trovato verrà sostituito, lasciando inalterato il resto del testo.

Ecco alcuni esempi:

```
(vl-string-subst "Non essere" "essere" "essere o non essere")
ritorna -> "Non essere o non essere"
```

```
(vl-string-subst "siamo" "essere" "essere o non essere" 5)
ritorna -> "essere o non siamo"
(vl-string-subst "siamo" "ciao" "essere o non essere")
ritorna -> "essere o non essere"
```

Ovviamente, la funzione ritornerà il testo risultante dopo la sostituzione. Una cosa interessante è il fatto che, nel caso in cui nulla venga sostituito, la funzione ritornerà lo stesso testo originale.

Purtroppo questa procedura non risolve un problema che si presenta spesso, cioè non consente di sostituire più parziali nella stringa. Quindi, se noi volessimo sostituire tutte le parole "essere" presenti con altro, questo richiede un poco più di codice.

Vediamo come sostituire più termini in una stringa:

```
(defun substAll (nuovaStr ricerca stringa / result)
    (setq result stringa)

    (while (/=
             (setq result (vl-string-subst nuovaStr ricerca
result))
             stringa)

         (setq stringa result)
    )

    result
)

(substAll "siamo" "essere" "essere o non essere")
ritorna -> "siamo o non siamo"
```

Questa procedura, **substAll** sostituisce tutte le occorrenze della parola "essere" con una nuova stringa. Bisogna però tenere presente che, cosi come è scritta, la funzione presenta un problema. Infatti non può sostituire testi che contengono se stessi. Ad esempio:

```
(substAll "non essere" "essere" "essere o non essere")
```

Questa istruzione non terminerà mai entrando in un ciclo infinito. Il motivo è che la parola da sostituire "essere" viene sostituita con una stringa che la contiene "non essere".

Andando oltre, un'altra funzione particolarmente utile è la **vl-string-mismatch** che viene utilizzata per analizzare due stringhe per sapere quanti caratteri coincidono tra le

due.

Ma cerchiamo di capire meglio. Iniziamo dalla definizione:

```
(vl-string-mismatch testo1 testo2 pos1 pos2 case)
```

I primi due argomenti sono i due testi da confrontare a seguire, opzionali, la posizione di partenza per il primo e il secondo testo, infine un **t** o **nil** per ignorare le differenze tra maiuscole o minuscole.

Vediamo alcuni esempi per capire l'utilità di questa funzione:

```
(vl-string-mismatch "essi" "essere o non essere")
ritorna -> 3
(vl-string-mismatch "essi" "essere o non essere" 0 13)
ritorna -> 3
(vl-string-mismatch "ciao" "essere o non essere")
ritorna -> 0
(vl-string-mismatch "Essi" "essere o non essere")
ritorna -> 0
(vl-string-mismatch "EsSi" "essere o non essere" 0 0 t)
ritorna -> 3
```

Iniziamo esaminando la prima istruzione, quella che confronta semplicemente "essi" con "essere o non essere". In questo caso viene ritornato 3, semplicemente perche è primi 3 caratteri delle due stringhe sono uguali. Tutto qui.

La seconda istruzione invece:

```
(vl-string-mismatch "essi" "essere o non essere" 0 13)
ritorna -> 3
```

In questo caso il confronto inizia, per la prima stringa dal suo primo carattere (che ha indice 0), mentre per la seconda stringa il confronto inizia dal 14 carattere cioè la "e". E anche in questo caso i caratteri in comune sono 3.

Un caso particolare è rappresentato dall'ultima istruzione:

```
(vl-string-mismatch "EsSi" "essere o non essere" 0 0 t)
ritorna -> 3
```

Che, utilizzando l'ultimo parametri per indicare di ignorare maiuscole e minuscole, confronta "EsSi" come se si trattasse di "essi", ritornando ovviamente 3.

Altre funzioni utili

Ci sono molte funzioni di Visual Lisp che andrebbero viste e commentate.

Una delle più importanti è sicuramente **vl-cmdf**. Prima di spiegarla, vediamone la definizione:

```
(vl-cmdf argomenti ...)
```

Quella che vediamo qui non è altro che una variante della famosissima funzione **command**, infatti **vl-cmdf** non fa altro che inviare al CAD uno o più comandi.

Tra **vl-cmdf** e **command** esiste però una differenza sostanziale, differenza che all'apparenza non si nota, ma è fondamentale. La funzione di Visual Lisp valuta tutti i parametri passati prima di eseguire il comando, mentre la funzione classica **command** esegue ogni parametro direttamente.

Esempio. Supponiamo di voler creare un rettangolo, definito da due punti (gli spigoli opposti. Il primo punto sarà specificato dall'utente, mentre il secondo punto verrà inserito da programma.

Ecco l'istruzione

```
    (command "._rectangle" (getpoint "primo angolo: ") '(10
10))
    (vl-cmdf "._rectangle" (getpoint "primo angolo: ") '(10
10))
```

La prima linea utilizza la classica **command**, la seconda la nuova **vl-cmdf**. Identiche ma con un comportamento diverso.

Nel primo caso, con **command**, ognuno dei tre parametri viene passato al CAD. Nel secondo caso, con **vl-cmdf**, i parametri vengono valutati ed eseguiti prima di essere inviati al CAD, e questo permette un maggior controllo sugli errori.

Nel caso appena visto, nella pratica, non esiste differenza, entrambe le istruzioni funzionano e producono il medesimo risultati.

Però, se le istruzioni non fossero quelle appena viste, ma fossero queste:

```
(command "._rectangle" (getpoint "primo angolo: ") '(10))
(vl-cmdf "._rectangle" (getpoint "primo angolo: ") '(10))
```

In questo caso il risultato sarebbe molto, molto, diverso. In queste due linee c'è un errore, il secondo parametro non specifica un punto, ma solamente la coordinata X (10). Questo causerà l'interruzione del comando di creazione del nostro rettangolo che, come è giusto, rifiuterà l'input giudicandolo non valido.

Utilizzando la funzione **command** si noterà che, l'istruzione lascia in sospeso il comando rettangolo che lascerà l'utente con un fastidioso messaggio a video che

richiede un punto. Al contrario, **vl-cmdf**, valutando preventivamente i parametri riuscirà a "capire" in anticipo che il secondo parametro non è valido e, invece di lanciare il comando lasciandolo in sospeso, eviterà di eseguire il rettangolo nel CAD, terminando immediatamente l'istruzione e, per segnalare il problema, restituirà **nil**.

Infine occorre ricordare che, la classica **command** non ritorna alcun valore, quindi è impossibile sapere se l'istruzione ha avuto esito positivo, al contrario **vl-cmdf** ritorna **t** nel caso l'istruzione abbia avuto successo, altrimenti ritorna **nil**.

Capitolo 21
Riferimenti e software utili da Internet

I CAD che supportano il linguaggio LISP

I CAD che supportano il linguaggio LISP sono ormai molti e disponibili per molti sistemi operativi.

In questa sezione sono raccolti i più noti, corredati dal sito di riferimento. Tutti questi programmi sono di tipo commerciale e disponibili in varie versioni:

- AutoCAD (http://www.autodesk.com). Sicuramente il CAD più noto, questo è il software che ha introdotto, per primo, l'utilizzo del linguaggio LISP. Disponibile sia su Windows, sia su Mac OS X;
- progeCAD (http://www.progecad.com). Disponibile per Windows;
- iCADMac (http://www.icadmac.com). Disponibile per Mac OS X;
- BricsCAD (http://www.bricscad.com). Disponibile per Windows e Linux;
- ZwCAD (http://www.zwcad.com). Disponibile per Windows;
- IntelliCAD.org (http://www.intellicad.org);
- DraftSight (http://www.3ds.com/it/products/draftsight/free-cad-software/). Cad a basso costo disponibile per Windows, Linux e Mac;
- Nanocad (http://nanocad.com/). Cad gratuito, compatibile con i file dwg e dotato di un interprete lisp. Disponibile per Windows;
- Interessante lista di programmi CAD da utilizzare con il sistema operativo Linux (es.: Ubuntu, Redhat, Mint, Suse, Debian, ecc...), alcuni supportano il linguaggio Lisp (http://www.tech-edv.co.at/lunix/CADlinks.html).

Un aspetto particolarmente interessante è dato dal fatto che progeCAD sia un software sviluppato e assistito in italiana.

I Siti che parlano di LISP

Ovviamente, su internet, esistono moltissimi siti che parlano di LISP in congiunzione agli ambienti CAD.

Ecco una breve raccolta di quelli che ritengo essere i siti più interessanti:

- Forum di CAD3D.it (http://www.cad3d.it/forum1/forumdisplay.php?f=55);
- Raccolta di procedure (http://www.jtbworld.com/lisp.htm);

- Sito italiano dedicato al CAD e al linguaggio LISP (http://web.mclink.it/MK1401/index.htm);
- Sito dedicato esclusivamente a LISP (http://www.afralisp.net/index.php);
- Un altro sito, dedicato ad articoli incentrati sul linguaggio LISP (http://www.caddigest.com/subjects/autocad/tutorials/autolisp.htm).

Documentazione sul formato DXF

Il formato DXF è alla base della struttura del database di disegno dei moderni sistemi CAD.

La documentazione ufficiale è disponibile sul sito AutoDESK a questo indirizzo :

http://usa.autodesk.com/adsk/servlet/item?linkID=10809853&id=12272454&siteID=123112

Da questa pagina sarà possibile scaricare i references dettagliati, in lingua inglese, relativi al formato DXF.

Editor di testo per Lisp e DCL

Lavorando con i linguaggi Lisp e DCL, una delle prime scelte da compiere è rappresentata da quella dell'editor di testo.

Certamente, se si utilizza progeCAD, IntelliCAD o AutoCAD precedente alla versione 2000, la scelta più immediata cadrà sul 'blocco note' (notepad), presente in tutte le versioni di Windows. Al contrario, se si utilizza AutoCAD 2000 o successivo, certamente si opterà per l'editor di 'Visual Lisp' integrato nel CAD.

Nel primo caso, il problema risiede nella pochezza di 'Blocco Note', infatti questo editor non dispone di alcun comando per agevolare la programmazione. Nel caso invece di 'Visual Lisp', l'impatto iniziale è decisamente buono, in quanto è progettato proprio per lo sviluppo in Lisp e DCL, il problema, almeno dal mio punto di vista, è rappresentato dalla modalità di indentazione che l'editor utilizza e da alcune funzionalità non proprio intuitive. Un ulteriore limite, di questo editor, è la sua unione indissolubile con AutoCAD che costringe, per l'editazione di altri tipi di file o in assenza di AutoCAD, all'utilizzo di un altro strumento, le cui modalità operative saranno certamente diverse. Al contrario sarebbe molto utile l'uso di un solo editor per la creazione e modifica di qualsiasi file di testo.

Nel panorama software attuale, si possono trovare editor per ogni esigenza, si va da semplici editor di testo a veri e propri sistemi di editazione per lo sviluppo di software.

Esistono molti software commerciali e altrettanti gratuiti (freeware o Open Source). Di seguito riporto l'elenco degli editor che reputo migliori, in grado di modificare sia i file Lisp sia i file DCL o qualsiasi altro tipo di file testuale.

SciTE

Prima di tutto parliamo dell'editor che preferisco ed utilizzo, SciTE. Questo software, utilizzabile sia sotto Windows che in Linux, consente la creazione e la modifica di qualsiasi file di testo, con la possibilità di evidenziare la sintassi in base al linguaggi, ed è studiato in modo particolare per i programmatori, fornendo tutte quelle funzioni utili al lavoro quotidiano. Caratteristica, non certo secondaria, è il fatto che sia Open Source, quindi gratuito e propenso a personalizzazioni e modifiche.

SciTE può essere scaricato, in versione italiana, dal sito internet dell'autore di questo testo (nella sezione 'software'):

http://www.redchar.net/

Oltre a possedere l'interfaccia in lingua italiana, è già predisposto per riconoscere sia il linguaggio Lisp di AutoCAD/progeCAD/IntelliCAD sia il linguaggio DCL. Per chi lo desiderasse, è possibile scaricare ed avere maggiori informazioni sulla versione originale di SciTE sul sito ufficiale:

http://scintilla.org

Leggendo le caratteristiche di questo editor ci si accorgerà che rappresenta uno dei più potenti e semplici sistemi di editazione attualmente disponibili.

Passiamo ora all'esame di altri editor, sia commerciali che gratuiti. Voglio ricordare che, nella maggior parte dei casi, non esiste un valido motivo per scegliere un prodotto commerciale al posto di uno Freeware o Open Source, in quanto il livello qualitativo di questi ultimi è uguale, se non superiori, ai corrispettivi a pagamento.

L'unica vera differenza tra le due tipologie di programmi è data dall'assistenza, di cui spesso, i prodotti commerciali dispongono, quindi rapida sistemazione dei bachi e possibilità di avere informazioni in maniera più tempestiva. Va certamente sottolineato, il fatto che, dato l'elevato livello qualitativo di tutti i prodotti che esamineremo, la necessità di assistenza è quasi sempre un'eventualità molto remota.

Editor di testo (Freeware e Open Source)

- RSciTE (Distribuzione italiana per Windows di SciTE). Disponibile su http://www.redchar.net
- SciTE (Windows/Linux). Disponibile su http://scintilla.org/SciTE.html

- Notepad++. Disponibile su http://notepad-plus-plus.org/
- ConText. Disponibile su http://www.fixedsys.com/context/
- Programmer's File Editor (PFE). Disponibile su http://download.com.com/3000-2352-904159.html

Editor di testo (Commerciali)

- UltraEdit-32. Disponibile su http://www.ultraedit.com/
- ZeusEdit. Disponibile presso http://www.zeusedit.com/
- EditPlus. Disponibile su http://www.editplus.com/
- ED for Windows. Disponibile su http://www.getsoft.com/
- Boxer. Disponibile su http://www.text-editors.com/?go .

Questo elenco non è certamente esaustivo, ma comunque fornisce quanto necessario per lavorare in maniera semplice e produttiva.

Capitolo 22
Licenza GNU per la Documentazione Libera

In questo capitolo viene riportata la licenza di questo libro. Qui troverete, sia la versione Italiana, sia la versione ufficiale in lingua Inglese.

Versione non ufficiale, in lingua Italiana

Questa è una traduzione italiana non ufficiale della Licenza GNU per la Documentazione Libera. Non è pubblicata dalla Free Software Foundation e non ha valore legale nell'esprimere i termini di distribuzione delle opere che la utilizzano. Solo la versione originale in inglese della licenza ha valore legale. Lo scopo di questa traduzione è quello di aiutare le persone di lingua italiana a capire meglio il significato della licenza Licenza GNU per la Documentazione Libera (FDL).

This is an unofficial translation of the GNU Free Documentation License into Italian. It was not published by the Free Software Foundation, and does not legally state the distribution terms for software that uses the GNU FDL - only the original English text of the GNU FDL does that. The aim of this traslation is to help Italian speakers understand the GNU FDL better.

Licenza GNU per la Documentazione Libera

Versione 1.2, Novembre 2002

59 Temple Place, Suite 330, Boston, MA 02111-1307 USA

A chiunque è consentito copiare e distribuire copie letterali di questo documento di licenza, ma non sono permesse modifiche.

0. PREAMBOLO

Lo scopo di questa Licenza è di realizzare un manuale, un libro di testo o un altro documento utile e funzionale che sia "libero" nel senso della libertà di: assicurare ad ognuno l'effettiva libertà di copiarlo e ridistribuirlo, con o senza modifiche, sia a scopo di lucro o meno. In secondo luogo, questa Licenza conserva per l'autore e l'editore un mezzo per ottenere il riconoscimento per il loro lavoro, senza essere considerati responsabili per le modifiche eseguite da altri.

Questa Licenza è una sorta di "copyleft", che significa che i lavori derivati dal documento devono essere liberi nello stesso senso. La Licenza è un complemento alla Licenza Pubblica Generale GNU, che è una licenza copyleft progettata per il software libero.

Abbiamo progettato questa Licenza per l'uso in manuali per il software libero, perché il software libero richiede documentazione libera: un programma libero dovrebbe essere fornito di manuali che garantiscano le stesse libertà che garantisce il software. Ma questa Licenza non è limitata ai manuali del software; può essere usata per qualunque lavoro tesuale, indipendentemente dall'argomento che tratta o dal fatto che sia

pubblicato come libro stampato. Noi raccomandiamo questa Licenza principalmente per queli lavori il cui scopo è l'istruzione o per i manuali.

1. APPLICABILITA' E DEFINIZIONI

Questa Licenza si applica a qualsiasi manuale o altro lavoro, realizzato con qualsiasi mezzo, che contenga una nota inserita dal titolare detentore del copyright, che dica che può essere distribuito nei termini di questa Licenza. Tale nota garantisce una licenza mondiale, priva di diritti d'autore e di durata illimitata ad usare tale lavoro nelle condizioni indicate qui. Il"Documento", nel seguito, si riferisce ad un qualsiasi tale manuale o lavoro. Chiunque tale da godere dei diritti della licenza è indicato come "voi". Voi accettate la licenza se copiate, modificate o distribuite il lavoro in un qualsiasi modo che richieda il permesso secondo le leggi sul diritto d'autore.

Una "Versione Modificata" del Documento indica un qualsiasi lavoro contenente il Documento o una porzione di esso, sia letterale, sia con modifiche oppure tradotta in un'altra lingua.

Una "Sezione Secondaria" è una appendice cui si fa riferimento o una sezione di copertina del Documento in relazione con gli editori o gli autori del Documento con l'argomento principale del Documento (o con argomenti collegati) e che non contiene nulla che possa ricadere direttamente nell'argomento principale. (Per esempio, se il Documento è in parte un libro di testo di Matematica, una Sezione Secondaria non può spiegare alcuna matematica.) La relazione può essere una questione di collegamento storico con l'argomento o con argomenti correlati o legali, commerciali, filosofici, etici o posizioni politiche che li riguardano.

Le "Sezioni Non Modificabili" sono particolari Sezioni Secondarie i cui titoli sono indicati, come facenti parte delle Sezioni Non Modificabili, nella nota che afferma che il Documento è rilasciato nell'ambito di questa Licenza. Se una sezione non rispetta la definizione di Secondaria allora non può essere indicata come Sezione Non Modificabile. Se il Documento non indica alcuna Sezione Non Modificabile, allora significa che non ce ne sono.

I "Testi di Copertina" sono brevi passaggi di testo che sono indicati, come Testi di Copertina o Testi di Retro-Copertina, nella nota che afferma che il Documento è rilasciato nell'ambito di questa Licenza. Un Testo di Copertina può contenere al massimo 5 parole e un Testo di Retro-Copertina può contenere al massimo 25 parole.

Una copia "Trasparente" del documento indica una copia leggibile da un elaboratore, rappresentata in un formato la cui specifica è disponibile al pubblico generico, che è direttamente utilizzabile per la revisione del documento con generici elaboratori di testo o (per immagini composte da pixel) con generici programmi di grafica o (per i disegni) da elaboratori di immagini ampiamente disponibili, e che è utilizzabile come sorgente per formattatori di testo o per traduzioni automatiche in vari formati sfruttabili come sorgenti per formattatori di testo. Una copia realizzata in un altro formato Trasparente i cui segnatori o assenza di segnatori sono stati disposti in modo da impedire o scoraggiare modifiche successive ai lettori non è Trasparente. Un formato di immagine

non è Trasparente se usato per rappresentare un’ampia quantità di testo. Una copia che non è "Trasparente" è chiamata "Opaca".

Esempi di formati utilizzabili per copie Trasparenti includono il semplice ASCII senza segnatori, il formato di ingresso Texinfo, il formato di ingresso LaTeX, SGML o XML con DTD pubblicamente disponibile e semplice HTML conforme agli standard, PostScript o PDF predisposti per la modifica manuale. Esempi di formati di immagini Trasparenti includono PNG, XCF e JPG. I formati Opachi comprendono i formati proprietari che possono essere letti e modificati solo da elaboratori di testo proprietari, SGML o XML per i quali non sono pubblicamente disponibili DTD o strumenti di elaborazione e HTML generato automaticamente, PostScript o PDF prodotto da alcuni elaboratori di testo a solo scopo di visualizzazione.

La "Pagina del Titolo" indica, per un libro stampato, la pagina del titolo stessa, più tutte le pagine seguenti necessarie per indicare, in modo leggibile, i dati che questa Licenza richiede appaiano nella pagina del titolo. Per i lavori in un formato che non prevede la pagina del titolo in quanto tale, "Pagina del Titolo" indica il testo in prossimità della più evidente indicazione del titolo del lavoro, immediatamente precedente il corpo del testo.

Una sezione "Titolata XYZ" indica una sottounità specifica del Documento, il cui titolo è esattamente XYZ oppure contiene XYZ in parentesi seguito dal testo che traduce XYZ in un’altra lingua. (Qui XYZ indica il nome di una sezione specifica menzionata di seguito, come "Ringraziamenti", "Dedica","Approvazione", o "Cronologia".) "Conservare il Titolo" di una tale sezione quando si modifica il Documento significa che deve rimanere una sezione"Titolata XYZ" in accordo con questa definizione.

Il Documento può includere dei Limiti di Garanzia in prossimità della nota che indica che questa Licenza si applica al Documento. Questi Limiti di Garanzia sono da considerarsi inclusi, come riferimento, in questa Licenza ma solamente per quanto li riguarda: qualsiasi altra implicazione che questi Limiti di Garanzia possono avere è nulla e non ha effetto sul significato di questa Licenza.

2. COPIA LETTERALE

Potete copiare e distribuire il Documento con qualsiasi mezzo, sia a scopo di lucro sia non, purché che questa Licenza, le note di copyright e la nota della Licenza che informa che questa Licenza si applica al Documento, siano riprodotte su tutte le copie che non aggiungiate nessun’altra condizione all’infuori di quelle di questa Licenza. Non potete usare accorgimenti tecnici per impedirlo controllare la lettura o le copie successive che realizzate o distribuite. Comunque, potete accettare dei contributi in cambio delle copie. Se distribuite un grande numero di copie dovete anche seguire le condizioni della sezione 3.

Potete prestare copie alle stesse condizioni indicate sopra, e potete mostrare in pubblico le copie.

3. COPIA IN GRANDI QUANTITA’

Se pubblicate copie stampate (o copie su un mezzo che comunemente ha copertine stampate) del Documento, numericamente più di 100, e la licenza del Documento richiede i "Testi di Copertina", dovere includere le copie in copertine che portino scritto, in modo chiaro e leggibile, tutti i seguenti "Testi di Copertina": Testi di Copertina sulla copertina anteriore, e Testi di Retro-Copertina sulla copertina posteriore. Entrambe le copertine devono anche indentificarvi in modo chiaro e leggibile come gli editori di tali copie. La copertina anteriore deve indicare il titolo completo con tutte le parole del titolo identicamente grandi e visibili. Potete aggiungere altro materiale sulle copertine a supplemento. Le copie con modifiche limitate alle sole copertine, fintantoché mantengono il titolo del Documento e soddisfano queste condizioni, possono essere trattate come copie letterali per quanto riguarda il resto.

Se i testi richiesti per le copertine sono troppo voluminosi per essere leggibili, dovete inserire il primo di essi (fintantoché sono ragionevolmente inseribili) sulla copertina effettiva e continuare con il resto nelle pagine adiacenti.

Se pubblicate o distribuite numericamente più di 100 copie Opache del Documento, dovete anche includere una copia Trasparente leggibile da un elaboratore insieme a ciascuna copia Opaca o indicare all'interno o insieme a ciascuna copia Opaca un indirizzo di rete informatica presso il quale il pubblico generico ha accesso per prelevare, tramite protocolli di rete standard e pubblici, una copia Trasparente completa del Documento, senza materiale aggiunto. Se usate l'ultima opzione dovete comportarvi in modo adeguato, quando distribuite copie Opache del Documento in grande quantità, per assicurare che la copia Trasparente rimanga sempre accessibile, alla posizione indicata, per almeno un anno dopo che avete distribuito l'ultima copia Opaca (direttamente o tramite vostri agenti o ridistributori) di tale edizione al pubblico.

E' richiesto, ma non necessario, che contattiate gli autori del Documento prima di ridistribuire un grande numero di copie, in modo da dar loro la possibilità di fornirvi una versione aggiornata del Documento.

4. MODIFICHE

Potete copiare e distribuire una Versione Modificata del Documento nelle condizioni delle sezioni 2 e 3 sopra, a patto che la Versione Modificata sia rilasciata esattamente nell'ambito di questa Licenza, con la Versione Modificata considerata come Documento, licenziando così la distribuzione e la modifica della Versione Modificata a chiunque possegga una copia della stessa. In aggiunta dovete fare le seguenti cose, alla Versione Modificata:

A. Usare, nella Pagina del Titolo (e sulle copertine, se presenti) un titolo differente da quello del Documento, e da quello delle versioni precedenti (che dovrebbero, se presente, essere indicati nella sezione Cronologia del Documento). Potete usare lo stesso titolo di una versione precedente se l'editore originale di quella versione ve ne dà il permesso.

B. Indicare, nella Pagina del Titolo, come autori, una o più persone o enti responsabili quali autori delle modifiche alla Versione Modificata, insieme con almeno cinque dei

principali autori del Documenti (tutti i suoi principali autori, se ne ha meno di cinque), a meno che i suddetti non vi autorizzino a non soddisfare questo requisito.

C. Indicate, nella pagina del Titolo, il nome dell'editore della Versione Modificata, come editore.

D. Conservare tutte le note di copyright del Documento.

E. Aggiungere un'appropriata nota di copyright, per le vostre modifiche, adiacente alle altre note di copyright.

F. Includere, subito dopo le note di copyright, la nota di licenza che dà il permesso pubblico di usare la Versione Mdificata nei termini di questa Licenza, nella forma mostrata nel Addendum più sotto.

G. Conservare in quella nota di licenza la lista completa delle Sezioni Invarianti e dei Testi di Copertina richiesti e indicata nella nota di licenza del Documento.

H. Includere una copia inalterata di questa Licenza.

I. Conservare la sezione Titolata "Cronologia", Conservare il suo Titolo e aggiungere ad essa una voce che indica, almeno, il titolo, l'anno, i nuovi autori ed editore della Versione Modificata come scritto nella Pagina del Titolo. Se non c'è alcuna sezione Titolata "Cronologia" nel Documento, createne una indicanto titolo, anno, autori e editore del Documento, così come mostrato nella Pagina del Titolo e quindi aggiungete una voce che descriva la Versione modificata così come indicato nella frase precedente.

J. Conversare l'indirizzo di rete, se esiste, indicato nel Documento come accesso pubblico ad una copia Trasparente del Documento e, similarmente, gli indirizzi di rete dati nel Documento riguardanti versioni precedenti su cui questo si basa. Questi possono essere inseriti nella sezione "Cronologia". Potete omettere l'indirizzo di rete per un lavoro pubblicato almeno quattro anni prima del Documento stesso, o se l'editore originale della versione cui si riferisce ve ne dà il permesso.

K. Per ogni sezione Titolata "Ringraziamenti" o "Dedica", Conservare il Titolo della sezione e conservare nella sezione tutta la sostanza e il tono usato per ciascuno dei ringraziamenti ai contributori e/o dediche scritte all'interno.

L. Conservare tutte le Sezioni Non Modificabili del Documento, inalterate nel loro testo e nei titoli. I numeri di sezione o equivalenti non sono da considerare parte dei titoli della sezione.

M. Cancellare ogni sezione Titolata "Approvazione". Tale sezione può non essere inserita nella Versione Modificata.

N. Non retitolare alcune sezione esistente con il titolo "Approvazione" o in conflitto con i titoli delle Sezioni Non Modificabili. O. Conservare i Limiti di Garanzia.

Se la Versione Modificata include nuove sezioni di copertina o appendici che si qualificano come Sezioni Secondarie e non contengono materiale copiato dal

Documento, potete a vostra discrezione definire alcune o tutte queste sezioni come non modificabili. Per fare questo, aggiungete i loro titoli alla lista delle Sezioni Non Modificabili nella nota di licenza della Versione Modificata. Questi titoli devono essere distinti da ogni altro titolo di sezione.

Potete aggiungere una sezione Titolata "Approvazione", purché non contenga nulla eccetto approvazioni alla vostra Versione Modificata da terze parti — per esempio, indicazioni di revisione delle bozze o che il testo è stato approvato da un'organizzazione come definizione d'autorità di uno standard.

E' possibile aggiungere una frase lunga fino a cinque parole come Testo di Copertina e una frase fino a 25 parole come Testo di Retro-Copertina, alla fine della lista dei Testi di Copertina della Versione Modificata. Solo una frase del Testo di Copertina e una del Testo di Retro-Copertina potrà essere aggiunta (o potrà essere inserita tramite arrangiamenti) da ciascun ente. Se un Documento contiene già un testo per la stessa copertina, precedentemente aggiunto da voi o arrangiato dallo stesso ente per cui collaborate o ne fate le veci, non potete aggiungere altro; potete sostituire il testo vecchio, su esplicito permesso del precedente editore che lo ha aggiunto.

Gli autori e gli editori del Documento tramite questa Licenza non danno il permesso di utilizzare i loro nomi pubblicamente o per dichiarazioni di approvazione implicita di qualsiasi Versione MOdificata.

5. COMBINAZIONE DI DOCUMENTI

Potete combinare il Documento con altri documenti rilasciati nell'ambito di questa Licenza, nei termini definiti dalla sezione 4 per le versioni modificate, purché includiate nella combinazione tutte le Sezioni Non Modificabili di tutti i documenti originali, non modificati e elencarle tutte come Sezioni Non Modificabili del vostro lavoro combinato nella sua nota di licenza e che conserviate tutti i Termini di Garanzia.

Il lavoro combinato deve contenere solamente una copia di questa Licenza, e Sezioni Non Modificabili multiple identiche possono essere sostituite con una singola copia. Se ci sono molteplici Sezioni Non Modificabili con lo stesso nome ma con contenuti differenti, trasformate in modo univoco il titolo di ciascuna di queste sezioni aggiungendo, alla fine dello stesso, tra parentesi, il nome dell'autore o editore originale di quella sezione se conosciuti, o altrimenti un numero univoco. Fate lo stesso aggiustamento con i titoli delle sezioni nella lista delle Sezioni Non Modificabili nella nota di licenza del lavoro combinato.

Nella combinazione, dovete combinare ogni sezione Titolata "Cronologia" dei vari documenti originali, a formare una sezione Titolata "Cronologia"; allo stesso modo combinate le sezioni Titolate "Ringraziamenti" e le sezioni Titolate "Dedica". Dovete cancellare tutte le sezioni Titolate "Approvazione".

6. INSIEMI DI DOCUMENTI

Potete fare un insieme costituito dal Documento e da altri documento rilasciati con questa Licenza, e sostituire le singole copie di questa Licenza nei vari documenti con

una singola copia inclusa nell'insieme, purché seguiate le regole di questa Licenza per le copie letterali di ciascun documento sotto tutti gli aspetti.

Potete estrarre un singolo documento da un tale insieme, e distribuirlo individualmente nell'ambito di questa Licenza, purché inseriate una copia di questa Licenza nel documento estratto, e seguiate questa Licenza sotto tutti gli altri aspetti a proposito della copia letterale di quel documento.

7. AGGREGAZIONE CON LAVORI INDIPENDENTI

Un insieme del Documento o di suoi derivati con altri documenti o lavori separati e indipendenti all'interno di un dispositivo di memorizzazione o di un mezzo di distribuzione è chiamato "aggregato" se il copyright risultante dall'insieme non è usato per limitare i diritti legali degli utenti dell'insieme oltre quello che i lavori individuali permettono. Quando il Documento è incluso in un aggregato, questa Licenza non si applica agli altri lavori dell'aggregato che non sono essi stessi lavori derivati del Documento.

Se il requisito sul Testo di Copertina della sezione 3 è applicabile a queste copie del Documento, allora se il Documento è meno della metà dell'intero aggregato, i Testi di Copertina del Documento possono essere inseriti sulle copertine che racchiudono il Documento all'interno dell'aggregato, o l'equivalente elettronico delle copertine se il Documento è in forma elettronica. Altrimenti devono apparire sulle copertine stampate che racchiudono l'intero aggregato.

8. TRADUZIONE

La traduzione è da considerare una sorta di modifica così potete distribuire traduzioni del Documento nei termini della sezione 4. Sostituire le Sezioni Non Modificabili con traduzioni richiede uno speciale permesso dei detentori del copyright, ma potete includere traduzioni di alcune o parte delle Sezioni Non Modificabili in aggiunta alla versione originale di tali Sezioni Non Modificabili. Potete inserire una traduzione di questa Licenza, e di tutte le note di licenza del Documento, e dei Limiti di Garanzia, purché includiate anche la versione originale in lingua Inglese di questa Licenza e le versioni originali delle note e dei terminilimiti di garanzia. In caso di disaccordo tra la traduzione e la versione originale di questa Licenza o di una nota o di un termine di garanzia, la versione originale prevale.

Se una sezione del Documento è Titolata "Ringraziamenti", "Dedica", o "Cronologia", il requisito (sezione 4) di Conservare il Titolo (sezione 1) richiederà tipicamente di cambiare il titolo corrente.

9. RISOLUZIONE DELLA LICENZA

Non potete copiare, modificare, sublicenziare o distribuire questo Documento ad esclusione di quanto esplicitamente indicato da questa Licenza. Qualsiasi altro tentativo di copiare, modificare, sublicenziare o distribuire il Documento è nullo e comporterà automaticamente la risoluzione della stessa Licenza nei Vostri confronti. Comunque, coloro che hanno ricevuto copie o diritti da voi nell'ambito di questa

Licenza non vedranno risolversi le loro licenze fintantoché tali destinatari ne rispetteranno integralmente i termini.

10. REVISIONI SUCCESSIVE DI QUESTA LICENZA

Free Software Foudantion potrà pubblicare nuove versioni riviste della Licenza GNU per la Documentazione Libera, di tanto in tanto. Tali nuovi versioni saranno simili nello spirito alla presente versione ma potranno differire nei dettagli per sopperire a nuovi problemi o fatti. Si veda http://www.gnu.org/copyleft/.

Ciascuna versione della Licenza ha un proprio numero di versione distintivo. Se il Documento specifica che si applica ad esso una particolare versione di questa Licenza "o qualsiasi versione successiva", avete la scelta di seguire i termini e le condizioni della versione indicata o di qualsiasi versione successiva pubblicata (non le bozze) da Free Software Foundation. Se il Documento non specifica un numero di versione della Licenza potete scegliere qualsiasi versione pubblicata (non come bozza) da Free Software Foundation.

ADDENDUM: Come usare questa Licenza per i vostri documenti

Per usare questa Licenza in un documento che avete scritto, includete una copia della Licenza nel documento e inserite il seguente copyright e nota di licenza subito dopo la pagina del titolo:

Copyright (c) ANNO VOSTRO NOME.

Si garantisce il permesso di copiare, distribuire e/o modificare

questo documento nei termini della Licenza GNU per la Documentazione

Libera, Versione 1.2 o qualsiasi versione successiva pubblicata da

Free Software Foundation; senza Sezioni Non Modificabili, senza Testi

di Copertina e senza Testi di Retro-Copertina.

Una copia della licenza è inclusa nella sezione intitolata

"Licenza GNU per la Documentazione Libera".

Se avete Sezioni Non Modificabili, Testi di Copertina e Testi di Retro-Copertina, sostituite il testo "senza Sezioni...Retro-Copertina" con il seguente:

con le seguenti Sezioni Non Modificabili ELENCARE I TITOLI, con il

seguente Testo di Copertina ELENCARE e con il seguente Testo di

Retro-Copertina ELENCARE.

Se avete Sezioni Non Modificabili senza Testi di Copertina o qualche altra combinazione dei tre, mescolate le due alternative per adeguarle alla situazione.

Se il vostro documento contiene esempi non banali di codice di programmazione, vi

raccomandiamo di rilasciare tali esempi in parallelo nell'ambito di una licenza per il software libero come la Licenza Pubblica Generale GNU.

Versione ufficiale in lingua Inglese

GNU Free Documentation License

Version 1.2, November 2002

59 Temple Place, Suite 330, Boston, MA 02111-1307 USA

Everyone is permitted to copy and distribute verbatim copies

of this license document, but changing it is not allowed.

0. PREAMBLE

The purpose of this License is to make a manual, textbook, or other functional and useful document "free" in the sense of freedom: to assure everyone the effective freedom to copy and redistribute it, with or without modifying it, either commercially or noncommercially. Secondarily, this License preserves for the author and publisher a way to get credit for their work, while not being considered responsible for modifications made by others.

This License is a kind of "copyleft", which means that derivative works of the document must themselves be free in the same sense. It complements the GNU General Public License, which is a copyleft license designed for free software.

We have designed this License in order to use it for manuals for free software, because free software needs free documentation: a free program should come with manuals providing the same freedoms that the software does. But this License is not limited to software manuals; it can be used for any textual work, regardless of subject matter or whether it is published as a printed book. We recommend this License principally for works whose purpose is instruction or reference.

1. APPLICABILITY AND DEFINITIONS

This License applies to any manual or other work, in any medium, that contains a notice placed by the copyright holder saying it can be distributed under the terms of this License. Such a notice grants a world-wide, royalty-free license, unlimited in duration, to use that work under the conditions stated herein. The "Document", below, refers to any such manual or work. Any member of the public is a licensee, and is addressed as "you". You accept the license if you copy, modify or distribute the work in a way requiring permission under copyright law.

A "Modified Version" of the Document means any work containing the Document or a portion of it, either copied verbatim, or with modifications and/or translated into another

language.

A "Secondary Section" is a named appendix or a front-matter section of the Document that deals exclusively with the relationship of the publishers or authors of the Document to the Document's overall subject (or to related matters) and contains nothing that could fall directly within that overall subject. (Thus, if the Document is in part a textbook of mathematics, a Secondary Section may not explain any mathematics.) The relationship could be a matter of historical connection with the subject or with related matters, or of legal, commercial, philosophical, ethical or political position regarding them.

The "Invariant Sections" are certain Secondary Sections whose titles are designated, as being those of Invariant Sections, in the notice that says that the Document is released under this License. If a section does not fit the above definition of Secondary then it is not allowed to be designated as Invariant. The Document may contain zero Invariant Sections. If the Document does not identify any Invariant Sections then there are none.

The "Cover Texts" are certain short passages of text that are listed, as Front-Cover Texts or Back-Cover Texts, in the notice that says that the Document is released under this License. A Front-Cover Text may be at most 5 words, and a Back-Cover Text may be at most 25 words.

A "Transparent" copy of the Document means a machine-readable copy, represented in a format whose specification is available to the general public, that is suitable for revising the document straightforwardly with generic text editors or (for images composed of pixels) generic paint programs or (for drawings) some widely available drawing editor, and that is suitable for input to text formatters or for automatic translation to a variety of formats suitable for input to text formatters. A copy made in an otherwise Transparent file format whose markup, or absence of markup, has been arranged to thwart or discourage subsequent modification by readers is not Transparent. An image format is not Transparent if used for any substantial amount of text. A copy that is not "Transparent" is called "Opaque".

Examples of suitable formats for Transparent copies include plain ASCII without markup, Texinfo input format, LaTeX input format, SGML or XML using a publicly available DTD, and standard-conforming simple HTML, PostScript or PDF designed for human modification. Examples of transparent image formats include PNG, XCF and JPG. Opaque formats include proprietary formats that can be read and edited only by proprietary word processors, SGML or XML for which the DTD and/or processing tools are not generally available, and the machine-generated HTML, PostScript or PDF produced by some word processors for output purposes only.

The "Title Page" means, for a printed book, the title page itself, plus such following pages as are needed to hold, legibly, the material this License requires to appear in the title page. For works in formats which do not have any title page as such, "Title Page" means the text near the most prominent appearance of the work's title, preceding the beginning of the body of the text.

A section "Entitled XYZ" means a named subunit of the Document whose title either is precisely XYZ or contains XYZ in parentheses following text that translates XYZ in another language. (Here XYZ stands for a specific section name mentioned below, such as "Acknowledgements", "Dedications", "Endorsements", or "History".) To "Preserve the Title" of such a section when you modify the Document means that it remains a section "Entitled XYZ" according to this definition.

The Document may include Warranty Disclaimers next to the notice which states that this License applies to the Document. These Warranty Disclaimers are considered to be included by reference in this License, but only as regards disclaiming warranties: any other implication that these Warranty Disclaimers may have is void and has no effect on the meaning of this License.

2. VERBATIM COPYING

You may copy and distribute the Document in any medium, either commercially or noncommercially, provided that this License, the copyright notices, and the license notice saying this License applies to the Document are reproduced in all copies, and that you add no other conditions whatsoever to those of this License. You may not use technical measures to obstruct or control the reading or further copying of the copies you make or distribute. However, you may accept compensation in exchange for copies. If you distribute a large enough number of copies you must also follow the conditions in section 3.

You may also lend copies, under the same conditions stated above, and you may publicly display copies.

3. COPYING IN QUANTITY

If you publish printed copies (or copies in media that commonly have printed covers) of the Document, numbering more than 100, and the Document’s license notice requires Cover Texts, you must enclose the copies in covers that carry, clearly and legibly, all these Cover Texts: Front-Cover Texts on the front cover, and Back-Cover Texts on the back cover. Both covers must also clearly and legibly identify you as the publisher of these copies. The front cover must present the full title with all words of the title equally prominent and visible. You may add other material on the covers in addition. Copying with changes limited to the covers, as long as they preserve the title of the Document and satisfy these conditions, can be treated as verbatim copying in other respects.

If the required texts for either cover are too voluminous to fit legibly, you should put the first ones listed (as many as fit reasonably) on the actual cover, and continue the rest onto adjacent pages.

If you publish or distribute Opaque copies of the Document numbering more than 100, you must either include a machine-readable Transparent copy along with each Opaque copy, or state in or with each Opaque copy a computer-network location from which the general network-using public has access to download using public-standard network protocols a complete Transparent copy of the Document, free of added material. If you use the latter option, you must take reasonably prudent steps, when you begin

distribution of Opaque copies in quantity, to ensure that this Transparent copy will remain thus accessible at the stated location until at least one year after the last time you distribute an Opaque copy (directly or through your agents or retailers) of that edition to the public.

It is requested, but not required, that you contact the authors of the Document well before redistributing any large number of copies, to give them a chance to provide you with an updated version of the Document.

4. MODIFICATIONS

You may copy and distribute a Modified Version of the Document under the conditions of sections 2 and 3 above, provided that you release the Modified Version under precisely this License, with the Modified Version filling the role of the Document, thus licensing distribution and modification of the Modified Version to whoever possesses a copy of it. In addition, you must do these things in the Modified Version:

A. Use in the Title Page (and on the covers, if any) a title distinct from that of the Document, and from those of previous versions (which should, if there were any, be listed in the History section of the Document). You may use the same title as a previous version if the original publisher of that version gives permission.

B. List on the Title Page, as authors, one or more persons or entities responsible for authorship of the modifications in the Modified Version, together with at least five of the principal authors of the Document (all of its principal authors, if it has fewer than five), unless they release you from this requirement.

C. State on the Title page the name of the publisher of the Modified Version, as the publisher.

D. Preserve all the copyright notices of the Document.

E. Add an appropriate copyright notice for your modifications adjacent to the other copyright notices.

F. Include, immediately after the copyright notices, a license notice giving the public permission to use the Modified Version under the terms of this License, in the form shown in the Addendum below.

G. Preserve in that license notice the full lists of Invariant Sections and required Cover Texts given in the Document’s license notice.

H. Include an unaltered copy of this License.

I. Preserve the section Entitled "History", Preserve its Title, and add to it an item stating at least the title, year, new authors, and publisher of the Modified Version as given on the Title Page. If there is no section Entitled "History" in the Document, create one stating the title, year, authors, and publisher of the Document as given on its Title Page, then add an item describing the Modified Version as stated in the previous sentence.

J. Preserve the network location, if any, given in the Document for public access to a Transparent copy of the Document, and likewise the network locations given in the Document for previous versions it was based on. These may be placed in the "History" section. You may omit a network location for a work that was published at least four years before the Document itself, or if the original publisher of the version it refers to gives permission.

K. For any section Entitled "Acknowledgements" or "Dedications", Preserve the Title of the section, and preserve in the section all the substance and tone of each of the contributor acknowledgements and/or dedications given therein.

L. Preserve all the Invariant Sections of the Document, unaltered in their text and in their titles. Section numbers or the equivalent are not considered part of the section titles.

M. Delete any section Entitled "Endorsements". Such a section may not be included in the Modified Version.

N. Do not retitle any existing section to be Entitled "Endorsements" or to conflict in title with any Invariant Section.

O. Preserve any Warranty Disclaimers.

If the Modified Version includes new front-matter sections or appendices that qualify as Secondary Sections and contain no material copied from the Document, you may at your option designate some or all of these sections as invariant. To do this, add their titles to the list of Invariant Sections in the Modified Version's license notice. These titles must be distinct from any other section titles.

You may add a section Entitled "Endorsements", provided it contains nothing but endorsements of your Modified Version by various parties--for example, statements of peer review or that the text has been approved by an organization as the authoritative definition of a standard.

You may add a passage of up to five words as a Front-Cover Text, and a passage of up to 25 words as a Back-Cover Text, to the end of the list of Cover Texts in the Modified Version. Only one passage of Front-Cover Text and one of Back-Cover Text may be added by (or through arrangements made by) any one entity. If the Document already includes a cover text for the same cover, previously added by you or by arrangement made by the same entity you are acting on behalf of, you may not add another; but you may replace the old one, on explicit permission from the previous publisher that added the old one.

The author(s) and publisher(s) of the Document do not by this License give permission to use their names for publicity for or to assert or imply endorsement of any Modified Version.

5. COMBINING DOCUMENTS

You may combine the Document with other documents released under this License,

under the terms defined in section 4 above for modified versions, provided that you include in the combination all of the Invariant Sections of all of the original documents, unmodified, and list them all as Invariant Sections of your combined work in its license notice, and that you preserve all their Warranty Disclaimers.

The combined work need only contain one copy of this License, and multiple identical Invariant Sections may be replaced with a single copy. If there are multiple Invariant Sections with the same name but different contents, make the title of each such section unique by adding at the end of it, in parentheses, the name of the original author or publisher of that section if known, or else a unique number. Make the same adjustment to the section titles in the list of Invariant Sections in the license notice of the combined work.

In the combination, you must combine any sections Entitled "History" in the various original documents, forming one section Entitled "History"; likewise combine any sections Entitled "Acknowledgements", and any sections Entitled "Dedications". You must delete all sections Entitled "Endorsements."

6. COLLECTIONS OF DOCUMENTS

You may make a collection consisting of the Document and other documents released under this License, and replace the individual copies of this License in the various documents with a single copy that is included in the collection, provided that you follow the rules of this License for verbatim copying of each of the documents in all other respects.

You may extract a single document from such a collection, and distribute it individually under this License, provided you insert a copy of this License into the extracted document, and follow this License in all other respects regarding verbatim copying of that document.

7. AGGREGATION WITH INDEPENDENT WORKS

A compilation of the Document or its derivatives with other separate and independent documents or works, in or on a volume of a storage or distribution medium, is called an "aggregate" if the copyright resulting from the compilation is not used to limit the legal rights of the compilation’s users beyond what the individual works permit. When the Document is included in an aggregate, this License does not apply to the other works in the aggregate which are not themselves derivative works of the Document.

If the Cover Text requirement of section 3 is applicable to these copies of the Document, then if the Document is less than one half of the entire aggregate, the Document’s Cover Texts may be placed on covers that bracket the Document within the aggregate, or the electronic equivalent of covers if the Document is in electronic form. Otherwise they must appear on printed covers that bracket the whole aggregate.

8. TRANSLATION

Translation is considered a kind of modification, so you may distribute translations of

the Document under the terms of section 4. Replacing Invariant Sections with translations requires special permission from their copyright holders, but you may include translations of some or all Invariant Sections in addition to the original versions of these Invariant Sections. You may include a translation of this License, and all the license notices in the Document, and any Warranty Disclaimers, provided that you also include the original English version of this License and the original versions of those notices and disclaimers. In case of a disagreement between the translation and the original version of this License or a notice or disclaimer, the original version will prevail.

If a section in the Document is Entitled "Acknowledgements", "Dedications", or "History", the requirement (section 4) to Preserve its Title (section 1) will typically require changing the actual title.

9. TERMINATION

You may not copy, modify, sublicense, or distribute the Document except as expressly provided for under this License. Any other attempt to copy, modify, sublicense or distribute the Document is void, and will automatically terminate your rights under this License. However, parties who have received copies, or rights, from you under this License will not have their licenses terminated so long as such parties remain in full compliance.

10. FUTURE REVISIONS OF THIS LICENSE

The Free Software Foundation may publish new, revised versions of the GNU Free Documentation License from time to time. Such new versions will be similar in spirit to the present version, but may differ in detail to address new problems or concerns. See http://www.gnu.org/copyleft/.

Each version of the License is given a distinguishing version number. If the Document specifies that a particular numbered version of this License "or any later version" applies to it, you have the option of following the terms and conditions either of that specified version or of any later version that has been published (not as a draft) by the Free Software Foundation. If the Document does not specify a version number of this License, you may choose any version ever published (not as a draft) by the Free Software Foundation.

How to use this License for your documents

To use this License in a document you have written, include a copy of the License in the document and put the following copyright and license notices just after the title page:

Copyright (c) YEAR YOUR NAME.

Permission is granted to copy, distribute and/or modify this document under the terms of the GNU Free Documentation License, Version 1.2 or any later version published by the Free Software Foundation; with no Invariant Sections, no Front-Cover Texts, and no Back-Cover Texts. A copy of the license is included in the section entitled "GNU

Free Documentation License".

If you have Invariant Sections, Front-Cover Texts and Back-Cover Texts, replace the "with...Texts." line with this:

with the Invariant Sections being LIST THEIR TITLES, with the

Front-Cover Texts being LIST, and with the Back-Cover Texts being LIST.

If you have Invariant Sections without Cover Texts, or some other combination of the three, merge those two alternatives to suit the situation.

If your document contains nontrivial examples of program code, we recommend releasing these examples in parallel under your choice of free software license, such as the GNU General Public License, to permit their use in free software.

Il LISP, nonostante sia un ottimo linguaggio di programmazione in ambiente CAD, poco conosciuto.

Al contrario di tutti gli altri linguaggi, LISP ha la particolarità di consentire la realizzazione di applicativi di piccola e media dimensione riducendo drasticamente la quantità di codice scritta inoltre, grazie particolarmente favorevole, consente la stesura delle prime procedure dopo pochissime ore di studio.

Essendo convinto che LISP, AutoLISP, possano soddisfare le esigenze della maggior parte dei disegnatori lavorano con i programmi CAD ho scritto, con la collaborazione di alcuni amici, un testo che spiega come utilizzarlo per automatizzare l'ambiente di lavoro, sia questo AutoCAD®, IntelliCAD®, progeCAD®, ZwCAD®, BricsCAD®, nanoCAD® ed in genere qualsiasi altro CAD che supporti tale linguaggio.

di Roberto Rossi
www.redchar.net