

*AMIGA*<sup>®</sup> OS 3.1

**ARexx**



 **Commodore**



## Copyright

Copyright © 1992 della Commodore Electronics Limited. Tutti i diritti sono riservati. Questo documento non può essere, in tutto o in parte, copiato, fotocopiato, riprodotto, tradotto o ridotto in qualsiasi forma o mezzo elettronico, senza previo consenso scritto della Commodore Electronics Limited.

Il materiale dichiarato in *USO DI AmigaDOS* è ricavato dal *MANUALE AmigaDOS*, Seconda Edizione, Copyright © 1987 della Commodore-Amiga, Inc. usato con autorizzazione scritta della Bantam Books. Tutti i diritti sono riservati. Le serie di caratteri Times Roman, Helvetica Medium e Courier incluse nella directory Fonts sul disco Fonts sono soggette a Copyright © 1985, 1987 Adobe Systems, Inc. Le serie di caratteri CG Times, Univers Medium e LetterGothic incluse nel disco Fonts sono soggette a Copyright © 1990 della Agfa Corporation ed a licenza della Agfa Corporation.

## Declinazione di responsabilità

La Commodore non presta alcuna assicurazione o garanzia, espressa o implicita, rispetto ai prodotti descritti in questo documento. Le informazioni contenute in questo documento sono fornite "COME SONO" ed espressamente soggette a modifica senza preavviso. Qualsiasi rischio derivante dall'uso di queste informazioni è a completo carico dell'utente. IN NESSUN CASO LA COMMODORE SARÀ RESPONSABILE PER DANNI DIRETTI, INDIRETTI, ACCIDENTALI O CONSEGUENZIALI RISULTANTI DA QUALSIASI RECLAMO CHE POSSA SORGERE DALLE ASSERTZIONI QUI PRESENTI, ANCHE SE C'È STATA INFORMAZIONE SULLA POSSIBILITÀ DI TALI DANNI. ALCUNI STATI NON PERMETTONO LA LIMITAZIONE DI TALI GARANZIE O DANNI, PERCIÒ LE LIMITAZIONI DI CUI SOPRA NON SONO VALIDE.

## Marchi di fabbrica

Commodore, il logo Commodore e CBM sono marchi di fabbrica registrati della Commodore Electronics Limited negli Stati Uniti e molti altri paesi. Amiga è un marchio di fabbrica registrato della Commodore-Amiga, Inc. AmigaDOS, Amiga Kickstart, Amiga Workbench, AutoConfig e Bridgeboard sono marchi di fabbrica della Commodore-Amiga, Inc. AmigaVision è un marchio di fabbrica della Commodore Electronics Limited e Commodore. MS-DOS è un marchio di fabbrica registrato della Microsoft Corporation.

Compugraphic, CG e Intellifont sono marchi di fabbrica registrati della Agfa Corp. CG Triumverate è un marchio della Agfa Corp. CG Times è basato su Times New Roman con licenza della The Monotype Corporation plc. Times New Roman è un marchio di fabbrica registrato della Monotype Corporation. Univers è un marchio di fabbrica registrato della Linotype AG. Universe è su licenza della Haas Typefoundry Ltd.

Diablo è un marchio di fabbrica registrato della Xerox Corporation; Epson è un marchio di fabbrica registrato della Epson America, Inc.; IBM e Proprinter XL sono marchi di fabbrica registrati della International Business Machines Corp; Imagewriter è un marchio di fabbrica della Apple Computer, Inc.; LaserJet e LaserJet PLUS sono marchi di fabbrica della Hewlett-Packard Company; NEC e Pinwriter sono marchi di fabbrica registrati della NEC Information Systems; Okidata è un marchio di fabbrica registrato della Okidata, una divisione della Oki America, Inc.; Okimate 20 è un marchio di fabbrica della Okidata, una divisione della Oki America., Inc.

Questo documento può anche contenere riferimenti ad altri marchi di fabbrica che si presume appartengano ai rispettivi costruttori.

*Coverdesign and Print by Village Tronic*

*Village Tronic Marketing GmbH, Wellweg 95, 31157 Sarstedt, Germany*

*Questo manuale è stato prodotto da Robert Spephenson Weir usando una varietà di sistemi Commodore.*

Codice: 368761-01



# ***Indice***

---

## ***Introduzione ad ARexx***

A chi serve ARexx? .....	1-1
ARexx su Amiga .....	1-2
Caratteristiche ARexx.....	1-3

## ***Predisposizioni operative***

Avvio di ARexx .....	2-1
Avvio automatico .....	2-1
Avvio manuale .....	2-2
Informazioni sui programmi ARexx.....	2-2
Uso dei programmi ARexx .....	2-3
Esempi di programma .....	2-4
Amiga.rexx .....	2-5
Età.rexx .....	2-6
Calc.rexx .....	2-7
Even.rexx .....	2-7
Square.rexx .....	2-8
Results.rexx.....	2-9
Grades.rexx.....	2-9

## ***Elementi di ARexx***

Indicatori .....	3-1
Commenti .....	3-2
Simboli.....	3-2
Stringhe .....	3-5

Operatori .....	3-5
Operatori aritmetici .....	3-6
Operatori di concatenazione .....	3-7
Operatori di comparazione .....	3-8
Operatori logici (booleani).....	3-9
Caratteri speciali .....	3-9
<b>Clausole .....</b>	<b>3-10</b>
Clausole nulle .....	3-11
Clausole etichetta.....	3-11
Clausole assegnazione .....	3-12
Clausole istruzione .....	3-12
Clausole comando .....	3-12
<b>Espressioni.....</b>	<b>3-13</b>
<b>Interfaccia di comando .....</b>	<b>3-15</b>
Indirizzo ospite .....	3-15
Creazione di una macro .....	3-17
Codici di ritorno .....	3-19
Shell .....	3-19
<b>Ambiente di esecuzione.....</b>	<b>3-20</b>
Ambiente esterno .....	3-20
Ambiente interno.....	3-21
Ricerca risorse .....	3-22

## ***Istruzioni***

Sintassi .....	4-1
Riferimento alfabetico.....	4-2

## ***Funzioni***

Chiamata di una funzione.....	5-1
<b>Tipi di funzione.....</b>	<b>5-2</b>
Funzioni interne .....	5-2
Funzioni integrate.....	5-3
Librerie di funzioni esterne .....	5-3
Lista delle librerie .....	5-4
Ospiti di funzioni esterne .....	5-4
<b>Ordine di ricerca.....</b>	<b>5-5</b>
<b>Lista clip .....</b>	<b>5-6</b>

<b>Funzioni integrate — Riferimento.....</b>	<b>5-7</b>
Sintassi.....	5-7
Riferimento alfabetico .....	5-8
Programma esemplificativo .....	5-37
<b>Funzioni REXXSupport.Library .....</b>	<b>5-40</b>

## ***Diagnostica***

<b>Analisi .....</b>	<b>6-1</b>
Uscita dell'analisi.....	6-3
Inibizione comando .....	6-4
Analisi interattiva.....	6-5
Gestione errori.....	6-6
Indicatore di analisi esterno .....	6-6
<b>Interruzioni.....</b>	<b>6-7</b>

## ***Analisi sintattica***

<b>Modelli.....</b>	<b>7-1</b>
Marcatori .....	7-2
Obiettivi .....	7-2
Oggetti modello.....	7-3
<b>Il processo di scansione .....</b>	<b>7-4</b>
<b>Esempi di analisi sintattica .....</b>	<b>7-5</b>
Analisi sintattica mediante identazione .....	7-5
Analisi sintattica mediante configurazione.....	7-6
Analisi sintattica mediante marcatori di posizione .....	7-6
Modelli multipli .....	7-7

## ***Appendice A***

### ***Messaggi di errore***

***Appendice B***  
***Utilità di comando***

***Glossario***

***Indice analitico***



# ***Benvenuti***

---

ARexx, l'omologo Amiga del linguaggio di programmazione IBM™ REXX, fornisce la possibilità di personalizzare l'ambiente di lavoro. E' particolarmente utile come linguaggio per gli script (file di comandi) che consentono di controllare e modificare le applicazioni e dirigere le modalità di interazione fra loro.

Scopo di questo manuale è presentare ARexx, indicare le modalità per la creazione di programmi ARexx e riportare l'elenco dei comandi ARexx.

**Capitolo 1. Introduzione ad ARexx:** Questo capitolo descrive le generalità di ARexx, le modalità operative su Amiga, e le caratteristiche basilari del linguaggio di programmazione.

**Capitolo 2. Predisposizioni Operative:** Questo capitolo indica le allocazioni per la memorizzazione dei programmi ARexx, le modalità esecutive per il programma ARexx e fornisce parecchi esempi di programmazione.

**Capitolo 3. Elementi di ARexx:** Questo capitolo descrive dettagliatamente le regole ed i concetti che compongono il linguaggio di programmazione ARexx.

**Capitolo 4. Istruzioni:** Questo capitolo contiene l'elenco alfabetico delle istruzioni ARexx che definiscono una operazione.

**Chapter 5. Funzioni:** Questo capitolo descrive l'uso di funzioni, che sono istruzioni di programma usate da ARexx, e fornisce l'elenco alfabetico delle funzioni ARexx.

**Capitolo 6. Diagnostica:** Questo capitolo focalizza le caratteristiche di correzione a livello sorgente usate nello sviluppo e verifica dei programmi.

**Capitolo 7. Analisi Sintattica:** Questo capitolo descrive le modalità per l'estrazione di strutture di informazioni da stringhe.

**Appendice A. Messaggi di Errore:** Questa appendice elenca i messaggi di errore ARexx.

**Appendice B. Utilità Comandi:** Questa appendice elenca i comandi ARexx eseguibili dalla Shell.

**Glossario.** Il glossario contiene i termini ARexx più comuni.

## Convenzioni grafiche

In questo manuale sono usate le convenzioni grafiche seguenti:

<b>PAROLE CHIAVE</b>	Sono indicate in maiuscolo, tuttavia gli argomenti sono inseribili sia in minuscolo che maiuscolo.
espressione	Gli argomenti richiesti sono indicati in minuscolo.
(barra verticale)	Le selezioni alternative sono separate da una barra verticale.
{ } (graffe)	Le alternative richieste sono racchiuse in parentesi graffe.
[ ] (quadre)	Le parti per le istruzioni opzionali sono racchiuse in parentesi quadre.
<n>	Le variabili sono indicate in parentesi angolari. Non introdurre le parentesi angolari nell'immissione della variabile.
Courier	Il testo indicato con il carattere Courier rappresenta l'uscita dai programmi ARexx o altre informazioni visualizzate sullo schermo.
Tasto1 + Tasto2	Sequenze di tasti indicate con il segno (+) che le collega per indicare la pressione contemporanea di tasti.

Tasto1, Tasto2	Sequenze di tasti indicate con una virgola (,) che le separa per indicare la pressione di tasti in sequenza.
Tasti Amiga	Due tasti sulla tastiera Amiga usati per funzioni speciali. Il tasto Amiga sinistro è posto a sinistra della barra spaziatrice e porta incisa la lettera A. Il tasto Amiga destro è posto a destra della barra spaziatrice con una A in corsivo.

## ***Bibliografia***

Ulteriori informazioni sull'apprendimento e l'uso di ARexx possono essere reperite nei seguenti volumi:

*Modern Programming Using REXX*, by R.P. O'Hara and D.G. Gomberg, Prentice-Hall, 1985

*The REXX Language: A Practical Approach to Programming*, by M. F. Cowlishaw, Prentice-Hall, 1985.

*Programming in REXX*, J. Ranade IBM Series, by Charles Daney.

*Using ARexx on the Amiga*, by Chris Zamara and Nick Sullivan, Abacus, 1991.

*Amiga ROM Kernel Reference Manual: Libraries*, Third Edition, Addison-Wesley, 1992.



## **Capitolo 1**

# **Introduzione ad ARexx**

---

Il linguaggio di programmazione ARexx agisce come nucleo centrale tra le applicazioni — anche quelle create da produttori terzi — con cui può scambiare dati e comandi. Per esempio, ARexx consente di istruire un programma di telecomunicazioni per contattare una BBS, scaricare i dati soldi dalla BBS, e quindi automaticamente passare i dati ad un programma di foglio elettronico per l'analisi statistica — senza alcun intervento dell'utente.

ARexx è un linguaggio interpretato che utilizza come file in ingresso file ASCII. L'interprete ARexx è il programma REXXMast, residente nel cassetto System di Workbench. REXXMast supervisiona l'esecuzione del programma ARexx. Se REXXMast riscontra un errore durante la traduzione o esecuzione di una linea, si arresta e visualizza un messaggio di errore sullo schermo. Questa verifica interattiva è sia uno strumento di apprendimento sia un aiuto per la correzione degli errori in quanto sono visualizzati sul momento e nelle posizioni in cui avvengono.

## **A chi serve ARexx?**

Non occorre che l'utente abbia grande esperienza del computer Amiga per usare i programmi e i file di comandi ARexx, ma occorre sapere come:

- Aprire una Shell ed introdurre i comandi AmigaDOS.
- Usare un programma di elaborazione testi, quali ED o MEmacs
- Creare un file User-startup

Tuttavia per modificare i file comandi o crearne dei personali, occorre avere una nozione basilare sia di Amiga Workbench sia degli ambienti AmigaDOS. Gli utenti Amiga esperti riscontreranno che ARexx è più semplice e più potente di AmigaDOS. Infatti, ARexx consente di potenziare o sostituire comandi e script AmigaDOS esistenti, ed anche creare applicazioni integrate.

## **ARexx su Amiga**

ARexx è supportato da tutte le configurazioni hardware Amiga. Partendo con la release di Amiga Workbench Versione 2.0, ARexx è stato integrato nel sistema operativo Amiga. Specificatamente, ARexx utilizza due importanti prestazioni del sistema operativo Amiga: la comunicazione multitasking e interprocesso.

Multitasking consente l'esecuzione di più di un programma. Per esempio, è possibile creare un file, formattare un disco e regolare i colori dello schermo.

La comunicazione interprocesso (IPC) consente lo scambio di informazioni fra applicazioni. La comunicazione interprocesso avviene attraverso l'uso di porte messaggi, un indirizzo contenuto in una applicazione che può ricevere ed inviare messaggi acclusi ad ogni programma. Ogni porta messaggio è definita da un nome e l'invio di un messaggio ad una applicazione richiede l'uso del nome della porta nello script ARexx.

La sequenza di eventi nell'inviare e ricevere un messaggio è la seguente:

1. All'inizializzazione l'applicazione apre la porta messaggi.
2. L'applicazione attende di ricevere un messaggio.
3. Il sistema operativo Amiga notifica all'applicazione che nella porta è presente un messaggio.
4. L'applicazione elabora un messaggio.
5. L'applicazione notifica al mittente (ARexx) che il messaggio è stato ricevuto ed elaborato.

Questo trasferimento di messaggi non è limitato ad una applicazione con ARexx, parecchie applicazioni possono scambiare messaggi usando ARexx come nucleo centrale di trasferimento. Tuttavia, tutte le applicazioni devono essere compatibili con ARexx.

## **Caratteristiche ARexx**

Le caratteristiche del linguaggio di programmazione ARexx sono:

- **Dati non definiti**— I dati sono trattati come stringhe di caratteri individuali e i valori variabili non sono dichiarati.
- **Esecuzione Interpretata** — La capacità di ARexx di leggere ed eseguire evita il passo aggiuntivo della compilazione programma.
- **Gestione Automatica Risorse** — L'allocazione automatica della memoria interna elimina testi e dati non necessari.
- **Esame, Rilevazione e Verifica** — L'esame e la rilevazione consentono il trattamento degli errori che normalmente terminano il programma. La verifica consente di visionare l'intero programma, riducendo il tempo di sviluppo.
- **Librerie Funzioni** — Le librerie di funzioni esterne forniscono le funzioni preprogrammate estese.





## **Capitolo 2**

# ***Predisposizioni operative***

---

Questo capitolo descrive le operazioni seguenti:

- Avvio di ARexx
- Salvataggio programmi
- Memorizzazione programmi
- Uso di programmi campione

## ***Avvio di ARexx***

Per avviare ARexx occorre attivare il programma REXXMAST. Il programma REXXMAST può avviarsi automaticamente o manualmente. Ogniqualvolta si avvia o si arresta ARexx appare un messaggio di testo.

## ***Avvio automatico***

L'avvio automatico di ARexx può avvenire in due modi: ponendo l'icona REXXMAST nel cassetto WBStartup o aggiornando il file S:User-Startup file.

Per porre l'icona REXXMAST nel cassetto WBStartup:

1. Aprire il cassetto System.
2. Trascinare l'icona REXXMAST sul cassetto WBStartup.
3. Riavviare Amiga.

Per aggiornare il file S:User-Startup:

1. Aprire un programma di aggiornamento testi.
2. Aprire il file S:User-Startup.

3. Digitare REXXMAST>NIL:.
4. Salvare il file.
5. Riavviare Amiga.

## **Avvio manuale**

L'avvio manuale di RexxMast può avvenire in due modi: attivare l'icona RexxMast nel Workbench o avviarla dalla Shell. Gli utenti di sistemi a dischetti possono risparmiare spazio disco avviando ARexx soltanto quando necessario.

Per avviare RexxMast da Workbench:

1. Aprire il cassetto System.
2. Attivare l'icona RexxMast.

Per avviare RexxMast dalla Shell:

1. Aprire una Shell.
2. Digitare REXXMAST >NIL: e premere invio.

## **Informazioni sui programmi ARexx**

I programmi ARexx sono generalmente memorizzati nella directory REXX: (che è generalmente assegnata alla directory SYS:S).

Quantunque i programmi possano essere memorizzati in qualsiasi directory, la memorizzazione in rexx: presenta parecchi vantaggi:

- Si può avviare il programma senza dover digitare il percorso completo.
- Tutti i programmi ARexx risiederanno nella stessa directory.
- La maggior parte delle applicazioni cercano i programmi ARexx in REXX:.

Analogamente alla memorizzazione, al programma ARexx può essere attribuito un nome qualsiasi. Tuttavia, adottando una convenzione semplice per la denominazione si ottiene una gestione programmi molto più semplice. I programmi che si avviano dalla

Shell devono avere l'estensione .rexex per distinguerli dai file generati dalle altre applicazioni.

## **Uso dei programmi ARexx**

Per usare un programma ARexx utilizzare il comando RX. Se al nome del programma è aggiunto un percorso intero, il programma viene cercato soltanto nella directory specificata. Se non è indicato il percorso, vengono controllate la directory attuale e REXX:.

Fino a quando il programma risiede nella directory REXX:, non è necessario aggiungere l'estensione .rexex quando si specifica il nome del programma. In altre parole, digitando:

```
RX Program.rexx
```

è come digitare:

```
RX Program
```

Un programma breve può essere introdotto direttamente sulla linea comandi racchiudendo la linea programma fra virgolette. Per esempio, il programma seguente invia 5 file denominati da myfile.1 a myfile.5 alla stampante.

```
RX "DO i=1 to 5;  
ADDRESS command 'copy myfile.' || i 'prt: '; END"
```

Quando una applicazione è compatibile con ARexx, è possibile usare programmi ARexx dall'interno dell'applicazione scegliendo una voce menu o specificando opzioni di comando. Per ulteriori informazioni riferirsi alla documentazione dell'applicazione.

I programmi ARexx possono essere avviati da Workbench creando una icona strumento o progetto per il programma. Occorre specificare il comando RX come programma associato nell'icona. Nella finestra Informazioni dell'icona introdurre:

```
Programma associato: SYS:Rexxc/RX
```

Quando l'icona è aperta, RX avvia RexxMast (se non è già avviato). Esegue i file associati con l'icona come programma ARexx.

ARexx accetta due tipi di parametri: Console, per specificare una finestra e CMD, per specificare una stringa comando. Introdurre

questi tipi di parametri nella finestra Informazioni dell'icona progetto nel modo seguente:

```
Console=CON:0/0/640/200/Example/Close  
CMD=rexxprogram
```

## **Esempi di programma**

Gli esempi seguenti illustrano le modalità d'uso di ARexx per visualizzare stringhe testo sullo schermo, eseguire calcoli e attivare il controllo errori.

I programmi possono essere introdotti con qualsiasi programma di gestione, testi quali ED o MEMacs, o un programma di elaborazione testi. Salvare il programma come file ASCII se si usa un programma di elaborazione testi. ARexx supporta il set di caratteri ASCII esteso (Å, Æ, ß). Questi caratteri estesi sono riconosciuti come caratteri normali di stampa e sono mappati dal minuscolo al maiuscolo.

Gli esempi illustrano anche l'uso di alcuni requisiti basilari di sintassi ARexx come:

- Linee di commento
- Regole di spaziatura
- Confronto maiuscolo/minuscolo
- Uso di virgolette ed apici

Ciascun programma ARexx comprende almeno una riga di commento che descrive il programma ed una istruzione per la visualizzazione del testo sulla console. I programmi ARexx devono sempre iniziare con una linea commento. La barra e l'asterisco iniziali (/\*) e l'asterisco e la barra finali (\*/) indicano all'interprete REXX/Master che è stato riconosciuto un programma ARexx. Se non sono presenti /\* e \*/, REXX/Master non riconosce il file come programma ARexx. Una volta iniziata l'esecuzione del programma, ARexx ignora qualsiasi linea commento aggiuntiva all'interno del file. Tuttavia, le linee commento sono estremamente utili quando si legge il programma. Esse sono di ausilio all'organizzazione e alla comprensione del programma.

Per rendere più conveniente l'uso di macro campione sperimentate con ARexx, creare uno script (file comandi) AmigaDOS speciale. Aprire il programma (per esempio ED) per creare il file S:edr ed inserirvi le 3 righe seguenti:

```
.key file  
ed rexx:<file>.rexx;  
oppure usare altri programmi diversi da ED  
rx <file>
```

Infine , impostare il bit di protezione s per questo file. Digitare nella Shell: Protect S:edr+s

Quindi, se si desidera provare un programma esempio denominato Test.rexx, digitare nella Shell: edr Test

Lo script edr invoca il programma nel punto in cui è stato digitato il testo sorgente del programma e, durante il salvataggio, edr chiama direttamente RX per avviare la nuova macro ARexx.

## ***Amiga.rexx***

Questo programma indica le modalità d'uso di SAY in un gruppo di istruzioni per visualizzare stringhe testo sullo schermo. Le istruzioni sono dichiarazioni che denotano l'operazione da eseguire. Ogni dichiarazione incomincia sempre con un simbolo. Nell'esempio seguente, il simbolo è SAY. (Durante l'esecuzione del programma i simboli sono sempre tradotti in maiuscolo.) SAY è seguito da una stringa di esempio. La stringa è un insieme di caratteri delimitato da una coppia di apici (') o di virgolette (").

### ***Programma 1. Amiga.rexx***

```
/*A simple program*/  
SAY 'Amiga. The Computer For the Creative Mind.'
```

Introdurre il programma precedente e salvarlo come REXX:Amiga.rexx. Per avviare il programma, aprire la finestra Shell e digitare:

```
RX Amiga
```

Anche se il percorso completo e il nome programma è REXX:Amiga.rexx, non è necessario digitare il nome della directory

REXX: o l'estensione .rexx se il programma è salvato nella directory .REXX:.

Nella finestra Shell appare il testo seguente:

Amiga. The Computer for the Creative Mind.

## ***Età.rexx***

Questo programma visualizza una richiesta per introdurre e leggere le informazioni introdotte.

### ***Programma 2. Età.rexx***

```
/*Calcola l'età in giorni*/  
SAY 'Inserire la vostra età:'  
PULL age  
SAY 'Avete circa' age*365 'giorni.'
```

Salvare questo programma come REXX:età.rexx ed avviarlo con il comando:

RX età

Questo programma inizia con una linea commento che ne descrive l'esecuzione. Tutti i programmi ARexx iniziano con un commento. L'istruzione SAY visualizza una richiesta per l'ingresso dati nella console.

L'istruzione PULL legge una linea di dati introdotta dall'utente, che in questo caso è l'età dell'utente. PULL prende l'ingresso, lo converte in maiuscolo e lo memorizza in una variabile. Le variabili sono simboli cui possono essere assegnati valori. Scegliere nomi descrittivi per le variabili. Questo esempio utilizza il nome variabile "age" per tenere il numero introdotto.

Le linee finali moltiplicano la variabile "age" per 365 e indicano all'istruzione SAY di visualizzare il risultato. La variabile "age" non deve essere dichiarata come numero perché il suo valore era stato controllato durante l'uso nell'espressione. Questo è un esempio di dato non definito. Per capire cosa sarebbe successo se age non fosse stato un numero, provare ad eseguire nuovamente il programma con un ingresso non numerico per age. Il messaggio di errore che appare indica il numero linea ed il tipo di errore avvenuto.

## **Calc.rexx**

Questo programma introduce l'istruzione DO, che ripete l'esecuzione del programma. Illustra inoltre l'operatore esponente (\*\*). Introdurre questo programma e salvarlo come REXX:Calc.rexx. Per avviare il programma, usare il comando "RX calc".

### **Programma 3. Calc.rexx**

```
/*Calcola le potenze alla 2 e alla 3.*/  
DO i = 1 to 10      /*Inizio ciclo - 10 interazioni*/  
    SAY i i**2 i**3  /*Fa i calcoli*/  
END                /*Fine del ciclo*/  
SAY 'Fatto!'
```

L'istruzione DO esegue ripetutamente le istruzioni tra DO e END. La variabile "i" è la variabile indice per il ciclo ed è incrementata di 1 per ogni iterazione (ripetizione). Il numero che segue il simbolo TO è il limite per l'istruzione DO e può essere una variabile o una espressione completa anziché la costante 10.

Generalmente, i programmi ARexx utilizzano la spaziatura singola fra i caratteri alfanumerici. (Non inserire uno spazio dopo ogni lettera!) Tuttavia, nel programma 3 la spaziatura è ravvicinata tra i caratteri esponente (\*\*) e le variabili e costanti (i, e 2, i e 3).

Le istruzioni entro il ciclo sono indentate. Ciò non è richiesto dal linguaggio ma rende il programma più leggibile, perché si visualizza l'inizio e la fine del ciclo.

## **Even.rexx**

L'istruzione IF consente l'esecuzione condizionata delle istruzioni. In questo esempio, i numeri da 1 a 10 sono classificati come dispari o pari dividendoli per due e quindi controllando il resto. L'operatore aritmetico // calcola il resto dopo l'operazione di divisione.

**Programma 4. Even.rexx**

```

/*Pari o dispari?*/
DO i = 1 to 10 /*Inizio ciclo - 10 interazioni*/
    IF i // 2 = 0 THEN type = 'pari'
    ELSE type = 'dispari'
    SAY i 'è' type
END
/*Fine ciclo*/

```

Se la linea IF dichiara che il resto della divisione della variabile "i" divisa per 2 è uguale a 0, allora imposta la variabile "type" a pari. Se il resto non è 0, il programma salta il punto decisionale THEN ed esegue il punto decisionale ELSE, impostando la variabile "type" a dispari.

**Square.rexx**

Questo esempio introduce il concetto di una funzione, un gruppo di dichiarazioni eseguite menzionando il nome funzione in un contesto appropriato. Le funzioni consentono di costruire grandi programmi complessi da moduli più piccoli. Le funzioni inoltre permettono lo stesso codice per operazioni analoghe in un programma differente.

In un'espressione le funzioni sono specificate come nome seguito da una parentesi aperta. (Non vi è lo spazio tra il nome e la parentesi.) La parentesi può essere seguita da una o più espressioni, denominate argomenti. L'ultimo argomento deve essere seguito da una parentesi chiusa. Questi argomenti passano le informazioni alla funzione per l'elaborazione.

**Programma 5. Square.rexx**

```

/*Definizione e richiamo di una funzione.*/
DO i = 1 to 5
    SAY i square(i) /*Richiama la funzione "square"*/
END
EXIT
square:
    /*Nome funzione*/
    ARG x
    /*Prende l'argomento*/
    RETURN x**2 /*Lo eleva al quadrato e ritorna*/

```

Iniziando con DO e terminando con END, il ciclo è impostato con una variabile indice "i", che si incrementa di 1, il ciclo si ripete 5 volte. Il ciclo contiene un'espressione che chiama la funzione



"square" quando è valutata l'espressione. Il risultato della funzione è visualizzato mediante l'istruzione SAY.

La funzione "square" è definita dalle istruzioni ARG e RETURN che calcola i valori quadrati. ARG reperisce il valore della stringa argomento "i" e RETURN passa il risultato della funzione nuovamente all'istruzione SAY

Una volta che la funzione è chiamata dal ciclo, il programma cerca il nome funzione "square:", reperisce l'argomento "i", esegue il calcolo, e ritorna alla linea con il ciclo DO/END. L'istruzione EXIT termina il programma dopo il ciclo finale.

## **Results.rexx**

L'istruzione TRACE attiva la prestazione controllo errori di ARexx.

### **Programma 6. Results.rexx**

```
/*Dimostra "risultati" di trace*/
TRACE results
sum = 0 ; sumsq = 0;
DO i = 1 to 5
    sum = sum + 1
    sumsq = sumsq + i**2
END
SAY 'sum=' sum 'sumsq=' sumsq
```

La console visualizza le linee sorgenti eseguite, ad ogni passata del ciclo DO/END, e i risultati finali dell'espressione. Togliendo l'istruzione TRACE, si visualizza soltanto il risultato finale sum = 15 sumsq = 55.

## **Grades.rexx**

Questo programma calcola il grado finale per un dato studente basato su quattro gradi di saggi e il grado di partecipazione della classe. La media del saggio 1 e del saggio 2 è valutata 30%, la media del saggio 3 e del saggio 4 è valutata 45%, e la partecipazione è valutata il 25% del livello finale.

Viene presentata l'opzione per continuare con un altro calcolo. La risposta è convalidata e se non uguaglia Q (abbandona), il ciclo continua. Se la risposta uguaglia Q, il programma abbandona il ciclo ed esce.

**Programma 7. Grades.rexx**

```
/*Programma di votazione*/
SAY "Salve, calcolerò la votazione per voi."
response = 0
DO while response ~ = "Q" /*Cicla se la risposta non è Q*/
  SAY "Inserire i voti dello studente."
  SAY "Esame 1:"
  PULL es1
  SAY "Esame 2:"
  PULL es2
  SAY "Esame 3:"
  PULL es3
  SAY "Esame 4:"
  PULL es4
  SAY "Partecipazioni:"
  PULL p
  Final = (((es1 + es2)/2*.3) + ((es3 + es4)/2*.45) + (p*.25))
  SAY "Il voto finale per questo studente è " Final
  SAY "Volete continuare? (Q per finire)."
  PULL response
END
EXIT
```

## **Capitolo 3**

# **Elementi di ARexx**

---

Questo Capitolo introduce le regole ed i concetti che costituiscono il linguaggio di programmazione ARexx e descrive le modalità di interpretazione di ARexx per i caratteri e le parole usati nei programmi.. Gli elementi descritti sono i seguenti:

- Indicatore (token) — l'elemento più piccolo del linguaggio ARexx
- Clausola — l'unità eseguibile più piccola, analoga ad una frase
- Espressione — un gruppo di indicatori valutati
- Interfaccia di Comando — il processo con cui i programmi ARexx comunicano con le applicazioni compatibili ARexx.

Questo Capitolo comprende anche la descrizione dell'ambiente esecutivo ARexx. Questa descrizione è destinata agli utenti Amiga avanzati e comprende dettagli tecnici sulla comunicazione fra processi.

## **Indicatori**

L'indicatori, l'entità più piccola distinta del linguaggio ARexx può essere un singolo carattere o una serie di caratteri. Vi sono cinque categorie di indicatori:

- commenti
- simboli
- stringhe
- operatori
- caratteri speciali

## Commenti

Il commento è un gruppo di caratteri che inizia con la sequenza `/*` (barra asterisco) e termina con `*/` (asterisco barra). Ogni programma ARexx deve iniziare con un commento. Ogni `/*` deve avere il corrispondente `*/`. Per esempio:

```
/*Questo è un commento ARexx*/
```

I commenti possono essere posti in qualsiasi punto del programma e possono anche essere collegati l'un l'altro. Per esempio:

```
/*A /*un commento*/ misto*/
```

Inserire i commenti lungo il programma. I commenti sono promemoria per l'autore e per i lettori, descriventi le intenzioni del programma. Poiché l'interprete ignora i commenti quando scandisce il programma, i commenti non ne rallentano l'esecuzione.

## Simboli

Il simbolo è un gruppo di caratteri a-z, A-Z, 0-9, e il punto (`.`), punto esclamativo (`!`), punto interrogativo (`?`), il segno dollaro (`$`), e il sottolineato (`_`). Durante la scansione del programma l'interprete traduce i simboli in maiuscolo, perciò il simbolo `MyName` è equivalente a `MYNAME`. I quattro tipi di simboli riconosciuti sono:

<b>Simboli fissi</b>	Una serie di caratteri numerici che iniziano con una cifra (0-9) o un punto ( <code>.</code> ). Il valore di un simbolo fisso è sempre il nome del simbolo stesso, tradotto in maiuscolo 12345 è un esempio di simbolo fisso.
<b>Simboli semplici</b>	Una serie di caratteri alfabetici che iniziano con una lettera A-Z. "MyName" è un esempio di simbolo semplice.
<b>Simboli radice</b>	Una serie di caratteri alfanumerici che terminano con un punto. "A." e "Radice9." sono simboli radice.
<b>Simboli composti</b>	Una serie di caratteri alfanumerici che comprendono uno o più punti all'interno dei caratteri. "A.1.Index" è un esempio di simbolo composto.

I simboli semplici, radice e composti sono denominati variabili e possono essere assegnati a valori nel corso dell'esecuzione del programma. Se alla variabile non è stato assegnato un valore, la variabile non viene inizializzata. Il valore per una variabile non inizializzata è il nome della variabile stessa (tradotto in maiuscolo, se consentito).

I simboli radice e composti hanno proprietà speciali che li rendono vantaggiosi per la costruzione di matrici e liste. I simboli radice forniscono il metodo per reinizializzare una classe completa di simboli composti. Il simbolo composto può essere considerato come costituito dalla struttura  $\text{stem.n}_1.\text{n}_2 \dots \text{n}_k$ , dove il nome iniziale è un simbolo radice e ogni nodo,  $\text{n}_1 \dots \text{n}_k$ , è un simbolo fisso o semplice.

Quando l'assegnazione è riferita ad un simbolo radice, assegna quel valore a tutti i simboli composti possibili derivati dalla radice. Così, il valore di un simbolo composto dipende dalle assegnazioni precedenti fatte a sé stesso o alle sue radici associate.

In un programma ogniqualevolta appare un simbolo composto, ne viene espanso il nome sostituendo ogni nodo con il suo valore attuale. La stringa valore può consistere di caratteri, compresi gli spazi interposti, e non viene convertita al maiuscolo. Il risultato dell'espansione è un nuovo nome utilizzato in luogo del simbolo composto. Per esempio, se J ha il valore 3 e K ha il valore 7, il simbolo composto A.J.K sarà espanso a A.3.7.

I simboli composti possono essere considerati come una forma di memoria associativa o a contenuto indirizzabile. Per esempio, si supponga di dover memorizzare e reperire un gruppo di nomi e numeri telefonici. L'approccio convenzionale è impostare due matrici, NOME e NUMERO, ognuno indicizzato da un intero partendo da uno al numero di elementi. Il numero viene ricercato scandendo la matrice dei nomi sino al rilevamento del nome dato, cioè in NOME.12, e quindi ricerca NUMERO.12. Con i simboli composti, il simbolo NAME può contenere il nome da ricercare, e NUMERO.NOME espande quindi il numero corrispondente, per esempio NUMERO.CBM.

I simboli composti possono anche essere usati come matrici indicizzate convenzionali, con il vantaggio aggiunto che è richiesta

soltanto una singola assegnazione (alla radice) per inizializzare l'intera matrice.

Per esempio, il programma seguente utilizza le radici "number." e "addr." per creare una guida telefonica computerizzata.

### **Programma 8. Phone.rexx**

```
/*Guida telefonica con uso di variabili complesse.*/
IF ARG() ~ = 1 THEN DO
  SAY "USO: rx phone nome"
  EXIT 5
END
/*Apertura finestra visualizzazione dati.*/
CALL OPEN out, "con:0/0/640/60/ARexx guida telefonica"
IF ~ result THEN DO
  SAY "Aperatura fallita ... spiacemente"
  EXIT 10
END
/*Definizione numeri*/
number. = '(non trovato)'
number.wsh = '(555) 001-0001'
addr. = '(non trovato)'
number.CBM = '(555) 002-0002'
addr.CBM = '1200 Wilson Dr., West Chester, PA, 19380'

/*(Il lavoro è fatto qui)*/
ARG name /*Il nome*/
CALL WRITELN out,name || "'il numero è" number.name
CALL WRITELN out,name || "'l'indirizzo è" addr.name
CALL WRITELN out, "Premi invio per uscire."
CALL READLN out
EXIT
```

Per eseguire il programma, attivare una finestra Shell e digitare:

```
RX Phone cbm
```

Su una finestra appare quindi il nome e l'indirizzo assegnato a CBM.

## Stringhe

Una stringa è un qualsiasi gruppo di caratteri che inizia e termina con un delimitatore apice (') o virgolette ("). Alle due estremità della stringa deve essere usato lo stesso delimitatore. Per comprendere il carattere delimitatore nella stringa, usare una sequenza con doppio delimitatore (' ' o " "). Per esempio:

```
"È l'ora giusta."  
'Non puoi vederlo?'
```

Il valore di una stringa è la stringa stessa. Il numero di caratteri definisce la lunghezza stringa. Se la stringa non contiene caratteri, è denominata stringa nulla.

Le stringhe che sono seguite dal carattere X o B sono definite rispettivamente come esadecimali o binarie, e devono essere composte di cifre esadecimali (0-9, A-F) o cifre binarie (0,1). Per esempio:

```
'4A 3B C0'X  
'00110111'B
```

Gli spazi sono ammessi nelle transizioni byte per migliorare la leggibilità. Le stringhe esadecimali o binarie sono utili per specificare caratteri non ASCII e informazioni specifiche di macchina, come gli indirizzi. Sono immediatamente convertiti nel formato interno di pacchetto (compressi).

## Operatori

Gli operatori sono una combinazione dei caratteri seguenti: ~ + - \* / = > < & | ^, come descritto in questo paragrafo. Sono presenti quattro tipi di operatore:

- Operatori aritmetici che richiedono uno o due operandi numerici e producono un risultato numerico.
- Operatori di concatenazione che congiungono due stringhe in una stringa singola.

- Operatori di comparazione che richiedono due operandi per produrre un risultato booleano (0 o 1).
- Operatori logici che richiedono uno o due operandi booleani e producono un risultato booleano.

Ad ogni operatore è associata una priorità che determina l'ordine in cui le operazioni vengono eseguite nelle espressioni. Gli operatori con priorità più alta (8) sono eseguiti prima di quelli a priorità più bassa (1).

### ***Operatori aritmetici***

Una classe importante di operandi sono quelli rappresentanti numeri. I numeri consistono di caratteri 0-9, il punto (.), il segno più (+), il segno meno (-), e gli spazi. Per indicare notazioni esponenziali, un numero può essere seguito da una "e" o "E" e da un intero (segnato).

Per specificare i numeri possono essere usati sia le stringhe sia i simboli. Poiché il linguaggio è senza definizione dati, le variabili non devono essere dichiarate come numeri prima di usare l'operazione aritmetica. Invece, ogni stringa valore è esaminata al momento dell'uso per verificare che rappresenti un numero. Gli esempi seguenti sono tutti numeri validi:

```
33
" 12.3 "
0.321e12
' + 15. '
```

Sono permessi gli spazi iniziali e finali. Gli spazi possono essere interposti tra un (+) o un (-) ma non all'interno del numero stesso.

È possibile modificare la precisione di base utilizzata per i calcoli aritmetici durante l'esecuzione del programma. Il numero di cifre significative usate nelle operazioni aritmetiche è determinato dall'impostazione di **NUMERIC Digits** e può essere modificata usando l'istruzione **NUMERIC** descritta nel Capitolo 4.

Il numero di posti decimali usati per un risultato dipende dall'operazione e dal numero di posti decimali negli operandi. **ARexx** preserva gli zeri finali per indicare la precisione del risultato. Se il numero totale di cifre richiesto per esprimere un valore eccede l'imposta-



zione attuale, il numero viene formattato in notazione esponenziale. Essi sono:

- Notazione scientifica — l'esponente è regolato affinché sia posta una cifra singola alla sinistra del punto decimale.
- Notazione tecnica — il numero è scalato affinché l'esponente sia un multiplo di 3 e le cifre a sinistra del punto decimale hanno una gamma compresa fra 1 e 999.

La Tabella 3-1 indica gli operatori aritmetici.

**Tabella 3-1. Operatori Aritmetici**

Operatore	Priorità	Esempio	Risultato
<b>+</b> (conversione prefisso)	8	'3.12'	3.12
<b>-</b> (negazione prefisso)	8	-"3.12"	-3.12
<b>**</b> (elevazione a potenza)	7	0.5 ** 3	0.125
<b>*</b> (moltiplicazione)	6	1.5*1.50	2.250
<b>/</b> (divisione)	6	6 / 3	2
<b>%</b> (divisione intero)	6	-8 % 3	-2
<b>//</b> (resto)	6	5.1//0.2	0.1
<b>+</b> (addizione)	5	3.1+4.05	7.15
<b>-</b> (sottrazione)	5	5.55 - 1	4.55

### **Operatori di concatenazione**

ARexx definisce due operatori di concatenazione. Il primo identificato dalla sequenza operatore **||** (due barrette verticali), congiunge due stringhe in una stringa singola senza alcun spazio. Questo tipo di concatenazione può anche essere specificato in modo implicito. Quando un simbolo ed una stringa sono introdotti senza spazi, ARexx si comporta come se l'operatore **||** fosse specificato. La seconda operazione di concatenazione è identificata dall'operatore **spazio** e congiunge le due stringhe operando con l'intervento di uno spazio.

La priorità di tutte le operazioni di concatenazione è 4. La tabella 3-2 riepiloga le differenti operazioni.

Tabella 3-2. Operatori di Concatenazione

Operatore	Operazione	Esempio	Risultato
<b>  </b>	Concatenazione	'perchè io, 'll'mamma?'	perchè io, mamma?
<b>spazio</b>	Concatenazione con Spazio	'buona' 'fortuna'	buona fortuna
<b>nessuno</b>	Concatenazione Implicita	uno'due'tre	UNOdueTRE

### Operatori di comparazione

ARexx supporta tre tipi di comparazioni

- Comparazioni esatte — vengono effettuate carattere per carattere
- Comparazioni stringa — ignorano gli spazi iniziali e aggiungono spazi alla stringa più corta.
- Comparazioni numeriche — convertono gli operandi in forma numerica interna usando l'impostazione di precisione corrente, quindi viene eseguita una comparazione aritmentica.

Le comparazioni risultano sempre in un valore booleano. I numeri 0 e 1 sono usati per rappresentare i valori booleani falso e vero. L'uso di un valore anziché 0 o 1 quando si attende un operando booleano genera un errore. Qualsiasi numero equivalente a 0 o 1, per esempio 0.000 o 0.1E1, è anche accettato come valore booleano.

Esclusi gli operatori di uguaglianza esatta(==) e di inuguaglianza esatta (~==) tutti gli operatori di comparazione determinano dinamicamente se una comparazione stringa o una comparazione numerica deve essere effettuata. Viene effettuata una comparazione numerica se entrambi gli operandi sono numeri validi. Altrimenti, gli operandi sono comparati come stringhe.

Tutte le comparazioni hanno priorità 3. La Tabella 3-3 elenca gli operatori di comparazione accettabili.

Tabella 3-3. Operatori di Comparazione

Operatore	Operazione	Modo
==	Uguaglianza esatta	Esatto
~=	Ineguaglianza esatta	Esatto
=	Uguaglianza	Stringa/Numerico
~=	Ineguaglianza	Stringa/Numerico
>	Maggiore di	Stringa/Numerico
>= o ~<	Maggiore di o Uguale a	Stringa/Numerico
<	Minore di	Stringa/Numerico
<= o ~>	Minore di o Uguale a	Stringa/Numerico

### Operatori logici (booleani)

ARexx definisce le quattro operazioni logiche NOT, AND, OR e OR Esclusivo, le quali richiedono tutte operandi booleani e producono un risultato booleano. Un tentativo di eseguire una operazione logica su un operando non booleano produce un errore. La Tabella 3-4 indica gli operatori logici accettabili.

Tabella 3-4. Operatori Logici

Operatore	Priorità	Operazione
~	8	NOT (Inversione)
&	2	AND
	1	OR
^ o &&	1	OR Esclusivo

### Caratteri speciali

In ARexx alcuni caratteri di interpunzione hanno significati speciali come indicato nella tabella 3-5.

Tabella 3-5. Caratteri Speciali

Carattere Speciale	Definizione
(:) Due punti	Il due punti definisce una etichetta quando preceduto da un indicatore simbolo (qualsiasi carattere alfanumerico o. ! ? \$).
( ) Parentesi	Le parentesi sono usate per raggruppare operatori e operandi in sotto espressioni per eludere le normali priorità operatore. Una parentesi aperta serve anche per indicare una chiamata funzione entro un'espressione. Un simbolo o stringa seguito immediatamente da una parentesi aperta definisce un nome funzione. All'interno dell'istruzione la parentesi deve sempre essere chiusa.
(;) Punto e virgola	Il punto e virgola agisce come operatore di istruzione. Se vi sono brevi dichiarazioni che stanno su una riga, separarle con punto e virgola.
(,) Virgola	La virgola agisce come carattere a fine riga di continuazione per le istruzioni interrotte a causa della fine della linea e come separatore di espressioni di argomento in una chiamata funzione.

## Clausole

La clausola è l'unità di linguaggio più piccola che può essere eseguita come istruzione, le clausole sono formate da raggruppamenti di indicatori.

Durante la lettura del programma, l'interprete del linguaggio divide il programma in gruppi di clausole. Questi gruppi di una o più clausole sono quindi divisi in indicatori ed ogni clausola è classificata come tipo particolare. Analogamente piccole differenze sintattiche possono cambiare completamente il contenuto semantico di una istruzione. Per esempio:

SAY 'Ciao, Bill'

è una clausola istruzione e visualizza "Ciao, Bill" sulla console, ma:  
`'SAY 'Ciao, Bill'`

è un clausola comando, e produce "SAY Ciao, Bill" come comando per un programma esterno. La presenza della stringa nulla iniziale (') modifica la classificazione da una clausola istruzione ad una clausola comando.

La fine di una linea normalmente agisce come la fine implicita di una clausola. Una clausola può essere continuata sulla linea successiva terminando la linea con una virgola. La virgola è ignorata dal programma, e la linea successiva è considerata come continuazione della clausola. Non vi è limite al numero di continuazioni che possono avvenire (ad eccezione dei limiti imposti dal buffer del comando).

Gli indicatori stringa e commenti sono continuati automaticamente se la linea termina prima del rilevamento del delimitatore di chiusura e il carattere della nuova linea (ritorno carrello) non è considerato appartenente all'indicatori.

## ***Clausole nulle***

Le clausole sono linee composte di spazi o commenti sparse nel programma. Non hanno alcuna funzione nell'esecuzione del programma, ad eccezione della migliore leggibilità e dell'incremento del conteggio linee.

## ***Clausole etichetta***

La clausola etichetta è un simbolo seguito da due punti (:). Un'etichetta agisce come segnale di posizione nel programma e non produce alcun altro effetto. Il due punti è considerato come terminatore implicito di clausola, perciò ogni etichetta diventa una clausola separata. Le clausole etichetta possono apparire in qualsiasi punto del programma. Per esempio:

```
start:      /*Inizia esecuzione*/  
syntax:    /*Errore di processo*/
```

## **Clausole assegnazione**

Le clausole assegnazione sono identificate da un simbolo di variabile seguito dall'operatore =. (In questo contesto la definizione normale dell'operatore = di comparazione di uguaglianza è eluso.) Gli indicatori alla destra di = sono elaborati come espressione e il risultato è assegnato alla variabile. Per esempio:

```
quando = 'Ora è il momento giusto'
risposta = 3.14 * fact(5)
```

Il segno uguale (=) assegna il valore 'Ora è il momento giusto' alla variabile 'quando', e assegna il risultato di  $3,14 * \text{fact}(5)$  alla variabile 'risposta'.

## **Clausole istruzione**

Le clausole istruzione iniziano con il nome dell'istruzione ed indicano ad ARexx di eseguire un'operazione. I nomi istruzione sono descritti nel Capitolo 4. Per esempio:

```
DROP a b c
SAY 'prego'
IF j > 5 THEN LEAVE;
```

## **Clausole comando**

Le clausole comando sono espressioni ARexx che non possono essere classificate nei precedenti tipi di clausole. Viene elaborata l'espressione ed il risultato è emesso come comando per un ospite esterno. Per esempio:

```
'delete' 'myfile' /*Comando AmigaDOS*/
'jump' current+10 /*Comando di editor*/
```

Il comando delete non è riconosciuto come comando ARexx, perciò viene inviato all'ospite esterno, in questo caso AmigaDOS. Nel secondo esempio, si presume, che il comando Jump (Salta a ) sia capito da un elaboratore di testi esterno.

## **Espressioni**

Le espressioni sono un gruppo di indicatori elaborati. La maggior parte delle elaborazioni contengono almeno una espressione. Le espressioni sono composte da:

- **Stringhe** — Il valore di una stringa è il valore della stringa stessa.
- **Simboli** — Il valore di un simbolo fisso è il simbolo fisso stesso, tradotto in maiuscolo. I simboli possono essere usati come variabili ed avere un valore assegnato.
- **Operatori** — Gli operatori hanno un ordine di priorità che ne determina il momento di esecuzione.
- **Parentesi** — Le parentesi possono essere usate per modificare l'ordine normale di elaborazione nell'espressione o per identificare le chiamate funzione. Un simbolo o una stringa seguiti immediatamente da una parentesi aperta definiscono il nome della funzione mentre gli indicatori tra le parentesi aperte e chiuse, formano la lista degli argomenti per la funzione. Per esempio, l'espressione:

```
J 'fattoriale è' fact(J)
```

è composta da:

- un simbolo — J
- un operatore spazio
- una stringa — il fattoriale è
- un altro spazio
- un simbolo — fact
- una parentesi aperta
- un simbolo — J
- una parentesi chiusa

In questo esempio, FACT è un nome funzione e (J) è la lista argomenti, l'espressione singola J.

Prima di procedere all'elaborazione di una espressione, ARexx deve ottenere un valore per ogni simbolo dell'espressione. Per i simboli fissi il valore è il nome simbolo stesso, ma i simboli di variabili devono essere letti nella tabella attuale dei simboli. Nell'esempio

precedente se al simbolo J era assegnato il valore 3, l'espressione dopo la risoluzione del simbolo diventa:

```
3 'fattoriale è' FACT(3)
```

Per evitare ambiguità nei valori assegnati ai simboli durante l'elaborazione, ARexx garantisce un rigoroso ordine di risoluzione da sinistra a destra. La risoluzione del simbolo procede indipendentemente dalla priorità dell'operatore o dal raggruppamento parentetico. Se viene rilevata una chiamata funzione, la risoluzione è sospesa durante l'elaborazione della funzione. Nell'espressione uno stesso simbolo può avere diversi valori.

Se l'esempio precedente fosse strutturato nel modo seguente:

```
FACT(J) 'è' J 'fattoriale'
```

la seconda ricorrenza del simbolo J si risolverebbe in 3?

Generalmente, le chiamate funzione possono avere effetti collaterali e modificare i valori delle variabili. Se l'esempio fosse strutturato, il valore di J potrebbe essere cambiato dalla chiamata in FACT.

Dopo la risoluzione di tutti i valori dei simboli, l'espressione è elaborata osservando la priorità degli operatori ed il raggruppamento di sottoespressione. ARexx non garantisce l'ordine di elaborazione tra operatori aventi stessa priorità e non impiega l'elaborazione "percorso veloce" delle operazioni booleane. Per esempio, nell'espressione:

```
(1 = 2) & (FACT(3) = 6)
```

la chiamata della funzione FACT viene eseguita anche se il primo termine dell'operazione AND (&) è 0. Questo esempio evidenzia che ARexx continua la lettura da sinistra a destra anche se l'esempio dato è falso e restituisce il valore 0.



## ***Interfaccia di comando***

L'interfaccia di comando ARexx è una porta messaggi pubblica. Le applicazioni compatibili ARexx devono avere questa porta messaggi. I programmi ARexx emettono comandi ponendo stringhe comando in un pacchetto messaggi e invia il pacchetto alla porta messaggi dell'ospite. Il programma sospende il funzionamento mentre l'ospite elabora i comandi e riprende il funzionamento quando ritorna il pacchetto messaggi di risposta.

### ***Indirizzo ospite***

Arexx contiene due indirizzi ospite impliciti, un valore corrente ed un valore precedente, come parte dell'ambiente di memorizzazione del programma. Questi valori possono essere modificati in qualsiasi momento usando l'espressione ADDRESS (o un sinonimo, SHELL). È possibile interrogare l'indirizzo ospite attuale con la funzione integrata ADDRESS(). La stringa indirizzo ospite predefinita è REXX, ma questa può essere elusa quando si chiama il programma. Molte applicazioni ospite forniscono il nome della porta pubblica quando chiamano un programma macro, affinché la macro possa inviare automaticamente i comandi all'ospite.

Viene riconosciuto un indirizzo ospite speciale. La stringa COMMAND indica che la macro deve essere inviata direttamente ad AmigaDOS. È presunto che tutti gli altri indirizzi ospite si riferiscano ad una porta messaggi pubblici. Il tentativo di inviare un comando ad una porta messaggi non esistenti genera un errore di sintassi "Ambiente ospite non trovato".

Il programma 9 indica l'interazione tra ARexx e il programma di creazione testi AmigaDOS ED. Il programma verifica se ED è in funzione, determina il nome della porta messaggi ed imposta alcune variabili radice.

**Programma 9. ED-status.rexx**

```

/*Stampa lo stato di ED. ED deve essere avviato prima
di eseguire questo programma. Le porte di ED sono
chiamate 'Ed', 'Ed_1', 'Ed_2', ecc. */
DEFAULT_ED = "Ed" /*Questo nome è sensibile alle
maiuscole*/
/*Procedura da seguire se ED non è attivo, o se un
secondo ED è attivo.*/
DO WHILE ~ SHOW('p',DEFAULT_ED) /*Controlla le
porte ARExx*/
SAY "Non esiste la porta" DEFAULT_ED
SAY "Porte ARExx disponibili:"
SAY SHOW('P') '0a'X
SAY "Inserire un nome di porta differente, o FINE per
terminare"
/* Scelta porta ARExx diversa se disponibile */
DEFAULT_ED = READLN(stdin)
IF STRIP(UPPER(DEFAULT_ED)) = 'FINE' then exit 10
/*Fine programma*/
END
SAY "Using ED port" DEFAULT_ED
/*Quando la porta è identificata, ARExx la indirizza
direttamente.*/
ADDRESS VALUE DEFAULT_ED
/*Inizializzazione variabili radice utili*/
STEM.0 = 15 /*Numero di variabili ARExx ED*/
STEM.1 = 'LEFT' /*Margine sinistro (SL)*/
STEM.2 = 'RIGHT' /*Margine destro (SR)*/
STEM.3 = 'TABSTOP' /*Impostazione Tab stop (ST)*/
STEM.4 = 'LMAX' /*Massimo Linee su schermo*/
STEM.5 = 'WIDTH' /*Larghezza schermo in caratteri*/
STEM.6 = 'X' /*Posizione X dello schermo da 1*/
STEM.7 = 'Y' /*Posizione Y dello schermo da 1*/
STEM.8 = 'BASE' /*Finestra base*/
/*La base è 0 a meno che lo schermo non sia
posizionato verso destra)*/
STEM.9 = 'EXTEND' /*Valore margine esteso (EX)*/
STEM.10 = 'FORCECASE' /*Flag confronto maiuscole*/
STEM.11 = 'LINE' /*Numero linea attuale*/
STEM.12 = 'FILENAME' /*File da modificare*/
STEM.13 = 'CURRENT' /*Testo linea attuale*/
STEM.14 = 'LASTCMD' /*Ultimo comando esteso*/
STEM.15 = 'SEARCH' /*Ultimo testo cercato*/
/*Chiede ad ED di inserire i valori nelle variabili
radice 'STEM.'*/
'RV' '/STEM/' /*RV è un comando di ED usato per
mandare dati ad ARExx*/

```

```
/*STEM.1 è LEFT, e STEM.LEFT ora contiene un valore  
dato da ED. Esiste un modo per stampare questa  
informazione.*/
```

```
DO i = 1 to STEM.0  
MD_VAR = STEM.i  
SAY STEM.i "=" STEM.ED_VAR /*Stampa una  
variabile/valore di ED*/  
END
```

## ***Creazione di una macro***

ARexx consente di scrivere programmi per qualsiasi applicazione ospite che comprenda l'interfaccia comando compatibile. Alcuni programmi applicativi sono disegnati con linguaggio macro incluso e possono comprendere molti comandi macro predefiniti.

Controllare la presenza di comandi "scorciatoia" nel programma macro. Alcuni programmi possono comprendere potenti funzioni implementate specificatamente per usarle in programmi macro.

L'interpretazione dei comandi ricevuti dipende completamente dall'applicazione ospite. Nel caso più semplice, le stringhe di comando corrispondono esattamente ai comandi che potrebbero essere introdotti direttamente dall'utente. Per esempio, i comandi di controllo posizione (su/giù) per un programma di creazione testi avrà probabilmente interpretazioni identiche. Altri comandi possono essere validi soltanto quando emessi da un programma macro. Probabilmente un programma che simula una operazione da menu non è introdotto dall'utente. Nel Programma 10, il programma ARexx è chiamato da ED per scambiare due caratteri di posto.

**Programma 10. Transpose.rexx**

```

/*Data la stringa '123', se il cursore è sul 3, la
macro la converte in '213'.*/

HOST = ADDRESS()      /*Trova quale ED richiama ARExx*/
ADDRESS VALUE HOST     /*. . . e si connette.*/
'rv' '/CURR/'          /*ED ha i dati nella radice CURR*/

/*Richiede le informazioni necessarie:*/
currpos = CURR.X /*Posizione del cursore sulla
linea*/
currlin = CURR.CURRENT /*Contenuto della linea
attuale*/

IF (currpos >2) /*Elaborazione con posizione giusta*/
THEN currpos = currpos - 1
ELSE DO              /*Riporta un errore ed esce*/
'sm //Il cursore deve essere alla pos. 3 o più a
desra/'
EXIT 10
END

/*Scambia i caratteri di CURRPOS e CURRPOS-1 e
sostituisce la linea attuale con quella modificata*/
DROP CURR. /* La variabile STEM CURR non è più
richiesta; libera un po' di memoria*/

'd' /*Dice ad ED di cancellare la linea attuale*/
currlin = swapch (currpos,currlin) /*Swap 2 chars*/
'i /'||currlin||'/' /*Inserisce la linea
modificata*/
DO i = 1 to currpos /*Porta il cursore in posizione*/
'cr' /*Comando ED 'cursore a destra'*/
END
EXIT /*FINITO*/
/*Funzione per scambiare due caratteri*/
swapch: procedure
PARSE ARG cpos,clin
chl = substr (clin, cpos, 1) /*Prende carattere*/
clin = delstr (clin, cpos, 1) /*Cancella dalla
stringa*/
clin = insert (chl,clin,cpos-2,1) /*Inserisce
scambiando*/
RETURN clin /*Ritorna la stringa modificata*/

```

Per eseguire questo esempio da ED, premere ESC quindi digitare:

```
Rx "Rexx:transpose.rexx"
```

Occorre specificare l'estensione e il percorso completi, altrimenti non opera. È anche possibile assegnare questa stringa ad un tasto funzione.

## **Codici di ritorno**

Al termine dell'elaborazione di un comando, l'ospite risponde con un codice di restituzione per indicare lo stato del comando. La documentazione per l'applicazione ospite generalmente descrive i codici di restituzione possibili per ogni comando.

Il codice di ritorno è posto nella variabile speciale ARexx RC affinché possa essere esaminato dalla macro. Il valore 0 significa che il comando è stato eseguito correttamente. Un intero positivo indica una condizione di errore. La gravità dell'errore aumenta all'aumentare del valore dell'intero. Il codice di ritorno consente al programma macro di determinare la corretta esecuzione del comando ed operare in caso di errore.

## **Shell**

Sebbene ARexx sia previsto per operare più efficacemente con programmi che supportano la sua interfaccia comando specifica, può essere ugualmente usato con qualsiasi programma shell che utilizzi i meccanismi standard I/O per ottenere il proprio flusso di ingresso. Un modo per usare ARexx è creare un file comandi attuale sul disco RAM, quindi passarlo direttamente alla Shell. Il programma 11 apre una nuova Shell per eseguire lo script EXECUTE standard.

**Programma 11. Shell.rexx**

```
/*Apri una nuova Shell*/  
ADDRESS command  
conwindow = "CON:0/0/640/100/Nuova/Close"  
  
/*Crea un file di comandi*/  
CALL OPEN out,"ram:temp",write  
CALL WRITELN out, 'echo "Questa è una prova"'  
CALL CLOSE out  
  
/*Apri una finestra per la Shell*/  
'newshell' conwindow "ram:temp"  
EXIT
```

## **Ambiente di esecuzione**

**Nota** Il soggetto trattato in questo paragrafo è indirizzato agli utenti Amiga esperti. Le descrizioni presumono una certa padronanza di lavoro del sistema operativo Amiga e conoscenza con i manuali Amiga ROM Kernel.

L'interprete ARexx, REXXMAST, fornisce un ambiente di esecuzione uniforme eseguendo ogni programma come elaborazione autonoma nel sistema operativo multitasking di Amiga. Ciò permette di avere un'interfaccia flessibile fra programma ospite esterno e REXXMAST. Il programma ospite esterno può procedere contemporaneamente con le proprie operazioni oppure può attendere che finisca l'interpretazione del programma ARexx. Ogni programma ARexx contiene un ambiente interno ed esterno.

### **Ambiente esterno**

L'ambiente esterno comprende la struttura di elaborazione, i flussi di ingresso e uscita e la directory attuale. Quando viene creata ogni elaborazione ARexx, questa eredita i flussi di ingresso e di uscita e la directory attuale dal cliente, il programma esterno che ha chiamato il programma ARexx. Per esempio se il programma ARexx è stato avviato da una shell, il programma eredita il flusso di ingresso e di uscita e la directory attuale di quella shell. La directory attuale

è usata come punto di partenza in ogni ricerca di file programma o dati. Le funzioni esterne sono limitate ad un massimo di 15 argomenti.

## **Ambiente interno**

L'ambiente interno di un programma ARexx consiste di una struttura statica globale ed uno o più ambienti di memorizzazione. I valori dei dati globali sono fissati (statici) alla chiamata del programma. Questi valori comprendono i codici sorgente del programma, le stringhe dati statiche, e le stringhe argomenti. Una volta che il programma è in esecuzione, questi valori non possono essere cambiati.

I programmi ARexx chiamati come comandi generalmente hanno soltanto una stringa argomento, quantunque l'opzione di indicazione comando ne possa fornire più di uno. Un programma chiamato come funzione interna può avere un numero indefinito di argomenti. Questi argomenti persistono per la durata del programma.

L'ambiente di memorizzazione comprende la tabella simboli utilizzata per i valori delle variabili, le opzioni numeriche, le opzioni trace, e le stringhe di indirizzo ospite. Mentre l'ambiente globale è unico, possono esistere molti ambienti di memorizzazione durante il corso dell'esecuzione programma. Ogniqualvolta è chiamata una funzione interna, viene attivato e inizializzato un nuovo ambiente di memorizzazione. I valori iniziali della maggior parte dei campi sono ereditati dall'ambiente precedente, ma i valori possono essere modificati dopo senza influire sull'ambiente del chiamante. Il nuovo ambiente persiste fino al ritorno del controllo dalla funzione.

Ogni ambiente di memorizzazione comprende una tabella simboli per memorizzare le stringhe valori che sono state assegnate alle variabili. Questa tabella simboli è organizzata come albero binario a due livelli. Il livello primario memorizza le entrate per simboli semplici e radici. Il livello secondario è usato per simboli composti. Tutti i simboli composti associati ad una particolare radice sono memorizzati in un albero in cui l'entrata della radice funge da radice dell'albero.

I simboli non vengono introdotti nella tabella fino a quando non viene indicata loro una assegnazione. Una volta create, le entrate al

livello primario non vengono rimosse, anche se il simbolo diventa successivamente inutile. Gli alberi secondari sono rilasciati ogniqualvolta viene indicata una nuova assegnazione alla radice associata a quell'albero.

## ***Ricerca risorse***

ARexx fornisce la ricerca completa di tutte le risorse allocate dinamicamente da utilizzare per l'esecuzione del programma. Queste risorse comprendono lo spazio memoria, i file DOS, le strutture correlate, e la struttura della porta messaggi. Il sistema di ricerca consente la chiusura di un programma in qualsiasi punto senza lasciare alcuna risorsa pendente.

È possibile uscire dalla rete di ricerca risorse dell'interprete effettuando chiamate direttamente al sistema operativo di Amiga dall'interno del programma ARexx. È responsabilità del programmatore cercare e presentare tutte le risorse allocate all'esterno del sistema di rilevamento risorse ARexx. ARexx fornisce una speciale funzione di interruzione affinché il programma possa mantenere il controllo dopo un errore di esecuzione, eseguire l'eliminazione richiesta e uscire.



## Capitolo 4

# Istruzioni

---

Una clausola istruzione inizia con il nome di una istruzione particolare ed indica ad ARexx di eseguire una data operazione. Questo capitolo fornisce un elenco alfabetico delle istruzioni disponibili in ARexx.

Ogni parola chiave dell'istruzione può essere seguita da una o più sottoparole chiave, espressioni, o altre informazioni specifiche dell'istruzione. Le parole chiave dell'istruzione e le sottoparole chiave sono riconosciute soltanto in questo contesto specifico. Ciò consente alle stesse parole chiave di essere usate in un contesto differente come variabili o nomi funzioni. Una parola chiave di istruzione non può essere seguita da un operatore due punti (:) o uguale (=).

## Sintassi

La sintassi di ogni istruzione è indicata a destra dell'intestazione della parola chiave. Le convenzioni usate nelle sintassi sono indicate nella Tabella 4-1:

**Tabella 4-1. Convenzioni di sintassi**

Convenzione	Definizione
<b>PAROLA CHIAVE</b>	Tutte le parole chiave e sottoparole chiave sono indicate in maiuscolo
<b>espressione</b>	Gli argomenti richiesti sono indicati in minuscolo
<b>I (barra verticale)</b>	Le selezioni alternative sono separate da una barra verticale

Convenzione	Definizione
<b>{ (graffe)</b>	Le alternative richieste sono racchiuse fra graffe
<b>[] (parentesi quadre)</b>	Parti di istruzioni opzionali sono racchiuse in parentesi quadre

Per esempio, il formato per l'istruzione CALL è il seguente:

```
CALL {simbolo | stringa} [espressione]
[,espressione,...]
```

Occorre fornire un simbolo o una stringa come argomento. La barra verticale identifica le selezioni alternative, e le graffe indicano che è richiesto l'uso di un argomento. La specificazione di una espressione è opzionale, come indicato dalle parentesi quadre.

Gli esempi sono riportati alla fine della specifica dell'istruzione. Le descrizioni o le valutazioni degli esempi sono indicati come commenti ARexx /\*...\*/.

## Riferimento alfabetico

Questo paragrafo fornisce l'elenco alfabetico delle istruzioni integrate di ARexx. La sintassi di ogni istruzione è illustrata a destra della parola chiave dell'istruzione.

**ADDRESS ADDRESS** [[simbolo | stringa] | [VALUE][espressione]]

Questa istruzione specifica l'indirizzo ospite emesso dall'interprete. Un indirizzo ospite è il nome della porta messaggi di una applicazione a cui sono inviati i comandi ARexx. ARexx ha due indirizzi ospite: un valore attuale ed un valore precedente. Ogniqualvolta è fornito un indirizzo ospite, l'indirizzo precedente è perso e l'indirizzo attuale diventa quello precedente. Questi indirizzi ospiti sono parte di un ambiente di memorizzazione del programma e sono preservati attraverso chiamate di funzioni interne. L'indirizzo attuale può essere reperito con la funzione integrata ADDRESS().

La parola chiave ADDRESS da sola interscambia gli ospiti attuali e precedenti. L'esecuzione ripetuta commuta tra i due indirizzi ospite.

ADDRESS {stringa | simbolo} specifica che il nuovo indirizzo ospite è la stringa o il simbolo. Il valore della stringa o simbolo è l'indicatore stesso. I nomi delle porte messaggi subiscono il confronto maiuscole. La sintassi appropriata di un comando programma ad una porta messaggi denominata MyPort è:

```
ADDRESS 'MyPort'
```

L'omissione degli apici attorno a MyPort induce ARexx a cercare la porta messaggi MYPORt e genera un errore. L'indirizzo ospite corrente diventa l'indirizzo precedente. Un'espressione specificata dopo una stringa o simbolo viene elaborata e il risultato emesso all'ospite specificato. Non vengono effettuate modifiche alle stringhe indirizzo attuale o precedente. Ciò fornisce un modo conveniente per emettere un comando singolo ad un ospite esterno senza interessare gli indirizzi ospiti attuali. Il codice di restituzione proveniente dal comando è trattato come se fosse proveniente da una clausola comando.

Se è specificata l'espressione ADDRESS [VALUE], ARexx usa il risultato dell'espressione come nuovo indirizzo ospite, e l'indirizzo attuale diventa l'indirizzo precedente. La parola chiave VALUE può essere omessa se il primo indicatore dell'espressione non è un simbolo o una stringa. Per esempio:

```
ADDRESS /*Scambia le porte precedente e attivate.*/  
ADDRESS edit *La nuova porta è EDIT.*/  
ADDRESS edit 'top' /*Si sposta su top.*/  
ADDRESS VALUE edit in /*Calcola un nuovo indirizzo  
ospite.*/
```

## **ARG**

**ARG [modello] [,modello ...]**

ARG è la forma abbreviata dell'istruzione PARSE UPPER ARG. Reperisce una o più stringhe argomento disponibili per il programma e assegna valori alle variabili nel modello. Le stringhe argomento disponibili a seconda se programma è stato chiamato da un comando o da una funzione. Le chiamate comando normalmente

hanno soltanto una stringa argomento, ma possono avere anche 15 funzioni. Le stringhe argomento non sono modificate dall'istruzione ARG. ARG restituisce lettere maiuscole. Per esempio:

```
ARG first,second /*Legge argomenti*/
```

La struttura e l'elaborazione delle maschere è brevemente descritta con l'istruzione PARSE a pagina 4-13.

## BREAK

## BREAK

L'istruzione BREAK è usata per uscire da un ciclo dell'istruzione DO o dall'interno di una stringa interpretata da INTERPRET. È valida soltanto in questi contesti. Se usata in una dichiarazione DO, BREAK esiste dalla dichiarazione DO più interna contenente BREAK. Questo contrasta con l'analoga istruzione LEAVE, che esiste soltanto da un DO iterativo (ripetizione). Per esempio:

```
DO /*Blocco iniziale*/
  IF a>3 THEN BREAK /*Finito?*/
  a = a + 1
  y.a = name
END /*Fine blocco*/
```

**CALL** CALL {simbolo | stringa} [espressione] [,espressione, ...]

L'istruzione CALL è usata per chiamare una funzione interna o esterna. Il nome della funzione è specificato dall'indicatore simbolo o stringa. Qualsiasi espressione che segue viene elaborata e diventa l'argomento per la funzione chiamata. Il valore restituito dalla funzione è assegnato alla variabile speciale RESULT. Non è un errore se una stringa risultato non viene restituita. In questo caso la variabile RESULT è tralasciata da DROP (diventa non inizializzata).

Il collegamento alla funzione viene stabilito dinamicamente al momento della chiamata. ARexx segue uno specifico ordine di ricerca nel tentare di localizzare la funzione chiamata. Per esempio:

```
CALL CENTER name,length+4,'+'
```

CENTER è la funzione chiamata. L'espressione viene elaborata e passata come argomenti a CENTER.

**DO** [[var=exp] | [exp] [TO exp] [BY exp]]  
[FOR exp] [FOREVER] [WHILE exp | UNTIL exp]

L'istruzione DO comincia un gruppo di istruzioni eseguite come blocco. La gamma dell'istruzione DO comprende tutte le dichiarazioni fino a ed inclusa una eventuale istruzione END.

Se l'istruzione DO non è seguita da sottoparole chiave, il blocco viene eseguito una volta. Le sottoparole chiave possono essere usate per iterare il blocco sino al rilevamento della condizione di terminazione. Una istruzione DO iterativa talvolta è denominata ciclo poiché ARexx "ricicla" per eseguire ripetutamente l'istruzione. Le varie parti dell'istruzione DO sono:

- Una espressione inizializzatrice nella forma "variabile=espressione" definisce la variabile indice del ciclo. L'espressione è elaborata quando la gamma DO è attivata per la prima volta e il risultato è assegnato alla variabile indice. Nelle iterazioni successive viene elaborata un'espressione nella forma "variabile = variabile + incremento" dove l'incremento è il risultato dell'espressione BY. Se specificato, l'espressione inizializzatrice deve precedere tutte le altre sottoparole chiave.
- L'espressione che segue un simbolo BY definisce l'incremento da aggiungere alla variabile indice in ogni iterazione successiva. L'espressione deve produrre un risultato numerico, che può essere positivo o negativo e non necessita che sia intero. L'incremento predefinito è 1.
- Il risultato dell'espressione TO specifica il limite superiore (o inferiore) per la variabile indice. Ad ogni iterazione la variabile indice è comparata al risultato TO. Se l'incremento (risultato di BY) è positivo e la variabile è maggiore del limite, l'istruzione

DO termina e il controllo passa alla dichiarazione seguente successiva all'istruzione END. Il ciclo termina anche se l'incremento è negativo e la variabile indice è minore del limite.

- L'espressione FOR deve produrre un numero intero positivo quando elaborato e specifica il numero massimo di iterazioni da eseguire. Il ciclo termina quando si raggiunge questo limite indipendentemente dal valore della variabile indice.
- Le espressioni inizializzatrici BY, TO e FOR sono elaborate soltanto quando l'istruzione è attivata per la prima volta, così l'incremento ed i limiti sono fissati per tutta l'esecuzione. Il limite non è richiesto. Per esempio, l'istruzione "DO i=1" conteggia indefinitivamente.
- La parola chiave FOREVER può essere usata se è richiesta l'istruzione DO iterativa ma non è necessaria la variabile indice. IL ciclo sarà terminato dall'istruzione LEAVE o BREAK contenuta nel ciclo.
- L'espressione WHILE è elaborata all'inizio di ogni iterazione e deve risultare in un valore booleano. L'iterazione procede se il risultato è 1 (vero); altrimenti, il ciclo termina.
- L'espressione UNTIL è elaborata al termine di ogni iterazione e deve risultare in un valore booleano. L'istruzione continua con l'iterazione successiva se il risultato è 0 (falso), altrimenti termina. (WHILE e UNTIL sono reciprocamente esclusive.)

#### **Programma 12. Iteration.rexx**

```
/*Esempio di DO*/
LIMIT = 20; number = 1
DO i=1 to LIMIT for 10 WHILE number < 20
    number = i * number
    SAY "Iterazione" i "numero=" number
END
number = number/3.345; i = 0
DO number for LIMIT/5
    i = i + 1
    SAY "Iterazione" i "numero=" number
END
```

L'uscita è indicata con linee di commento per la descrizione. I commenti non appaiono sullo schermo.

```

Iterazione 1 numero = 1 /*1 * 1 = 1*/
Iterazione 2 numero = 2 /*2 * 1 = 2*/
Iterazione 3 numero = 6 /*3 * 2 = 6*/
Iterazione 4 numero = 24 /*4 * 6 = 24*/
Iterazione 1 numero = 7.17488789 /*24/3.345 =
7.17488789*/
Iteration 2 number = 7.17488789 /*il numero non
cambia*/
Iteration 3 number = 7.17488789 /*limite/5 = 20/5 =
4*/
Iteration 4 number = 7.17488789 /*operazione
ripetuta 4 volte*/

```

**Nota** Se è presente anche il limite FOR, l'espressione iniziale è ancora elaborata ma il risultato non occorre che sia un positivo intero.

## **DROP**

**DROP** variabile [variabile ...]

I simboli variabile specificati sono ripristinati al loro stato non inizializzato, nel quale il valore della variabile è il nome della variabile stessa. Non è un errore usare DROP per una variabile che sia già non inizializzata. L'uso di DROP per un simbolo radice è equivalente ad usare DROP per i valori di tutti i simboli composti possibili derivati dall'ultima radice. Per esempio:

```

a = 123 /*Assegna un valore ad a */
DROP a b /*Toglie i valori per A e B*/
SAY a b /*Risulta in A B.*

```

## **ECHO**

**ECHO** [espressione]

L'istruzione ECHO è un sinonimo dell'istruzione SAY. Visualizza il risultato dell'espressione sulla console. Per esempio:

```

ECHO "Non è say"

```

**ELSE**

ELSE [;] [dichiarazione condizionata]

L'istruzione ELSE fornisce il salto condizionato alternativo alla dichiarazione IF. È valida soltanto nella gamma di un'istruzione IF e deve seguire la dichiarazione condizionale del punto decisionale THEN. Se il punto THEN non è stato eseguito, viene eseguita la dichiarazione che segue la clausola ELSE. Se si desidera eseguire più istruzioni nella discriminazione mafuscolo/minuscolo di ELSE, delimitare questo blocco di istruzione con DO...END.

Le clausole ELSE si collegano sempre alla dichiarazione IF precedente più vicina. Può rendersi necessario fornire clausole ELSE "fittizie" per le gamme interne IF di una dichiarazione IF composta per consentire punti decisionali alternativi per le dichiarazioni IF esterne. Non è sufficiente far seguire a ELSE il punto e virgola o una clausola nulla. Invece, può essere usata l'istruzione NOP (no-operation). Per esempio:

```
IF i > 2 THEN SAY 'Realmente?'
    ELSE SAY 'Va bene'
```

**END**

END [variabile]

L'istruzione END termina la gamma delle istruzioni DO o SELECT. Se è fornito il simbolo variabile opzionale questo è comparato con la variabile indice della dichiarazione DO (che deve essere iterativa). Se i simboli non corrispondono viene generato un errore. Per esempio:

```
DO i=1 to 5      /*La variabile indice è i*/
    SAY i
    END i        /*Fine del ciclo "i"*/
```

**EXIT**

EXIT [espressione]

L'istruzione EXIT termina l'esecuzione di un programma. È valido in qualsiasi punto del programma. L'espressione elaborata è reinviata al chiamante come risultato della funzione o comando.

L'elaborazione del risultato EXIT è differente se la stringa risultato è stata richiesta dal programma chiamante o se l'invocazione attuale è risultata da una chiamata comando o funzione.



- Se una stringa risultato è stata richiesta, il risultato dell'espressione è copiato su un blocco di memoria allocata ed è restituito un puntatore al blocco come risultato secondario della chiamata.
- Se il chiamante non ha richiesto la stringa risultato e il programma è stato invocato come comando, viene fatto un tentativo per convertire il risultato dell'espressione in un numero intero. Questo valore è quindi restituito come risultato primario, con 0 come risultato secondario. Ciò permette all'informazione EXIT di essere interpretata come codice di restituzione dal chiamante.

Per esempio:

```
EXIT                /*Codice di ritorno non richiesto*/  
EXIT 10             /*È occorso un errore*/
```

## **IF**                    IF espressione [THEN] [;] [dichiarazione condizionata]

L'istruzione IF viene usata congiuntamente alle istruzioni THEN e ELSE per eseguire in modo condizionato una dichiarazione. Il risultato dell'espressione deve essere un valore booleano. Se il risultato è 1 (Vero), viene eseguita la dichiarazione che segue il simbolo THEN. Altrimenti, il controllo passa alla dichiarazione successiva. La parola chiave THEN non deve necessariamente seguire immediatamente l'espressione IF, ma può apparire come clausola separata.

L'istruzione è analizzata come "IF espressione; THEN; dichiarazione". L'espressione che segue la dichiarazione IF stabilisce la condizione di verifica che determina se verranno eseguite le clausole THEN o ELSE. Qualsiasi dichiarazione valida può seguire il simbolo THEN. In particolare, il gruppo "DO . . . END;" consente di eseguire condizionatamente una serie di dichiarazioni. Per esempio:

```
IF result < 0 THEN exit                    /*Finito?*/
```

## **INTERPRET**

INTERPRET espressione

Il comando INTERPRET tratta l'espressione come se fosse un blocco di dichiarazioni sorgente. L'espressione viene elaborata e il risultato è eseguito come una o più dichiarazioni di programma. Le dichiarazioni sono considerate come gruppo, come se fossero circondate dalla combinazione "DO . . . END". Qualsiasi dichiarazione può essere

compresa nella sorgente interpretata da INTERPRET, incluse le istruzioni DO o SELECT. L'istruzione BREAK può essere usata per terminare l'elaborazione di dichiarazioni interpretate.

Un'istruzione INTERPRET, al momento dell'esecuzione attiva un livello di controllo, che funge da limite per le istruzioni LEAVE e ITERATE. Queste istruzioni possono soltanto essere usate con i cicli DO definiti in INTERPRET. Non è un errore includere clausole etichetta all'interno della stringa interpretata, vengono ricercate soltanto le etichette definite nel programma originale durante il trasferimento del controllo.

L'istruzione INTERPRET può essere usata per costruire dinamicamente programmi per poi eseguirli. I frammenti di programmi sono passati come argomenti alle funzioni, le quali poi li INTERPRETano. Per esempio:

```
inst = 'SAY'                /*Una istruzione*/  
INTERPRET inst hello       /*. . . "SAY HELLO"*/
```

## ITERATE

ITERATE [variabile]

L'istruzione ITERATE termina l'iterazione attuale di un'istruzione DO ed inizia l'iterazione successiva. In realtà, il controllo passa alla dichiarazione END e quindi (in funzione del risultato dell'espressione UNTIL) ritorna alla dichiarazione DO. L'istruzione normalmente agisce sul ciclo DO iterativo interno. Se l'istruzione ITERATE non è contenuta nell'istruzione DO iterativa avviene un errore.

Se esistono parecchi cicli annidati il simbolo variabile opzionale specifica quale ciclo DO deve essere abbandonato. La variabile è rilevata come valore letterale e deve corrispondere alla variabile indice dell'istruzione DO attualmente attiva. Se non è rilevata l'istruzione DO corrispondente avviene un errore. Per esempio:

```
DO i=1 to 5  
  IF i = 3 THEN ITERATE i  
  SAY i  
END
```

**LEAVE****LEAVE** [variabile]

**LEAVE** provoca un'uscita immediata dal ciclo **DO** iterativo contenente l'istruzione. Se l'istruzione **LEAVE** non è contenuta nell'istruzione **DO** iterativa avviene un errore. Se esistono parecchie cicli annidati il simbolo variabile opzionale specifica quale ciclo **DO** deve essere abbandonato. La variabile è rilevata come valore letterale e deve corrispondere alla variabile indice dell'istruzione **DO** attualmente attiva. Se non è rilevata l'istruzione **DO** corrispondente avviene un errore. Per esempio:

```
DO i = 1 to limit
  IF i > 5 THEN LEAVE /*Numero massime iterazioni*/
END
```

**NOP****NOP**

L'istruzione **NOP** (**NO-oPeration**) è fornita per controllare il collegamento delle clausole **ELSE** in dichiarazioni **IF** composte. Per esempio:

```
IF i = j THEN          /*Primo IF (esterno)*/
  IF j = k THEN a = 0  /*IF interno*/
    ELSE NOP          /*Else più interno*/
  ELSE a = a + 1      /*Else esterno*/
```

**NUMERIC**

**NUMERIC** {**DIGITS** | **FUZZ**} espressione  
**NUMERIC FORM** {**SCIENTIFIC** | **ENGINEERING**}

- L'istruzione **NUMERIC** imposta opzioni relative alla precisione numerica e al formato. Le opzioni numeriche sono preservate quando è chiamata una funzione interna.
- L'opzione dell'espressione **DIGITS** specifica il numero di cifre di precisione per i calcoli aritmetici. L'espressione deve elaborare un numero intero positivo.
- L'opzione dell'espressione **FUZZ** specifica il numero di cifre da ignorare nelle operazioni di comparazione numerica. Il numero deve essere intero positivo, inferiore all'impostazione **DIGITS** attuale.

- L'opzione FORM SCIENTIFIC specifica che i numeri richiedenti la notazione esponenziale siano espressi con notazione scientifica. L'esponente è regolato affinché la mantissa per numeri diversi da zero sia compresa tra 1 e 9. Questo è il formato predefinito.
- L'opzione FORM ENGINEERING seleziona il formato engineering per numeri che richiedono la notazione esponenziale. Il formato engineering normalizza un numero affinché l'esponente sia un multiplo di tre e la mantissa (se non è zero) sia compresa tra 1 e 999.

```
NUMERIC DIGITS 12    /*12 cifre di precisione*/
NUMERIC FORM SCIENTIFIC /*Notazione scientifica
notation*/
```

## OPTIONS

```
OPTIONS [FAILAT espressione]
OPTIONS [PROMPT espressione]
OPTIONS [RESULTS]
OPTIONS [CACHE]
```

L'istruzione OPTIONS è usata per impostare vari valori predefiniti interni. L'espressione FAILAT imposta il limite in cui, od oltre cui, i codici di restituzione comando saranno segnalati come errore. Il limite deve essere un valore intero. L'espressione PROMPT fornisce una stringa da usare come testo di richiesta con l'istruzione PULL (o PARSE PULL). La parola chiave RESULTS indica che l'interprete deve richiedere una stringa risultato al momento dell'invio dei comandi all'ospite esterno.

Le opzioni interne controllate da questa istruzione sono preservate lungo le chiamate di funzione senza influenzare l'ambiente del chiamante. Se con l'istruzione OPTIONS non è specificata alcuna parola chiave, tutte le opzioni controllate ripristinano le proprie impostazioni predefinite. L'istruzione OPTIONS accetta anche la parola chiave NO per ripristinare l'opzione selezionata al valore predefinito, rendendo più conveniente ripristinare l'attributo RESULTS per un singolo comando senza dover ripristinare le opzioni FAILAT e PROMPT.

OPTIONS accetta anche la parola chiave CACHE che può essere usata per abilitare o disabilitare la funzione per la memorizzazione cache. Cache è normalmente abilitata. Per esempio:

```
OPTIONS FAILAT 10
OPTIONS PROMPT "Ok capo?"
OPTIONS RESULTS
```

## **OTHERWISE**      OTHERWISE [;] [dichiarazione condizionata]

Questa istruzione è valida soltanto con la gamma di una istruzione SELECT e deve seguire tutte le dichiarazioni "WHEN . . . THEN". Se nessuna delle clausole WHEN precedenti è riuscita, viene eseguita la dichiarazione che segue l'istruzione OTHERWISE. OTHERWISE non è obbligatoria nella gamma SELECT. Tuttavia, se la clausola OTHERWISE è omessa e nessuna delle istruzioni WHEN è riuscita, avviene un errore. Per esempio:

```
SELECT
  WHEN i=1 THEN say 'uno'
  WHEN i=2 THEN say 'due'
  OTHERWISE SAY 'altro'
END
```

## **PARSE**      PARSE [UPPER] sorgenteingresso [modello] [,modello ...]

L'istruzione PARSE fornisce un meccanismo per estrarre una o più sottostringhe da una stringa ed assegnarle alle variabili. La stringa di ingresso può provenire da varie sorgenti, fra cui stringhe argomento, un'espressione o da console.

L'esame è controllato mediante modello, che può consistere di simboli, stringhe, operatori e parentesi. Il modello fornisce sia le variabili cui dare valori sia il modo per determinare le stringhe valore. Durante l'operazione di analisi sintattica la stringa ingresso è divisa in sottostringhe assegnate ai simboli variabile nel modello. Il processo continua sino ad assegnare un valore a tutte le variabili nel modello. Se la stringa di ingresso è "esaurita" alle variabili rimanenti sono assegnati valori nulli.

Quando una variabile nel modello è seguita immediatamente da un'altra variabile, la stringa valore è determinata dividendo la

stringa di ingresso in parole separate da spazi. Non sono permessi spazi iniziali e finali. Ogni parola viene assegnata ad una variabile nel modello. Normalmente l'ultima variabile riceve il resto non indicizzato della stringa di ingresso perché non è seguito da un simbolo. Il simbolo segnaposto, un punto (.), forza la terminazione della variabile con il punto al primo spazio nel flusso di ingresso. I segnaposto si comportano come variabili, ma a loro non è mai assegnato un valore.

Il modello può essere omissso se l'istruzione è intesa soltanto per creare la stringa di ingresso. I modelli sono descritti nel Capitolo 7.

L'obiettivo dell'operazione di analisi sintattica è associare una posizione attuale e la successiva con ciascun simbolo variabile nel modello. La sottostringa tra queste posizioni viene quindi assegnata come valore alla variabile.

Le opzioni dell'istruzione sono descritte nel prosieguo.

- La parola chiave opzionale UPPER può essere usata con qualsiasi sorgente di ingresso e specifica che la stringa di ingresso deve essere tradotta in maiuscolo prima dell'analisi sintattica. Deve essere il primo indicatore dopo PARSE.
- Le sorgenti per le stringhe di ingresso sono specificate dai simboli parola chiave descritte nel seguito. Quando sono forniti modelli multipli ogni modello riceve una nuova stringa di ingresso, quantunque per alcune opzioni sorgenti la nuova stringa sia identica alla precedente. La stringa sorgente di ingresso viene copiata prima dell'analisi, affinché le stringhe originali non siano mai alterate dal processo di analisi.
- L'opzione di ingresso ARG reperisce le stringhe argomento fornite alla chiamata del programma. Normalmente le invocazioni comando hanno soltanto una stringa argomento singola, ma le funzioni possono avere 15 stringhe argomento.
- La stringa di ingresso EXTERNAL viene letta dal flusso STDERR, (vedere il Capitolo 6) in modo da non interferire con i dati trattati da PUSH o QUEUE. Se sono forniti modelli multipli, ogni modello legge una stringa nuova. Questa opzione sorgente è analoga a PULL.

- L'opzione di ingresso **NUMERIC** pone le opzioni numeriche attuali in una stringa nell'ordine **DIGITS**, **FUZZ** e **FORM**, separate da spazio singolo.
- L'opzione di ingresso **PULL** legge una stringa dalla console di input. Se sono forniti modelli multipli, ogni modello legge una stringa nuova.
- L'opzione di ingresso **SOURCE** reperisce le stringhe " in ingresso" per il programma. Il formato della stringa è il seguente.

```
{COMMAND|FUNCTION} {0|1} CALLED RESOLVED EXT HOST
```

dove:

- {COMMAND | FUNCTION} indica se il programma è stato chiamato come comando o come funzione.
- {0 | 1} è un marcatore booleano che indica se una stringa risultato era stata richiesta dal chiamante.
- CALLED è il nome usato per chiamare questo programma.
- RESOLVED è il nome finale risolto del programma.
- EXT è l'estensione del file da usare per ricerca ("REXX" è l'estensione predefinita).
- HOST è l'indirizzo ospite iniziale per i comandi.

L'opzione **SOURCE** ora ritorna al nome percorso completo del file programma **ARexx**. Prima era stato dato solo un nome relativo, che non era sufficiente a localizzare il file sorgente del programma.

La stringa di ingresso "VALUE espressione WITH" è il risultato dell'espressione fornita. La parola chiave **WITH** è richiesta per separare l'espressione dal modello. Il risultato dell'espressione può essere analizzato ripetutamente mediante i modelli multipli, ma l'espressione non è rielaborata.

L'opzione di ingresso "VAR variabile" utilizza il valore della variabile specificata come stringa di ingresso. Quando sono forniti modelli multipli, ogni modello usa il valore attuale della variabile. Questo valore può cambiare se la variabile è compresa in un obiettivo di assegnazione in qualsiasi modello.

L'opzione di ingresso **VERSION** della configurazione attuale dell'interprete **ARexx** è fornita nella forma:

ARexx VERSION CPU MPU VIDEO FREQ

dove:

- **VERSION** è il livello di release dell'interprete, con formato 1.14.
- **CPU** indica il processore che attualmente elabora il programma ed avrà uno dei valori 68000, 68010, 68020, 68030 o 68040.
- **MPU** può essere **NONE**, 68881, o 68882, a seconda del processore matematico disponibile.
- **VIDEO** indica **NTSC** o **PAL**.
- **FREQ** fornisce la frequenza della linea 60Hz o 50Hz.

Per esempio:

```
/*Numeric string is: "9 0 SCIENTIFIC"*/  
PARSE NUMERIC DIGITS FUZZ FORM .  
SAY digits      /*9*/  
SAY fuzz       /*0*/  
SAY form       /*SCIENTIFIC*/  
myvar = 1234567890  
PARSE VAR myvar 1 a 3 b +2 c 1 d  
SAY a  
SAY b  
SAY c  
SAY d
```

Questa è l'uscita:

```
12  
34  
567890  
1234567890
```

## **PROCEDURE**      **PROCEDURE** [**EXPOSE** variabile [variabile...]]

L'istruzione **PROCEDURE** è usata in una funzione interna per creare una nuova tabella simboli. Questa protegge i simboli definiti nell'ambiente del chiamante da eventuali alterazioni causate dall'esecuzione della funzione. **PROCEDURE** è generalmente la prima dichiarazione nella funzione, quantunque sia valida in qualunque punto all'interno del corpo funzione. È un errore



eseguire due dichiarazioni PROCEDURE all'interno della stessa funzione.

La sottoparola chiave EXPOSE fornisce un meccanismo selettivo per accedere alla tabella simboli del chiamante, e per passare variabili globali alla funzione. Le variabili che seguono la parola chiave EXPOSE sono prese come riferimenti ai simboli nella tabella del chiamante. Eventuali successive modifiche effettuate su queste variabili saranno indicate anche nell'ambiente del chiamante.

Le variabili nell'elenco EXPOSE possono comprendere simboli radice o composti, nel qual caso l'ordine delle variabili diventa significativo. L'elenco EXPOSE è elaborato da sinistra a destra ed i simboli composti sono espansi in base ai valori validi nella nuova generazione. Per esempio, si supponga che il valore del simbolo J nella generazione precedente sia 123, e che J sia non inizializzato nella nuova generazione. Allora PROCEDURE EXPOSE J A.J espone J e A.123, mentre PROCEDURE EXPOSE A.J J espone A.J e J. L'esposizione di una radice ha l'effetto di esporre tutti i simboli composti possibili derivati da quella radice. Cioè, PROCEDURE EXPOSE A. espone A.I, A.J, A.J.J, A.123, ecc. Per esempio:

```
fact: PROCEDURE      /*Una funzione recursiva*/  
      ARG i  
      IF i = 1  
      THEN RETURN 1  
      ELSE RETURN i * fact(i-1)
```

## **PULL**

**PULL** [modello] [,modello...]

Pull è l'abbreviazione dell'istruzione PARSE UPPER PULL. Legge una stringa proveniente dalla console, la traduce in maiuscolo e l'analizza sintatticamente mediante modello. Possono essere lette stringhe multiple fornendo modelli aggiuntivi. L'istruzione legge dalla console anche se non sono forniti modelli. (I modelli sono descritti nel Capitolo 7). Per esempio:

```
PULL first last .      /*Lettura nomi*/
```

**PUSH**

PUSH [espressione]

L'istruzione PUSH è usata per preparare un flusso di dati da leggere mediante una shell comandi o altro programma. Appende una nuova linea al risultato dell'espressione quindi la impila o la spinge nel flusso STDIN. Le linee impilate sono collocate nel flusso nell'ordine "last-in, first-out" (ultimo entrato, primo uscito) e sono disponibili per la lettura come fossero state introdotte interattivamente. Per esempio, dopo l'emissione delle istruzioni:

```
PUSH line 1
PUSH line 2
PUSH line 3
```

il flusso viene letto nell'ordine linea 3, linea 2 e linea 1.

PUSH consente al flusso STDIN di essere usato come blocco note privato per preparare i dati per l'elaborazione successiva. Per esempio, parecchi file possono essere concatenati con delimitatori fra essi semplicemente leggendo i file in ingresso, PUSH spinge la linea nel flusso ed inserisce il delimitatore dove richiesto. Per esempio:

```
DO i=1 to 5
  PUSH 'echo "Linea 'i' "'
END
```

**QUEUE**

QUEUE [espressione]

L'istruzione QUEUE è usata per preparare un flusso di dati da leggere mediante una shell comandi o altro programma. È analoga all'istruzione PUSH e differisce soltanto per il fatto che le linee dati sono collegate nel flusso STDIN nell'ordine "first-in, first-out". In questo caso, le istruzioni:

```
QUEUE line 1
QUEUE line 2
QUEUE line 3
```

si leggono nell'ordine linea 1, linea 2 e linea 3. Le linee trattate da **QUEUE** precedono sempre tutte le linee introdotte interattivamente e seguono sempre le linee impilate da **PUSH**. Per esempio:

```
DO i=1 to 5
  QUEUE 'echo "Linea 'i'"'
END
```

**RETURN****RETURN** [espressione]

**RETURN** è usata per abbandonare una funzione e cedere il controllo al punto di chiamata della funzione precedente. L'espressione elaborata è restituita come risultato della funzione. Se non viene fornita l'espressione, può avvenire un errore nell'ambiente del chiamante. Le funzioni chiamate dall'interno di un'espressione devono restituire la stringa risultato e generano un errore se il risultato non è disponibile. Le funzioni chiamate mediante l'istruzione **CALL** non devono necessariamente restituire alcun risultato.

**RETURN** emessa dall'ambiente base di programma non è un errore ed è equivalente all'istruzione **EXIT**. Riferirsi all'istruzione **EXIT** per la descrizione sulle modalità di trasferimento delle stringhe risultato al chiamante esterno. Per esempio:

```
RETURN 6*7    /*Ritorna 42*/
```

**SAY****SAY** [espressione]

Il risultato dell'espressione elaborata è scritto sulla console di uscita, con appeso il carattere nuova linea. Se l'espressione è omessa viene inviata alla console una stringa nulla. Per esempio:

```
SAY 'La risposta è ' value
```

**SELECT****SELECT**

**SELECT** inizia un gruppo di istruzioni contenente una o più clausole **WHEN** e eventualmente una clausola **OTHERWISE** singola, ognuna seguita da una dichiarazione condizionata. Viene eseguita soltanto una delle istruzioni condizionate all'interno del gruppo

SELECT. Ogni dichiarazione WHEN è eseguita in successione sino alla riuscita di una. Se non ne riesce nessuna, viene eseguita la dichiarazione OTHERWISE. La gamma SELECT deve essere selezionata da una dichiarazione END. Per esempio:

```
SELECT
    WHEN i=1 THEN SAY 'uno'
    WHEN i=2 THEN SAY 'due'
    OTHERWISE SAY 'altro'
END
```

**SHELL** SHELL [simbolo | stringa] | [[VALUE] [espressione]]

L'istruzione **SHELL** è un sinonimo dell'istruzione **ADDRESS**. Per esempio:

```
SHELL edit    /*In posta ospite su 'EDIT'*/
```

<b>SIGNAL</b>	SIGNAL {ON   OFF} condizione
	SIGNAL [VALUE] espressione

SIGNAL {ON | OFF} controlla lo stato degli indicatori di interruzione interni. Le interruzioni consentono al programma di rilevare e tenere il controllo al rilevamento di alcuni errori. In questa forma SIGNAL deve essere seguito da una delle parole chiave ON o OFF ed una delle parole di condizione elencate nel seguito. L'interruttore di interruzione specificato dal simbolo di condizione viene quindi impostato allo stato indicato. Le condizioni del segnale valido sono:

<b>BREAK_C</b>	Rilevata interruzione Ctrl+C.
<b>BREAK_D</b>	Rilevata interruzione Ctrl+D.
<b>BREAK_E</b>	Rilevata interruzione Ctrl+E.
<b>BREAK_F</b>	Rilevata interruzione Ctrl+F.
<b>ERROR</b>	Codice non zero restituito dal comando ospite
<b>HALT</b>	Rilevata una richiesta HALT esterna.
<b>IOERR</b>	Rilevato errore del sistema I/O.
<b>NOVALUE</b>	Uso di variabili non inizializzate.
<b>SYNTAX</b>	Rilevato errore di sintassi o esecuzione.

Le parole chiave di condizione sono interpretate come etichette cui il controllo viene trasferito se avviene la condizione selezionata. Per esempio, se viene abilitata l'interruzione **ERROR** ed un comando restituisce il codice non zero, **ARexx** trasferisce il controllo all'etichetta **ERROR**:. L'etichetta di condizione deve essere definita nel programma; altrimenti, avviene l'errore **SINTAX** immediato con uscita del programma.

Nell'espressione **SIGNAL [VALUE]**, gli indicatori che seguono **SIGNAL** sono elaborati come espressione. Viene generata un'interruzione immediata che trasferisce il controllo all'etichetta specificata dal risultato dell'espressione. L'istruzione agisce così come "goto condizionale".

Ogniqualevolta avviene un'interruzione, tutte le gamme di controllo attive attuali (**IF**, **DO**, **SELECT**, **INTERPRET**, o interattive **TRACE**) sono eliminate prima del trasferimento del controllo. Così, il trasferimento non deve essere usato per saltare in una gamma di ciclo **DO** o altre strutture di controllo. La condizione **SIGNAL** influisce soltanto sulle strutture di controllo dell'ambiente corrente, rendendone possibile l'uso dall'interno della funzione senza influire sullo stato dell'ambiente del chiamante.

La variabile speciale **SIGL** è impostata sul numero di linea attuale ogniqualevolta avviene un trasferimento del controllo. Il programma può ispezionare **SIGL** per determinare quale linea era in corso di esecuzione prima del trasferimento. Se un **ERROR** o una condizione **SINTAX** causa una interruzione, viene impostata la variabile speciale **RC** sul codice errore che ha causato l'interruzione. Per la condizione **ERROR**, questo codice generalmente è un livello di gravità dell'errore. Per ulteriori informazioni sui codici di errore e livelli di gravità riferirsi all'appendice A. La condizione **SINTAX** indica sempre un codice errore **ARexx**:. Per esempio:

```
SIGNAL on error      /*Abilita interruzione*/  
SIGNAL off syntax    /*Disabilita SINTAX*/  
SIGNAL start         /*Salta a START*/
```

**WHEN** WHEN espressione [THEN [:] [dichiarazione condizionata]]

L'istruzione WHEN è analoga all'istruzione IF, ma è valida soltanto nella gamma SELECT. Ogni espressione WHEN viene elaborata in successione e deve risultare in un valore booleano. Se il risultato è 1, viene eseguita la dichiarazione condizionata e il controllo passa alla dichiarazione END che termina l'istruzione SELECT. Analogamente all'istruzione IF, THEN non deve necessariamente far parte della stessa clausola. Per esempio:

```
SELECT
  WHEN i<j THEN SAY 'minore'
  WHEN i=j THEN SAY 'uguale'
  OTHERWISE SAY 'maggiore'
END
```

## **Capitolo 5**

# **Funzioni**

---

La funzione è un programma o gruppo di istruzioni che vengono eseguite ogniqualevolta il nome della funzione appare in un contesto particolare. Una funzione può essere parte di un programma interno, parte di una libreria o un programma esterno separato. Le funzioni costituiscono un blocco costruttivo importante di programmi modulari perché consentono di costruire grandi programmi partendo da una serie di piccoli moduli facilmente sviluppabili.

Questo capitolo descrive i vari tipi di funzione e le modalità di elaborazione. Fornisce inoltre un elenco alfabetico della libreria di funzioni integrata ARexx.

## ***Chiamata di una funzione***

Nel programma ARexx, una funzione è definita come simbolo o stringa seguita immediatamente da una parentesi aperta. Il simbolo o la stringa (preso come valore letterale) specifica il nome funzione, e la parentesi aperta inizia la lista degli argomenti. Tra l'apertura e la chiusura della parentesi vi sono zero o più espressioni di argomenti, separati da virgole, che forniscono i dati da passare alla funzione.

Le chiamate valide di funzione sono:

```
CENTER ('titolo',20)
ADDRESS()
'ALLOCMEM' (256*4,1)
```

Ogni espressione argomento è elaborata in sequenza e le stringhe risultanti sono passate come lista argomento alla funzione. Ogni espressione argomento, generalmente un valore letterale singolo,

può comprendere operazioni aritmetiche o di stringa o anche altre chiamate di funzioni. Le espressioni argomento sono elaborate da sinistra a destra.

Le funzioni possono anche essere chiamate mediante l'istruzione CALL. L'istruzione CALL, descritta nel Capitolo 4, può essere usata per chiamare una funzione che può non restituire il valore.

## ***Tipi di funzione***

Sono possibili tre tipi di funzione:

- Funzioni interne — definite all'interno del programma ARexx.
- Funzioni integrate — fornite dal linguaggio di programmazione ARexx.
- Librerie funzioni — una libreria Amiga speciale.

### ***Funzioni interne***

La funzione interna è identificata da una etichetta nel programma. Quando viene chiamata la funzione interna, ARexx crea un nuovo ambiente di memorizzazione affinché l'ambiente chiamante precedente sia preservato. Il nuovo ambiente eredita i valori del predecessore, ma le modifiche successive alle variabili ambiente non influiscono sull'ambiente precedente.

I valori specifici preservati sono:

- Indirizzi ospite attuale e precedente
- Le impostazioni NUMERIC DIGITS, FUZZ, e FORM
- L'opzione trace, indicatore di inibizione, e indicatore interattivo
- Lo stato degli indicatori di interruzione come definiti dall'istruzione SIGNAL
- La stringa carattere di richiesta attuale come impostata dall'istruzione OPTIONS PROMPT

Il nuovo ambiente non ottiene automaticamente una nuova tabella simboli, quindi inizialmente tutte le variabili dell'ambiente precedente sono disponibili per la funzione chiamata. L'istruzione PROCEDURE può essere utilizzata per creare una tabella e quindi



proteggere i valori dei simboli del chiamante. **PROCEDURE** può anche essere usata per consentire l'utilizzo dello stesso nome variabile in due aree differenti con due valori differenti.

L'esecuzione della funzione interna procede fino all'esecuzione di un'istruzione **RETURN**. A questo punto il nuovo ambiente è eliminato, e il controllo riprende dal punto della chiamata funzione. L'espressione fornita con l'istruzione **RETURN** è elaborata e restituita al chiamante come risultato della funzione.

## ***Funzioni integrate***

**ARexx** fornisce una cospicua libreria di funzioni predefinite come parte del sistema di linguaggio. Queste funzioni sono sempre disponibili e sono ottimizzate per operare con le strutture dati interne. In generale, le funzioni integrate operano molto più velocemente di una funzione equivalente interpretata, perciò se ne raccomanda l'uso.

Parecchie funzioni integrate creano e manipolano file **AmigaDOS** esterni. I file sono riferiti mediante un nome logico, un nome con confronto maiuscole che è assegnato al file quando viene aperto la prima volta. I flussi di ingresso e uscita iniziali sono forniti nei nomi **STDIN** (ingresso standard) e **STDOUT** (uscita standard). Teoricamente non vi è limite la numero di file che possono essere aperti contemporaneamente, quantunque il limite viene imposto dalla memoria disponibile. Tutti i file aperti sono chiusi automaticamente all'uscita del programma.

## ***Librerie di funzioni esterne***

La libreria funzione è una raccolta di una o più funzioni organizzate come la libreria **Amiga** condivisa. La libreria deve risiedere in **LIBS:**, ma può risiedere nella memoria o su disco. Le librerie residenti su disco sono caricate e aperte secondo le necessità.

La libreria deve essere personalizzata in modo speciale affinché sia utilizzabile da **ARexx**. Ogni libreria funzioni deve contenere un nome libreria, una priorità di ricerca, lo spostamento del punto di ingresso, ed un numero di versione. Quando **ARexx** ricerca una funzione, l'interprete apre ciascuna libreria e controlla il proprio punto

di ingresso "query". Questo punto di ingresso deve essere specificato come distanza intera (p.e. "-30") dalla libreria base. Il codice di restituzione proveniente dalla chiamata query indica se è stata trovata la funzione desiderata. Se è trovata, viene chiamata con i parametri passati dall'interprete, e il risultato della funzione è restituito al chiamante. Se non è trovata, viene restituito il codice di errore "Funzione non trovata" e la ricerca continua sulla libreria successiva della lista. Le librerie funzione sono sempre chiuse dopo il controllo affinché il sistema operativo possa ottenere lo spazio memoria se richiesto.

### ***Lista delle librerie***

Il processo residente ARexx aggiorna la lista librerie funzioni e ospiti funzioni correntemente disponibile denominata lista librerie. Il programma applicativo può aggiungere o togliere librerie funzioni come richiesto.

La Lista Librerie è tenuta aggiornata e ordinata per priorità. Ogni voce ha una priorità di ricerca associata nella gamma 100 (più alta) a -100 (più bassa). Le voci possono essere aggiunte ad una priorità appropriata per controllare la risoluzione del nome funzione. Le librerie con priorità più alta sono cercate per prime. All'interno di un dato livello di priorità, le librerie aggiunte prima sono cercate prima. I livelli di priorità sono significativi se ciascuna libreria ha duplicate le definizioni dei nomi funzione, poiché la funzione posta ulteriormente più in basso nella catena di ricerca potrebbe non essere mai chiamata.

### ***Ospiti di funzioni esterne***

Il nome associato ad un ospite di funzione è quello della porta messaggi pubblica. Le chiamate di funzioni sono passate all'ospite come pacchetto messaggi; è quindi compito dell'ospite individuale determinare se il nome funzione specificato è noto. La risoluzione del nome è completamente interna all'ospite, così gli ospiti di funzione forniscono un meccanismo di collegamento naturale per implementare chiamate di procedure remote ad altre macchine in rete. Il processo residente ARexx è una funzione ospite ed è installato nella Lista Librerie con priorità -60.

## Ordine di ricerca

I collegamenti funzione in ARexx sono stabiliti al momento della chiamata della funzione. È seguito un ordine di ricerca specifico sino al rilevamento della corrispondenza simbolo nome o stringa. Se non può essere localizzata la funzione specificata, si genera un errore e termina l'elaborazione dell'espressione. L'ordine completo di ricerca è:

**Funzioni interne**

Viene esaminata la sorgente del programma per la corrispondenza dell'etichetta con il nome funzione. Se c'è corrispondenza, viene creato un ambiente di memorizzazione nuovo ed il controllo è trasferito all'etichetta.

**Funzioni integrate**

Nella libreria funzioni integrate viene cercato il nome specificato. Queste funzioni sono definite da nomi in maiuscolo.

**Librerie di funzioni e funzioni ospite**

Le librerie di funzioni e le funzioni ospite sono aggiornate nella Lista Libreria, che è letta partendo dalla priorità più alta sino al rilevamento della funzione richiesta o al termine della lista. Le funzioni dell'ospite sono chiamate mediante un protocollo a messaggio passante analogo a quello usato per i comandi e può essere utilizzato come porta di collegamento per chiamate di procedure remote ad altre macchine in rete.

**Programmi ARexx esterni**

Il passo finale consiste nel cercare il file programma ARexx esterno inviando un messaggio di chiamata al processo residente ARexx. La ricerca inizia sempre nella directory attuale e segue lo stesso percorso di ricerca della chiamata programma ARexx originale. Il processo di corrispondenza del nome non esegue il confronto maiuscole.

La procedura di corrispondenza del nome funzione può tener conto della differenza tra lettere maiuscole e minuscole per alcune fasi della ricerca ma non per altre. La procedura di corrispondenza usata in una libreria di funzioni o in un ospite di funzione è a discrezione del programmatore. Le funzioni definite con nomi contenenti lettere maiuscole e minuscole devono essere chiamate mediante un

indicatore stringa, perché i nomi simbolo sono sempre tradotti in maiuscolo.

L'ordine completo di ricerca avviene ogniqualvolta il nome funzione è definito da un indicatore simbolo. Tuttavia, la ricerca di funzioni interne viene elusa se il nome è specificato mediante indicatore stringa. Ciò consente alle funzioni interne di usurpare i nomi delle funzioni esterne, come nell'esempio seguente:

```
CENTER:                /*"CENTER" interna*/
ARG string,length      /*lettura argomenti*/
length = MIN(length,60) /*calcolo lunghezza*/
return 'CENTER'(string, length)
```

Qui la funzione integrata CENTER() è stata sostituita da una funzione interna dopo la modifica dell'argomento lunghezza.

## Lista clip

La Lista Clip è un meccanismo con accesso pubblico utilizzato come blocco note generale per comunicazioni intertask. Molte funzioni usano la clipboard per reperire tipi differenti di informazioni, come costanti o stringhe predefinite.

La Lista Clip aggiorna un gruppo di coppie (nome, valore), da utilizzare per scopi diversi. (SETCLIP() è usato per aggiungere coppie alla lista.) Ogni voce nella lista consiste di una stringa nome e valore e può essere localizzata mediante il nome. In generale, i nomi usati devono essere unici per un'applicazione onde evitare duplicazioni indesiderate con altri programmi. La lista può contenere un indefinito numero di voci.

Un'applicazione potenziale della Lista Clip consiste in un meccanismo per il caricamento di costanti predefinite nel programma ARexx. Per esempio:

```
pi=3.14159; e=2.718; sqrt2=1.414 . . .
```

(cioè, una serie di assegnazioni separate da punto e virgola). Nella pratica, tale stringa può essere reperita per nome usando la funzione integrata GETCLIP() e interpretata con INTERPRET all'in-

terno del programma. Le dichiarazioni di assegnazione entro la stringa creano quindi le definizioni costanti richieste. Per esempio:

```
/*Si assume che la stringa chiamata "numeri" sia  
disponibile*/  
numeri= GETCLIP('numeri')  
INTERPRET numeri      /*. . . assegnazioni*/
```

Le stringhe non devono essere ristrette per contenere soltanto dichiarazioni di assegnazione, ma possono comprendere dichiarazioni ARexx valide. La Lista Clip può così fornire una serie di programmi per inizializzare altri compiti di elaborazione.

Il processo residente supporta le operazioni di aggiunta e cancellazione per l'aggiornamento della Lista Clip. È presunto che i nomi nella coppia (nome, valore) siano in caratteri maiuscoli e minuscoli e quindi vengono aggiornati affinché siano unici nella lista. Un tentativo di aggiungere una stringa con nome esistente aggiorna soltanto la stringa valore. Le stringhe nome e valore vengono copiate quando la voce è inserita nella lista, così il programma che aggiunge una voce non deve aggiornare le stringhe.

Le voci inserite nella Lista Clip rimangono disponibili se non esplicitamente cancellate. La Lista Clip è automaticamente rilasciata al termine del processo residente.

## ***Funzioni integrate — Riferimento***

Questo paragrafo fornisce l'elenco alfabetico delle funzioni integrate. La sintassi di ogni funzione è indicata a destra della parola chiave della funzione.

### ***Sintassi***

Gli argomenti opzionali sono indicati in parentesi e generalmente hanno un valore predefinito usato se l'argomento è omissso. Quando viene specificata una parola chiave opzione come argomento, è significativo soltanto il primo carattere. Le parole chiave opzione non subiscono il confronto maiuscole.



**ADDLIB()**                      ADDLIB(nome,priorità[,spostamento,versione])

Aggiunge una libreria funzioni o un ospite funzioni alla lista librerie aggiornata dal processo residente. L'argomento nome specifica il nome di una libreria funzioni o porta messaggi pubblica associata ad un ospite funzione. Il nome esegue il confronto maiuscole. Ciascuna libreria specificata deve risiedere nella directory di sistema LIBS:.

L'argomento priorità specifica la priorità di ricerca e deve essere un numero intero tra 100 e -100, compresi. Gli argomenti distanza e versione sono validi soltanto per le librerie. L'offset è lo spostamento dell'intero rispetto al punto di introduzione della "query" della libreria, e la versione è un intero che specifica il minimo accettabile della libreria.

La funzione restituisce un risultato booleano che indica se l'operazione è riuscita. Se è specificata una libreria, in questo momento non è aperta. Analogamente, ARexx non verifica se la porta ospite funzione specificata è aperta. Per esempio:

```
SAY ADDLIB("rexxsupport.library",0,-30,0) → 1  
CALL ADDLIB "EtherNet",-20                /*A gateway*/
```

**ADDRESS()**                      ADDRESS()

Restituisce la stringa indirizzo ospite attuale. L'indirizzo ospite è la porta messaggi cui i comandi sono inviati. La funzione SHOW() può essere usata per controllare se l'ospite esterno richiesto è realmente disponibile. Vedere anche SHOW(). Per esempio:

```
SAY ADDRESS()                → REXX
```

**ARG()** ARG([numero][, 'EXISTS' | 'OMITTED'])

ARG() restituisce il numero di argomenti forniti all'ambiente attuale. Se è fornito soltanto un parametro numero, è restituita la stringa argomento corrispondente. Se è dato il numero e la parola chiave Exists (Esiste) o Omitted (Omesso), la restituzione booleana indica lo stato dell'argomento corrispondente. La verifica di esistenza o omissione non indica se la stringa ha valore nullo, ma soltanto se è stata fornita una stringa. Per esempio:

```
/*Assunzione argomenti dove: ('uno',,10)*/  
SAY ARG()      → 3  
SAY ARG(1)     → uno  
SAY ARG(2, 'O') → 1
```

**B2C()** B2C(stringa)

Converte una stringa di cifre binaria (0,1) nella rappresentazione caratteri corrispondente (pacchetto). La conversione è la stessa quantunque la stringa argomento non sia stata specificata come stringa binaria letterale (p.e. '1010'B). Gli spazi sono consentiti nella stringa ma soltanto nei byte di transizione. Questa funzione è particolarmente utile per creare stringhe da usare come maschere di bit. Vedere anche X2C(). Per esempio:

```
SAY B2C('00110011') → 3  
SAY B2C('01100001') → a
```

**BITAND()** BITAND(stringa1,stringa2[,riempimento])

Le stringhe argomento sono collegate logicamente da AND, e la lunghezza del risultato è data dalla più lunga delle due stringhe operando. Se è fornito un carattere di riempimento, la stringa più corta è riempita a destra. Altrimenti, l'operazione termina alla fine della stringa più corta, e il resto della stringa più lunga è appeso al risultato. Per esempio:

```
BITAND('0313'x, 'FFF0'x) → '0310'x
```



**BITCHG()****BITCHG(stringa,bit)**

Modifica lo stato del bit specificato nella stringa argomento. I numeri bit sono definiti affinché il bit 0 sia il bit di ordine inferiore del byte all'estrema destra della stringa. Per esempio:

```
BITCHG('0313'x,4) → '0303'x
```

**BITCLR()****BITCLR(stringa,bit)**

Azzera il bit specificato nella stringa argomento. I numeri bit sono definiti affinché il bit 0 sia il bit di ordine inferiore del byte all'estrema destra della stringa. Per esempio:

```
BITCLR('0313'x,4) → '0303'x
```

**BITCOMP()****BITCOMP(stringa1,stringa2[,riempimento])**

Confronta le stringhe argomento bit per bit, iniziando dal numero bit 0. Il valore restituito è il numero del primo bit in cui le stringhe differiscono, o -1 se le stringhe sono identiche. Per esempio:

```
BITCOMP('7F'x,'FF'x) → 7 /*Settimo bit*/  
BITCOMP('FF'x,'FF'x) → -1
```

**BITOR()****BITOR(stringa1,stringa2[,riempimento])**

Le stringhe argomento sono collegate logicamente da OR, e la lunghezza del risultato è data dalla più lunga delle due stringhe operando. Altrimenti l'operazione termina alla fine della stringa più corta, e il resto della stringa più lunga viene aggiunto al risultato. Per esempio:

```
BITOR('0313'x,'003F'x) → '033F'x
```

**BITSET()**

BITSET(stringa,bit)

Imposta a 1 il bit specificato nella stringa argomenti. I numeri di bit sono definiti affinché il bit 0 sia il bit inferiore del byte della stringa. Per esempio:

```
BITSET('0313'x,2)    → '0317'x
```

**BITTST()**

BITTST(stringa,bit)

La restituzione booleana indica lo stato del bit specificato nella stringa argomento. I numeri di bit sono definiti affinché il bit 0 sia il bit inferiore del byte della stringa. Per esempio:

```
BITTST('0313'x,4)    → 1
```

**BITXOR()**

BITXOR(stringa1,stringa2[,riempimento])

Le stringhe argomento sono logicamente collegate con XOR, e la lunghezza del risultato è data dalla più lunga delle due stringhe operando. Se viene fornito un carattere di riempimento, la stringa più corta viene completata con caratteri di riempimento a destra; altrimenti, l'operazione termina alla fine della stringa più corta e la stringa più lunga viene appesa al risultato. Per esempio:

```
BITXOR('0313'x,'001F'x) → '030C'x
```

**C2B()**

C2B(stringa)

Converte la stringa di carattere nella stringa equivalente di cifre binarie. Vedere anche C2X(). Per esempio:

```
SAY C2B('abc')    → 01100000101100001001100011
```

**C2D()****C2D(stringa[,n])**

Converte l'argomento stringa dalla rappresentazione del carattere al numero decimale corrispondente, espresso come cifre ASCII (0-9).

Se viene fornito n, la stringa di caratteri è considerata come numero espresso in n byte. La stringa è troncata e completata con caratteri nulli a sinistra come richiesto, e il bit di segno viene esteso per la conversione. Per esempio:

```
SAY C2D('0020'x)    → 32
SAY C2D('FFFF ffff'x) → -1
SAY C2D('FF0100'x,2) → 256
```

**C2X()****C2X(stringa)**

Converte l'argomento stringa dalla rappresentazione del carattere al numero esadecimale corrispondente, espresso con caratteri ASCII 0-9 e A-F. Vedere anche C2B(). Per esempio:

```
SAY C2X('abc')    → 616263
```

**CENTER()****CENTER(stringa, lunghezza[,riempimento])****CENTRE()****CENTRE(stringa, lunghezza[,riempimento])**

Centra l'argomento stringa in una stringa con la lunghezza specificata. Se la lunghezza è maggiore di quella della stringa, i caratteri di riempimento o gli spazi sono aggiunti secondo le necessità. Per esempio:

```
SAY CENTER('abc',6)      → ' abc  '
SAY CENTER('abc',6,'+')  → '+abc++'
SAY CENTER('123456',3)   → '234'
```

**CLOSE()**

CLOSE(file)

Chiude il file specificato da un dato nome logico. Il valore restituito è un indicatore di riuscita booleano ed è pari a 1 a meno che il file specificato non sia stato aperto. Per esempio:

```
SAY CLOSE('input') → 1
```

**COMPARE()**

COMPARE(stringa1,stringa2[,riempimento])

Confronta due stringhe e restituisce l'indice della prima posizione in cui esse differiscono oppure 0 se le stringhe sono identiche. La stringa più corta viene completata con caratteri di riempimento come richiesto usando il carattere fornito o gli spazi. Per esempio:

```
SAY COMPARE('abcde','abcce') → 4
```

```
SAY COMPARE('abcde','abcde') → 0
```

```
SAY COMPARE('abc++','abc+-','+') → 5
```

**COMPRESS()**

COMPRESS(stringa[,lista])

Se l'argomento lista viene omissso, la funzione rimuove gli spazi iniziali, finali o interposti dall'argomento stringa. Se viene fornita la lista opzionale, specifica i caratteri che devono essere rimossi dalla stringa. Per esempio:

```
SAY COMPRESS(' why not ') → whynot
```

```
SAY COMPRESS('++12-34-+', '+-') → 1234
```

**COPIES()**

COPIES(stringa,numero)

Crea una nuova stringa concatenando il numero specificato di copie dell'originale. L'argomento numero può essere zero, nel qual caso viene restituita la stringa nulla. Per esempio:

```
SAY COPIES('abc',3) → abcabcabc
```

**D2C()**

D2C(numero)

Crea una stringa il cui valore è la rappresentazione binaria (pacchetto) del numero decimale dato. Per esempio:

D2C(65) → A

**D2X()**

D2X(numero[,cifre])

Converte un numero decimale in esadecimale. Per esempio:

D2X(31) → 1F

**DATE()**

DATE([opzione],[data],[formato])

Restituisce la data corrente nel formato specificato. L'opzione predefinita ('NORMAL') restituisce la data nel formato GG MMM AAAA (giorno, meso, anno), come in 20 APR 1988. Le opzioni riconosciute sono:

<b>BASEDATE</b>	Numero di giorni dal 1 gennaio del millennio in corso
<b>CENTURY</b>	Numero di giorni dal 1 gennaio del secolo corrente
<b>DAYS</b>	Numeri dei giorni dal 1 gennaio dell'anno corrente
<b>EUROPEAN</b>	Data nella forma GG/MM/AA
<b>INTERNAL</b>	Giorni del sistema interno
<b>JULIAN</b>	Data nella forma AAGGG
<b>MONTH</b>	Mese corrente (in caratteri misti)
<b>NORMAL</b>	Data nella forma GG MMM AAAA
<b>ORDERED</b>	Data nella forma AA/MM/GG
<b>SORTED</b>	Data nella forma AAAMMGG
<b>USA</b>	Data nella forma MM/GG/AA
<b>WEEKDAY</b>	Giorno della settimana (in caratteri misti)

Queste opzioni possono essere ridotte al solo primo carattere.

La funzione DATE() accetta anche secondi e terzi argomenti opzionali per fornire la data nella forma di giorni del sistema interno oppure nella forma ordinata indicata con "sorted" AAAAMMGG. Il

secondo argomento è un numero intero che indica che specifica o i giorni del sistema (predefinito) o una data nella forma "sorted". Il terzo argomento specifica la forma della data e può essere 'T' oppure 'S'. La data corrente nei giorni del sistema può essere recuperata usando DATE('INTERNAL'). Per esempio:

```
SAY DATE()           → 14 Jul 1992
SAY DATE('M')       → July
SAY DATE(S)         → 19920714
SAY DATE('S',DATE('I')+21) → 19920804
SAY DATE('W',19890609,'S') → Friday
```

## DATATYPE()

DATATYPE(stringa[,opzione])

Se è specificato soltanto l'argomento stringa, DATATYPE() verifica se il parametro stringa è un numero valido e restituisce NUM o CHAR. Se è data una parola chiave opzione, la restituzione booleana indica se la stringa ha soddisfatto la verifica richiesta. Vengono riconosciute le seguenti parole chiave opzione:

<b>ALPHANUMERIC</b>	Accetta caratteri alfabetici (A-Z, a-z) o numerici (0-9)
<b>BINARY</b>	Accetta una stringa di cifre binarie
<b>LOWERCASE</b>	Accetta caratteri alfabetici minuscoli (a-z)
<b>MIXED</b>	Accetta caratteri misti maiuscoli/minuscoli
<b>NUMERIC</b>	Accetta numeri validi
<b>SYMBOL</b>	Accetta simboli REXX validi
<b>UPPER</b>	Accetta caratteri alfabetici maiuscoli (A-Z)
<b>WHOLE</b>	Accetta numeri interi
<b>X</b>	Accetta stringhe di cifre esadecimali

Per esempio:

```
SAY DATATYPE('123') → NUM
SAY DATATYPE('1a f2','X') → 1
SAY DATATYPE('aBcde','L') → 0
```

**DELSTR()****DELSTR(stringa,n[,lunghezza])**

Cancella la sottoscritta dell'argomento stringa ad iniziare dall'n-esimo carattere per la lunghezza in caratteri specificata. La lunghezza predefinita è la lunghezza restante della stringa. Per esempio

```
SAY DELSTR('123456',2,3)    → 156
```

**DELWORD()****DELWORD(stringa,n[,lunghezza])**

Cancella la sottoscritta dell'argomento stringa ad iniziare dall'n-esima parola per la lunghezza in parole specificata. La lunghezza predefinita è la lunghezza restante della stringa. La stringa cancellata comprende qualsiasi spazio finale che segue l'ultima parola. Per esempio:

```
SAY DELWORD('Tell me a story',2,2)    → 'Tell story'
```

```
SAY DELWORD('one two three',3)    → 'one two '
```

**DIGITS()****DIGITS()**

Restituisce l'impostazione di cifre numeriche attuali. Per esempio:

```
NUMERIC DIGITS 6
```

```
SAY DIGITS()    → 6
```

**EOF()****EOF(file)**

Controlla il nome del file logico specificato e restituisce il valore booleano 1 (Vero) se è stata raggiunta la fine del file, e 0 (Falso) in caso contrario. Per esempio:

```
SAY EOF(infile)    → 1
```

**ERRORTEXT()**

ERRORTEXT(n)

Riporta il messaggio d'errore associato al codice d'errore ARexx specificato. Se il numero non è un codice d'errore valido viene restituita una stringa nulla. Per esempio:

```
SAY ERRORTEXT(41)    → Invalid expression
```

**EXISTS()**

EXISTS(nomefile)

Controlla se esiste un file esterno con nomefile dato. La stringa nome può comprendere specifiche del dispositivo e della directory. Per esempio:

```
SAY EXISTS('SYS:C/ED')    → 1
```

**EXPORT()** EXPORT(indirizzo[,stringa][,lunghezza][,riempimento])

Copia i dati dalla stringa opzionale in un'area di memoria precedentemente assegnata, che deve essere specificata come indirizzo da 4 byte. Il parametro lunghezza specifica il numero massimo di caratteri da copiare. Il valore predefinito è la lunghezza della stringa. Se la lunghezza specificata è superiore alla stringa, l'area rimanente è completata con caratteri di riempimento o con caratteri nulli ('00'x). Il valore restituito è il numero di caratteri copiati.

**Avvertenza**    **Qualsiasi area di memoria può essere sovrascritta, a volte causando uno stallo del sistema. Il task switching annullato durante la copia; di conseguenza la funzionalità del sistema può essere ridotta se vengono copiate stringhe lunghe.**

Vedere anche IMPORT() e STORAGE(). Per esempio:

```
count = EXPORT('0004 0000'x,'La risposta')
```



**FORM()**

FORM()

Restituisce l'impostazione attuale nella FORMA NUMERICA. Per esempio:

```
NUMERIC FORM SCIENTIFIC  
SAY FORM() → SCIENTIFIC
```

**FIND()**

FIND(stringa,frase)

La funzione FIND() localizza un insieme di parole in una stringa di parole più grande e restituisce il numero di posizione della parola corrispondente trovata. Per esempio:

```
SAY FIND('Oggi andiamo al mare','andiamo al') → 2
```

**FREESPACE()**

FREESPACE ()

Restituisce un blocco di memoria di data lunghezza al polo interpreti interno. L'argomento indirizzo deve essere una stringa di 4 byte ottenuta da una chiamata precedente a GETSPACE(), l'allocatore interno. Non è sempre necessario cedere memoria interna allocata, poiché è rilasciata al sistema quando il programma termina. Tuttavia, se è stato allocato un grande blocco, la restituzione al polo può evitare problemi di spazio memoria. Il valore restituito è un indicatore booleano ad esito positivo. Vedere anche GETSPACE ().

```
FREESPACE('00042000'x,32) → 1
```

Chiamando FREESPACE() senza argomenti viene restituita la quantità di memoria disponibile nel pool interno dell'interprete ARexx.

**FUZZ()**

FUZZ()

Restituisce l'impostazione corrente NUMERIC FUZZ. Per esempio:

```
NUMERIC FUZZ 3  
SAY FUZZ() → 3
```

**GETCLIP()**

GETCLIP(nome)

Cerca nella Lista Clip una voce che corrisponda col parametro nome fornito e restituisce la stringa del valore associata. La corrispondenza del nome esegue il confronto maiuscole; se non è trovato il nome viene restituita la stringa nulla. Vedere anche SETCLIP(). Per esempio:

```
/*Assume che 'numeri' contenga 'PI=3.14159'*/  
SAY GETCLIP('numeri')    → PI=3.14159
```

**GETSPACE()**

GETSPACE(lunghezza)

Assegna un blocco di memoria della lunghezza specificata dal pool interno dell'interprete. Il valore restituito è l'indirizzo da 4 byte del blocco allocato, che non viene rimesso a zero o altrimenti inizializzato. La memoria interna viene automaticamente restituita al sistema quando il programma ARexx si chiude, quindi questa funzione non deve essere usata per l'assegnazione di memoria a programmi esterni. La REXX Support Library comprende la funzione ALLOCMEM(), che alloca memoria dalla lista libera del sistema. Vedere anche FREESPACE(). Per esempio:

```
SAY C2X(GETSPACE(32))    → '0003BF40'x
```

**HASH()**

HASH(stringa)

Restituisce l'attributo calcolato di una stringa come numero decimale e aggiorna il valore interno calcolato della stringa. Per esempio:

```
SAY HASH('1')    → 49
```

**IMPORT()****IMPORT(indirizzo[,lunghezza])**

Crea una stringa copiando dei dati dall'indirizzo a 4 byte specificato. Se non viene fornito il parametro lunghezza, la copia termina quando viene trovato un byte nullo. Vedere anche EXPORT(). Per esempio:

```
extval = IMPORT('0004 0000'x,8)
```

**INDEX()****INDEX(stringa,modello[,inizio])**

Cerca la prima occorrenza dell'argomento modello nell'argomento stringa, incominciando dalla posizione di inizio specificata. La posizione di inizio predefinita è 1. Il valore restituito è l'indice del modello corrispondente oppure 0 se non è stato trovato il modello. Per esempio:

```
SAY INDEX("123456", "23") → 2
```

```
SAY INDEX("123456", "77") → 0
```

```
SAY INDEX("123123", "23", 3) → 5
```

**INSERT()****INSERT(nuova,vecchia[,inizio][,lunghezza]  
[,riempimento])**

Inserisce una stringa nuova nella stringa vecchia dopo la posizione di inizio specificata. La posizione di inizio predefinita è 0. La stringa nuova è troncata o completata con caratteri di riempimento alla lunghezza specificata come richiesto, usando il carattere di riempimento o gli spazi forniti. Se la posizione di inizio è oltre la fine della stringa vecchia, questa viene completata con caratteri di riempimento a destra. Per esempio:

```
SAY INSERT('ab', '12345') → ab12345
```

```
SAY INSERT('123', '++', 3, 5, '-') → ++-123--
```

**LASTPOS()**

LASTPOS(modello,stringa[,inizio])

Cerca a ritroso la prima occorrenza dell'argomento modello nell'argomento stringa, incominciando dalla posizione di inizio specificata. La posizione di inizio predefinita è la fine della stringa. Il valore restituito è l'indice del modello corrispondente oppure 0 se il modello non è stato trovato. Per esempio:

```
SAY LASTPOS('2','1234')      → 2
SAY LASTPOS('2','1234234')    → 5
SAY LASTPOS('2','123234',3)    → 2
SAY LASTPOS('2','13579')      → 0
```

**LEFT()**

LEFT(stringa,lunghezza[,riempimento])

Restituisce la sottostringa all'estrema sinistra in un dato argomento stringa con la lunghezza specificata. Se la stringa è più corta della lunghezza richiesta, viene completata a destra con i caratteri di riempimento o gli spazi forniti. Per esempio:

```
SAY LEFT('123456',3)        → 123
SAY LEFT('123456',8,'+')    → 123456++
```

**LENGTH()**

LENGTH(stringa)

Restituisce la lunghezza della stringa. Per esempio:

```
SAY LENGTH('three')        → 5
```

**LINES()**

LINES(file)

Restituisce il numero di righe accodate o digitate davanti al file logico, che deve riferirsi ad un flusso interattivo. Il conteggio delle righe viene ottenuto come risultato secondario di una chiamata WaitForChar(). Per esempio:

```
PUSH 'una linea'
PUSH 'un'altro linea'
SAY LINES(STDIN)           → 2
```

**MAX()** MAX(numero,numero[,numero, . . . ])

Restituisce il massimo degli argomenti forniti, i quali devono essere tutti numerici. Devono essere forniti almeno due parametri. Per esempio:

SAY MAX(2.1,3,-1) → 3

**MIN()** MIN(numero,numero[,numero, . . . ])

Restituisce il minimo degli argomenti forniti, i quali devono essere tutti numerici. Devono essere forniti almeno due parametri. Per esempio:

SAY MIN(2.1,3,-1) → -1

**OPEN()** OPEN(file,nomefile,['APPEND' | 'READ' | 'WRITE'])

Apre un file esterno per un'operazione specifica. L'argomento file definisce il nome logico col quale il file verrà riferito. Il nomefile è il nome esterno del file e può comprendere specifiche del dispositivo e della directory. La funzione restituisce un valore booleano che indica se l'operazione è riuscita. Non vi sono limiti quanto al numero di file aperti contemporaneamente e tutti i file aperti vengono chiusi automaticamente all'uscita dal programma. Vedere anche CLOSE(), READ(), e WRITE(). Per esempio:

SAY OPEN('MyCon', 'CON:160/50/320/100/MyCon/cds')  
→ 1

SAY OPEN('outfile','ram:temp','W') → 1

**OVERLAY()** OVERLAY(nuovo,vecchio[,inizio]  
[ ,lunghezza][ ,riempimento])

Sovrappone la stringa nuova alla stringa vecchia incominciando dalla posizione di inizio specificata, che deve essere positiva. La posizione di inizio predefinita è 1. La stringa nuova viene troncata o completata con caratteri di riempimento alla lunghezza specificata come richiesto, usando il carattere di riempimento o gli spazi forniti.

Se la posizione di inizio è oltre la fine della stringa vecchia, quest'ultima viene completata con caratteri di riempimento a destra. Per esempio:

```
SAY OVERLAY('bb','abcd') → bbcd
```

```
SAY OVERLAY('4','123',5,5,'-') → 123-4----
```

## POS()

POS(modello,stringa[,inizio])

Cerca la prima occorrenza dell'argomento modello nell'argomento stringa, incominciando dalla posizione specificata dall'argomento inizio. La posizione di inizio predefinita è 1. Il valore restituito è l'indice della stringa corrispondente oppure 0 se il modello non è stato trovato. Per esempio:

```
SAY POS('23','123234') → 2
```

```
SAY POS('77','123234') → 0
```

```
SAY POS('23','123234',3) → 4
```

## PRAGMA()

PRAGMA(opzione[,valore])

Questa funzione permette a un programma di modificare vari attributi relativi all'ambiente di sistema all'interno del quale il programma opera. L'argomento opzione è una parola chiave che specifica un attributo dell'ambiente. L'argomento valore fornisce il nuovo valore attribuito da installare. Il valore restituito dalla funzione dipende dall'attributo selezionato. Alcuni attributi restituiscono il valore precedente installato, mentre altri possono impostare un indicatore di riuscita booleano.

Le parole chiave opzione correntemente definite sono:

- DIRECTORY** Specifica una nuova directory attuale. La directory attuale viene usata come radice per i nomefile che non includono esplicitamente una specifica di dispositivo. Il risultato è il nome della directory vecchia. PRAGMA ('D') equivale a PRAGMA ('D',''); restituisce il percorso della directory attuale senza modificarla.
- PRIORITÀ** Specifica una nuova priorità del task. Il valore priorità deve essere un numero intero compreso tra -128 e 127, ma in pratica l'intervallo è molto più limitato. I programmi ARexx non devono essere eseguiti con una priorità superiore a quella del processo residente, che viene eseguito correntemente con priorità 4. Il valore restituito è il livello di priorità precedente.
- ID** Restituisce l'identificativo (ID) del processo (l'indirizzo del processo) come stringa esadecimale da 8 caratteri. L'ID del compito è un identificativo unico della chiamata particolare ARexx e può essere usato per crearne un nome unico.
- STACK** Specifica un nuovo valore dello stack per il programma attuale ARexx. Quando è dichiarato un nuovo valore viene restituito il valore precedente.

Le opzioni correntemente utilizzabili sono:

- PRAGMA('W',{'NULL'|'WORKBENCH'})** Gestisce il campo WindowPtr del compito. Impostandolo su 'Null' si sopprime qualsiasi carattere di richiesta che potrebbe altrimenti essere generato da una chiamata DOS.
- PRAGMA('\*',[nome])** Definisce il nome logico specificato come gestore della console attuale ("\*"), permettendo all'utente di aprire due flussi su una finestra. Se il nome è omissso, il gestore della console viene impostato su quello del processo del cliente.

```
SAY PRAGMA('D','DF0:C')    → Extras
SAY PRAGMA('D','DF1:C')    → Workbench:C
SAY PRAGMA('PRIORITY',-5)  → 0
SAY PRAGMA('ID')           → 00221ABC
CALL PRAGMA '*',STDOUT
SAY PRAGMA("STACK",8092)   → 4000
```

**RANDOM()**                      RANDOM([MIN][,MAX][,valore di partenza])

Restituisce un numero intero pseudocasuale nell'intervallo specificato dagli argomenti min e max. Il valore minimo predefinito è 0, mentre quello massimo è 999. L'intervallo max-min deve essere inferiore o uguale a 1000. Se è richiesto un intervallo di numeri interi casuali maggiore, i valori della funzione RANDU() possono essere adeguatamente dimensionati e tradotti. L'argomento valore di partenza può essere fornito per inizializzare lo stato interno del generatore di numeri casuali. Vedere anche RANDU(). Per esempio:

```
thisroll = RANDOM(1,6)     /*Potrebbe essere 1*/  
nextroll = RANDOM(1,6) /*Valori da 1 a 6*/
```

**RANDU()**                      RANDU([valore di partenza])

Restituisce un numero pseudocasuale uniformemente distribuito, tra 0 e 1. Il numero di cifre di precisione del risultato è sempre uguale all'impostazione Numeric Digits attuale. Con la scelta di valori di scala e di traduzione adeguati, RANDU() può essere usata per generare numeri pseudocasuali in un intervallo arbitrario.

L'argomento opzionale valore di partenza intero viene usato per inizializzare lo stato interno del generatore di numeri casuali. Vedere anche RANDOM(). Per esempio:

```
firsttry = RANDU()     /*0.371902021?*/  
NUMERIC DIGITS 3  
tryagain = RANDU()     /*0.873?*/
```

**READCH()**                      READCH(file,lunghezza)

Legge il numero di caratteri specificato dal file logico dato in una stringa. La lunghezza della stringa riportata è l'effettivo numero di caratteri letti e può essere inferiore alla lunghezza richiesta se, ad esempio è stata raggiunta la fine del file. Vedere anche READLN(). Per esempio:

```
instring = READCH('input',10)
```



**READLN()**

READLN(file)

Legge i caratteri dal file logico dato in una stringa fino al rilevamento del carattere "nuova riga". La stringa riportata non comprende la "nuova riga". Vedere anche READCH(). Per esempio:

```
instring = READLN('MyFile')
```

**REMLIB()**

REMLIB(nome)

Rimuove una voce con nome dato dalla Lista della libreria aggiornata dal processo residente. Il risultato booleano è 1 se la voce è stata trovata ed effettivamente rimossa. Questa funzione non distingue tra librerie di funzioni e ospiti di funzione, ma rimuove una voce nominata. Vedere anche ADDLIB(). Per esempio:

```
SAY REMLIB('MyLibrary.library') → 1
```

**REVERSE()**

REVERSE(stringa)

Inverte la sequenza di caratteri della stringa.. Per esempio:

```
SAY REVERSE('?ton yhw') → why not?
```

**RIGHT()**

RIGHT(stringa, lunghezza[, riempimento])

Restituisce la sottostringa all'estrema destra nell'argomento stringa con la lunghezza specificata. Se la sottostringa è più corta della lunghezza richiesta, viene completata a sinistra con il carattere di riempimento o gli spazi forniti. Per esempio:

```
SAY RIGHT('123456',4) → 3456
```

```
SAY RIGHT('123456',8,'+') → ++123456
```

**SEEK()**      **SEEK(file,spostamento[, 'BEGIN' | 'CURRENT' | 'END'])**

Sposta il cursore in una nuova posizione nel file logico dato, specificato come spostamento da una posizione di aggancio. L'aggancio predefinito è Current. Il valore restituito è la nuova posizione relativa all'inizio del file. Per esempio:

```
SAY SEEK('input',10,'B')      → 10
SAY SEEK('input',0,'E') → 356 /*file length*/
```

**SETCLIP()**      **SETCLIP(nome[,valore])**

Aggiunge una coppia nome-valore alla Lista Clip aggiornata dal processo residente. Se esiste già una voce con lo stesso nome, il suo valore viene aggiornato alla stringa del valore fornito. Le voci possono essere rimosse specificando un valore nullo. La funzione restituisce un valore booleano che indica se l'operazione è riuscita. Per esempio:

```
SAY SETCLIP('path','DF0:s')      → 1
SAY SETCLIP('path')      → 1
```

**SHOW()**      **SHOW(opzione[,nome][,riempimento])**

Restituisce i nomi nella lista risorse specificata dall'argomento opzione, oppure verifica se è disponibile una voce con nome specificato. Le parole chiave opzione correntemente utilizzabili sono:

<b>CLIP</b>	Esamina i nomi nella Lista Clip
<b>FILES</b>	Esamina i nomefile logici correntemente aperti
<b>LIBRARIES</b>	Esamina i nomi nella Lista delle librerie, che sono librerie di funzioni o ospiti di funzione.
<b>PORTS</b>	Esamina i nomi nella Lista porte di sistema

Se è omesso l'argomento nome, la funzione restituisce una stringa con i nomi delle risorse separata da uno spazio vuoto o dal carattere di riempimento, se fornito. Se viene dato l'argomento nome, il valore booleano riportato indica se il nome è stato trovato nella lista delle risorse. Le immissioni dei nomi subiscono il confronto maiuscole.

**SIGN()****SIGN(numero)**

Restituisce 1 se l'argomento numero è positivo oppure 0 e -1 se numero è negativo. L'argomento deve essere numerico. Per esempio:

```
SAY SIGN(12)      → 1
SAY SIGN(-33)     → -1
```

**SOURCELINE()****SOURCELINE([line])**

Restituisce il testo di una riga specificata del programma ARexx eseguito correntemente. Se l'argomento riga viene omesso, la funzione restituisce il numero totale di righe del file. Questa funzione viene spesso usata per integrare le informazioni "di guida" in un programma. Per esempio:

```
/*Programma di prova*/
SAY SOURCELINE()   → 3
SAY SOURCELINE(1) → /*Programma di prova*/
```

**SPACE()****SPACE(stringa,n[,riempimento])**

Riformatta l'argomento stringa in modo che vi siano n spazi (caratteri vuoti) tra ogni coppia di parole. Se il carattere di riempimento viene specificato, è usato invece degli spazi come carattere separatore. Specificando n come 0 tutti gli spazi vengono rimossi dalla stringa. Per esempio:

```
SAY SPACE('Now is the time',3)
    → 'Now   is   the   time'
SAY SPACE('Now is the time',0)
    → 'Nowisthetime'
SAY SPACE('1 2 3',1,'+')   → '1+2+3'
```

**STORAGE()**

STORAGE([indirizzo],[stringa]  
[,lunghezza],[riempimento])

STORAGE() senza argomenti restituisce la memoria di sistema disponibile. Se dato, l'argomento indirizzo deve essere una stringa da 4 byte. La funzione copia i dati dalla stringa (opzionale) all'indirizzo di memoria indicato. Il parametro lunghezza specifica il numero massimo di byte da copiare e torna alla lunghezza della stringa predefinita. Se la lunghezza specificata è superiore alla stringa, l'area rimanente è completata con il carattere di riempimento o caratteri nulli ('00'x).

Il valore restituito è il contenuto precedente dell'area memoria. Questa può essere utilizzata in una chiamata seguente per recuperare il contenuto originale. Vedere anche EXPORT().

**Avvertenza**    **Qualsiasi area di memoria può essere sovrascritta, causando a volte un arresto anomalo del sistema. Il cambio di compito è vietato quando la copia è in corso; di conseguenza la funzionalità del sistema può essere ridotta se vengono copiate delle stringhe lunghe.**

Per esempio:

```
SAY STORAGE()      → ' 248400
oldval = STORAGE('0004 0000'x,'The answer')
CALL STORAGE '0004 0000'x,,32,'+'

```

**STRIP()**

STRIP(stringa[,{'B' | 'L' | 'T'}][,riempimento])

Se non è fornito nessun parametro opzionale, la funzione rimuove gli spazi iniziali e finali dall'argomento stringa. Il secondo argomento specifica se devono essere rimossi i caratteri Leading (iniziali), Trailing (finali) o Both (entrambi). L'argomento di riempimento (o di svuotamento) opzionale seleziona il carattere da rimuovere. Per esempio:

```
SAY STRIP(' say what? ') → say what?
SAY STRIP(' say what? ','L') → say what?
SAY STRIP('++123++','B','+') → 123

```

**SUBSTR()**                      SUBSTR(stringa,inizio[,lunghezza][,riempimento])

Restituisce la sottostringa dell'argomento stringa incominciando dalla posizione di inizio specificata per la lunghezza specificata. La posizione di inizio deve essere positiva e la lunghezza predefinita è data dalla lunghezza rimanente della stringa. Se la sottostringa è più corta della lunghezza richiesta, viene completata a destra con degli spazi o il carattere di riempimento specificato. Per esempio:

```
SAY SUBSTR('123456',4,2)      → 45
```

```
SAY SUBSTR('myname',3,6,'=')      → name==
```

**SUBWORD()**                      SUBWORD(stringa,n[,lunghezza])

Sostituisce la sottostringa dell'argomento stringa incominciando dall'nesima parola per la lunghezza in parole specificata. La lunghezza predefinita è data dalla lunghezza rimanente della stringa. La stringa restituita non ha mai spazi iniziali o finali. Per esempio:

```
SAY SUBWORD('Now is the time ',2,2)      → is the
```

**SYMBOL()**                      SYMBOL(nome)

Controlla se l'argomento nome è un simbolo valido ARexx. Se non lo è, la funzione restituisce la stringa BAD. Se il simbolo non è inizializzato restituisce LIT. Se è stato assegnato un valore al simbolo, restituisce VAR. Per esempio:

```
SAY SYMBOL('J')      → VAR
```

```
SAY SYMBOL('x')      → LIT
```

```
SAY SYMBOL('++')      → BAD
```

**TIME()**

TIME(opzione)

Restituisce l'ora corrente del sistema o controlla il contatore interno del tempo trascorso. Le parole chiave valide sono:

<b>CIVIL</b>	Ora corrente in formato 12 ore (a.m./p.m.) con ore e minuti
<b>ELAPSED</b>	Tempo trascorso in secondi dall'inizio programma
<b>HOURS</b>	Ora corrente in ore dalla mezzanotte
<b>MINUTES</b>	Ora corrente in minuti dalla mezzanotte
<b>NORMAL</b>	Ora corrente in formato 24 ore con ore/minuti/secondi
<b>RESET</b>	Reimposta l'orologio del tempo trascorso
<b>SECONDS</b>	Ora corrente in secondi dalla mezzanotte

Se non viene specificata alcuna opzione, la funzione restituisce l'ora del sistema corrente nella forma HH:MM:SS. Per esempio:

```
/*Si suppone che siano le 1:02 AM . . .*/
SAY TIME('C')      → 1:02 AM
SAY TIME('HOURS')   → 1
SAY TIME('M')       → 62
SAY TIME('N')       → 01:02:54
SAY TIME('S')       → 3720
call TIME('R')      /*risetta temporizzatore*/
SAY TIME('E')       → .020
SAY TIME()          → 01:02:00
```

**TRACE()**

TRACE(opzione)

Imposta il modo di analisi (tracing) (vedi capitolo) su quanto specificato dalla parola chiave opzione, che deve essere una delle opzioni alfabetiche o opzioni prefisso valide. La funzione TRACE() altera il modo di analisi anche durante l'analisi interattiva, quando sono ignorate le istruzioni TRACE nel programma sorgente. Il valore restituito è il modo in uso prima della chiamata funzione. Questo consente di ripristinare in seguito il modo di analisi precedente. Per esempio:

```
/*Modo analisi è ?ALL*/
SAY TRACE('Results') → ?A
```



**UPPER()**

UPPER(stringa)

Traduce la stringa in maiuscolo. L'operazione di questa funzione equivale a quella di TRANSLATE(stringa), ma è leggermente più veloce per stringhe corte. Per esempio:

```
SAY UPPER('One Fine Day')    →  ONE FINE DAY
```

**VALUE()**

VALUE(nome)

Restituisce il valore del simbolo rappresentato dall'argomento nome. Per esempio:

```
/*Assume che J valga 12*/  
SAY VALUE('j')    → 12
```

**VERIFY()**

VERIFY(stringa,lista,['MATCH'])

Se l'argomento Match è omissso, la funzione restituisce l'indice del primo carattere nell'argomento stringa che non è contenuto nell'argomento lista oppure 0 se tutti i caratteri sono nella lista. Se viene fornita la parola chiave Match, la funzione restituisce l'indice del primo carattere che è nella lista oppure 0 se nessuno dei caratteri si trova nella lista. Per esempio:

```
SAY VERIFY('123456','0123456789')    → 0  
SAY VERIFY('123a56','0123456789')    → 4  
SAY VERIFY('123a45','abcdefghij','m') → 4
```

**WORD()**

WORD(stringa,n)

Restituisce l'nesima parola dell'argomento stringa o la stringa nulla se vi sono meno di n parole. Per esempio:

```
SAY WORD('Now is the time ',2)    → is
```



**WORDINDEX()****WORDINDEX(stringa,n)**

Restituisce la posizione di inizio dell'nesima parola della stringa argomento oppure 0 se vi sono meno di n parole. Per esempio:

SAY WORDINDEX('Now is the time ',3) → 8

**WORDLENGTH()****WORDLENGTH(stringa,n)**

Restituisce la lunghezza dell'nesima parola nell'argomento stringa. Per esempio:

SAY WORDLENGTH('one two three',3) → 5

**WORDS()****WORDS(stringa)**

Restituisce il numero di parole dell'argomento stringa. Per esempio:

SAY WORDS("You don't SAY!") → 3

**WRITECH()****WRITECH(file,stringa)**

Scrive l'argomento stringa nel file logico dato. Il valore riportato è il numero effettivo di caratteri scritti. Per esempio:

SAY WRITECH('output','Testing') → 7

**WRITELN()****WRITELN(file,stringa)**

Scrive l'argomento stringa nel file logico dato con l'aggiunta di una "nuova riga". Il valore restituito è il numero effettivo di caratteri scritti. Per esempio:

SAY WRITELN('output','Testing') → 8

**X2C()**

X2C(stringa)

Converte una stringa di cifre esadecimali nella rappresentazione carattere (pacchetto). Gli spazi sono permessi nella stringa argomento ai limiti dei byte:

```
SAY X2C('12ab')    → '12ab'x
SAY X2C('12 ab')   → '12ab'x
SAY X2C(61)        → a
```

**X2D()**

X2D(esadecimale,cifre)

Converte un numero esadecimale in decimale. Per esempio:

```
SAY X2D('1f')    → 31
```

**XRANGE()**

XRANGE([inizio][,fine])

Genera una stringa consistente di tutti i caratteri numericamente presenti tra i valori di inizio e fine specificati. Il carattere di inizio predefinito è '00'x, e quello di fine è 'FF'x. Soltanto il primo carattere degli argomenti di inizio e di fine è significativo. Per esempio:

```
SAY XRANGE()      → '00010203 . . . FDFEFF'x
SAY XRANGE('a','f') → 'abcdef'
SAY XRANGE(, '0A'x) → '000102030405060708090A'x
```

## Programma esemplificativo

Il seguente programma illustra molte delle funzioni integrate che manipolano le stringhe di caratteri.

```
/*Questo programma ARExx indica l'effetto delle
funzioni integrate che modificano le stringhe. Le
funzioni si dividono in due gruppi, di cui uno
manipola i singoli caratteri e l'altro manipola
intero stringhe.*/

teststring1 = " every good boy does fine "

/*Il primo gruppo è composto delle funzioni STRIP(),
COMPRESS(), SPACE(), TRIM(), TRANSLATE(), DELSTR(),
DELWORD(), INSERT(), OVERLAY(), e REVERSE().*/

/*STRIP() rimuove soltanto i caratteri iniziali e finali.*/

/*Stampa la stringa originale, per la comparazione.
Inseriamo un punto alla fine della stringa, per poter
vedere ciò che accade agli spazi alla fine della
stringa.*/
SAY " every good boy does fine "

/*La stessa stringa con gli spazi iniziali/finali
eliminati*/
SAY STRIP(" every good boy does fine ")."

/*Un tentativo fallito di rimuovere le "e" iniziali e
finali*/
SAY STRIP(" every good boy does fine
",,"e")."

/*Le "e" erano protette dagli spazi iniziali e
finali. La loro rimozione le espone agli effetti di
STRIP()*/
SAY STRIP("every good boy does fine",,"e")."

/*Rimuove le "e" e gli spazi dalla stringa
originale*/
SAY STRIP(" every good boy does fine ",,"
e")."
```

```
/*Passiamo ad usare la variabile "teststring1",
definita sopra. Rimuove soltanto gli spazi finali
nella stringa testo.*/
SAY STRIP(teststring1, T)". "

/*Rimuovere gli spazi finali e le "e"*/
SAY STRIP(teststring1,T," e)". "

/*Compress() rimuove i caratteri in qualsiasi punto
della stringa. Questo rimuove tutti gli spazi dalla
stringa di verifica "teststring".*/
SAY COMPRESS(teststring1)

CALL TIME('r')
SAY TIME('Civil') /*Formato ore 0-12 HH:MM{AM | PM}*/
SAY TIME('h') /*Ore dalla mezzanotte*/
SAY TIME('m') /*Minuti dalla mezzanotte*/
SAY TIME('s') /*Secondi dalla mezzanotte*/
SAY TIME('e') /*Tempo trascorso dall'inizio
programma*/

/*Funzione: TRACE Uso: TRACE([opzione])*/
SAY TRACE()
SAY TRACE(TRACE()) /*Lascia immutato*/
/*Funzione:TRANSLATE
Uso:
TRANSLATE(stringa[,uscita][,entrata][,riempimento])*/
SAY TRANSLATE('aBCdef') /*Traduce in Maiuscolo*/
SAY TRANSLATE ('abcdef','1234')
SAY TRANSLATE('654321','abcdef','123456')
SAY TRANSLATE('abcdef','123','abcdef','+')

/*Funzione: TRIM Uso: TRIM(stringa)*/
SAY TRIM(' abc ')

/*Funzione: TRUNC Uso: TRUNC(numero[,posti])*/
SAY TRUNC(123.456)
SAY '$'TRUNC(134566.123,2)

/*Funzione: UPPER Uso: UPPER(stringa)*/
SAY UPPER('aBCdef12')

/*Funzione: VALUE Uso: VALUE(nome)*/
abc = 'mio nome'
SAY VALUE('abc')

/*Funzione: VERIFY Uso: VERIFY(stringa,lista[, 'M'])*/
SAY VERIFY('123a45','0123456789')
```

```
SAY VERIFY('abc3de','012456789','M')

/*Funzione: WORD  Uso: WORD(stringa,n)*/
SAY WORD('Now is the time',3)

/*Funzione: WORDINDEX  Uso: WORDINDEX(stringa,n)*/
SAY WORDINDEX('Now is the time ',3)

/*Funzione: WORDLENGTH  Uso: WORDLENGTH(stringa,n)*/
SAY WORDLENGTH('Now is the time ',4)

/*Funzione: WORDS  Uso: WORDS(stringa)*/
SAY WORDS('Now is the time')
/*Funzione: WRITECH  Uso: WRITECH(logico,stringa)*/
IF OPEN('test','ram:test$$','W') THEN DO
    SAY WRITECH('test','messaggio') /*Scrive la
stringa*/
    CALL CLOSE 'test'
END

/*Funzione: WRITELN  Uso: WRITELN(logico,stringa)*/
IF OPEN('test','ram:test$$','W') THEN DO
    SAY WRITELN('test','messaggio')
    /*Scrive la stringa (con nuova riga)*/
    CALL CLOSE 'test'
END

/*Funzione: X2C  Uso: X2C(stringa esadecimale)*/
SAY X2C('616263') /*Converte in carattere
(pacchetto)*/

/*Funzione: XRANGE  Uso: XRANGE([inizio][,fine])*/
SAY C2X(xrange('f0'x))
SAY XRANGE('a','g')
EXIT
```

Il risultato del programma 13 è:

```

every good boy does fine
every good boy does fine.
every good boy does fine .
very good boy does fin.
very good boy does fin.
every good boy does fine.
every good boy does fin.
everygoodboydoesfine /
1:23PM /*Questi risultati variano conformemente*/
13 /*all'ora di esecuzione.*/
803
48199
0.80
N
N
ABCDEF
abcdef
fedcba
123+++
abc
123
$134566.12
ABCDEF12
mio nome
4
0
the
8
4
4
7
8
abc
F0F1F2F3F4F5F6F7F8F9FAFBFCFDFE FF
abcdefg

```

## ***Funzioni REXXSupport.Library***

Le funzioni elencate in questo capitolo fanno parte di REXXSupport.library (libreria di supporto REXX). Possono essere usate soltanto se la libreria è aperta. La procedura per aprire la libreria è la seguente.

**Programma 14. OpenLibrary.rexx**

```
/*Aggiunge rexxsupport.library se non è ancora
aperta.*/
IF ~ SHOW('L', "rexxsupport.library") THEN DO
/*Se la libreria non è ancora aperta, cercare di
aprirla.*/
IF ADDLIB('rexxsupport.library', 0, -30,0)
THEN SAY "Aperta rexxsupport.library."
ELSE DO
    SAY 'Libreria di supporto ARexx non disponibile,
uscita'
    EXIT 10      /*Termina se ADDLIB() fallisce*/
END
END
```

**ALLOCMEM()****ALLOCMEM(lunghezza[,attributo])**

Assegna un blocco di memoria della lunghezza specificata nel polo di memoria libera del sistema e ne restituisce l'indirizzo come stringa da 4 byte. Il parametro attributo opzionale deve essere un indicatore di assegnazione di memoria EXEC standard, fornito come stringa da 4 byte. L'attributo predefinito è per memoria "PUBBLICA" (non liberata). Vedere il *Manuale di Riferimento Amiga ROM Kernel: Librerie* per informazioni sui tipi di memoria e i parametri attributo.

Questa funzione deve essere usata ogniquale volta la memoria viene assegnata per l'uso da parte di programmi esterni. È compito dell'utente liberare lo spazio di memoria quando non è più necessario. Vedere anche FREEMEM(). Per esempio:

```
SAY C2X(ALLOCMEM(1000))      → 00050000
SAY C2X(ALLOCMEM (1000,'00 01 00 0 1'X )) → 00228400
/*1000 byte di memoria pubblica allocata*/
```

**CLOSEPORT()**

CLOSEPORT(nome)

Chiude la porta messaggi specificata dall'argomento nome, che deve essere stata assegnata da una chiamata a OPENPORT() all'interno del programma ARexx corrente. Qualsiasi messaggio ricevuto ma non rispedito con REPLY viene automaticamente rinviato con il codice di restituzione impostato a 10. Vedere anche OPENPORT(). Per esempio:

```
CALL CLOSEPORT myport
```

**FREEMEM()**

FREEMEM(indirizzo, lunghezza)

Libera un blocco di memoria della lunghezza data per l'elenco dei blocchi liberi del sistema. Il parametro indirizzo è una stringa da 4 byte, ottenuta generalmente da una chiamata precedente ad ALLOCMEM(). FREEMEM() non può essere usato per liberare la memoria assegnata usando GETSPACE(), il programma di assegnazione memoria interna di ARexx. Il valore restituito è l'indicatore di riuscita booleano. Vedere anche ALLOCMEM(). Per esempio:

```
MemoryRequest = 1024
MyMem = ALLOCMEM(MemoryRequest)
SAY C2X(MyMem) → 07C987B0
SAY FREEMEM(MyMem, MemoryRequest) → 1
/*Or: SAY FREEMEM('07C987B0'x,1024)*/
```

**Avvertenza** Prima che termini il programma bisogna usare la funzione FREEMEM() per liberare la quantità esatta di memoria assegnata ad ogni ALLOCMEM(). Altrimenti il sistema può bloccarsi o la memoria non è disponibile fino al riavvio.

**GETARG()**

GETARG(pacchetto[,n])

Estrae un comando, un nome funzione o una stringa argomento da un pacchetto messaggi. L'argomento pacchetto deve essere un indirizzo da 4 byte ottenuto da una chiamata precedente a GETPKT(). L'argomento opzionale [n] specifica la posizione con-



tenente la stringa da estrarre e deve essere inferiore o uguale al conteggio effettivo degli argomenti del pacchetto. I comandi e i nomi funzioni sono sempre nella posizione 0; i pacchetti funzioni possono avere stringhe argomento nella posizione 1-15. Per esempio:

```
command = GETARG(packet)
function = GETARG(packet,0) /*nome stringa*/
arg1 = GETARG(packet,1)    /*primo argomento*/
```

## **GETPKT()**

**GETPKT(nome)**

Controlla la porta messaggi specificata dall'argomento nome per verificare la presenza messaggi. La porta messaggi indicata deve essere aperta da una chiamata precedente ad **OPENPORT()** all'interno del programma **ARexx** attuale. Il valore restituito è l'indirizzo da 4 byte del primo pacchetto messaggi oppure '0000 0000'x in assenza di pacchetti.

La funzione risponde immediatamente in presenza o in assenza di pacchetto nella coda della porta messaggi. I programmi non devono essere creati come "ciclo di occupazione" (busy loop) sulla porta messaggi. Se non vi è lavoro utile prima che arrivi il successivo pacchetto messaggi, il programma deve chiamare **WAITPKT()** e consentire ad altri compiti di procedere. Vedere anche **WAITPKT()**. Per esempio:

```
packet = GETPKT('MyPort')
```

## **OPENPORT()**

**OPENPORT(nome)**

Crea una porta messaggi pubblica con il nome dato. Il valore booleano restituito indica se l'apertura della porta è riuscita. Se esiste già un'altra porta con lo stesso nome o se non è stato possibile assegnare un bit segnale avviene un errore di inizializzazione. La porta messaggi è allocata come nodo **Risorsa Porta** ed è collegata alla struttura dati globali del programma. Le porte vengono chiuse automaticamente quando esce il programma e nessun messaggio pendente è rinviato al mittente. Vedere anche **CLOSEPORT()**. Per esempio:

```
success = OPENPORT("MyPort")
```

**REPLY()****REPLY(pacchetto,rc)**

Restituisce un messaggio al mittente, con il campo risultato primario impostato sul valore dato dall'argomento rc. Il risultato secondario viene azzerato. L'argomento pacchetto deve essere fornito come indirizzo da 4 byte e l'argomento rc deve essere un numero intero. Per esempio:

```
CALL REPLY(packet,10) /*Ritorno errore*/
```

**SHOWDIR()**
**SHOWDIR(directory['ALL' | 'FILE' | 'DIR']  
[riempimento])**

Restituisce il contenuto della directory specificata come stringa di nomi separati da spazi. Il secondo parametro è una parola chiave opzione che determina se saranno incluse tutte le voci, soltanto i file o soltanto le subdirectory. Per esempio:

```
SAY SHOWDIR('SYS:REXXC', 'f', ';')
```

```
→ WaitForPort;TS;TE;TCO;RXSET;RXLIB;RXC;RX;HI
```

**SHOWLIST()**
**SHOWLIST({'A' | 'D' | 'H' | 'I' | 'L' | 'M' | 'P' |  
'R' | 'S' | 'T' | 'V' | 'W'})[,nome][,riempimento])**

L'argomento si introduce digitandone la lettera iniziale. Gli argomenti sono:

- A**    ASSIGNS and Assigned Device (ASSEGNA e Dispositivo Assegnato)
- D**    Device Drivers (Programmi di Gestione)
- H**    Handlers (Gestori)
- I**    Interrupts (Interruzioni)
- L**    Libraries (Librerie)
- M**    Memory List Items (Voci di Elenchi di Memoria)
- P**    Ports (Porte)
- R**    Resources (Risorse)
- S**    Semaphores (Semafori)

- T**     Tasks (Ready) (Processi (Pronti))
- V**     Volume Names (Nomi di Volumi)
- W**     Waiting Tasks (Processi in Attesa)

Se viene fornito soltanto un argomento, SHOWLIST() riporta una stringa separata da spazi. Se viene fornito un carattere di riempimento, i nomi saranno separati dal carattere anziché da spazi. Se viene fornito il parametro nome, SHOWLIST() restituisce un valore booleano che indica se l'elenco specificato contiene quel nome. Per i nomi viene eseguito il confronto maiuscole. Per fornire un'immagine accurata dell'elenco attuale, il cambio di processi è vietato quando viene letto l'elenco. Per esempio:

```
SAY SHOWLIST('P')      → REXX MyCon
SAY SHOWLIST('P',,';') → REXX;MyCon
SAY SHOWLIST('P','REXX') → 1
```

**STATEF()****STATEF(nomefile)**

Restituisce una stringa contenente informazioni su un file esterno. La stringa viene formattata come:

```
"{DIR | FILE} length blocks protection days minutes
ticks comment."
```

L'indicatore (token) lunghezza dà la lunghezza del file in byte, mentre l'indicatore blocco specifica la lunghezza del file in blocchi.

```
SAY STATEF("LIBS:REXXSupport.library")
/*might give "File 2524 5 ----RW-D 4866 817 2088*/
```

**WAITPKT()****WAITPKT(nome)**

Attende che sia ricevuto un messaggio alla porta specificata (con nome), che deve essere stata aperta da una chiamata a OPENPORT() all'interno del programma ARexx corrente. Il valore booleano restituito indica se nella porta è disponibile un pacchetto messaggi. Normalmente il valore restituito è 1 (Vero), poiché la funzione attende che si verifichi un evento alla porta messaggi.

Il pacchetto deve quindi essere rimosso da una chiamata GETPKT() e infine restituito usando la funzione REPLY(). Qualsiasi pacchetto messaggi ricevuto ma non restituito, quando esce dalla funzione REPLY, viene automaticamente restituito un programma ARexx al mittente con codice 10. Per esempio:

```
CALL WAITPKT 'MyPort'    /*Aspetta pacchetto*/
```

## **Capitolo 6**

# **Diagnostica**

---

ARexx fornisce le utilità di analisi e diagnostica a livello sorgente. Queste utilità visualizzano istruzioni selezionate di un programma durante l'esecuzione. Durante l'analisi di una clausola sulla console sono visualizzati il numero linea, il testo sorgente, e le informazioni relative.

L'interruzione interna consente al programma ARexx di rilevare alcuni eventi sincroni o asincroni e prendere provvedimenti speciali quando avvengono tali eventi. Eventi come un errore di sintassi o una richiesta di arresto esterna che normalmente causano l'uscita dal programma possono invece essere interrotti per eseguire le azioni correttive.

## **Analisi**

L'operazione di analisi seleziona quali clausole sorgenti saranno analizzate e prevede due indicatori di modifica che controllano l'inibizione comando e l'analisi interattiva. Le opzioni Trace possono essere abbreviate ad una lettera. Le opzioni di Trace sono:

### **ALL (TUTTE)**

Vengono analizzate tutte le clausole

### **BACKGROUND (BASE)**

L'analisi non viene eseguita e non è possibile imporre al programma di effettuare un'analisi interattiva.

### **COMMANDS (COMANDI)**

Tutte le clausole comando vengono analizzate prima di essere inviate all'ospite esterno. I codici risultanti diversi da zero sono visualizzati sulla console.

<b>ERRORS (ERRORI)</b>	I comandi che generano un codice risultante diverso da zero sono analizzati dopo l'esecuzione della clausola.
<b>INTERMEDIATES (INTERMEDI)</b>	Tutte le clausole vengono analizzate e i risultati intermedi sono visualizzati durante la risoluzione dell'espressione. Questi comprendono valori relativi a variabili, nomi composti espansi e risultati di chiamate funzione.
<b>LABELS (ETICHETTE)</b>	Vengono analizzate tutte le clausole etichetta durante l'esecuzione. È visualizzata una etichetta ogni volta che si verifica un trasferimento di controllo.
<b>NORMAL Default (NORMALE Predefinito)</b>	Le clausole comando con codici di restituzione che superano il livello di errore corrente sono analizzate dopo l'esecuzione e la visualizzazione di un messaggio d'errore.
<b>OFF</b>	Termine analisi.
<b>RESULTS (RISULTATI)</b>	Tutte le clausole sono analizzate prima dell'esecuzione e viene visualizzato il risultato finale di ogni espressione. Sono visualizzati anche i valori assegnati alle variabili dalle istruzioni ARG, PARSE o PULL . Questa opzione è raccomandata per verifiche generali.
<b>SCAN (SCANSIONE)</b>	Opzione speciale che analizza tutte le clausole e controlla gli errori, ma sopprime l'esecuzione vera e propria delle istruzioni. È utile come fase filtro preliminare per un programma appena creato.

Il modo analisi può essere impostato usando l'istruzione **TRACE** o la funzione **TRACE()** integrata. L'analisi può essere disabilitata selettivamente all'interno del programma per saltare parti di programma precedentemente controllate.

Ogni riga di analisi visualizzata sulla console è indentata per indicare il livello di controllo (annidamento) effettivo di quella clausola ed è identificata da uno speciale codice di tre caratteri, come indicato nella Tabella 6-1. La sorgente di ogni clausola è preceduta dal rispettivo numero di riga del programma. I risultati finali o intermedi dell'espressione sono racchiusi tra virgolette affinché gli spazi iniziali e finali siano evidenti.

Tabella 6-1. Codici Speciali a Tre Caratteri

Codice	Valori visualizzati
+++	Errore di comando o di sintassi
>C>	Nome composto espanso
>F>	Risultato di una chiamata funzione
>L>	Clausola etichetta
>O>	Risultato di operazione diadica
>P>	Risultato di operazione con prefisso
>U>	Variabile non inizializzata
>V>	Valore di una variabile
>>>	Risultato di espressione o modello
>.>	Valore indicatore "segnaposto"

## Uscita dell'analisi

L'uscita dell'analisi dal programma viene sempre inviata ad uno di due flussi logici. L'interprete prima cerca il flusso `STDERR` e vi dirige l'eventuale uscita. In caso contrario l'uscita va al flusso uscita standard `STDOUT` e sarà interlacciata con l'uscita normale della console programma. I flussi `STDERR` e `STDOUT` possono essere aperti e chiusi sotto controllo del programma, quindi il programmatore ha il controllo completo sulla destinazione dell'uscita analisi.

In alcuni casi il programma può non avere un flusso di uscita predefinita. Ad esempio, un programma richiamato da un'applicazione ospite che non fornisca flussi di ingresso e uscita non ha la console di uscita. Per fornire un'utilità analisi per tali programmi, il processo residente può aprire una speciale console di analisi globale, utilizzabile da parte di qualunque programma attivo. All'apertura di questa console, l'interprete apre automaticamente il flusso `STDERR` per ogni programma `ARexx` in cui `STDERR` non è definita correntemente. Quindi il programma indirizza la sua uscita analisi verso il nuovo flusso.

La console di analisi globale può essere aperta e chiusa usando l'utilità di comando `TCO`. I programmi `ARexx` devieranno automa-

ticamente la loro uscita analisi a una nuova finestra, aperta come console standard di AmigaDOS e che l'utente può spostare e ridimensionare secondo le necessità.

La console analisi serve anche come flusso di ingresso per i programmi durante l'analisi interattiva. Quando un programma si interrompe temporaneamente per analizzare l'ingresso, la riga di entrata deve essere immessa nella console di analisi. Un indefinito numero di programmi può usare contemporaneamente la console di analisi, anche se si raccomanda di analizzare soltanto un programma per volta.

La console globale può essere chiusa usando il comando TCC . La chiusura è ritardata finché non sono soddisfatte tutte le richieste di lettura della console. È effettivamente chiusa quando tutti i programmi attivi indicano che non utilizzano più la console.

## ***Inibizione comando***

ARexx fornisce il modo analisi denominato inibizione comando, che sopprime i comandi ospite. In questo modo le clausole comando sono calcolate normalmente, ma il comando non viene effettivamente inviato all'ospite esterno, e il codice di restituzione è impostato a zero. Questo fornisce un modo per provare programmi che inviano comandi potenzialmente distruttivi, come la cancellazione di file o la formattazione dischi. L'inibizione comando non riguarda le clausole comando immesse in modo interattivo. Questi comandi vengono sempre eseguiti, ma il valore della variabile speciale RC resta immutato.

L'inibizione comando può essere usata con qualunque opzione analisi. È controllata dal carattere "!", che può apparire da solo o precedere qualsiasi opzione alfabetica in un'istruzione TRACE. Ogni occorrenza del carattere "!" commuta il modo inibizione presente. L'inibizione comando è disabilitata quando l'analisi è impostata su OFF.



## ***Analisi interattiva***

L'analisi interattiva è un'utilità diagnostica che permette all'utente di immettere istruzioni sorgenti durante l'esecuzione di un programma. Queste istruzioni possono essere usate per esaminare o modificare i valori variabili, per immettere comandi o altrimenti interagire col programma. Qualunque istruzione di linguaggio valida può essere immessa in modo interattivo, con le stesse regole e restrizioni che si applicano all'istruzione INTERPRET. In particolare, le istruzioni composte come DO e SELECT devono essere complete all'interno della riga immessa.

L'analisi interattiva può essere usata con qualsiasi opzione analisi. Nel modo analisi interattivo, l'interprete fa una pausa dopo ogni clausola analizzata e chiede un'ingresso con il codice "+++". Ad ogni pausa sono possibili tre tipi di risposte dell'utente:

- Se viene immessa una riga nulla, il programma continua fino al punto pausa successivo.
- Se viene immesso il carattere "=", la clausola precedente è nuovamente eseguita.
- Qualunque altro ingresso viene trattato come istruzione diagnostica e viene letta ed eseguita.

I punti di pausa durante l'analisi interattiva sono determinati dall'opzione analisi correntemente attiva, in quanto l'interprete fa una pausa soltanto dopo una clausola analizzata. Tuttavia, alcune istruzioni non possono essere rieseguite in modo sicuro, quindi l'interprete non farà pause dopo averle eseguite. Le istruzioni "senza pause" sono CALL, DO, ELSE, IF, THEN, e OTHERWISE. L'interprete fa la pausa dopo clausole che abbiano generato un errore di esecuzione.

L'analisi interattiva è gestita dal carattere "?", da solo o con un'opzione analisi alfabetica. Un'opzione può essere preceduta da un numero indefinito di caratteri "?" ed ogni occorrenza commuta il modo presente. Ad esempio, se le opzioni analisi correnti sono NORMAL, allora "TRACE ?R" imposta l'opzione su RESULTS e seleziona il modo analisi interattivo. Un'ulteriore "TRACE ?" disattiva l'analisi interattiva.

## **Gestione errori**

L'interprete ARexx fornisce una speciale gestione errori mentre esegue le istruzioni di diagnostica. Gli errori rilevati durante la diagnostica interattiva vengono registrati ma non fanno terminare il programma. Questa speciale gestione è valida soltanto per le istruzioni immesse in modo interattivo.

ARexx disabilita anche gli indicatori di interruzione interni durante la diagnostica interattiva. Ciò evita il trasferimento accidentale del controllo dovuto a un errore o a una variabile non inizializzata. Tuttavia, se viene immessa un'istruzione "SIGNAL label", il trasferimento ed eventuali ingressi interattivi rimanenti vengono abbandonati. L'istruzione SIGNAL può ancora essere usata per modificare gli indicatori di interruzione, e le nuove impostazioni saranno valide quando l'interprete tornerà alla normale elaborazione.

Ogni compito ARexx inizializza il proprio livello di errore comando al livello errore del cliente (di solito 10) e sopprime la stampa degli errori. Il livello di errore può essere modificato usando `OPTIONS FAILAT`. Gli errore comando ( $RC > 0$ ) e i fallimenti ( $RC \geq \text{FAILAT}$ ) possono essere rilevati separatamente usando `SIGNAL ON ERROR` e `SIGNAL ON FAILURE`.

## **Indicatore di analisi esterno**

ARexx gestisce un indicatore di analisi esterno usato per obbligare i programmi ad usare il modo analisi interattivo. Quando è impostato questo indicatore di analisi, usando l'utilità di comando `TS`, qualsiasi programma che non sia ancora nel modo analisi interattiva vi entra immediatamente. L'opzione analisi interna viene impostata su `RESULTS` a meno che non sia correntemente impostata su `INTERMEDIATES` oppure `SCAN`, nel qual caso rimane immutata. I programmi chiamati mentre è impostato l'indicatore di analisi esterno iniziano l'esecuzione nel modo analisi interattiva.

L'indicatore di analisi esterno può riprendere il controllo sui programmi ciclici o senza risposta. Un programma nel modo analisi interattiva consente all'utente di controllare le istruzioni del programma e diagnosticare il problema. L'indicatore di analisi esterno è un indicatore globale, ha effetto su tutti i programmi corrente-

mente attivi. L'indicatore di analisi rimane impostato finché viene disabilitato usando l'utilità comando TE . Ogni programma conserva una copia interna dell'ultimo stato dell'indicatore di analisi e ne imposta l'opzione analisi su OFF quando disabilitato. I programmi nel modo analisi BACKGROUND non rispondono all'indicatore di analisi esterno.

## **Interruzioni**

ARexx gestisce un sistema di interruzione interno usato per rilevare e catturare certe condizioni di errore. Quando viene abilitata un'interruzione e si presenta la condizione corrispondente, avviene un trasferimento di controllo all'etichetta specifica di quell'interruzione. Questo consente al programma di mantenere il controllo in circostanze che altrimenti farebbero terminare il programma. Le condizioni di interruzione possono essere causate da eventi sincroni come un errore di sintassi o da eventi asincroni come una richiesta di sospensione Ctrl+C.

Nota        Queste interruzioni interne sono completamente separate dal sistema di interruzione hardware gestito dal sistema operativo EXEC.

Il nome assegnato ad ogni interruzione è effettivamente l'etichetta alla quale verrà trasferito il controllo. In tal modo, un'interruzione SYNTAX trasferirà il controllo all'etichetta "SYNTAX:". Le interruzioni possono essere abilitate o disabilitate usando l'istruzione SIGNAL. Ad esempio, l'istruzione "SIGNAL ON SYNTAX" abilita l'interruzione SYNTAX.

Le interruzioni supportate da ARexx sono:

<b>BREAK_C</b>	Cattura (rileva e tratta come segnale, e non come uscita normale) una richiesta di sospensione Ctrl+C generata da AmigaDOS. Se l'interruzione non è abilitata, il programma termina immediatamente col messaggio d'errore "Execution halted" (esecuzione interrotta) e restituisce il codice d'errore 2.
<b>BREAK_D</b>	Cattura una richiesta di sospensione Ctrl+D inviata da AmigaDOS. La richiesta di sospensione viene ignorata se l'interruzione non è abilitata.
<b>BREAK_E</b>	Cattura una richiesta di sospensione Ctrl+E inviata da AmigaDOS. La richiesta di sospensione viene ignorata se l'interruzione non è abilitata.
<b>BREAK_F</b>	Cattura una richiesta di sospensione Ctrl+F inviata da AmigaDOS. La richiesta di sospensione viene ignorata se l'interruzione non è abilitata.
<b>ERROR</b>	Viene generata da qualsiasi comando ospite che restituisce un codice diverso da zero.
<b>HALT</b>	Se abilitata cattura una richiesta di arresto esterna. In caso contrario il programma termina immediatamente con il messaggio d'errore "Execution halted" (esecuzione interrotta).
<b>IOERR</b>	Se abilitata cattura gli errori rilevati dal sistema I/O.
<b>NOVALUE</b>	Se abilitata avviene un'interruzione a causa di una variabile non inizializzata. Può essere usata all'interno di un'espressione, nell'istruzione UPPER o con la funzione integrata VALUE().
<b>SYNTAX</b>	Interruzione generata da un errore di sintassi o di esecuzione. Però non è possibile catturare tutti questi errori. In particolare, alcuni errori avvengono prima dell'esecuzione programma e quelli individuati dall'interfaccia esterna di ARexx non possono essere catturati dall'interruzione SINTAX

Quando un'interruzione impone un trasferimento di controllo, tutti i gruppi di comandi correntemente attivi vengono eliminati e l'interruzione che ha provocato il trasferimento viene disabilitata.

Quest'ultima azione è necessaria per evitare un possibile ciclo di interruzioni ricorsivo. Soltanto le strutture di controllo dell'ambiente corrente sono interessate, quindi un'interruzione generata all'interno di una funzione non influisce sull'ambiente del programma chiamante.

Un'interruzione ha un effetto su due variabili speciali:

<b>SIGL</b>	Sempre impostata sul numero di riga corrente prima che avvenga il trasferimento del controllo. Ciò consente di determinare la riga del sorgente eseguita.
<b>RC</b>	Viene impostata sul codice d'errore che ha provocato tale condizione. Per le interruzioni ERROR, questo valore è un codice di restituzione comando e generalmente è interpretato come livello di gravità errore. Il valore delle interruzioni SYNTAX è sempre un codice d'errore ARexx.

Le interruzioni sono utili per consentire la correzione errori. Cioè informare i programmi esterni che è avvenuto un errore o ulteriori diagnosi per isolare il problema. Il programma 15 invia un comando "messaggio" ad un ospite esterno chiamato "MyEdit" al rilevamento di un errore di sintassi.

***Programma 15. Interrupt.rexx***

```
/*Un programma macro per 'MyEdit'*/  
SIGNAL ON SYNTAX      /*Abilita interruzione*/  
(normal processing)  
EXIT  
SYNTAX:                /*Trovato errore di sintassi*/  
ADDRESS 'MyEdit'  
'message' 'error' RC errortext(RC)  
EXIT 10
```



## Capitolo 7

# Analisi sintattica

---

L'analisi sintattica estrae sottostringhe da una stringa e le assegna a variabili. L'analisi sintattica è effettuata usando l'istruzione `PARSE` o le sue varianti `ARG` e `PULL`. L'ingresso per l'operazione è la stringa di analisi sintattica e può provenire da diverse sorgenti comprese le stringhe argomento, espressioni e la console.

Le funzioni di manipolazione stringhe come `SUBSTR()` e `INDEX()` possono anche essere usate per l'analisi sintattica, ma l'istruzione `PARSE` è più efficiente soprattutto per estrarre molti campi da una stringa.

## Modelli

L'analisi sintattica è gestita da un modello, cioè un gruppo di indicatori che specifica sia le variabili cui assegnare valori sia il modo per determinare le stringhe valore. La disposizione degli indicatori nel modello determina se uno dei due oggetti basilari del modello è un marcatore o un obiettivo.

<b>Marcatore</b>	Determina la posizione iniziale e finale della stringa di analisi sintattica o la posizione di scansione.
<b>Obiettivo</b>	Un simbolo cui è assegnato un valore dall'operazione di analisi. Il valore è una sottostringa determinata dalle posizioni dei marcatori.

## **Marcatori**

Vi sono tre tipi di oggetti marcatori:

<b>Marcatori assoluti</b>	Posizione indice effettiva nella stringa di analisi sintattica.
<b>Marcatori relativi</b>	Spostamento positivo o negativo dalla posizione attuale.
<b>Marcatori di configurazione</b>	Confrontano la configurazione con la stringa di analisi sintattica iniziando dalla posizione di scansione corrente.

## **Obiettivi**

Gli obiettivi, come i marcatori, possono influire sulla posizione di scansione se le stringhe valore sono estratte mediante identazione (tokenizzazione). L'analisi sintattica mediante identazione estrae parole (identificatori) dalla stringa di analisi sintattica e viene usata ogni volta che un obiettivo è seguito immediatamente da un altro obiettivo. Durante l'identazione la posizione di scansione attuale viene fatta avanzare davanti a tutti gli spazi fino all'inizio della parola successiva. L'in-dice di fine è la posizione immediatamente successiva alla fine della parola, quindi la stringa valore non ha spazi né iniziali né finali.

Gli obiettivi sono specificati dai simboli variabili. Il segnaposto denotato da un punto (.), è un tipo speciale di obiettivo e agisce come un obiettivo normale a meno che non abbia un valore assegnato.



## Oggetti modello

Ogni oggetto modello è specificato da uno o più indicatori:

<b>Simboli</b>	Un simbolo può specificare un obiettivo o un marcatore. E' un marcatore se segue un operatore (+, - or =) e il valore sim-bolo viene usato come posizione assoluta o relativa. I simboli racchiusi tra parentesi specificano i marcatori di configura-zione, e il valore del simbolo viene usato come stringa di con-figurazione. Specifica un obiettivo se non si verifica nessuno dei casi precedenti e il simbolo è una variabile. I simboli fissi specificano sempre marcatori assoluti e devono essere numeri interi, eccetto il simbolo punto (.), che definisce un obiettivo segnaposto.
<b>Stringhe</b>	Una stringa rappresenta sempre un marcatore di configura-zione.
<b>Parentesi</b>	Un simbolo racchiuso tra parentesi è un marcatore di configu-razione e il valore del simbolo viene usato come stringa di configurazione. Mentre il simbolo può essere fisso o vari-abile, di solito è una variabile. Una configurazione fissa può essere data più semplicemente come stringa.
<b>Operatori</b>	I tre operatori (+, - e =) sono validi all'interno di un modello e devono essere seguiti da un simbolo fisso o variabile. Il valore del simbolo usato come marcatore deve quindi rap-presentare un numero intero. Gli operatori "+" e "-" indicano un marcatore relativo, il cui valore viene negato dall'operatore "-". L'operatore "=" indica un marcatore assoluto ed è opzio-nale se il marcatore è definito da un simbolo fisso.
<b>Virgole</b>	La virgola (,) segna la fine di un modello ed è usata come se-paratore quando vengono forniti più modelli con un'istruzione. L'interprete ottiene una nuova stringa di analisi sintattica prima di elaborare ogni modello successivo. Per alcune opzioni sorgente, la nuova stringa sarà identica a quella precedente. Le opzioni ARG, EXTERNAL e PULL generalmente forniscono una stringa diversa; lo stesso vale per l'opzione VAR se la variabile è stata modificata.

L'analizzatore comandi interfaccia ARexx è stato generalizzato per riconoscere sequenze con doppio delimitatore entro un file stringa (fra virgolette). La convenzione delle virgolette è conveniente per programmi corti, ma è facile restare senza livelli di virgolette in programmi più lunghi. Gli apici e le virgolette in un programma REXX sono equivalenti, ma l'ambiente esterno può fare una distinzione.

AmigaDOS usa le virgolette. Le stringhe introdotte da una Shell devono iniziare con le virgolette, specialmente se si desidera inserire i punti e virgola. Per esempio:

```
RX "SAY 'Buon giorno. È una bella giornata!' "
→ Buon giorno. È una bella giornata!
```

```
RX "SAY '""Ciao!""'" → "Ciao!"
```

## ***Il processo di scansione***

Le posizioni di scansione sono espresse come indice nella stringa di analisi sintattica e possono variare da 1 (l'inizio della stringa) a tutta la lunghezza della stringa più 1 (la fine.)

La sottostringa specificata dai due indici di scansione include i caratteri dalla posizione di inizio alla posizione di fine esclusa. Ad esempio, gli indici da 1 a 10 specificano i caratteri 1-9 nella stringa di analisi sintattica. Se il secondo indice di scansione è minore di o uguale al primo; il resto della stringa di analisi sintattica è usato come sottostringa. Questo significa che una specifica del modello come:

```
PARSE ARG 1 all 1 first second
```

assegnerà l'intera stringa di analisi sintattica alla variabile ALL. Se l'indice di esame corrente si trova già alla fine della stringa di analisi sintattica, il resto è la stringa nulla.

Quando un marcatore di configurazione è confrontato con la stringa di analisi sintattica, la posizione del marcatore è l'indice del primo carattere della configurazione confrontata o la fine della stringa se non è stata trovata corrispondenza. La configurazione viene rimossa dalla stringa ogni volta che viene trovata una corrispondenza. Questa è l'unica operazione che modifica la stringa di analisi sintattica durante l'analisi sintattica.

I modelli sono scanditi da sinistra a destra con l'indice di scansione iniziale impostato su 1. La posizione di scansione viene aggiornata ogni volta che viene incontrato un oggetto marcatore, a seconda del tipo e del valore del marcatore. Ogni volta che viene trovato un oggetto obiettivo, il valore da assegnare è determinato scandendo l'og-

getto obiettivo successivo. Se l'oggetto successivo è un altro obiettivo, la stringa valore è determinata indentando la stringa di analisi sintattica. In caso contrario la posizione di esame corrente viene usata come inizio della stringa valore e la posizione specificata dal marcatore seguente è usata come punto di fine.

La scansione continua, e termina quando sono stati usati tutti gli oggetti del modello. A ogni obiettivo sarà assegnato un valore. Una volta che è stata esaurita la stringa di analisi sintattica, la stringa nulla viene assegnata a qualsiasi obiettivo rimanente..

## ***Esempi di analisi sintattica***

### ***Analisi sintattica mediante identazione***

I programmi per computer spesso dividono una stringa nelle parole o nelle parti che la compongono. Questa operazione è compiuta con un modello che consiste interamente di variabili (obiettivi).

```
/*Assume "hammer 1 each $600.00" inserito*/  
PULL item qty units cost .
```

In questo esempio la riga di ingresso dall'istruzione PULL viene divisa in parole assegnata alle variabili del modello. L'item variabile riceve il valore "hammer", qty viene impostato a "1", unit a "each" e cost riceve il valore "\$600.00". Al segnaposto finale (.) viene dato un valore nullo, poiché vi sono solo quattro parole nell'ingresso. Tuttavia, impone alla variabile cost un valore identato. Se il segnaposto fosse omesso, il resto della stringa di analisi sintattica sarebbe assegnato a cost, che in tal caso avrebbe uno spazio iniziale.

```
answer = "Only Amiga makes it possible."  
DO forever  
  PARSE VAR answer first answer  
/*Mette la prima parola in 'first' e il resto in  
'answer.'*/  
  IF first =='' THEN LEAVE  
  /*Si ferma se non ci sono altre parole*/  
  SAY answer  
END
```

La prima parola di una stringa viene rimossa e il resto viene rimesso nella stringa. Il processo continua finché non viene più estratta nessuna parola. L'uscita è:

```
Amiga makes it possible.  
makes it possible.  
it possible.  
possible.
```

## ***Analisi sintattica mediante configurazione***

I marcatori di configurazione estraggono i campi desiderati. In questo caso la "configurazione" è molto semplice — un solo carattere — ma in generale può essere una stringa arbitraria di qualsiasi lunghezza. Questa forma di analisi sintattica è utile ogni volta che i caratteri delimitatori sono presenti nella stringa di analisi sintattica.

```
/*Assume un argomento stringa "12,35.5,1" */  
ARG hours ',' rate ',' withhold
```

La configurazione viene effettivamente rimossa dalla stringa di analisi sintattica quando viene trovata una corrispondenza. Se la stringa di analisi sintattica viene analizzata nuovamente dall'inizio, la sua lunghezza e la sua struttura possono essere diverse rispetto all'inizio dell'analisi sintattica. Tuttavia, il sorgente originale della stringa non viene mai modificato.

## ***Analisi sintattica mediante marcatori di posizione***

L'analisi sintattica mediante i marcatori di posizione viene usata quando è noto che i file si trovano in certe posizioni nella stringa.

```
/* Record tipo: */  
/* Inizio: 1-5 */  
/* Lunghezza: 6-10 */  
/* Nome: @ (inizio, lunghezza)*/  
PARSE value record with 1 inizio +5 lunghezza +5  
=inizio nome +lunghezza
```

I dati in elaborazione contengono un campo di lunghezza variabile. La posizione di inizio e la lunghezza del campo si trovano nella prima parte del dato e un marcatore di posizione variabile viene usato per estrarre il campo desiderato.

La sequenza "=inizio" è un marcatore assoluto il cui valore è dato dalla posizione posta nella variabile inizio in precedenza durante l'esame. La sequenza "+lunghezza" fornisce la lunghezza effettiva del campo.

## **Modelli multipli**

E' possibile specificare più di un modello con un'istruzione separando i modelli con una virgola. L'istruzione ARG (o PARSE UPPER ARG) accede alle stringhe argomento fornite alle chiamate del programma. Ogni modello accede alla stringa argomento successiva. Per esempio:

```
/*Assume argomenti ('one two',12,sort)*/  
ARG first second,amount,action,option
```

Il primo modello consiste delle variabili first e second, che sono rispettivamente impostate sui valori "one" e "two". Nei due modelli successivi amount riceve il valore "12" e action viene impostato su "SORT". L'ultimo modello consiste della variabile "option", che è impostata sulla stringa nulla poiché erano disponibili soltanto tre argomenti.

Quando sono usati più modelli con le opzioni sorgente EXTERNAL o PULL, ogni modello aggiuntivo chiede all'utente una riga aggiuntiva di ingresso:

```
/*Legge last, first, e middle names e ssn*/  
PULL last ',' first middle,ssn
```

Vengono lette due righe di ingresso. E' atteso che la prima riga di ingresso abbia tre parole, la prima delle quali seguita da una virgola che vengono assegnate alle variabili "last", "first", e "middle". L'intera seconda riga di ingresso è assegnata alla variabile "ssn".

L'uso di più modelli può risultare utile anche con un'opzione sorgente che restituisce la stringa di analisi sintattica identica. Se il primo modello comprende dei marcatori di configurazione che alterano la stringa di analisi sintattica, i modelli successivi possono

ancora accedere alla stringa originale. Le stringhe di analisi sintattica successive ottenute dalla sorgente VALUE non provocano una nuova risoluzione dell'espressione, ma recuperano il risultato precedente.

## ***Appendice A***

# ***Messaggi di errore***

---

Quando l'interprete ARexx rileva un errore nel programma, restituisce un codice d'errore per indicare la natura del problema. Gli errori vengono normalmente gestiti visualizzando il codice d'errore, il numero di riga sorgente in cui l'errore si è verificato e un breve messaggio che spiega la condizione di errore. Il programma quindi termina a meno che l'interruzione SYNTAX non sia stata abilitata precedentemente (usando l'istruzione SIGNAL), il controllo ritorna al programma chiamante. La maggior parte degli errori di sintassi e di esecuzione possono essere catturati da SYNTAX, permettendo all'utente di mantenere e eseguire l'elaborazione speciale dell'errore richiesta. Certi errori sono generati all'esterno del contesto del programma ARexx e quindi non possono essere catturati nel modo suddetto. Riferirsi a "Analisi e interruzioni" per maggiori informazioni sulla cattura e elaborazione degli errori.

Associato ad ogni codice d'errore è un livello di gravità che viene dato al programma chiamante come codice di risultato primario. I valori di questi codici risultato sono 5 (poco grave), 10 (mediamente grave), 20 (molto grave). Il codice d'errore stesso è restituito come risultato secondario. La successiva propagazione o l'indicazione di questi codici dipende dal programma (chiamante) esterno.

Le pagine seguenti elencano tutti i codici d'errore definiti, insieme con il livello di gravità associato e la stringa messaggio.

Tabella A-1. Codici e messaggi d'errore

Errore	Codice risultato	Messaggio	Descrizione
1	5	Program not found	Il programma indicato non è stato trovato o non è un programma ARexx. I programmi ARexx devono iniziare con un commento ( <i>/*...*/</i> ). Questo errore è rilevato dall'inter-faccia esterna e non può essere catturato dall'interruzione SYNTAX.
2	10	Execution halted	È stata ricevuta una sospensione Ctrl+C o una richiesta di arresto esterna e il programma è terminato. Questo errore viene catturato se l'interruzione HALT è abilitata.
3	20	Insufficient memory	L'interprete non è stato in grado di allocare sufficiente memoria per l'operazione. Poiché lo spazio memoria è richiesto per tutte le operazioni di analisi sintattica e di esecuzione, questo errore generalmente non può essere catturato dall'interruzione SYNTAX.
4	10	Invalid character	È stato rilevato un carattere non ASCII nel programma. I codici di controllo e altri caratteri non ASCII possono essere usati nel programma definendoli come stringhe esadecimali o binarie. È un errore della fase di esame e non può essere catturato dall'interruzione SYNTAX.



<b>Errore</b>	<b>Codice risultato</b>	<b>Messaggio</b>	<b>Descrizione</b>
5	10	Unmatched quote	Manca l'apice o le virgolette di chiusura. Controllare che ciascuna stringa sia delimitata correttamente. È un errore della fase di esame e non può essere catturato da SYNTAX.
6	10	Unterminated comment	Non è stata rilevata la sequenza di chiusura */ per un campo commento. I commenti possono essere annidati, quindi ogni sequenza /* deve essere completata da */. È un errore della fase di esame non può essere catturato dall'interruzione SYNTAX.
7	10	Clause too long	Clausola troppo lunga per il buffer interno. Dividere la riga sorgente in parti più piccole. È un errore della fase di esame e non può essere catturato dall'interruzione SYNTAX.
8	10	Invalid token	È stato rilevato un indicatore lessicale sconosciuto oppure non è possibile classificare adeguatamente una clausola. È un errore della fase di esame e non può essere catturato dall'interruzione SYNTAX.
9	10	Symbol or string too long	È stato fatto un tentativo per creare una stringa più lunga del massimo consentito.

Errore	Codice risultato	Messaggio	Descrizione
10	10	Invalid message packet	È stato rilevato un codice di operazione non valido in un pacchetto messaggi inviato al processo residente ARexx. Il pacchetto è stato restituito senza essere elaborato. Questo errore viene rilevato dall'interfaccia esterna e non può essere catturato dall'interruzione SYNTAX.
11	10	Command string error	Non è stato possibile elaborare una stringa di comando. Questo errore viene rilevato dall'interfaccia esterna e non può essere catturato dall'interruzione SYNTAX.
12	10	Error return from function	Una funzione esterna ha riportato un codice d'errore diverso da zero. Controllare che la funzione contenga i parametri corretti.
13	10	Host environment not found	Non è stata trovata la porta messaggi corrispondente a una stringa indirizzo ospite. Controllare che l'ospite esterno richiesto sia attivo.
14	10	Requested library not found	È stato fatto un tentativo per aprire una libreria funzioni compresa nella Lista libreria, ma non è stato possibile aprirla. Controllare che il nome e la versione corretti siano stati specificati al momento dell'introduzione nell'elenco delle risorse.

<b>Errore</b>	<b>Codice risultato</b>	<b>Messaggio</b>	<b>Descrizione</b>
15	10	Function not found	È stata chiamata una funzione che non è stato possibile trovare in nessuna libreria correttamente accessibile, né essere localizzata come programma esterno. Controllare che siano state aggiunte alla Lista le librerie funzioni appropriate.
16	10	Function did not return value	È stata chiamata una funzione che non ha restituito la stringa risultato, ma neppure un errore. Controllare che la funzione sia stata programmata correttamente o chiamarla usando l'istruzione CALL.
17	10	Wrong number of arguments	È stata fatta una chiamata a una funzione che attendeva un numero diverso di argomenti. Questo errore è generato se viene chiamata una funzione integrata o esterna con più argomenti di quanti ne possa contenere il pacchetto messaggi usato per le comunicazioni esterne.
18	10	Invalid argument to function	Ad una funzione è stato fornito un argomento non valido oppure manca un argomento richiesto. Controllare i requisiti dei parametri specificati per la funzione.
19	10	Invalid PROCEDURE	È stata data un'istruzione PROCEDURE in un contesto non valido: non era attiva nessuna funzione interna o era già stata data un'istruzione PROCEDURE nell'ambiente di memorizzazione corrente.

Errore	Codice risultato	Messaggio	Descrizione
20	10	Unexpected THEN or WHEN	È stata eseguita un'istruzione WHEN o THEN all'esterno di un contesto valido. L'istruzione WHEN è valida soltanto all'interno di un dominio SELECT, mentre THEN deve essere l'istruzione successiva a IF o WHEN.
21	10	Unexpected ELSE or OTHERWISE	È stata trovata un'istruzione ELSE o OTHERWISE all'esterno di un contesto valido. L'istruzione OTHERWISE è valida soltanto all'interno di un dominio SELECT. ELSE è valida soltanto quando segue la condizione THEN di un campo IF.
22	10	Unexpected BREAK, LEAVE or ITERATE	L'istruzione BREAK è valida soltanto all'interno di un dominio DO o di una stringa INTERPRET. Le istruzioni LEAVE e ITERATE sono valide soltanto all'interno di un dominio DO iterativo.
23	10	Invalid statement in SELECT	È stata incontrata un'istruzione non valida all'interno di un dominio SELECT. Soltanto le istruzioni WHEN, THEN e OTHERWISE sono valide all'interno di un dominio SELECT, eccezion fatta per le istruzioni condizionali che seguono le clausole THEN o OTHERWISE.
24	10	Missing or multiple THEN	Non è stata trovata una clausola THEN attesa o un'altra THEN è stata trovata dopo che una era già stata eseguita.
25	10	Missing OTHERWISE	Nessuna delle clausole WHEN in SELECT è riuscita, ma non è stata fornita nessuna clausola OTHERWISE.

<b>Errore</b>	<b>Codice risultato</b>	<b>Messaggio</b>	<b>Descrizione</b>
26	10	Missing or unexpected END	La sorgente del programma è finita prima di aver trovato END per un'istruzione DO o SELECT, oppure END è stata incontrata all'esterno di un dominio DO o SELECT.
27	10	Symbol mismatch	Il simbolo specificato in un'istruzione END, ITERATE o LEAVE non concorda con la variabile di indice del dominio DO. Controllare che i cicli attivi siano stati annidati correttamente.
28	10	Invalid DO syntax	È stata eseguita un'istruzione DO non valida. Deve essere data un'espressione inizializzatrice se viene specificata un'espressione TO o BY, mentre un'espressione FOR deve produrre un risultato intero non negativo.
29	10	Incomplete IF or SELECT	Un dominio IF o SELECT è terminato prima di aver trovato tutte le istruzioni richieste. Controllare se è stata omessa l'istruzione condizionale che segue una clausola THEN, ELSE o OTHERWISE.
30	10	Label not found	Un'etichetta specificata da un'istruzione SIGNAL o citata implicitamente da un'interruzione abilitata non è stata trovata nella sorgente del programma. Le etichette definite dinamicamente da un'istruzione INTERPRET o da un ingresso interattivo non sono comprese nella ricerca.

Errore	Codice risultato	Messaggio	Descrizione
31	10	Symbol expected	È stato trovato un indicatore che non è un simbolo dove è valido soltanto un indicatore simbolo. Le istruzioni DROP, END, LEAVE, ITERATE e UPPER possono essere seguite soltanto da un indicatore simbolo. Questo messaggio è emesso anche quando manca un simbolo richiesto.
32	10	Symbol or string expected	È stato trovato un indicatore non valido in un contesto in cui è valido soltanto un simbolo o una stringa.
33	10	Invalid keyword	È stato identificato come parola chiave un indicatore simbolo in una clausola istruzione, ma non è valido nel contesto specifico.
34	10	Required keyword missing	Una clausola istruzione richiede la presenza di uno specifico indicatore parola chiave, ma non è stato fornito. Questo messaggio viene inviato se un'istruzione SIGNAL ON non è seguita da una parola chiave di interruzione (es. SYNTAX).
35	10	Extraneous characters	È stata eseguita un'istruzione apparentemente valida, ma sono stati trovati altri caratteri alla fine della clausola.
36	10	Keyword conflict	Due parole chiave reciprocamente esclusive sono state incluse in una clausola istruzione o una parola chiave è stata inclusa due volte nella stessa istruzione.

<b>Errore</b>	<b>Codice risultato</b>	<b>Messaggio</b>	<b>Descrizione</b>
37	10	Invalid template	Il modello fornito con un'istruzione ARG, PARSE o PULL non è costruito correttamente.
38	10	Invalid TRACE request	La parola chiave alfabetica fornita con un'istruzione TRACE o come argomento della funzione integrata TRACE() non è valida.
39	10	Uninitialized variable	È stato fatto un tentativo per usare una variabile non inizializzata mentre era abilitata l'interruzione NOVALUE.
40	10	Invalid variable name	È stato fatto un tentativo di assegnare un valore a un simbolo fisso.
41	10	Invalid expression	È stato rilevato un errore durante la risoluzione di un'espressione. Controllare che ogni operatore abbia il numero giusto di operandi e che nell'espressione non appaiano indicatori estranei. Questo errore è rilevato soltanto nelle espressioni effettivamente calcolate. Non avvengono controlli sulle espressioni delle clausole che sono saltate.
42	10	Unbalanced parentheses	È stata trovata un'espressione con un numero diverso di parentesi aperte e chiuse.
43	10	Nesting limit exceeded	Il numero di sottoespressioni di un'espressione è maggiore del massimo consentito. L'espressione deve essere semplificata dividendola in due o più espressioni intermedie.

<b>Errore</b>	<b>Codice risultato</b>	<b>Messaggio</b>	<b>Descrizione</b>
44	10	Invalid expression result	Il risultato di un'espressione non è valido all'interno del relativo contesto. Questo messaggio è inviato se un'espressione incremento o un'espressione limite in un'istruzione DO produce un risultato non numerico.
45	10	Expression required	È stata omessa un'espressione in un contesto che la richiede. Ad esempio, l'istruzione SIGNAL, se non è seguita dalle parole chiave ON o OFF deve essere seguita da un'espressione.
46	10	Boolean value not 0 or 1	Il risultato dell'espressione doveva essere un risultato booleano, ma ha calcolato un valore diverso da 0 o 1.
47	10	Arithmetic conversion error	È stato usato un operando non numerico in un'espressione che richiede operandi numerici. Questo messaggio viene anche generato da una stringa esadecimale o binaria non valida.
48	10	Invalid operand	Operando non valido per l'operazione desiderata. Questo messaggio è generato se viene fatto un tentativo di dividere per 0 o se viene usato un esponente frazionario in un'operazione di elevamento a potenza.



## **Appendice B**

# **Utilità di comando**

---

Le utilità di comando ARexx, residenti nella Directory REXXC, forniscono varie funzioni di controllo. Sono moduli eseguibili gestiti dalla Shell. Sono pertinenti soltanto quando è attivo il processo residente ARexx.

### **HI**

### **HI**

Imposta l'indicatore di arresto globale, affinché tutti i programmi ARexx attivi ricevano una richiesta di arresto esterna. Ogni programma esce immediatamente, a meno che non ne sia stata abilitata l'interruzione HALT. L'indicatore di arresto non rimane impostato, ma viene azzerato automaticamente quando tutti i programmi correnti hanno ricevuto la richiesta.

### **RX**

### **RX nome [argomenti]**

Lancia un programma ARexx. Se il nome specificato include un percorso esplicito, il programma viene cercato soltanto in quella directory, in caso contrario viene cercato nella directory corrente e in REXX:. La stringa argomento opzionale viene passata al programma.

**RXSET**

RXSET [nome [[=] valore]]

Aggiunge una coppia (nome, valore) alla Lista Clip. Si presume che le stringhe nome contengano lettere maiuscole e minuscole. Se esiste già una coppia con lo stesso nome, il suo valore viene sostituito con la stringa corrente. Se viene dato un nome senza una stringa valore, l'introduzione viene rimossa dalla Lista Clip. Se RXSET è chiamato senza argomenti, emette un elenco di tutte le coppie (nome, valore) della Lista Clip.

**RXC**

RXC

Chiude il processo residente. La porta pubblica di "REXX" viene immediatamente ritirata e il processo residente esce appena è terminato l'ultimo programma ARexx. Dopo una richiesta di chiusura non può essere lanciato nessun nuovo programma.

**TCC**

TCC

Chiude la console di analisi globale quando non è più usata dai programmi attivi. Tutte le richieste lette e accodate alla console devono essere soddisfatte prima della chiusura.

**TCO**

TCO

Apri la console di analisi globale. L'uscita di analisi da tutti i programmi attivi viene dirottata automaticamente alla nuova console. La finestra della console può essere spostata e ridimensionata dall'utente e chiusa con il comando "TCC".

**TE**

TE

Azzera l'indicatore di analisi globale, e pone su OFF il modo analisi per tutti i programmi ARexx attivi.

**TS****TS**

Inizia l'analisi interattiva impostando l'indicatore di analisi esterno, che pone nel modo analisi interattivo tutti i programmi ARexx attivi. I programmi incominciano a produrre uscite di analisi e fanno una pausa dopo l'istruzione successiva. Questo comando è utile per riprendere il controllo di programmi caduti in cicli infiniti o che si comportano in modo anomalo. L'indicatore di analisi rimane impostato finché viene azzerato dal comando TE, quindi le chiamate programma seguenti vengono eseguite nel modo analisi interattivo.

**WaitForPort****WaitForPort** [attesa nome porta]

WaitForPort attende 10 secondi affinché appaia la porta specificata. Il codice 0 indica che la porta è stata trovata, mentre 5 indica che l'applicazione non è in corso o che la porta non esiste. I nomi porta subiscono il controllo maiuscole.

```
WaitForPort ED_1  
WaitForPort MyPort
```



# Glossario

---

Questo glossario presenta le definizioni dei termini usati nel manuale ARexx.

**ambiente di memorizzazione (storage environment)**

I componenti variabili del programma ARexx, comprese le stringhe tabella simboli, opzioni numeriche, opzioni trace e indirizzo ospite.

**ambiente globale (global environment)**

I componenti fissi di un programma ARexx, compresi i codici sorgente del programma, stringhe dati statiche e le stringhe argomento.

**analisi (tracing)**

Visualizzazione delle linee di un programma ARexx durante l'esecuzione del programma. Ciò consente all'utente di determinare esattamente la posizione in cui è avvenuto l'errore.

**analisi sintattica (parsing)**

Divisione di una stringa in unità più piccole

**argomento (argument)**

Un'informazione aggiuntiva compresa in un'istruzione o funzione. L'argomento determina l'operazione da intraprendere.

**booleano (boolean)**

Che ha due stati possibili: 0 (falso) o 1 (vero).

**clausola (clause)**

La più piccola unità linguaggio eseguibile.

**clausola comando (command clause)**

L'espressione ARexx in cui il risultato è emesso come comando ad una applicazione esterna.

**clausola di assegnazione (assignment clause)**

Un simbolo variabile (semplice, radice o simbolo composto) seguito da un operatore =. In una clausola di assegnazione, gli indicatori a destra del segno = sono elaborati e il risultato viene assegnato al simbolo della variabile.

**clausola nulla (null clause)**

Una linea di spazi o una linea commento.

**codice di restituzione (return code)**

Livello di gravità di un errore. Questo numero (5, 10, o 20) è visualizzato quando avviene un errore nel programma ARexx.

**commento (comment)**

Un gruppo di caratteri racchiusi nei simboli /\*....\*/. Ogni programma ARexx inizia con un commento.

**comunicazione interprocesso (interprocess communication)**

Lo scambio di informazioni tra applicazioni.

**concatenazione (concatenation )**

Il processo per collegare due stringhe insieme.

**delimitatore (delimiter)**

Un carattere che indica l'inizio e la fine di una stringa. I delimitatori ARexx sono degli apici (') o delle virgolette (").

**diagnostica (debugging)**

Ricerca ed eliminazione di errori in un programma.

**dichiarazione (statement)**

Un assegnazione, istruzione, o clausola comando.

**etichetta (label)**

Indicatore simbolo seguito da due punti (:). Le etichette identificano un posizione particolare nel programma.

**espressione (expression)**

Un gruppo di indicatori da elaborare. Le espressioni sono composte da stringhe, simboli, operatori e parentesi.

**funzione (function)**

Un gruppo di dichiarazioni che possono essere eseguite in blocco. Le funzioni consentono all'utente di costruire programmi complessi da moduli più piccoli.

**identazione (tokenization)**

Il processo di divisione di una dichiarazione nei suoi gettoni singoli.

**indicatori (tokens)**

L'entità più piccola del linguaggio ARexx.

**indirizzo (address)**

Un numero di identificazione assegnato ad ogni byte di informazione nella memoria del computer e ad ogni settore di un disco.

**indirizzo ospite (host address)**

In un'applicazione esterna, l'indirizzo ospite corrisponde alla porta messaggi cui possono essere inviati i messaggi ARexx.

**interfaccia di comando (command interface)**

Un porta messaggi attraverso la quale ARexx emette comandi agli applicativi compatibili.

**interruzioni (interrupts)**

Gli indicatori interni di ARexx che consentono al programma di rilevare errori e tenere il controllo quando il programma terminerebbe a causa di un errore. Le interruzioni sono controllate dall'istruzione SIGNAL.

**istruzione (instruction)**

Una clausola che inizia con una certa parola chiave e indica ad ARexx di eseguire un'operazione.

**iterazione (iteration)**

Ripetizione di un segmento di programma.

**librerie funzione (function libraries)**

Una raccolta di una o più funzioni organizzate analogamente alla libreria condivisa Amiga. Una libreria funzioni può contenere un nome libreria, sulla priorità di ricerca, uno spostamento dal punto di ingresso ed un numero versione.

**lista clip (clip list)**

La clipboard utilizzata per comunicazione fra processi. Alla lista clip possono essere aggiunte coppie di nomi e valori con la funzione SETCLIP().

### **lista librerie (library list)**

Una lista interna, aggiornata da ARexx, di tutte le librerie funzioni e ospite funzione disponibili correntemente. Le applicazioni possono aggiungere o togliere librerie funzioni come necessario.

### **macro (macro)**

Un altro nome per un programma ARexx.

### **marcatore (marker)**

Un indicatore che determina l'inizio e la fine di una stringa analisi.

### **modello (template)**

Un gruppo di indicatori che specifica le variabile usate in un'operazione di analisi. Specifica inoltre il modo in cui sono determinati i valori.

### **obiettivo (target)**

Un simbolo, generalmente un simbolo variabile, cui è assegnato un valore durante un'operazione di analisi..

### **operatori (operators)**

Un carattere come (+), (-), o (|) usato in una operazione aritmetica, in una concatenazione di comparazione o logica.

### **ospiti funzione (function hosts)**

Un'applicazione esterna che contiene una porta ARexx. Il nome di un ospite funzione è il nome della porta messaggi pubblica del programma.

### **porta messaggi (message port)**

Un interfaccia nell'applicazione Amiga che consente al programma di comunicare con un programma ARexx.

### **precisione numerica (numeric precision)**

Il numero di posti decimali in un risultato aritmetico. Al diminuire del numero dei posti decimali, il risultato diventa meno preciso.

### **RexxMast**

Il programma che agisce come interprete per i programmi ARexx.



**simbolo (symbol)**

Qualsiasi gruppo di caratteri alfanumerici a-z, A-Z, 0-9, punto (.), punto esclamativo (!), punto interrogativo (?), segno di dollaro (\$), o sottolineato (\_).

**simbolo composto (compound symbol)**

Un indicatore composto di caratteri alfanumerici o ., !, ?, \$, \_ con uno o più punti all'interno del nome. I simboli composti hanno la struttura radice.n1.n2...nk.

**simbolo fisso (fixed symbol)**

Un indicatore che incomincia con una cifra o un punto.

**simbolo radice (stem symbol)**

Un indicatore che contiene un punto al termine del suo nome. I simboli radice sono usati per inizializzare i simboli composti.

**simbolo semplice (simple symbol)**

Un indicatore che non inizia con una cifra o punto.

**stringa (string)**

Un gruppo di caratteri che inizia e termina con un delimitatore (apice o virgolette). Il valore della stringa è la stringa stessa.

**tabella simboli (symbol table)**

Una tabella simboli creata da ARexx che memorizza le stringhe valori che sono state assegnate alle variabili in un programma.

**variabile (variable)**

Un simbolo cui può essere assegnato un valore.



# Indice analitico

---

## A

- ABBREV(), 5-8
- ABS(), 5-8
- ADDLIB(), 5-9
- ADDRESS, 3-15, 4-2, 4-20
- ADDRESS(), 3-15, 4-2, 5-9
- ALLOCMEM(), 5-20, 5-41
- ambiente, 3-20
  - esterno, 3-20
  - globale, 3-21
  - interno, 3-21
  - memorizzazione, 3-21, 4-2, 5-2, 5-5
- ambiente di memorizzazione, 3-21, 4-2, 5-2, 5-5
- ambiente globale, 3-21
- ambienti di memorizzazione, 3-21
- analisi
  - console di analisi globale, 6-3
  - formattazione visualizzazione, 6-2
  - interattiva, 6-1, 6-5
  - interruzioni, 6-7
  - opzioni, 6-5, 6-6
  - uscita, 6-3
- analisi interattiva, 6-1, 6-5
- Analisi Sintattica
  - Obiettivi, 7-2
- analisi sintattica, 7-1
  - il processo di scansione, 7-4
  - marcatore, 7-1, 7-2
  - mediante configurazione, 7-6

- mediante indentazione, 7-2, 7-5
- modelli, 7-1
- modelli multipli, 7-3, 7-7
- obiettivi, 7-1, 7-4, 7-5
- ARG, 4-3, 4-17, 6-2, 7-1
- ARG(), 5-10
- avvio dei programmi, 2-7

## B

- B2C(), 5-10
- BITAND(), 5-10
- BITCHG(), 5-11
- BITCLR(), 5-11
- BITCOMP(), 5-11
- BITOR(), 5-11
- BITSET(), 5-12
- BITTST(), 5-12
- BITXOR(), 5-12
- BREAK, 4-4, 4-10

## C

- C2B(), 5-12
- C2D(), 5-13
- C2X(), 5-13
- CALL, 4-2, 4-4, 5-2
- CENTER(), 5-13

- chiamata funzioni, 4-4
- cicli, 2-7, 2-9, 4-5
- ciclo, 2-10
- Cifre, 4-11
  - notazione scientifica, 4-12
  - notazioni engineering, 4-12
- Cifre Numeriche, 5-17
- clausole, 3-1, 3-10
  - assegnazione, 3-12
  - comando, 3-12, 6-2, 6-4
  - etichetta, 3-11, 4-10, 5-2, 6-2
  - istruzione, 3-12
  - nulle, 3-11
- clausole assegnazione, 3-12
- clausole comando, 3-11, 6-4
- clausole etichetta, 3-11, 4-10, 5-2, 6-2
- clausole istruzione, 3-11
- clausole nulle, 3-11
- CLOSE(), 5-14
- CLOSEPORT(), 5-42
- codici di restituzione, 4-12, 6-1, 6-2
- codici di ritorno, 3-19
- comando
  - clausole, 3-12, 6-2
  - interfaccia, 3-1, 3-15
- commenti, 2-4, 2-6, 3-1, 3-11
- COMPARE(), 5-14
- COMPRESS(), 5-14
- comunicazione interprocesso, 1-2
- condizione del segnale, 4-20
- configurazione
  - analisi sintattica, 7-6
  - marcatore, 7-3, 7-4
  - marcatori, 7-2, 7-3
- continuazione clausole, 3-11
- controllo Errori
  - Rilevamento, 6-1
- controllo errori, 2-9
- conversione
  - a decimale, 5-13
  - a esadecimale, 5-13
  - caratteri, 5-12

- cifra binaria, 5-10
- cifre decimali, 5-36
- da decimale a esadecimale, 5-15
- da esadecimale a decimale, 5-36
- COPIES(), 5-14

## **D**

- D2C(), 5-15
- D2X(), 5-15
- DATATYPE(), 5-17
- DATE(), 5-15
- delimitatore, 3-5
- DELSTR(), 5-17
- DELWORD(), 5-17
- diagnostica, 6-1
- dichiarazione condizionata, 4-8, 4-9, 4-19, 4-22
- dichiarazioni condizionata, 4-11
- DIGITS(), 5-17
- DO, 2-7, 4-4, 4-5, 4-8, 4-10, 4-11, 6-5
- DROP, 4-7

## **E**

- ECHO, 4-7
- ELSE, 4-8
- END, 4-5, 4-8, 4-10, 4-20
- EOF(), 5-18
- ERRORTXT(), 5-18
- esame, 1-3
  - sorgente di ingresso, 4-14
- espressione inizializzatrice, 4-5
- espressioni, 3-1, 3-12, 3-13

**esterno**

- ambiente, 3-20
- file, 5-3
- indicatore di analisi, 6-6
- librerie funzione, 5-3

EXISTS(), 5-18

EXIT, 2-9, 4-8, 4-19

EXPORT(), 5-18

**F**

**file esterni**

- apertura, 5-23

FIND(), 5-19

flusso di ingresso, 3-19, 3-20, 6-4

flusso di uscita, 3-20

flusso uscita, 3-20, 6-3

FORM(), 5-19

FREEMEM(), 5-42

FREESPACE(), 5-19

funzioni, 2-8, 2-9, 5-1

- chiamata, 4-4, 5-1

- integrate, 5-3, 5-5

- interne, 3-21, 5-5

funzioni integrate, 5-3, 5-5

funzioni interne, 3-21

funzioni ospite, 5-5

FUZZ(), 5-20

**G**

gestione errori, 6-6

GETARG(), 5-42

GETCLIP(), 5-6, 5-20

GETPKT(), 5-43

GETSPACE(), 5-20

**globale**

- ambiente, 3-21
- console di analisi, 6-3

**H**

HALT, 4-20

HASH(), 5-21

**I**

IF, 2-7, 2-8

IMPORT(), 5-18, 5-21

indentazione, 7-5

INDEX(), 5-21, 7-1

indicatore, 3-1

indicatore (token), 3-1

indicatori di interruzione, 4-20, 5-2, 6-6

indicatori di interruzione interni, 4-20

indicatori di interruzioni, 6-8

indicazione, 3-21

indirizzo ospite, 3-15, 3-21, 4-2, 5-9

INSERT(), 5-22

**interne**

- funzioni, 5-2, 5-5

**interni**

- predefiniti, 4-12

**interno**

- ambiente, 3-21

INTERPRET, 4-4, 4-9, 6-5

interruzioni, 6-7

interruzioni interne, 6-1, 6-6

Istruzioni, 4-1

istruzioni, 2-5, 2-7, 2-9, 3-12

ITERATE, 4-10

### **L**

LEAVE, 4-4, 4-10

LEFT(), 5-22

LENGTH(), 5-22

librerie

    condivisa, 5-3

    funzione, 5-3

librerie di funzioni, 5-5

librerie funzioni, 1-3, 5-4

LINES(), 5-23

Lista Clip

    aggiunta valori, 5-28

    esaminazione, 5-28

    ricerca, 5-20

lista Clip, 5-6

lista libreria, 5-4

    esaminazione, 5-28

    rimozione voci, 5-27

### **M**

macro, 3-17

marcatori, 7-1, 7-2

    posizione, 7-6

marcatori di posizione, 7-6

MAX(), 5-23

memoria

    assegnazione blocchi, 5-20,

    5-41

    rilascio blocchi, 5-42

memorizzazione ambiente, 3-21

MIN(), 5-23

modelli

    multipli, 4-14, 7-7

modelli multipli, 4-14, 4-15, 7-3,  
7-7

modello, 4-3, 4-13, 4-17, 7-1

    multiplo, 4-14, 4-15, 7-3

    oggetti, 7-3

    ricerca, 5-24

modo analisi, 5-32

multitasking, 1-2

### **N**

NOP, 4-8, 4-11

notazione engineering, 4-12

notazione scientifica, 4-12

NUMERIC, 3-6, 4-11, 5-2

NUMERIC Digits, 3-6

numero casuale, 5-26

### **O**

Obiettivi, 7-2

obiettivi, 7-1, 7-4, 7-5

OPEN(), 5-23

OPENPORT(), 5-43

operatori, 3-1, 3-5, 3-13, 7-3

    comparazione, 3-8

    concatenazione, 3-7

    logici, 3-9

operatori aritmetici, 3-5, 3-6

operatori di comparazione, 3-6

operatori di concatenazione, 3-5

operatori logici, 3-6

OPTIONS, 5-2, 6-6

ospiti di funzioni, 5-4

OTHERWISE, 4-13

OVERLAY(), 5-23

### **P**

parentesi, 3-13, 7-3

parole chiave, 4-1

PARSE, 6-2, 7-1

porte messaggi  
    chiusura, 5-42  
    controllo, 5-43  
    creazione, 5-43  
POS(), 5-24  
posizione di scan, 7-4  
posizione di scansione, 7-1, 7-2  
PRAGMA(), 5-24  
precisione, 3-8  
PROCEDURE, 4-16, 5-2, 5-3  
PULL, 2-6, 4-17, 6-2, 7-1  
PUSH, 4-18

## **R**

RANDOM(), 5-26  
RANDU(), 5-26  
READCH(), 5-27  
READLN(), 5-27  
REMLIB(), 5-27  
REPLY(), 5-44  
RETURN, 2-9, 4-19, 5-3  
REVERSE(), 5-27  
REXX:, 2-2, 2-3  
RexxMast, 2-1, 2-4, 3-20  
REXXSupport.library  
    apertura, 5-40  
REXXSupport.library, 5-40  
RIGHT(), 5-28  
rilevamento, 6-1  
RX, 2-3

## **S**

SAY, 2-5, 2-6, 2-7, 2-9, 4-7, 4-19  
scansione posizione, 7-2  
SEEK(), 5-28  
SELECT, 4-8, 4-19, 6-5  
SETCLIP(), 5-28

SHELL, 4-20  
shell, 3-19  
shell comandi, 4-18  
SHOW(), 5-9, 5-28  
SHOWDIR(), 5-44  
SHOWLIST(), 5-44  
SIGN(), 5-29  
SIGNAL, 4-20, 5-2, 6-6, 6-7  
simboli, 2-5, 2-7, 3-1, 3-2, 3-13, 7-3  
    composti, 3-2, 3-21, 4-17  
    composto, 3-3, 4-7  
    fissi, 7-3  
    radice, 3-2, 3-3, 3-21  
    semplici, 3-2, 3-21  
simboli composti, 3-2, 3-3, 3-21, 4-7, 4-17  
simboli fissi, 3-13, 7-3  
simboli radice, 3-2, 3-3, 3-21, 4-17  
simboli semplici, 3-2, 3-21  
SOURCELINE(), 5-29  
SPACE(), 5-30  
spazi, 3-6  
STATEF(), 5-45  
STDERR, 4-14, 6-3  
STDIN, 4-18  
STDOUT, 6-3  
STORAGE(), 5-18, 5-30  
stringhe, 2-5, 3-1, 3-5, 3-11, 3-13, 7-3  
    comparazione, 5-11, 5-14  
    esame, 4-13  
    inversione caratteri, 5-27  
    rimozione spazi, 5-30  
stringhe argomento, 2-8, 3-21, 4-4, 7-1  
STRIP(), 5-30  
SUBSTR(), 5-31, 7-1  
SUBWORD(), 5-31  
SYMBOL(), 5-31  
SYNTAX, 6-7

## **T**

tabella simboli, 3-13, 3-21, 4-16,  
5-2  
TCC, 6-4  
TCO, 6-3  
TE, 6-7  
TIME(), 5-32  
TRACE, 2-9, 6-2  
trace  
    opzioni, 3-21, 5-2, 6-1  
TRACE(), 5-32, 6-2  
TRANSLATE(), 5-33  
TRIM(), 5-33  
TRUNC(), 5-33  
TS, 6-6

## **U**

UPPER(), 5-34

## **V**

validata, 2-10  
VALUE(), 5-34  
variabile RC, 3-19, 6-9  
variabile RESULT, 4-4  
variabile SIGL, 4-21, 6-9  
variabili, 2-6, 3-3, 3-6, 3-21, 4-13,  
4-17, 7-1  
    valori ripristinati, 4-7  
verifica errori  
    esame, 1-3  
VERIFY(), 5-34  
visualizzazione uscita, 4-19

## **W**

WAITPKT(), 5-45  
WHEN, 4-22  
WORD(), 5-34  
WORDINDEX(), 5-35  
WORDLENGTH(), 5-35  
WORDS(), 5-35  
WRITECH(), 5-35  
WRITELN(), 5-35

## **X**

X2C(), 5-36  
X2D(), 5-36  
XRANGE(), 5-36





AMIGA

OS 3.1 AREXX