M30 M40

PASCAL +
Preparazione ed Esecuzione dei Programmi

olivetti L1

Copyright © 1983, by Olivetti
All rights reserved

PUBBLICAZIONE EMESSA DA:

Ing. C. Olivetti & C., S.p.A.
Direzione Documentazione
77, Via Jervis - 10015 IVREA (Italy)

4004470 J

PREFAZIONE

Questo manuale e' destinato ai programmatori PASCAL+ che vogliano preparare ed eseguire i loro programmi sul sistema L1 MOS.

Si assume che il lettore abbia conoscenza del Sistema operativo MOS e delle caratteristiche del linguaggio di programmazione PASCAL+.

SOMMARIO

Il manuale descrive gli strumenti messi a disposizione di un programmatore PASCAL+ per preparare ed eseguire un programma sul sistema L1 MOS.

- Il capitolo 1 descrive il compilatore PASCAL+
- Il capitolo 2 descrive il linker PASCAL+
- Il capitolo 3 descrive il debugger PASCAL+
- Il capitolo 4 descrive un programma per il link dinamico di l-moduli
- Il capitolo 5 descrive alcune utility
- L'appendice A contiene l'elenco degli errori di run time.

RIFERIMENTI

In precedenza, leggere...

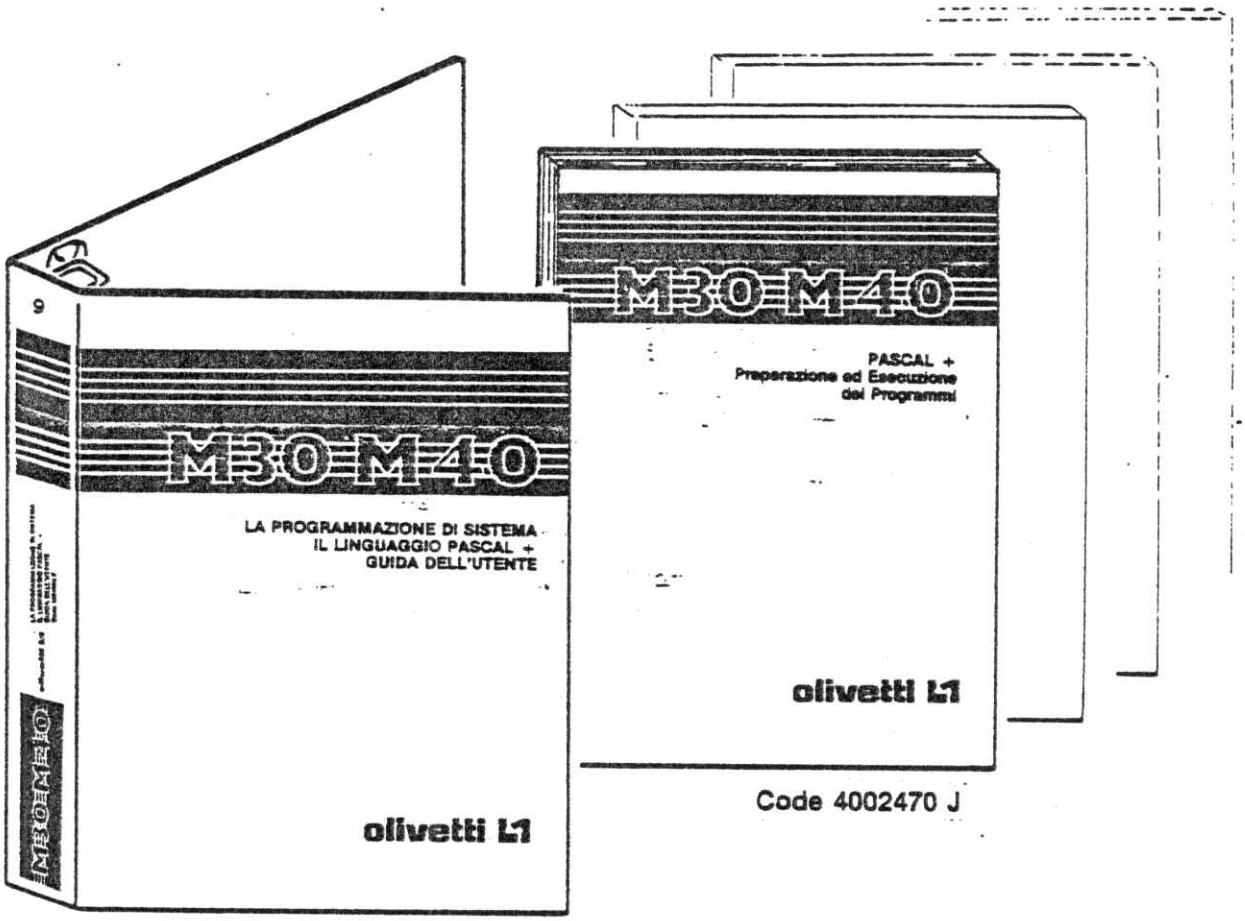
Linguaggio PASCAL+ Manuale di Consultazione - Code 4002450 Q

MOS - EDITOR Manuale di Consultazione - Code 4002430 N

Per ulteriori informazioni, leggere...

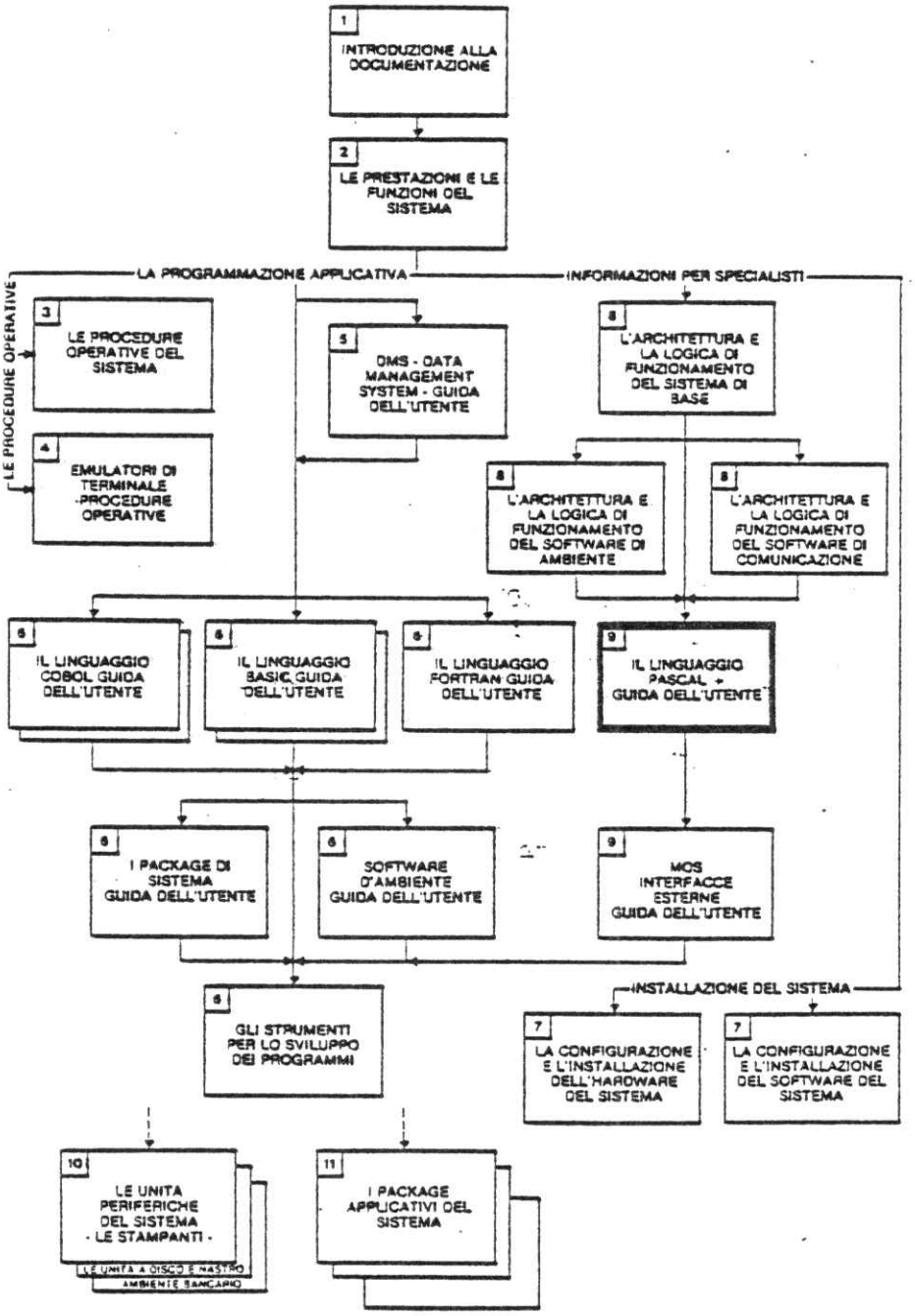
MOS - SHELL I Comandi Operativi - Manuale di Consultazione - Code 4002760P

PRIMA EDIZIONE: Settembre 1983, Release 2.0



Code 4002470 J

Code 4004980 F


1
INTRODUZIONE ALLA DOCUMENTAZIONE
2
LE PRESTAZIONI E LE FUNZIONI DEL SISTEMA
LA PROGRAMMAZIONE APPLICATIVA
INFORMAZIONI PER SPECIALISTI
LE PROCEDURE OPERATIVE
3
LE PROCEDURE OPERATIVE DEL SISTEMA
4
EMULATORI DI TERMINALE -PROCEDURE OPERATIVE
5
DMS - DATA MANAGEMENT SYSTEM - GUIDA DELL'UTENTE
8
L'ARCHITETTURA E LA LOGICA DI FUNZIONAMENTO DEL SISTEMA DI BASE
8
L'ARCHITETTURA E LA LOGICA DI FUNZIONAMENTO DEL SOFTWARE DI AMBIENTE
8
L'ARCHITETTURA E LA LOGICA DI FUNZIONAMENTO DEL SOFTWARE DI COMUNICAZIONE
5
IL LINGUAGGIO COBOL GUIDA DELL'UTENTE
5
IL LINGUAGGIO BASIC GUIDA DELL'UTENTE
5
IL LINGUAGGIO FORTRAN GUIDA DELL'UTENTE
9
IL LINGUAGGIO PASCAL - GUIDA DELL'UTENTE
6
I PACKAGE DI SISTEMA GUIDA DELL'UTENTE
6
SOFTWARE D'AMBIENTE GUIDA DELL'UTENTE
9
MOS INTERFACCE ESTERNE GUIDA DELL'UTENTE
INSTALLAZIONE DEL SISTEMA
6
GLI STRUMENTI PER LO SVILUPPO DEI PROGRAMMI
7
LA CONFIGURAZIONE E L'INSTALLAZIONE DELL'HARDWARE DEL SISTEMA
7
LA CONFIGURAZIONE E L'INSTALLAZIONE DEL SOFTWARE DEL SISTEMA
10
LE UNITA PERIFERICHE DEL SISTEMA - LE STAMPANTI -
LE UNITÀ A DISCO E NASTRO
AMBIENTE BANCARIO
11
I PACKAGE APPLICATIVI DEL SISTEMA

INDICE

PAGINA

1-1 1. COMPILAZIONE DI UNITA' PASCAL+

1-2 CHIAMATA DEL COMPILATORE

1-4 OPZIONI DI LINEA DI COMANDO DEL COMPILATORE

1-6 OPZIONI DEL COMPILATORE NEL SORGENTE

1-9 IL COMANDO DI INCLUDE

1-10 FORMATO DEL LISTING

1-12 FORMATO DEL FILE OGGETTO E NOMI DI SEZIONE

1-13 MESSAGGI DI ERRORE DI COMPILAZIONE

2-1 2. COLLEGAMENTO E CARICAMENTO DI UNITA' PASCAL+: IL LINKER ZLOC

2-2 CARATTERISTICHE DEL LINKER

2-3 LA CHIAMATA DEL LINKER

2-3 ORGANIZZAZIONE DELLA LISTA DEI COMANDI

2-8 CLASSIFICAZIONE DEI COMANDI

2-9 ORDINE DEI COMANDI

2-9 DESCRIZIONE DEI COMANDI

2-10 ATTRIBUTES

2-11 BLOCK DESCRIPTOR

2-12 COMMAND

2-13 ENTRY

2-14 INPUT

2-14 INSYM

2-15 LIBRARY

PAGINA

2-15 MAP

2-16 MESSAGE

2-17 NOWARNINGS

2-17 OPTIMIZE

2-17 OUTPUT

2-18 QUIET

2-18 SQUEEZE

2-18 STATISTICS

2-18 SYMBOL

2-20 TYPE

2-22 ESEMPI

3-1 3. IL DEBUGGER SIMBOLICO PASCAL+

3-2 PROCEDURE OPERATIVE

3-4 COMANDI DEL DEBUGGER

3-4 COMANDI DI CONTROLLO

3-4 COMANDI RELATIVI AI DATI

3-5 COMANDI DI BASE

3-5 ELENCO COMANDI

4-1 4. ZSTUB : UNO STRUMENTO PER IL COLLEGAMENTO FRA L-MODULI

4-1 INTRODUZIONE

4-2 USO DI ZSTUB

4-2 PROCEDURE OPERATIVE

4-3 IL COMANDO INPUT

4-5 IL COMANDO TABLE

4-6 IL COMANDO INTER

4-7 IL COMANDO INTRA

4-7 CARATTERISTICHE IMPLEMENTATIVE

PAGINA

4-8 COME LANCIARE ZSTUB

4-9 ESEMPIO

5-1 5. TOOLS VARI UTILI NELLA PREPARAZIONE ED ESECUZIONE DI PROGRAMMI

5-2 MAN

5-3 MORE, PAGE

5-5 ZPCMESG

A-1 A. MESSAGGI DI ERRORE DI RUN TIME

A-2 ELENCO DEI MESSAGGI

1. COMPILAZIONE DI UNITA' PASCAL+

Un programma PASCAL+ e' costituito da un certo numero di unita' (denotate con MODULE, MONITOR o PROGRAM) contenenti un'unica unita' principale (denotata con PROGRAM).

Scopo di questo capitolo e' di mostrare come si compila un programma PASCAL+, usando i comandi e le opzioni del compilatore.

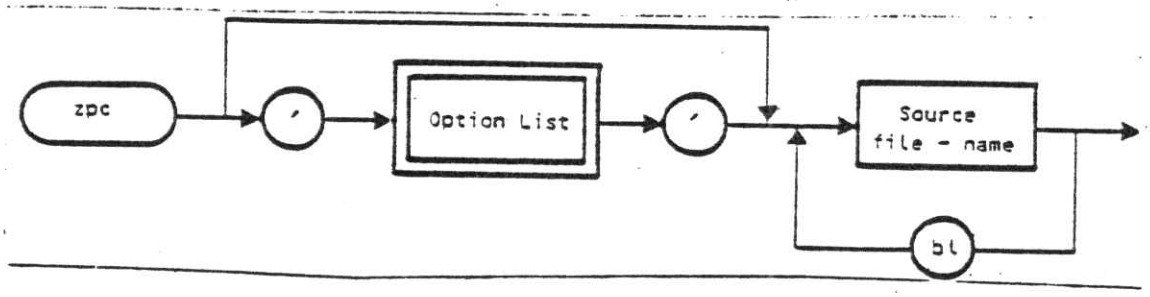
Sia per le istruzioni di un programma Pascal+ presenti negli esempi che per le opzioni che vanno all'interno dei programmi, sono stati usati per comodita' caratteri ASCII maiuscoli; il compilatore infatti non fa alcuna differenza fra caratteri maiuscoli e minuscoli.

I comandi del compilatore si dividono in due tipi. Il primo tipo contiene le opzioni che fanno parte della linea di comando che chiama il compilatore. Il secondo tipo contiene quelle opzioni o comandi che fanno parte del programma sorgente e che non vanno compilati, ma piuttosto eseguiti dal compilatore stesso quando vengono incontrati.

Sia la compilazione sia il programma compilato (e successivamente collegato col "linker") sono eseguiti sulla macchina M30/M40.

CHIAMATA DEL COMPILATORE

Il compilator PASCAL+, il cui nome e' "zpc", e' chiamato in ambiente Shell dal comando la cui sintassi e' illustrata dal seguente diagramma:



dove:

Option List e' la lista di opzioni il cui diagramma sintattico sara' illustrato nella prossima sezione e che fanno parte della linea di comando che chiama il compilatore "zpc".
L'utente puo' specificarne un numero qualsiasi (anche nullo) all'interno della linea di comando : esse costituiscono direttive al compilatore e il loro significato sara' esaminato nel prossimo paragrafo.

Source-file-name e' il nome di uno qualsiasi dei file di input o argomento della compilazione.

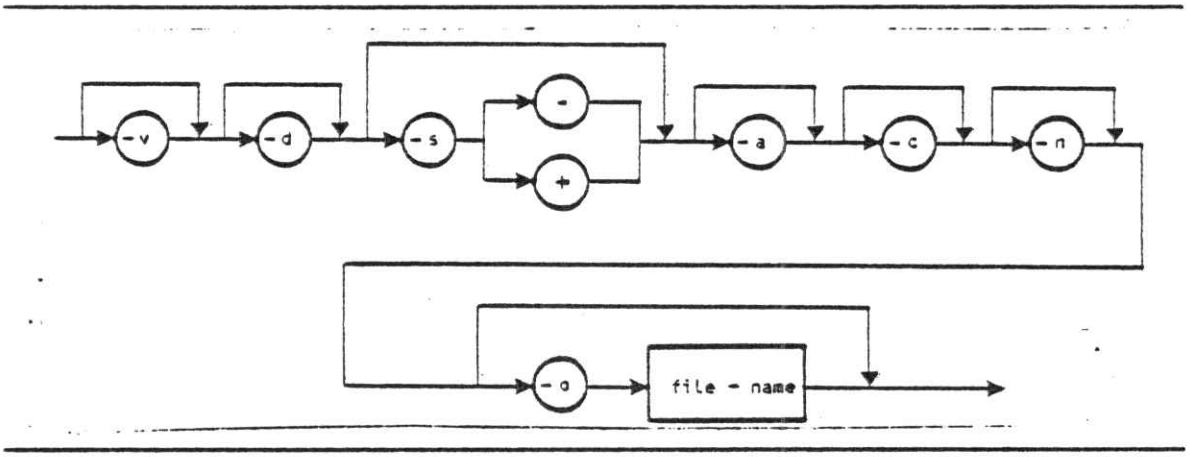
I nomi dei file argomento della compilazione vengono posti al termine della linea di comando di chiamata del compilatore.
Tutti gli argomenti (file), eccetto quelli il cui nome finisce con ".obj", sono considerati dal compilatore come programmi sorgenti PASCAL+; essi sono compilati e ciascun programma oggetto prodotto dalla compilazione e' posto in un file il cui nome e' quello del sorgente con l'estensione finale ".obj". Se il nome del file sorgente finisce con ".p", allora il nome del file oggetto e' quello del sorgente con il suffisso ".obj" al posto di ".p".
Gli argomenti il cui nome finisce con il suffisso ".obj" sono considerati programmi oggetto Z8000 PASCAL+ compatibili: essi possono essere il risultato di precedenti compilazioni o possono costituire librerie di routine PASCAL+ (o PASCAL+ compatibili).

Questi programmi, insieme ai file oggetto risultati dalla compilazione di tutti i sorgenti presenti nella linea di comando, sono concatenati successivamente dal linker "zloc" per costituire un unico programma eseguibile. Se l'utente non specifica, fra le opzioni della linea di comando, l'opzione "-c", cio' avviene automaticamente e i comandi di link

vengono presi da un file standard gia' definito dal sistema. Se l'opzione "-c" e' specificata, sara' compito dell'utente lanciare il "linker" con gli opportuni comandi per ottenere un programma eseguibile. Il nome di questo programma e' "z.out", a meno che sia stata usata l'opzione "-o" (si veda la prossima sezione).
Si tenga presente che i nomi dei file argomento devono essere separati da blank e naturalmente devono seguire la sintassi del sistema operativo MOS.

OPZIONI DI LINEA DI COMANDO DEL COMPILATORE

Il seguente diagramma illustra la sintassi della "Option List" presente all'interno della Linea di chiamata del compilatore.



-v: opzione verbose

L'opzione **-v** fa si' che il compilatore stampi tutte le informazioni relative ad ogni passo di compilazione per ogni file sorgente.

-d: opzione debug

L'opzione **-d** fa si' che il compilatore produca dei file di informazione per il debugger. Inoltre vengono aggiunte opportune direttive al linker "zloc"(si veda il capitolo sul debugger).

-s: opzione share

L'opzione **-s** fa si' che il programma eseguibile generato successivamente dal linker "zloc" sia collegato rispettivamente con un supporto run-time sharabile (-s+) o non sharabile (-s-).
Il valore di default e' "-s+".

-a: opzione di assembly language

L'opzione **-a** fa si' che il compilatore produca un listing disassemblato del file oggetto.

-c: opzione di compilazione

L'opzione -c sopprime la fase di "linking" della compilazione e fa si' che sia prodotto solo il codice oggetto. Se tale opzione viene omessa, il "linker" viene lanciato automaticamente e in modo trasparente all'utente, con i comandi di linking presi da un file di sistema : in tal caso l'output della compilazione sara' il programma eseguibile.

-n: opzione di compilazione per il solo controllo sintattico

L'opzione -n fa eseguire al compilatore il solo controllo sintattico del sorgente. Questa opzione sopprime il passo della compilazione relativo alla generazione del codice oggetto.

-o: opzione di file oggetto

L'opzione -o file-name fa si' che il codice eseguibile generato successivamente dal linker "zloc" sia posto nel file "file-name.out". Se non e' presente, il file eseguibile avra' il nome "z.out".
Uno o piu' spazi devono seguire il comando "-o" nella linea di comando prima che il 'file name' sia specificato.

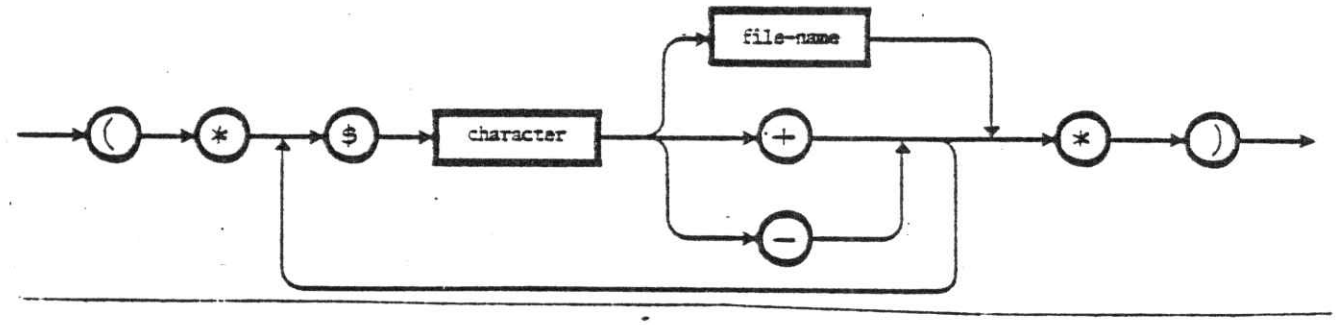
Nota

Nella linea di comando che chiama il compilatore le opzioni descritte, se presenti, devono essere sempre poste fra apici, come indica il diagramma sintattico della Chiamata del Compilatore, e separate da blank.

OPZIONI DEL COMPILATORE NEL SORGENTE

Queste opzioni previste per il compilatore PASCAL+ sono un tipo speciale di commento nel programma sorgente PASCAL+ e permettono l'abilitazione o meno di alcune funzioni fornite dal compilatore durante la compilazione.

La forma generale di tali opzioni e' la seguente:



All'interno di una opzione di compilazione non sono ammessi separatori, cioe' ne' blanks ne' altri caratteri possono apparire fra i simboli specificati nel precedente diagramma sintattico.

Tutte le opzioni di compilazione hanno valori di default assunti implicitamente in caso di loro assenza; questi valori sono specificati nella seguente descrizione.

$L : opzione List

Questa opzione abilita o disabilita la scrittura del listing del programma (messaggi di errore inclusi).
La forma "(*$L+*)" fa si' che il listing sia scritto sul terminale.
La forma "(*$L-*)" disabilita la creazione del listing.
La forma "(*$Lfile-name*)" fa si' che il listing sia scritto nel file specifico file-name.
Il valore di default e' (*$L-*).

Esempio:

```
PROGRAM PEANUT;
 .
 .
VAR ...
 .
 .
(*$L+*)
PROCEDURE ZEUS;
BEGIN
 .
 .
END;
(*$L-*)
 .
 .
CONST ...
 .
 .
TYPE ...
 .
 .
BEGIN (*$LRICHARD*)
 .
 .
(*$L-*)
END.
```

In questo esempio, il compilatore listera' sul terminale la procedura ZEUS e listera' nel file RICHARD il programma main; nessuna altra parte del testo sara' listata.

$R : controllo del Range

Questa opzione abilita o disabilita la generazione di codice addizionale usato per controlli run time, controlli sull'indice di array, controlli di overflow, controlli di divisione per 0, controlli sui limiti di parametri, ...
La forma (*$R+*) causa l'inserimento di tale codice addizionale nel codice generato dalla compilazione del testo che segue tale opzione.
La forma (*$R-*) disabilita la generazione del codice addizionale per la parte di testo che la segue.
Quest'ultima forma e' utile quando si desidera ridurre la dimensione del programma oggetto (ma e' bene essere sicuri della correttezza del programma).
Questa opzione puo' essere attivata o disattivata ovunque all'interno del testo dell'unita' e puo' apparire quante volte si voglia.
Il valore di default e' (*$R+*).

Esempio:

```
PROGRAM DOLL;
VAR A,B,Z : INTEGER;
    M : ARRAY [-180..33] OF REAL;
  .
  .
(*$R-*)
BEGIN
  A:=B+1;
  (*$R+*) A:=ADIVB; (*$R-*)
  (*assicura contro divisioni per 0*)
   .
   .
  (*$R+*)
  IF M[B+18]=0.0 THEN (*assicura contro errori di indice*)
   .
   .
  (*$R-*)
  Z:=Z-14; (*controllo run time non necessario*)
   .
   .
   .
END.
```

$Z : opzione di Clear

La forma (*$Z+*) provoca l'inizializzazione a zero di tutti i dati statici al momento della compilazione, e l'azzeramento di tutti i dati di HEAP a run time quando questi sono allocati.
Il valore di default e' "(*$Z-*)".

$F

La forma (*$F-*) fa si' che tutte le variabili, le procedure, le funzioni, i tipi e le costanti importate in un certo dominio non debbano essere ridefinite all'interno di quel dominio.
La forma (*$F+*) fa si' che ogni variabile, procedura o funzione importata, debba essere ridefinita all'interno del dominio che la importa, e non e' consentito importare costanti o tipi.
Il valore di default e' "(*$F+*)".

$B : opzione di Buffering

Questa opzione consente l'accumulo (buffering) di file run time di output. La forma (*$B-*) fa si' che i buffer di output siano liberati alla fine di ogni istruzione di write, altrimenti questo avviene solo nel caso di "writeln" o di lettura (reading) dal file standard di input

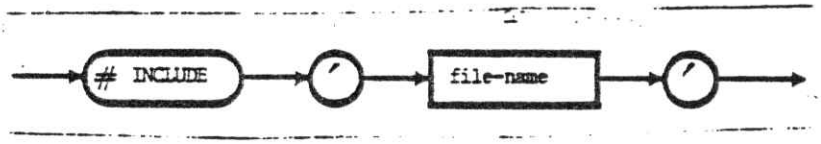
(tastiera). Il porre tale opzione uguale a "(*$B-*)" comporta notevoli conseguenze di performance su molti programmi.
Il valore di default e' "(*$B+*)".

$V : opzione Verbose

La forma (*$V+*) fa si' che per ciascuna procedura o funzione nel programma sorgente il compilatore stampi il nome e la dimensione del suo "Stack Frame".
La forma (*$V-*) fa si' che il compilatore giri senza stampare tali informazioni.
Il valore di default e' (*$V-*).

IL COMANDO DI INCLUDE

Di particolare utilita' risulta il comando #INCLUDE con il quale e' possibile inserire all'interno del codice sorgente ulteriore codice sorgente contenuto in altri file.
Questo comando puo' apparire ovunque nel testo del programma, con l'unica condizione che deve iniziare nella prima colonna del testo.
La sua sintassi e' diversa da quella delle precedenti opzioni ed e' illustrata dal diagramma seguente:



Questo comando e' utile specialmente quando piu' unita' PASCAL+ richiedono una parte comune di testo: in tal caso si puo' scrivere tale testo in un file appropriato ed includerlo in ciascuna unita'.
Da notare inoltre che il comando INCLUDE puo' essere presente all'interno anche della parte istruzioni di una unita' e puo' essere nidificato, cioe' il file da includere puo' a sua volta includere un altro file, ma non sono permessi piu' di due livelli di inclusione.

FORMATO DEL LISTING

Il formato del listing PASCAL+ e' piuttosto semplice: riproduce il testo dell'unita' che e' stata compilata, affiancando ad ogni riga il relativo numero progressivo e il numero progressivo delle singole istruzioni all'interno di ciascuna funzione o procedura.
Si osservi che il numero della seconda colonna si riferisce all'ultima istruzione vista dal compilatore quando e' stata raggiunta la fine di una riga. Percio', se in una riga di sorgente sono presenti due istruzioni, il numero stampato e' riferito alla seconda istruzione.

Esempio

```
 0.    0   (*$L+*)
 1.    0   PROGRAM COPY;
 2.    0    VAR F:FILE OF REAL;
 3.    0    PROCEDURE ASK_AND_RESET;
 4.    0     VAR EXT_NAME:PACKED ARRAY
 5.    0           [1..10] OF CHAR;
 6.    0       I:INTEGER;
 7.    0       C:CHAR;
 8.    0     BEGIN
 9.    2       FOR I:=1 TO 10 DO EXT_NAME [I]:='';
10.    3       I:=0;
11.    4       REPEAT
12.    5        READ(C);
13.    6        I:=I+1;
14.    8        IF I<=10 THEN EXT_NAME [I]:=C
15.    8       UNTIL EOLN;
16.    9        RESET(F,EXT_NAME)
17.    9     END;
18.    0    BEGIN
19.    1      ASK_AND_RESET;
20.    2     WHILE NOT EOF(F) DO
21.    4        BEGIN WRITE(FΩ);
22.    5        GET(F) END;
23.    6     CLOSE(F)
24.    6   END.
```

I numeri della colonna piu' a sinistra (seguiti da un punto) sono i numeri delle righe (a partire da 0)

La seconda colonna di numeri contiene i numeri delle istruzioni: questi rappresentano i numeri progressivi dell'ultima istruzione (non la prima) della riga.
Le istruzioni nidificate sono contate come fossere istruzioni diverse; le dichiarazioni non sono contate.

Un esempio significativo e' il seguente:

```
0.    0   PROGRAM EGG;
1.    0   VAR A: CHAR;
2.    0   BEGIN
3.    1    BEGIN
4.    2     BEGIN
5.    4      READ(A); WRITE(A);
6.    4     END
7.    4    END
8.    4   END
```

Il numero progressivo di istruzione e' utile in fase di "debugging" di un programma, poiche' si puo' far riferimento a questo numero quando si vuole interrompere o riprendere l'esecuzione del programma mettendo dei "breakpoint".

I messaggi di errore, qualora qualche errore di sintassi sia rilevato in fase di compilazione, sono inseriti all'interno del testo del listing, a partire dalla posizione dove e' scoperto l'errore e stampando il numero dell'errore nella linea successiva. La forma e' quella mostrata nell'esempio che segue. Si noti che la riga contenente l'errore di compilazione viene stampata interrota al punto dell'errore (segnalato da <<<<) e riscritta completamente nella riga successiva alla riga di segnalazione.
Per avere una breve spiegazione dell'errore commesso si puo'usare il programma "zpcmesg" descritto nel capitolo 5.

Esempio

```
      .
      .
35.  15  IF A=B THEN C=<<<< ;
***** error number 40 in line 35 *****
35.  15  IF A=B THEN C=8 ;
36.  16  Z:=15 ;
37.  17  WHILE C<>0 DO
      .
      .
      .
```

FORMATO DEL FILE OGGETTO E NOMI DI SEZIONE

Per capire meglio i comandi del "linker" di cui si parlera' nel prossimo capitolo, e' utile descrivere alcune convenzioni riguardanti i nomi dei file sorgenti PASCAL+ e quelli dei file oggetto prodotti dalla compilazione.
Nel seguito si intende che i file il cui nome termina con ".p" sono file sorgenti PASCAL+, mentre i file oggetto prodotti dal compilatore hanno lo stesso nome del file sorgente, con il suffisso ".obj" al posto di ".p".
I file oggetto sono composti da varie "sezioni" i cui nomi sono composti concatenando una "radice" e un "suffisso", assegnati dal compilatore in accordo con le seguenti convenzioni:

1. La radice per il nome delle sezioni che appartengono a un "programma" PASCAL+ e' sempre "PROG".

2. La radice per il nome delle sezioni relative ad un Monitor o un Modulo PASCAL+ e' il nome stesso specificato nell'istruzione "monitor" o "module" all'inizio del file; tuttavia tali nomi hanno sempre i caratteri convertiti in minuscolo.

3. Tutte le sezioni di codice eseguibile hanno un nome fatto della radice (definita come in 1. e 2.), concatenata con il suffisso "_p".

4. Tutti i nomi delle sezioni dei dati di sola lettura sono ottenuti concatenando il nome radice con il suffisso "_k".

5. Tutte le sezioni dei dati di lettura-scrittura hanno nomi il cui suffisso e' "_d".

6. Tutte le sezioni di stack hanno nomi il cui suffisso e' "_s".

7. Le sezioni che identificano l'inizio di un'area di heap hanno il suffisso "_i" e devono sempre essere allocate all'offset 0 all'interno di un segmento.

8. Le sezioni che identificano la fine di un'area di heap hanno il suffisso "_e".

9. In aggiunta a quanto sopra, le sezioni i cui nomi terminano con "_P", "_D" e "_S" esistono come parte del supporto di run-time.
   Tali sezioni appartengono alle stesse classi delle corrispondenti sezioni con i suffissi minuscoli "_p", "_d" e "_s".

Inoltre, l'entry point di un programma PASCAL+ e' definito dal simbolo "Pmain".
I nomi di procedure, funzioni o variabili che siano esportati da un modulo o da un monitor PASCAL+ sono creati in accordo alle seguenti convenzioni: il nome del simbolo a livello del file oggetto e' il nome del modulo/monitor, con i caratteri convertiti in minuscolo, seguito da un carattere di underscore ("_") e dal nome della procedura, funzione o variabile, convertito in caratteri minuscoli e troncato al 12° carattere. Si noti anche che la lunghezza totale dei simboli riconosciuti dal linker e' 16 caratteri. Tutti i caratteri seguenti sono ignorati.

MESSAGGI DI ERRORE DI COMPILAZIONE

In questa sezione viene riportata una lista completa dei codici di errore seguiti ciascuno da una breve spiegazione (in italiano).
Si tenga presente che solo il "numero di errore" appare, dopo la compilazione di un programma nel quale siano stati commessi errori di sintassi,o stampato sul listing o visualizzato sul terminale.
Se il programmatore vuole avere visualizzata su terminale una spiegazione sintetica .dell'errore commesso, puo' usare il programa "zpcmesg" descritto nel Capitolo 5 di questo manuale.

Lista dei codici di errore con spiegazione sintetica :

9 L'opzione di debug deve essere riportata prima della prima linea di programma

10 Troppe cifre

11 Cifra attesa dopo il . in numero reale

12 Overflow di intero

13 Cifra attesa ad esponente

14 Fine di linea incontrato nella costante di tipo string

15 Carattere illegale in input

16 Cifra ottale attesa

17 Incontrata inattesa fine di file (eof)

18 Linea di "include file" non corretta

19 Troppi livelli di inclusione di file

20 Simbolo illegale

21 Errore nel simple type

22 Errore nella parte di dichiarazione

23 Errore nella lista dei parametri

24 Errore nella costante

25 Errore nel tipo

26 Errore nella lista dei campi

27 Errore nel factor

28 Errore nella variabile

29 Atteso un identificatore

30 Atteso un integer

31 Atteso '('

32 Atteso ')'

33 Atteso '['

34 Atteso ']'

35 Atteso ':'

36 Atteso ';'

37 Atteso '='

38 Atteso ','

39 Atteso '*'

40 Atteso ':='

41 'PROGRAM' atteso

42 'OF' atteso

43 'BEGIN' atteso

44 'END' atteso

45 'THEN' atteso

46 'UNTIL' atteso

47 'DO' atteso

48 'TO'/'DOWNTO' atteso

49 'FILE' atteso

50 'IF' atteso

51 '.' atteso

52 'FROM' atteso

100 Identificatore dichiarato due volte

101 Identificatore non della classe appropriata

102 Identificatore non dichiarato

103 Segno non permesso

104 Numero atteso

105 Limite inferiore oltrepassa limite superiore

106 Tipi subrange incompatibili

107 Il tipo di costante deve essere integer

108 Il tipo non deve essere real

109 Il tagfield deve essere scalar o subrange

110 Tipo incompatibile col tipo del tagfield

111 Il tipo dell'index non deve essere real

112 Il tipo dell'index deve essere scalar o subrange

113 Il tipo dell'index non deve essere integer

114 Forward reference non soddisfatta

115 Tipo identificatore di forward reference nella dichiarazione delle variabili

116 Dichiarato forward : ripetizione della lista di parametri non permessa

117 Funzione dichiarata forward : ripetizione del result type non permessa

118 Il tipo della funzione deve essere scalar, subrange, o pointer

119 Parametro file value non permesso

120 Omesso il tipo della funzione nella dichiarazione della funzione

121 F-format solo per reali

122 Errore nel tipo di parametro della funzione standard

123 Errore nel tipo di parametro della procedura standard

124 Numero di parametri non in accordo con la dichiarazione

125 Sostituzione illegale di parametri

126 Tipo del risultato di funzione parametrica non in accordo con la dichiarazione

127 L' espressione non e' di tipo set

128 Test permessi solo su uguaglianze

129 Inclusione stretta non permessa

130 Confronto di file non permesso

131 Tipo di operando(i) non legale

132 Il tipo di operando deve essere boolean

133 Il tipo dell'elemento del set deve essere scalar o subrange

134 Tipi di elementi del set non compatibili

135 Il tipo di variabile non e' array o string

136 Il tipo dell'index non e' compatibile con la dichiarazione

137 Il tipo della variabile non e' record

138 Il tipo della variabile deve essere file o pointer

139 Tipo illegale per la variabile di controllo del loop

140 Tipo illegale di espressione

141 Assegnamento di file non permesso

142 Tipo della label non compatibile con l'espressione selezionante

143 I limiti del subrange devono essere scalari

144 Conflitto di tipi fra gli operandi

145 Assegnamento a funzione standard non permesso

146 Assegnamento a funzione permesso solo nel corpo della funzione

147 Nessun campo simile in questo record

148 Errore di tipo in lettura

149 Il parametro attuale deve essere una variabile

150 Case label definita piu' di una volta

151 Omessa la corrispondente dichiarazione di parte variante

152 Tagfield real o string non permessi

153 Una precedente dichiarazione non era stata resa forward

154 La sostituzione di proc/func standard non e' permessa

155 Label definita piu' di una volta

156 Label dichiarata piu' di una volta

157 Label non definita

158 Label non dichiarata

159 Value parameter atteso

160 Variante del record multidefinita

161 File non ammesso qui

162 Direttiva al compilatore sconosciuta (ne' 'external' ne' 'forward')

163 Una variabile non puo' essere un packed field

164 L'insieme dei reali non e' permesso

165 I campi di packed record e gli elementi di packed array non devono essere parametri "var" o argomenti usati per lettura

166 L'espressione case selector deve essere scalar o subrange

167 Il parametro attuale deve essere una costante

200 Il file nel programma non e' nel header ne' dichiarato come file globale

201 Identificatore esportato non dichiarato come routine globale

202 La procedura esportata da accesso a un tipo pointer

203 Il tipo del risultato della funzione esportata da accesso a un tipo pointer

204 Nessuna definizione per procedure o funzioni importate

205 Il descrittore da accesso a un tipo pointer

206 Restrizione nell'implementazione : gli export devono apparire prima della procedura

207 Forza il descrittore a fornire accesso ad un puntatore nel risolvere un descrittore dichiarato forward

208 Direttiva AT permessa solo per variabili globali

209 Una condizione non puo' essere passata come valore

210 Tentativo di fornire accesso a puntatore ad una procedura importata

211 Nessuna siffatta routine in questo monitor o modulo

212 Il tipo del risultato della procedura importata da accesso a puntatore

213 Sezioni di import ed export permesse solo a livello globale

214 Definizione di procedura che non e' importata

215 Non si puo' rendere forward o external una procedura importata

216 Non si puo' fornire un corpo ad una procedura importata

217 File globali solo per il programma, non per monitor o moduli

218 Costante set fuori dal range da 0 a 127

219 Array conformi non possono essere confrontate in questo modo

220 Le liste di argomenti di procedure formali e procedure attuali non combaciano

221 I tipi dei risultati di procedure formali e procedure attuali non combaciano

222 Non si puo' passare una procedura standard come parametro di procedura

223 Procedure e funzioni non possono essere scambiate l'una per l'altra

224 Stringa nome-file richiesta per reset e rewrite

225 Wait e signal richiedono condizioni come argomento

225 Wait e signal disponibili solo nei monitor

226 Attesa un'array

227 Packed array non permessa in questo contesto

228 Tipi incompatibili con il tipo dell'index dell'array

229 Packed array richiesta in questo contesto

230 Elementi dell'array di tipi non compatibili

231 Atteso descrittore

232 Atteso puntatore

233 La procedura precedente contiene un goto ad una statement strutturata o un goto a una label inesistente

234 La variabile del for deve essere dichiarata localmente

235 Variabile di controllo di un for loop assegnata o letta all'interno del for loop

236 Chiamate permesse solo per procedure sottoposte a fork

237 Fork non permesso per procedure formali o standard

238 Il numero di un segmento per un indirizzo assoluto deve essere in 0..127

239 Usare := negli assegnamenti

240 Non possono essere copiate condizioni in statement di assegnamento

241 Attesa costante di tipo string per la versione

300 Troppi record scope nidificati

301 Limiti del set fuori del range

302 Limiti di stringa fuori del range

303 Troppe procedure/function nidificate

304 Troppi registri richiesti, espressione troppo complicata

305 Troppi registri di floating point richiesti

306 Troppi registri dispari richiesti

307 Non disponibile coppia di registri per DIV o MOD

1000 Errore del compilatore, riferire al system manager

2. COLLEGAMENTO E CARICAMENTO DI UNITA' PASCAL+: IL LINKER ZLOC

Scopo di questo capitolo e' mostrare come si collegano le unita' compilate che costituiscono un programma PASCAL+ e come esse vadano allocate per una successiva esecuzione del programma.

Le operazioni di collegamento e allocazione sono ottenute da un unico programma sull'M30/M40 chiamato "linker" .

Sono descritte per prima cosa le caratteristiche del linker e le principali operazioni da esso effettuate. In seguito si parla della "chiamata del linker" e dei suoi comandi.

CARATTERISTICHE DEL LINKER

Il linker PASCAL+ e' uno strumento interattivo che accetta come input un gruppo di file oggetto e produce in output un unico file eseguibile sotto il controllo di una lista di comandi ,preparata dall'utente se nella chiamata del compilatore e' presente l'opzione -c ; se tale opzione non e' presente, il linker e' lanciato automaticamente dal sistema al termine della compilazione e i comandi sono presi da un file di sistema.

Possono essere richieste ulteriori attivita', come ad esempio la produzione di un file mappa, contenente una copia della lista dei comandi eseguiti, messaggi diagnostici, una mappa ordinata per locazioni di memoria delle sezioni e una lista ordinata alfabeticamente dei nomi delle sezioni e dei simboli globali con le loro corrispondenti locazioni.

E' possibile richiedere un compattamento del codice per rendere minima la dimensione del file di output e piu' veloce il caricamento. E' fornita anche una facility di ricerca di libreria. Un'altra opzione fa si' che venga generata una versione, comprensibile al sistema, del file map, cioe' un file binario con i nomi delle sezioni e i relativi indirizzi fisici assoluti, per l'uso del debugger simbolico.

Al momento del link ogni sezione e' posta in corrispondenza con un segmento del sistema; l'utente ha esplicito controllo sulla disposizione delle sezioni di programma in segmenti di memoria mediante il parametro Block Descriptor.

Come lanciare un programma eseguibile

Una volta creato il programma eseguibile nel file "file-name".out, esso puo' essere lanciato dall'utente in ambiente Shell semplicemente digitando sulla tastiera il nome "file-name".out, eventualmente preceduto dal pathname della directory che contiene tale file, o digitando il pathname di un qualsiasi altro file in cui sia stato posto il programma eseguibile.

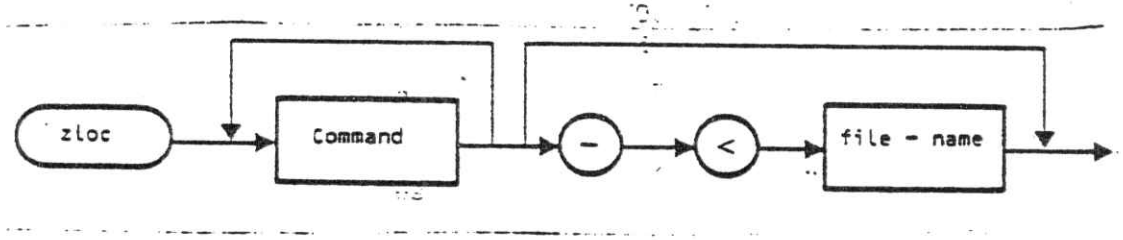
LA CHIAMATA DEL LINKER

Il linker PASCAL+ viene chiamato, in ambiente Shell, specificandone il nome "zloc" (in caratteri minuscoli) e facendo seguire una lista di comandi mediante i quali vengono passati i parametri desiderati.

ORGANIZZAZIONE DELLA LISTA DEI COMANDI

La lista dei comandi viene specificata come argomento del comando "zloc". Se non vengono forniti argomenti, la lista dei comandi e' richiesta all'operatore in modo interattivo. Se invece gli argomenti sono dati, essi sono considerati come lista dei comandi con il simbolo "-<" posto a significare la redirezione dell'input dalla tastiera al file il cui nome segue il carattere < (si veda il manuale "MCL Guida dell'utente").

La sintassi di chiamata del comando di link e' illustrata dal seguente diagramma:



dove:

Command e' uno qualsiasi dei comandi che saranno descritti piu' avanti.

-< indica la redirezione dell'input verso il file "file-name". Cio' significa che gli ulteriori comandi/argomenti di "zloc" saranno presi dal file "file-name".

file-name e' il nome di un file che contiene a sua volta una lista di comandi.

Nota : Il comando

COMMAND file-name

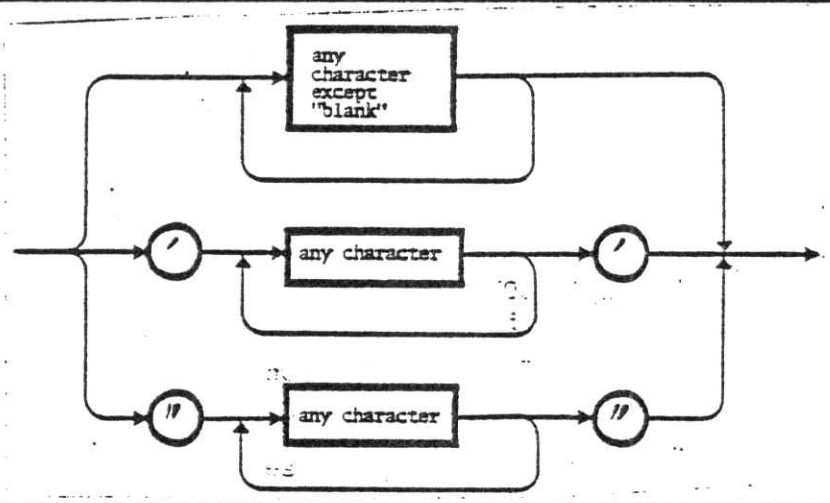
puo' essere usato all'interno della sequenza dei comandi per inserire comandi prendendoli da un altro file indicato. Solo un livello e'

permesso, cioe' non si puo' inserire nuovamente il comando COMMAND all'interno del file da esso specificato.

Stringhe, Espressioni, Termini e Commenti

I comandi del linker devono seguire una sintassi specifica.
Descriveremo la sintassi dei comandi facendo riferimento alle voci seguenti:

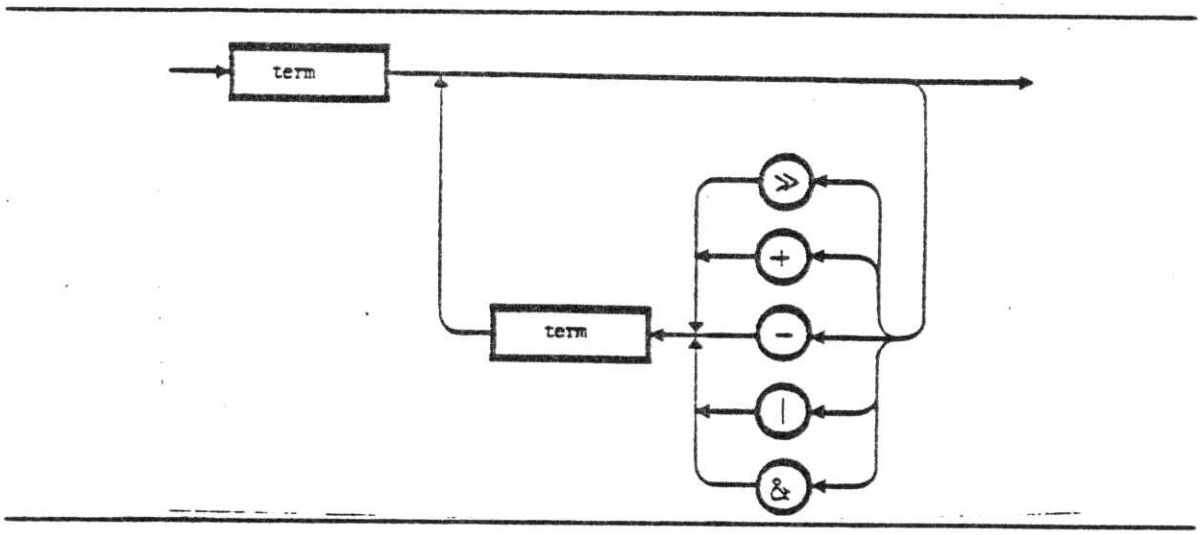
Stringhe



Esempi di stringhe:

ABC
"A B CDEF"

Espressione



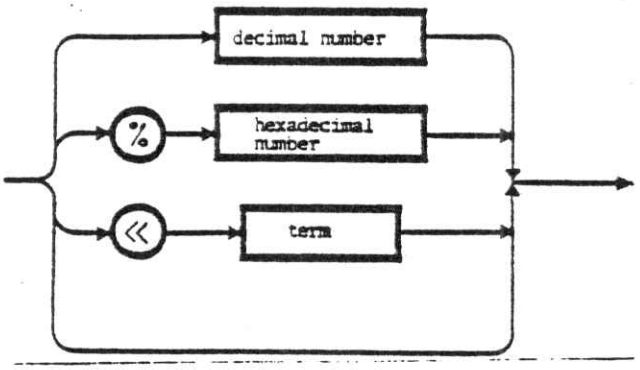
dove :

term
e' la voce "termine" spiegata qui di seguito

Esempi di Espressioni:

%3AF
50+28
<<%7F>>11&5
All'interno di una espressione non sono permessi blank, ne' commenti, ne' segni di punteggiatura.

Termine



Il significato dell'operatore all'interno del termine e' il seguente:

\+ addizione
\- sottrazione
| OR logico
& AND logico
>> esegue la seguente operazione:

sposta a sinistra di 24 bit l'operando di sinistra e gli somma l'operando di destra. Questo operatore e utile nel caso si debbano rappresentare indirizzi segmentati.

Le seguenti espressioni rappresentano lo stesso valore numerico e sono percio' equivalenti:

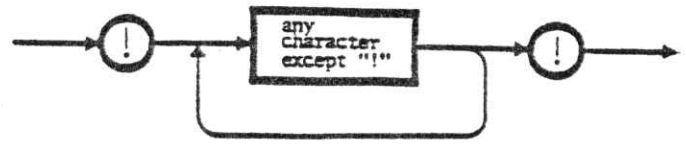
%0E00143C
%0E>>%143C
<<%0E>>%143C

Il termine nullo e' permesso solo se segue immediatamente l'operatore >>, e rappresenta lo zero.
Le seguenti espressioni sono equivalenti:

<<33>>
<<33>>0

Commenti

I commenti sono sequenze di caratteri che non costituiscono comandi al linker; essi sono racchiusi fra punti esclamativi, come mostra il diagramma seguente:



Commenti, spazi, tab, carriage return, virgole e punti e virgola possono capitare ovunque come separatori, eccetto che all'interno delle espressioni.

CLASSIFICAZIONE DEI COMANDI

I comandi che vengono passati come parametri al linker sono tutti, tranne uno, di tipo keyword, cioe' costituiti da una parola chiave eventualmente seguita da altri parametri. Li possiamo elencare ordinati nel modo seguente:

1. Comandi di tipo "multifile keyword", che hanno come parametro uno o piu' "file name". Sono:

   - INPUT
   - LIBRARY

2. Comandi di tipo "file keyword", che hanno come parametro un solo "file name". Sono:

   - COMMAND
   - MAP
   - OUTPUT

3. Comandi di tipo "symbol keyword", che hanno come parametri un "file-name" ed uno o piu' "symbol selector":

   - INSYM
   - SYMBOL

4. Comandi di tipo "simple keyword" costituiti dalla sola parola chiave:

   - NOWARNINGS
   - OPTIMIZE
   - QUIET
   - SQUEEZE
   - STATISTICS

5. Comandi di tipo "value keyword", che hanno come parametro un'espressione:

   - ATTRIBUTES
   - TYPE

6. Comando "entry command" avente come parametro un'espressione o una stringa di caratteri:

   - ENTRY

7. Comando "message command" avente come parametro una stringa di caratteri:

   - MESSAGE

8. Comando "Block Descriptor", che determina l'ordine con cui le sezioni di programma saranno caricate in memoria. Non e' introdotto da alcuna parola chiave, ma e' costituito da successive coppie "espressione pattern".

ORDINE DEI COMANDI

L'ordine con cui appaiono i comandi non ha grande importanza sul risultato dell'operazione di linking eccetto che nella descrizione del BLOCK DESCRIPTOR.
Ha invece importanza l'ordine relativo dei comandi TYPE e ATTRIBUTES e del BLOCK DESCRIPTOR, poiche' i valori indicati dai comandi TYPE e ATTRIBUTES correnti sono assegnati al primo segmento del primo BLOCK DESCRIPTOR che li segue nell'ordine dato.

Altri effetti dell'ordine dei comandi consistono nel fatto che i file sono aperti e dei flag sono settati quando ricorrono i relativi comandi. Per esempio, i comandi che compaiono prima del comando MAP non sono scritti nel file MAP, e comandi prima del comando QUIET sono ripetuti sul terminale, a meno che il comando QUIET compaia nella linea di comandi prima del simbolo "-<".
Il comando SQUEEZE non ha effetto a meno che non appaia prima del comando MAP, dato che i buffer di I/O di map vengono allocati quando e' incontrato il comando MAP.

DESCRIZIONE DEI COMANDI

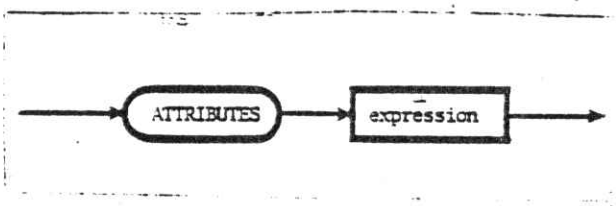
Viene riportato qui di seguito un elenco ordinato alfabeticamente di tutti i comandi con la sintassi relativa e una breve descrizione. Il lettore potra' usare questa parte del manuale come reference.

ATTRIBUTES

Il comando "ATTRIBUTES" e' usato per assegnare un valore al byte "attribute" del segmento successivamente creato dal "BLOCK DESCRIPTOR" e stabilire il tipo di protezione hardware di tale segmento. Infatti ogni segmento creato nel file eseguibile di output ha un byte "attribute" e un byte "type".
Il valore di default del byte attribute e' 0.
Le costanti usate per dicharare l'attributo di un segmento sono:

- 0 : rende il segmento accessibile per scrittura lettura ed esecuzione (RDWRT)
- 1 : rende il segmento accessibile solo per lettura ed esecuzione (RDONLY)
- 8 : rende il segmento accessibile solo per l'esecuzione (XQTONLY)
- 32 : segnala che il segmento ha funzione di stack (STKATTR)

La sintassi del comando e' la seguente:



Questo comando puo' ricorrere un numero qualsiasi di volte. Il valore dell'espressione deve essere minore di 256 (anche se attualmente per il sistema operativo MOS il valore massimo e' 32).

BLOCK DESCRIPTOR

Il "BLOCK DESCRIPTOR" specifica, per mezzo dei nomi di sezione (definiti come nei moduli di input), esattamente quali sezioni di codice debbano essere allocate ed in quali segmenti. Puo' essere specificato un qualsiasi numero di BLOCK DESCRIPTOR ; uno almeno e' sempre richiesto. L'espressione con cui inizia il BLOCK DESCRIPTOR fornisce l'indirizzo di memoria di partenza per la prima sezione. Tale espressione e' un indirizzo di 32 bit col numero di segmento nella posizione corretta per un "long address" dello Z8000 (cioe' il numero di segmento nei bit 24-30, l'offset in 0-15). Ogni sezione nominata (o individuata mediante caratteri jolly) nello stesso descrittore, e' posta in memoria immediatamente di seguito a quella precedente. In caso di codice non segmentato, tutti i blocchi debbono essere allocati nel segmento 0.

I "pattern" in genere sono nomi di sezioni presenti nel file oggetto di input, ma possono includere i seguenti caratteri jolly: "*" che individua una qualsiasi stringa alfanumerica, "?" che individua ogni singolo carattere, "[ab...]" che individua uno qualsiasi dei caratteri fra parentesi quadre e "[a-b]" che individua un qualsiasi carattere nell'intervallo a-b.
Un "pattern" e' valido per tutti i nomi di sezione che sono individuati da lui, e che non sono stati usati nel descrittore corrente o in descrittori precedenti. I nomi sono presi nell'ordine nel quale essi sono presenti nei moduli oggetto di input (vale a dire prima le sezioni di ordine inferiore).
Questa caratteristica puo' essere usata, ad esempio, per separare sezioni di programma da sezioni di dati.
Se a tutte le sezioni di codice, ad esempio, e' stata assegnata l'estensione _p e a tutte le sezioni di dati e' stata assegnata l'estensione _d, allora specificando il BLOCK DESCRIPTOR :

```
"0 *_p *_d"
```

si otterra' che tutte le sezioni di codice siano raccolte per prime seguite poi da tutte le sezioni di dati e sia le une che le altre saranno poste nel segmento 0.

Nota

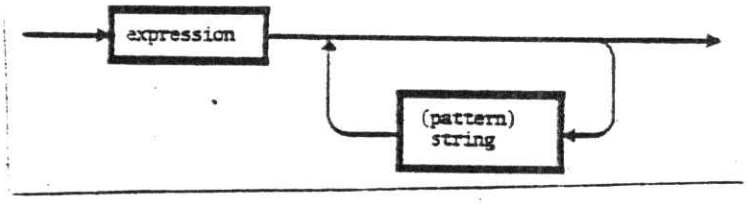
Si osservi che il BLOCK DESCRIPTOR dell'esempio precedente non seleziona le eventuali sezioni con suffisso _P o _D; se l'utente desidera selezionare anche queste sezioni, dovra' specificare il seguente BLOCK DESCRIPTOR:

```
"0 *_[pP] *_[dD]"
```

Se l'ordine delle sezioni di programma non ha importanza, allora e' sufficiente specificare il seguente BLOCK DESCRIPTOR

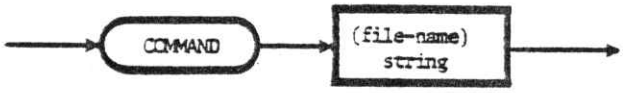
```
"0 *"
```

La sintassi del BLOCK DESCRIPTOR e' illustrata dal seguente diagramma :



COMMAND

Il comando COMMAND puo' essere usato nella sequenza dei comandi per "inserire" comandi prendendoli dal file specificato come parametro. Questo comando puo' incorrere un numero qualsiasi di volte, ma solo un livello e' permesso, cioe' il file specificato non puo' contenere a sua volta un comando COMMAND.
La sintassi e' illustrata dal diagramma seguente:

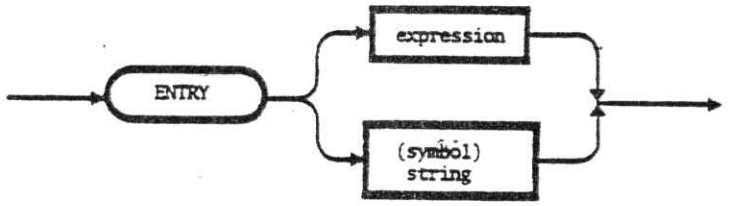


ENTRY

Il comando ENTRY puo' ricorrere al piu' una volta. Esso fornisce o un valore numerico o un nome di simbolo globale che deve costituire l'"entry point" del programma eseguibile. L'"entry point" e' determinato come segue:

- se e' presente il comando ENTRY, allora e' usato l'operando espressione o simbolo, indipendentemente da qualsiasi definizione all'interno dello stesso modulo di input
- se il comando ENTRY non e' presente, allora l'"entry point" e' posto come definito nel modulo di input
- se il modulo di input non definisce un "entry point", allora diventa "entry point" l'indirizzo del primo BLOCK DESCRIPTOR (si veda il comando BLOCK DESCRIPTOR)

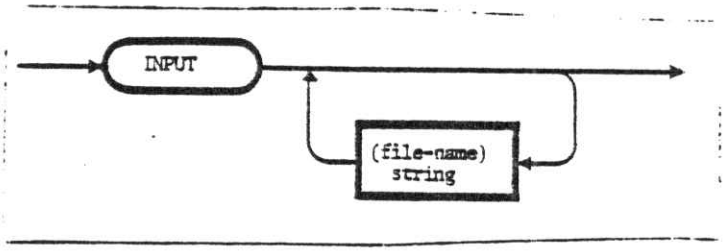
La sintassi e' la seguente:



le nome del simbolo globale e': nome modulo_ nome procedure in minuscolo.

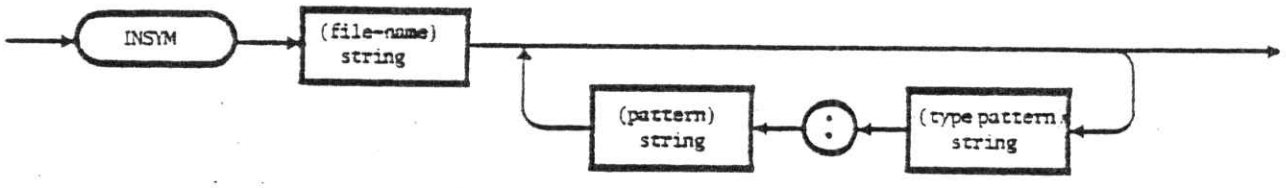
INPUT

Il comando INPUT puo' ricorrere un qualsiasi numero di volte nella lista dei comandi. Esso indica i file oggetto che contengono tutte le sezioni di codice da collegare tramite il linker "zloc".
Tutti i moduli devono essere moduli segmentati o moduli non segmentati.



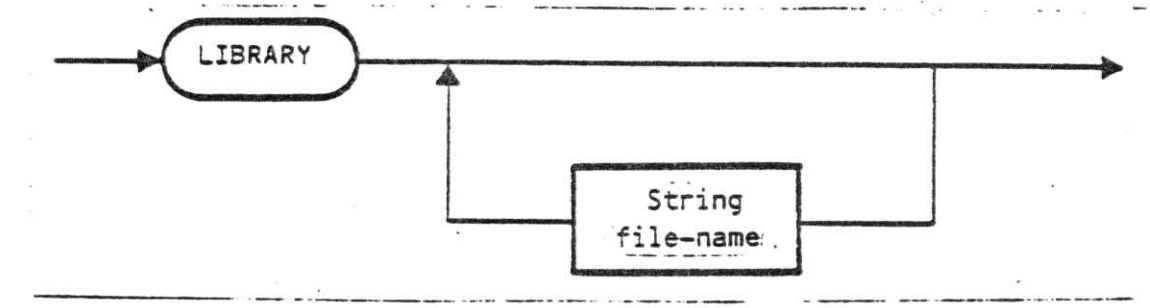
INSYM

Il comando INSYM puo' ricorrere un qualsiasi numero di volte. Esso seleziona i simboli del file di simboli specificato in accordo con i "pattern" specificati.
Questo comando e' usato per scambiare valori di simboli fra due distinte operazioni di "zloc". La sintassi e' illustrata dal diagramma seguente:



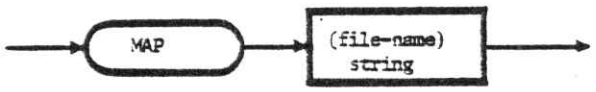
LIBRARY

Il comando library e' usato per istruire il programma a selezionare dai file di libreria nominati solo quei moduli che sono stati usati nel file di input.
La sintassi del comando e' la seguente:



MAP

Il comando MAP puo' ricorrere al piu' una volta. Esso indica il file in cui deve essere posta la Mappa formattata del programma eseguibile. Se tale file non esiste esso e' creato, altrimenti e' completamente sostituito con la nuova Mappa. Se il comando MAP non viene dato, il file mappa non e' creato.
Il file Mappa contiene una copia della lista di comandi eseguita, messaggi diagnostici, una lista dei nomi dei file, dei moduli e delle sezioni di input, una mappa ordinata per locazioni di memoria delle sezioni e una lista ordinata alfabeticamente dei nomi delle sezioni e dei simboli globali con le loro corrispondenti locazioni di memoria.
La sintassi del comando MAP e' illustrata dalla figura seguente:



MESSAGE

Il comando MESSAGE puo' ricorrere una sola volta. Esso fornisce il testo da includere nella parte messaggio della testata del file eseguibile MOS. Se non viene dato alcun comando MESSAGE, un messaggio nullo e' incluso nel file eseguibile prodotto dal linker.
La sintassi e' illustrata dalla figura seguente:

NOWARNINGS

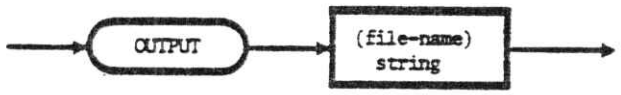
Per default, vari messaggi di warning sono inclusi nei messaggi diagnostici prodotti da "zloc"
Se e' presente il comando NOWARNINGS, allora tutti i messaggi di warning sono soppressi.
Questo comando non ammette alcun parametro.

OPTIMIZE

Il comando OPTIMIZE nella sequenza dei comandi fa si' che il file di output sia ottimizzato in un numero minimo di Sezioni di dati. Cio' implica che venga effettuata una ulteriore passata, quindi rallenta l'operazione di "linking".
Non si usi questo comando se non e' strettamente necessario.
Questo comando non ammette alcun parametro.

OUTPUT

Il comando OUTPUT ricorre una ed una sola volta nella lista dei comandi. Esso specifica il file dove si vuole che sia posto il file eseguibile (formato MOS). Il file e' creato se non esiste o e' completamente sostituito col nuovo output se gia' esiste.
La sintassi del comando e' la seguente:



QUIET

Il comando di QUIET provoca la soppressione dell'output del linker che e' normalmente spedito verso il terminale, fatta eccezione per i messaggi di "fatal error". Se nessun comando di QUIET e' dato, vengono spedite importanti informazioni al terminale. Queste consistono normalmente nella ripetizione della lista dei comandi e nei messaggi di warning (a meno che sia stato dato il messaggio NOWARNINGS).
Questo comando non ammette alcun parametro.

SQUEEZE

Il comando SQUEEZE fa si' che il "linker" usi buffer di dimensione minima possibile. L'effetto di questo comando e' permettere l'uso del maggior spazio possibile di memoria per le strutture di dati a link time e per la symbol table; lo svantaggio e' una certa diminuizione di velocita'.
Questo comando non ammette alcun parametro.

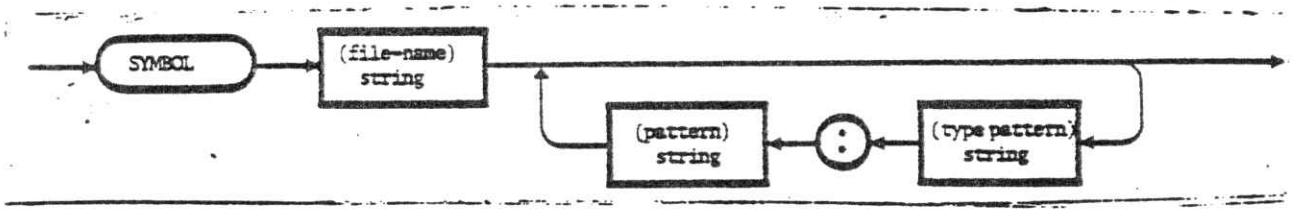
STATISTICS

Il comando STATISTICS, se specificato, fa si' che il programma produca in output delle statistiche su quanta memoria di "zloc" e' stata usata.
Questo comando non ammette alcun parametro.

SYMBOL

Il comando SYMBOL specifica che deve essere creato un "file di simboli" contenente una forma, leggibile dal calcolatore, del file MAP.

Il file dei simboli e' un file binario con i nomi dei simboli e i loro indirizzi fisici assoluti, che possono risultare utili per successive operazioni di "zloc" o per il debugger simbolico.
La sintassi di questo comando e' illustrata dalla figura seguente:



Questo comando puo' ricorrere piu' di una volta. Per ogni occorrenza del comando SYMBOL viene creato un file di simboli.
Tale file ha la seguente forma:

<testata> <corpo>

<Testata> : e' costituita dai primi 10 byte del file e contiene il nome del programma seguito dal numero (ASCII) di release di 6 byte;

<corpo> e' la parte rimanente del file.

Il <corpo> del file e' costituito da record di lunghezza variabile aventi il formato seguente:

<indirizzo> : e' un valore di4-byte (primo va considerato il byte piu' significativo o di ordine piu' alto) che fornisce la locazione di memoria del simbolo.

<tipo> : e' un simbolo tipo di un byte.
0 significa nome globale
1 significa nome di sezione
2 significa nome comune

<lunghezza> : e' un byte, che fornisce la lunghezza in byte del seguente campo <simbolo>.

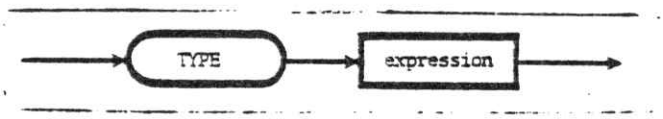
<simbolo> : e' il testo ASCII del simbolo che e' lungo <lunghezza> byte. Questo e' l'unico campo di lunghezza variabile del record.

TYPE

Il comando TYPE e' usato per assegnare un valore al byte "type" del segmento successivamente creato dal BLOCK DESCRIPTOR. Questo valore informa il sistema sul contenuto di tale segmento e su come debba essere trattato e non puo' superare 256 (anche se attualmente il valore massimo per il sistema operativo MOS e' 10).
Le costanti usate per dichiarare il tipo di segmento sono:

- 1 : codice solo eseguibile (XQTCODE)
- 2 : codice di sola lettura (RDCODE)
- 3 : codice di lettura-scrittura (RWCODE)
- 4 : dati di sola lettura (RDDATA)
- 5 : dati per lettura e scrittura (RWDATA)
- 6 : segmento di stack (STACK)
- 7 : codice rilocabile per sola lettura (RLRDCODE)
- 8 : codice rilocabile per lettura e scrittura (RLRWCODE)
- 9 : dati rilocabili per sola lettura (RLRDDATA)
- 10: dati rilocabili per lettura e scrittura (RLRWDATA)

Questo comando puo' ricorrere un qualsiasi numero di volte.
La sintassi del comando e' la seguente:



N.B. : nel assegnare un valore al byte "attribute" e al byte "type" di un segmento deve essere rispettata la seguente relazione di compatibilita' :

Attribute value	compatible types
0	3, 5, 8, 10
1	2, 4, 7, 9
8	1
32	6

ESEMPI

Per creare il file eseguibile "reader" dal file oggetto "reader.obj" ponendo tutto il file nel segmento 0 e lasciando una mappa di allocazione nel file "reader.map" , basta digitare il comando :

```
zloc input reader.obj output reader
map reader.map  0 *
```

Come si vede il BLOCK DESCRIPTOR "0 *" indica che tutte le sezioni di programma (il carattere jolly * infatti le individua tutte) devono essere poste nel segmento 0.

L'esempio che segue e' piu' complesso; l'utente, mediante la lista dei comandi, da' una serie di direttive al "linker".
Lo scopo e' creare un file eseguibile "reader" dal file oggetto "reader.obj" specificando l'entry point" al simbolo globale "START" e ponendo tutte le sezioni i cui nomi terminano con "_C" nel segmento 3 all'offset hex 2000, tutte quelle i cui nomi terminano con "_D" eccetto che "STRINGS_D" nel segmento 4 e la sezione "STRINGS_D" nel segmento hex 7F.
Al segmento 3 deve essere assegnato l' attributo di hex 20 ed il tipo 1, al segmento 4 l' attributo di hex 30 e il tipo 2, e al segmento hex 7f un attributo di hex 38 e un tipo di 2.

Nota : la non corrispondenza di tipi e attributi usati nell'esempio con quelli definiti nei comandi TYPE e ATTRIBUTES dipende solo da una differente configurazione di sistema operativo .

I comandi devono essere posti nel file "reader.loc".
Il campo "messaggio" del file eseguibile contiene solitamente il proprio numero di versione ed in questo caso particolare si desidera porre il testo "Debug Version" in questo campo. Poiche' non si desidera ricevere molti messaggi in output al terminale, viene dato il comando QUIET.

```
zloc Quiet Message  'Debug Version'  -< reader.loc
```

e il file "reader.loc" (locater) contiene:

```
Map     reader.map              ! Definisce il file map con il nome standard !

!    File di comandi per "reader"    !

Input   reader.obj              ! Modulo oggetto Z-type da qui               !
Output  reader                  ! File eseguibile                            !
Type 1, Attributes %20
<<3>>%2000                      ! Segmento di codice, dopo lo spazio di      !
                                ! memoria ROM                                !
        *_C                     ! Seleziona tutte le sezioni di codice       !

Type 2, Attributes %38
<<%7f>>0                        ! Segmento condiviso (shared) contenente     !
        STRINGS_D               ! stringa dati                               !

Type 2, Attributes %30
<<4>>                           ! Segmento dati privato                      !
        *_D                     ! Tutte le altre sezioni di dati             !


Entry   START                   ! L'entry point di questo programma e' a     !
                                ! START                                      !
```

il file mappa, reader.map, ha i seguenti contenuti:

```
Olivetti ZLOC  —  Release 4.0

Comandi:

zloc Quiet Message 'Debug Version' -

Map     reader.map              ! Definisce il file map con il nome standard !

!    File di comandi per "reader"    !

Input   reader.obj              ! Modulo oggetto Z-type da qui               !
Output  reader                  ! File eseguibile                            !
Type 1, Attributes %20
<<3>>%2000                      ! Segmento di codice, dopo lo spazio di      !
                                ! memoria ROM                                !
        *_C                     ! Seleziona tutte le sezioni di codice       !
```

```
Type 2, Attributes %38
<<%7f>>0                        ! Segmento condiviso (shared) contenente     !
        STRINGS_D               ! stringa dati                               !

Type 2, Attributes %30
<<4>>                           ! Segmento dati privato                      !
        *_D                     ! Tutte le altre sezioni di dati             !

Entry   START                   ! L'entry point di questo programma e' a     !
                                ! START                                      !
```

```
Warnings:

(Pass One - reader.obj)
    Section "Rid06" has not been included (matches no pattern)
    Section "MAXIM_TAB" has not been included (matches no pattern)
(Pass Two - reader.obj)
    Undefined Global Symbol "SYSTEM" referenced by section "main_C"
    Excluded section "Rid06" referenced by section "main_C"
(Generating Maps)
```

```
Input Map:

File            Module      Section     Size (HEX)

reader.obj
                reader
                            gorgon_C    036F
                            gorgon_D    0033
                            Rid06       024F
                            MAXIM_TAB   C46E
                            signet_C    08D6
                            signet_D    C56D
                            Elab_C      05F8
                            Elab_D      03A8
                            CODE_C      01E8
                            CODE_D      013F
                            STRINGS_D   0272
                            main_C      0207
                            main_D      0138
```

```
Segment Map (all values are in HEX):

Segment Offset Size End Section


<<03>>
          2000    036F    236E    gorgon_C
          2370    0207    2576    main_C
          2578    08D6    2E4D    signet_C
          2E4E    01E8    3035    CODE_C
          3036    05F8    362D    Elab_C
<<04>>
          0000    0033    0032    gorgon_D
          0034    0138    016B    main_D
          016C    C56D    C6D9    signet_D
          C6DA    013F    C818    CODE_D
          C81A    03A8    CBC1    Elab_D

<<7F>>
          0000    0272    0271    STRINGS_D
```

```
Global Symbols and Section Names (- indicates Section Name):

Symbol          Location (HEX)          Section

CODE_C            <<03>>2E4E              -
CODE_D            <<04>>C6DA              -
Contents          <<03>>2622              Signet_C
DERIVATE          <<03>>351C              Elab_C
delete            <<03>>2FF6              CODE_C
ELABORATE         <<03>>3036              Elab_C
Elab_C            <<03>>3036              -
Elab_D            <<04>>C81A              -
EXPLICATE         <<03>>334E              Elab_C
gorgon_C          <<03>>2000              -
gorgon_D          <<04>>0000              -
gor_Answer        <<03>>212F              gorgon_C
gor_Demand        <<03>>2344              gorgon_C
gor_Names         <<04>>0002              gorgon_D
gor_Quest         <<03>>2000              gorgon_C
gor_Reply         <<03>>2250              gorgon_C
Heading           <<03>>2D10              signet_C
insert            <<03>>2F26              CODE_C
ModelList         <<04>>C6DA              CODE_D
main              <<03>>2370              main_C
main_C            <<03>>2370              -
```

```
main_D          <<04>>0034          -
quit            <<03>>256E          main_C
START           <<03>>2378          main_C
signet_C        <<03>>2578          -
signet_D        <<04>>016C          -
STRINGS_D       <<7F>>0000          -
SYSTEM          <<7F>>FFFF          ** UNDEFINED **
SubHeading      <<03>>282C          signet_C
Table           <<04>>016C          signet_D
Title           <<03>>2578          signet_C

ZLOC Complete
```

3. IL DEBUGGER SIMBOLICO PASCAL+

Il Debugger Simbolico Pascal+ offre un controllo interattivo completo di un programma Pascal+ e permette all'utente di individuare quegli errori di programma che non possano essere individuati in fase di compilazione. Puo' capitare infatti, ad un certo punto del programma, che la sintassi sia giusta ma che l'algoritmo sia sbagliato, per cui il risultato ottenuto dall'esecuzione del programma non e' quello desiderato (ad esempio un numero errato di loop in una istruzione, un cambiamento ai valori di una variabile non desiderato ma sintatticamente corretto, e simili). Il Debugger e' orientato verso l'utente e non e' necessaria la conoscenza della sottostante architettura del sistema; il debugger permette all'utente di controllare il suo programma usando simboli e strutture come definiti nel suo programma sorgente. Questo significa che l'utente ha a che fare con istruzioni PASCAL+, procedure e variabili piuttusto che istruzioni macchina e registri.

L'utente controlla l'esecuzione del programma tramite dei breakpoint di controllo inseriti alle istruzioni PASCAL+ scelte. Essi costituiscono dei punti in cui l'esecuzione del programma viene temporaneamente sospesa dal Debugger.
Il Debugger e' implementato come un processo separato che controlla il programma utente, come fosse un processo figlio, in risposta ai comandi passatigli dall'utente.

La visibilita' per l'utente degli identificatori di programma e' realizzata dal Debugger accedendo alle informazioni immagazzinate in alcuni file esterni : di questi file si parlera' nel prossimo paragrafo.

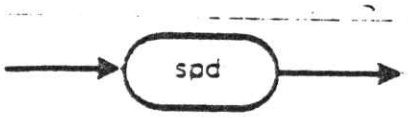
PROCEDURE OPERATIVE

Prima di chiamare il Debugger l'utente deve compiere le seguenti operazioni:

- Tutte le unita' di compilazione PASCAL+ devono essere compilate con l'opzione "- d" nella linea di chiamata del compilatore. Il compilatore, per ciascuna unita', crea due file i cui nomi sono quelli del sorgente (eventualmente provvisto del suffisso ".dfo" e ".cfo" rispettivamente). I file con il suffisso ".dfo" contengono informazioni sulle dichiarazioni PASCAL+, quelli con il suffisso ".cfo" contengono informazioni sulle istruzioni. Inoltre l'opzione "-d", se non e' presente nella linea di chiamata del compilatore anche l'opzione "-c" (si veda capitolo I), aggiunge alla chiamata del linker "zloc" (che avviene automaticamente) la direttiva "SYMBOL" e viene percio' prodotto in output da "zloc" il file "symbol" contenente una forma, comprensibile al sistema, della mappa. Il nome del file e' quello dell'opzione "-o" con il suffisso ".sym" (oppure "z.sym" se non c'e' l'opzione "-o"). I file ".dfo", ".cfo" e ".sym" devono essere notificati al debugger simbolico a run time.

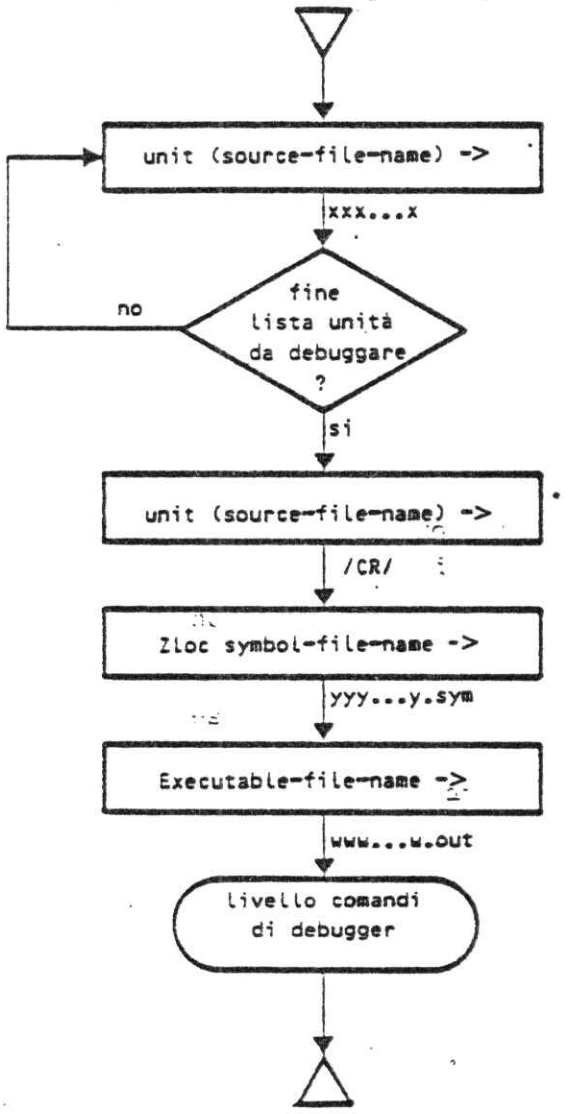
- Nel caso sia presente, nella linea di comando che chiama il compilatore, l'opzione "-c", le unita' di compilazione del programma devono essere linkate mediante "zloc", specificando un file di simboli (comando SYMBOL), a cura dell'utente.

Il Debugger e' lanciato in ambiente Shell, battendo dalla tastiera il comando "spd" seguito da /CR/.



Dopo che il comando e' stato battuto, vengono richiesti dei parametri in modo interattivo.
Il Debugger richiede, tramite prompt a terminale, che gli vengano passati in successione i nomi dei file di informazione simbolici: l'utente deve battere il nome del file sorgente di ciascuna unita' di compilazione da controllare con il debugger (senza i suffissi ".cfo" e ".dfo") seguito da /CR/. Per chiudere la lista dei file di informazione basta dare un altro

/CR/.
Viene poi richiesto, sempre con prompt a terminale, di battere il nome del file "symbol" (yyy...y.sym) e il nome del file eseguibile (www...w.out) (prodotti in output dal linker "zloc") entrambi seguiti da /CR/. La sequenza delle operazioni viene descritta dal diagramma seguente :



A questo punto l'utente comunica con il livello comandi del Debugger, segnalato su terminale dal prompt "->". Il programma utente e' arrestato proprio all'inizio della sua esecuzione e il Debugger aspetta i comandi dell'utente; esso accetta comandi costituiti da caratteri singoli eventualmente seguiti da parametri.

COMANDI DEL DEBUGGER

Tutti i comandi del Debugger sono costituiti da singoli caratteri eventualmente seguiti da uno o piu' parametri. Essi possono venire raggruppati secondo la seguente classificazione.

COMANDI DI CONTROLLO

I comandi di controllo forniscono i mezzi per controllare il flusso dell'esecuzione del programma. Essi permettono di:

- definire, togliere ed elencare i breakpoint posti dall'utente ('b', 'c', 'l')
- definire la unita' di programma controllata ('u')
- visualizzare le unita' di programma prima dell'esecuzione ('t')
- lanciare il programma utente ('r', 's')
- ottenere una traccia delle procedure attualmente attive ('h').

COMANDI RELATIVI AI DATI

I comandi del debugger relativi ai dati consistono in servizi per mezzo dei quali e' possibile :

- visualizzare il contenuto di una variabile o di una costante ('p')
- modificare il contenuto di una variabile ('m')
- visualizzare l'indirizzo di memoria di una variabile ('a')
- cambiare il contesto PASCAL+ attualmente accessibile (per poter accedere ad altre variabili) ('e')
- elencare tutti gli identificatori visibili nell'ambito presente ('n')
- accedere alla memoria mediante indirizzamento diretto ('d', 'g').

I comandi relativi ai dati sono interamente di alto livello, ossia si ha pieno accesso agli identificatori del programma utente (con le eccezioni mostrate nella sezione delle restrizioni) in conformita' alle regole della compatilita' dei tipi del PASCAL+.

COMANDI DI BASE

Questi comandi permettono di

- indicare il punto in cui e' sospeso il programma ('w')
- visualizzare i nomi delle unita' di compilazione in questa sessione di debugging ('i')
- chiudere la sessione di debugging ('q').

Restrizioni

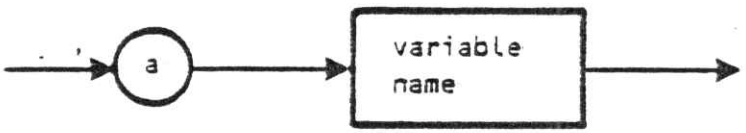
Tipi condition, descrittori, moduli instanziabili e monitor non sono gestiti.

ELENCO COMANDI

Viene riportato qui di seguito un elenco, ordinato alfabeticamente, di tutti i comandi del debugger per una facile consultazione : di ogni comando viene fornita la sintassi di chiamata ed una breve spiegazione del funzionamento.

a: INDIRIZZO DI UNA VARIABILE SIMBOLICA

Il comando 'a' produce come risultato, visualizzato a terminale, l'indirizzo di installazione in memoria della variabile, indicata come parametro, nella forma esadecimale "Segment-number / %offset". E' utile infatti conoscere, ad esempio, gli indirizzi delle strutture packed al fine di accedervi con i comandi di accesso alla memoria.

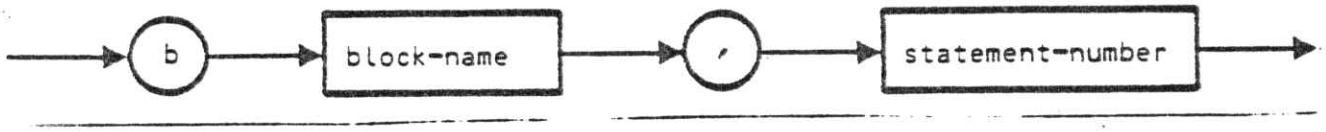


dove :

variabile name e' il nome usato all'interno del programma utente per identificare la variabile di cui si vuole conoscere l'indirizzo di installazione in memoria.

b: POSIZIONAMENTO DEI BREAKPOINT UTENTE

Il comando 'b' pone un breakpoint di controllo ad una istruzione facente parte di un blocco di programma.
Dopo che l'esecuzione del programma e' ripresa, quando il breakpoint e' raggiunto, il processo utente sara' arrestato prima dell'esecuzione della istruzione indicata e il breakpoint viene identificato e visualizzato su terminale dal Debugger. Il Debugger quindi passera' in "command mode" per accettare nuovamente comandi in modo interattivo. Il breakpoint rimane quindi presente fino a quando l'utente non lo rimuova esplicitamente.
Non vi e' limite al numero dei breakpoint utente che possono essere presenti contemporaneamente nel programma.



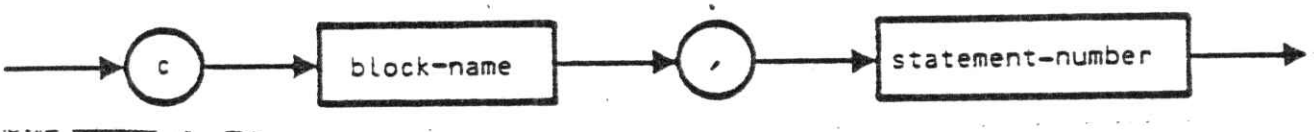
dove :

block-name e' il nome identificativo del blocco (funzione, procedura, modulo o programma) che contiene l'istruzione alla quale si intende porre il breakpoint, eventualmente preceduto dal nome del blocco immediatamente esterno che lo contiene (outer-block) per risolvere casi di ambiguita' od omonimie.

statement-number e' il numero che identifica l'istruzione dove si vuol porre il breakpoint, all'interno del blocco specificato in "block name". Tale numero e' ricavabile dal listing del programma sorgente.
Esso comincia da 0 (begin); le statement sono numerate sequenzialmente all'interno di ciascun blocco, e il loro numero e' indicato nella seconda colonna del listing. Si noti che il numero si riferisce alla ultima istruzione vista dal compilatore quando esso ha raggiunto la fine della linea che sta trattando.

c: ANNULLAMENTO DEI BREAKPOINT UTENTE

Il comando 'c' annulla il breakpoint utente alla statement indicata all'interno di un blocco, se esso e' stato posto dall'utente (si veda il comando 'b').



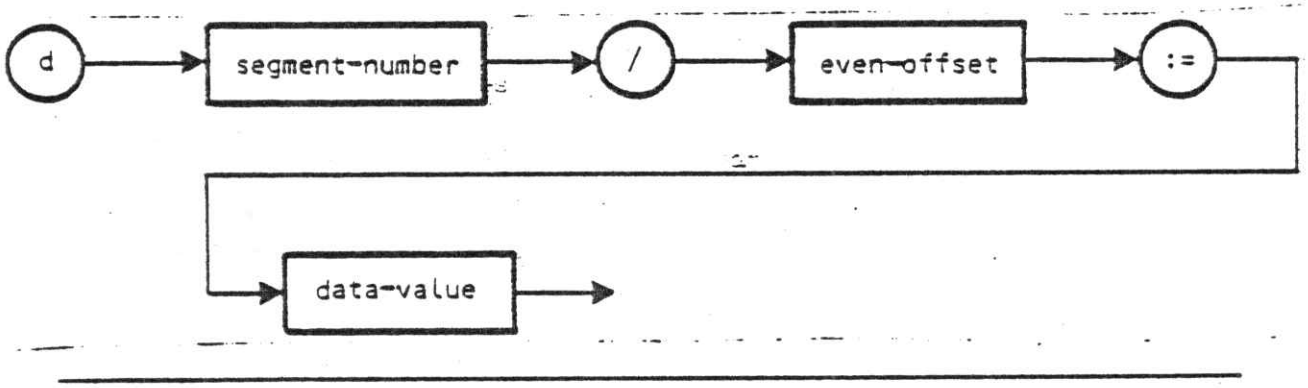
dove :

block-name e' il nome identificativo del blocco (funzione, procedura, modulo o programma) che contiene l'istruzione dalla quale si intende rimuovere il breakpoint, eventualmente preceduto dal nome del blocco immediatamente esterno che lo contiene (outer-block) per risolvere casi di ambiguita' od omonimie.

statement-number e' il numero che identifica l'istruzione dalla quale si vuole rimuovere il breakpoint, all'interno del blocco specificato in "block name". Tale numero e' ricavabile dal listing del programma sorgente.
Esso comincia da 0 (begin); le statement sono numerate sequenzialmente all'interno di ciascun blocco, e il loro numero e' indicato nella II colonna del listing. Si noti che il numero si riferisce alla ultima istruzione quando esso ha raggiunto la fine della linea che sta trattando.

d: DEPOSITO DEL CONTENUTO DI UNA WORD

Il comando 'd' consente di depositare un valore scelto nella word specificata.



dove :

segment-number e' il numero del segmento contenete la word cui si vuole accedere. Puo' essere usata una delle seguenti quattro forme : esadecimale con prefisso %, decimale, ottale con prefisso #, mista.

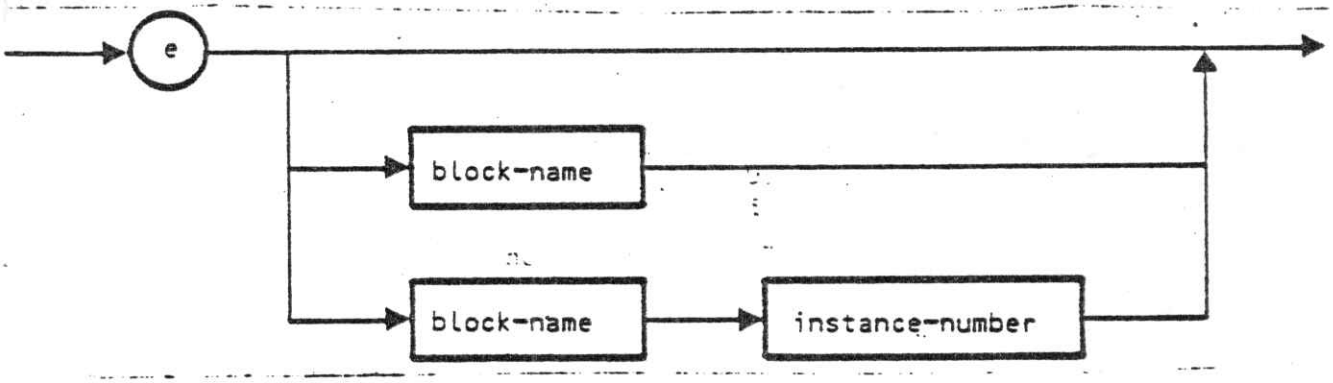
even-offset e' l'offset pari dall'inizio del segmento alla locazione (word) di memoria cui si vuole accedere. Per la forma da usare vale la stessa regola esposta per il segment number.

data-value e' il valore numerico da depositare nella word. Per la forma da usare vale la stessa regola esposta per il segment number.

Nota: e' sconsigliabile usare questo comando in caso generale : esso infatti risulta utile in casi particolari, ad esempio per accedere ad aree di memoria non direttamente visibili come dati del programma PASCAL+.

e: CAMBIAMENTO DI CONTESTO

Il comando 'e' cambia il contesto attuale, relativo alla procedura interrotta con l'ultimo breakpoint raggiunto dal Debugger. Il nome di un modulo, monitor o programma consente di accedere ai dati statici (sempre attivi). Il comando senza parametri ripristina il vecchio contesto.



dove:

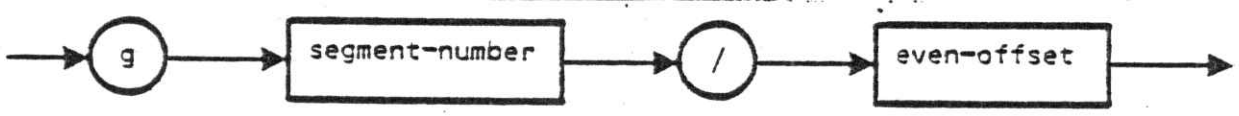
block-name e' il nome che identifica il blocco nel quale si desidera porre il nuovo contesto.

instance-number e' il numero che specifica quale istanza del blocco 'block name' deve essere introdotta. Il valore 1 corrisponde all'ultima attivazione del blocco. Il valore di default e' 1.

Nota : il block name e l'instance-number devono essere separati da ",".

g: VISUALIZZAZIONE DEL CONTENUTO DI UNA WORD

Il comando 'g' consente di visualizzare il contenuto della word indirizzata.



dove:

segment-number e' il numero del segmento contenente la word di cui si intende visualizzare il contenuto; per esso puo' essere usata una delle seguenti quattro forme : esadecimale con prefisso %, decimale, ottale con prefisso #, mista.

even-offset e' l'offset, a partire dall'inizio del segmento, della word di cui si intende visualizzare il contenuto; per questo numero puo' essere usata la stessa forma usata per il segment number.

h: VISUALIZZAZIONE DI PROCEDURE O FUNZIONI ATTIVE

Il comando 'h' consente di visualizzare il nome di procedure o di funzioni PASCAL+ (ivi compreso il nome del programma main) che sono attive al momento in cui tale comando viene eseguito.

Questo comando non ammette parametri.

i: VISUALIZZAZIONE DI INFORMAZIONI

Il comando 'i' consente di visualizzare a terminale i nomi di tutte le unita' di compilazione presenti in questa sessione di debug.

Questo comando non ammette parametri.

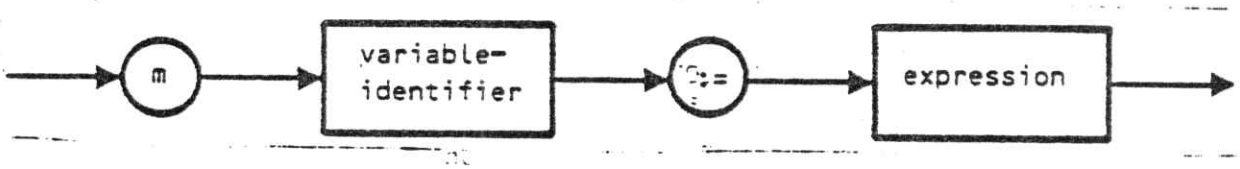
l: LISTA DEI BREAKPOINT DELL'UTENTE

Il comando 'l' consente di visualizzare su terminale la lista di tutti i breakpoint posti dall'utente, identificati dal "block-name" e dallo "Statement-number" (si vedano i comandi 'b' e 'c').

Questo comando non ammette parametri.

m: MODIFICA DEL CONTENUTO DI UNA VARIABILE

Il comando 'm' consente di modificare il contenuto di una variabile con il risultato di un espressione. Naturalmente l'assegnamento del valore ad una variabile deve rispettare le regole di compatibilita' dei tipi del PASCAL+.



dove

variabile-identifier e' il nome identificatore della variabile di cui si intende modificare il contenuto.

expression: si veda Cap. 2 pag. 2-5.

n: ELENCO DI TUTTI I NOMI VISIBILI

Il comando 'n' consente di visualizzare a terminale tutti gli identificatori (di costanti, variabili, procedure, funzioni) visibili a partire dalla procedura in corso e passando alla procedura o funzione che immediatamente la contiene , andando per livelli superiori. Per ciascun livello sono dati:

- il numero di livello (1 indica il livello piu esterno)
- il nome della procedura
- tutti gli identificatori di quel livello con indicazione della classe (constant, type, etc.) ed il tipo (array, pointer, integer, etc.)

Questo comando non ammette parametri.

p: STAMPA DEL CONTENUTO DI VARIABILI O COSTANTI

Il comando 'p' consente di visualizzare su terminale il contenuto di una variabile o di una costante. Questo contenuto, nel caso di variabili, puo' ovviamente variare a seconda del punto in cui si e' arrestato il programma eseguibile; le costanti o le variabili riferite dal comando devono essere allocate nel contesto in corso. Valgono inoltre le seguenti regole:

- per oggetti di tipo 'scalar', il contenuto e' dato sotto la forma associata al loro tipo. Ad esempio:

  var a : char ;

  il comando "p a" produce come risultato su terminale

  a = 'x' con x carattere

  var b : boolean ;

  il comando "p b" produce come risultato a terminale

  b = true o b = false

- i record sono visualizzati campo per campo (ciascun campo identificato dal suo nome). Per record con campi variabili vi sono due casi

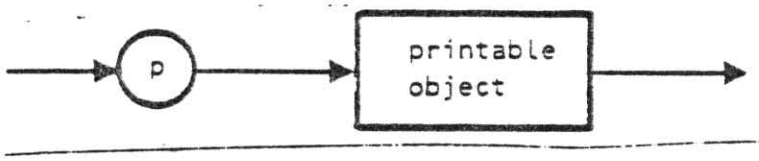
  . in caso di "tag field" implicito, viene visualizzato sia il valore del tag che il valore della variante relativa

  . in caso di "tag field" esplicito, viene calcolato il tag (selettore di variante) e viene visualizzata solo la variante relativa

- le array sono visualizzate componente per componente. Ciascuna componente e' visualizzata in una linea diversa. Invece per oggetti di tipo scalare, subrange e pointer, piu' componenti sono messe in una stessa linea

- oggetti di tipo set o string sono visualizzati nella forma Pascal+

- i puntatori sono visualizzati nella forma esadecimale 'SegmentNumber/Offset'. Il valore nil (puntatore nullo) e' riconosciuto e quindi normalmente visualizzato.



dove:

printable-object e' il nome identificatore della variabile o della costante di cui si vuole visualizzare il contenuto.

q: USCITA DAL DEBUGGER

Il comando 'q' chiude la sessione di debug.

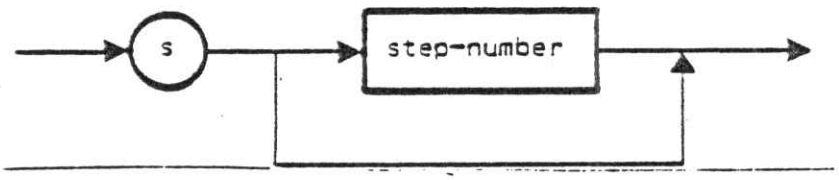
Questo comando non ha parametri.

r: LANCIO DEL PROGRAMMA

Il comando 'r' provoca l'esecuzione del programma dall'inizio o la riprende dal breakpoint corrente. Il Debugger riprende il controllo dopo l'esecuzione di una sola unita' di programma e ritorna in "command mode" quando incontra un breakpoint dell'utente ; se nessun breakpoint utente e' incontrato, il programma termina la sua esecuzione e la sessione di Debug e' chiusa.
Questo comando non ammette parametri.

s: LANCIO DEL PROGRAMMA A STEP

Il comando 's' provoca l'esecuzione di "step number" unita' di programma (per la definizione di unita' di programma controllata si veda il comando 'u'), dopodiche' il Debugger torna in "command mode" ed accetta nuovamente comandi utente. Il programma utente e' comunque arrestato qualora sia raggiunto un breakpoint posto dall'utente. Il valore di default per "step-number" e' 1, che provoca l'esecuzione del programma unita' per unita'.

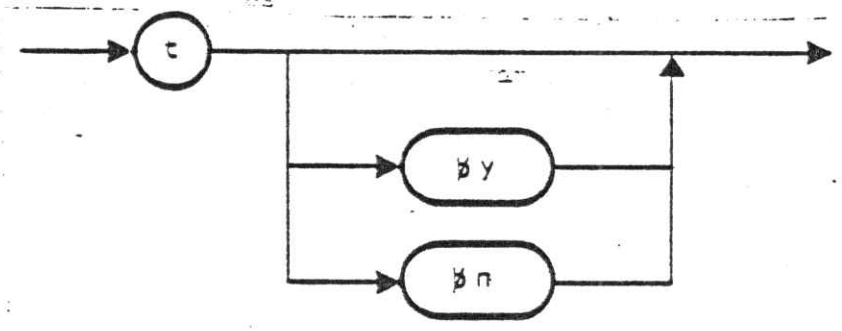


dove:

step-number e' il numero di unita' (step) di programma che si desidera siano eseguite.

t: VISUALIZZAZIONE DELLE UNITA'

Il comando 't' abilita o disabilita la visualizzazione di ciascuna unita' di programma prima dell'esecuzione. Il valore di default e' "n" (nessuna visualizzazione).



dove:

y abilita la visualizzazione.

n disabilita la visualizzazione.

u: DEFINIZIONE DELLA UNITA' DI PROGRAMMA CONTROLLATA

Il comando 'u' definisce le unita' di programma che devono essere controllate dal Debugger. Le Unita' riconosciute sono:

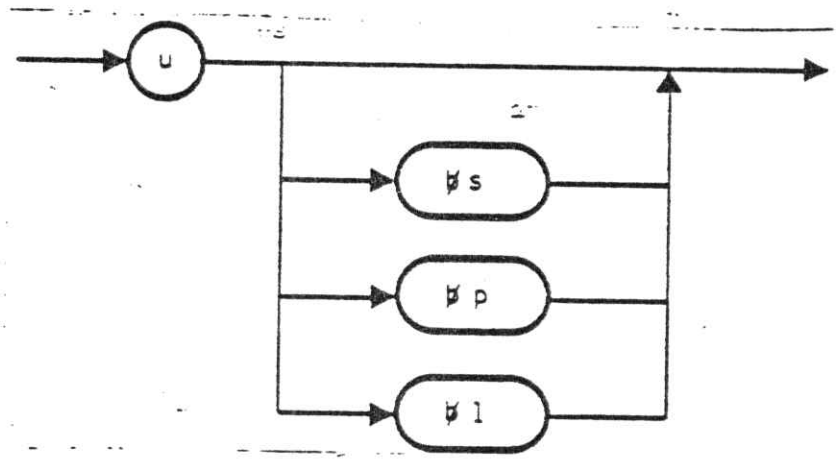
- s: la singola statement Pascal; questo da all'utente l'intero controllo del programma

- p: ogni procedura o funzione come un tutt'uno, essendone controllato solo l'inizio e l'uscita da essa; questa via permette all'utente di eseguire velocemente un programma per vedere quali routine vengono via via chiamate

- l: le istruzioni al livello in corso del programma; cio' significa che le procedure o le funzioni chiamate all'interno del programma saranno completate prima che il Debugger riprenda il controllo del programma, consentendo all'utente di bypassare l'esecuzione dettagliata di routine gia' corrette.

In ogni momento solo un valore unita' e' valido.

Se nessun parametro viene battuto dopo il comando 'u', l'unita' e' la porzione di programma tra il breakpoint corrente e il successivo.



w: VISUALIZZAZIONE DELLA POSIZIONE CORRENTE

Il comando 'w' visualizza su terminale sia l'istruzione e la procedura dove il processo utente e' sospeso sia l'ambito corrente visibile del programma.

Questo comando non ammette parametri.

4. ZSTUB : UNO STRUMENTO PER IL COLLEGAMENTO FRA L-MODULI

Il presente capitolo contiene la descrizione del programma ZSTUB.
Nell'introduzione e nella prima sezione vengono descritti in generale l'uso di ZSTUB e i vantaggi da esso offerti.
Nella seconda sezione vengono descritte le procedure operative per usare ZSTUB e precisamente:

- i comandi da dare in input al programma ZSTUB e il file "input file" contenente la lista delle procedure che si vogliono rendere globali
- la procedura di lancio di ZSTUB.

INTRODUZIONE

E' prassi comune, nella programmazione PASCAL+, che piu' l-moduli scambino fra loro delle risorse, con evidenti vantaggi di economia complessiva di codice, e questo scambio puo' avvenire sia tra l-moduli utente sia tra l-moduli utente e l-moduli di sistema.
Si ricordi che un l-modulo e' un modulo eseguibile risultato di un operazione di linking "statico" ottenuto mediante un linker standard (ad esempio "zloc").
In genere questo scambio di risorse avviene con l'"import/export" di procedure (si veda a questo proposito il manuale "Linguaggio PASCAL+ : Manuale di consultazione). Esiste percio' la necessita', sia per gli l-moduli che forniscono le procedure "globali" (cioe' messe a disposizione del sistema nel suo complesso) sia per gli l-moduli che le chiamano, di includere, in fase di link, degli speciali "moduli di interfaccia" usati poi dal sistema per collegare fra loro dinamicamente i vari l-moduli quando essi sono caricati dal sistema.

Il programma ZSTUB, descritto in questo capitolo, e' in grado di creare questi moduli di interfaccia.

USO DI ZSTUB

I moduli di interfaccia da creare con ZSTUB sono due: una "tabella di procedure" e una "libreria di interfaccia". Le "tabelle di procedure" (procedure-table o p-table) sono usate nel sistema, insieme ad un'unica m-table (module-table), come riferimento per risolvere le chiamate di "procedure globali", ossia di procedure che sono note fra diversi l-moduli nel sistema.
Ogni l-modulo che esporti alcune sue procedure e le renda cosi' disponibili anche ad altri l-moduli, deve fornire la propria p-table. Una procedure-table e' in realta' un modulo oggetto z-code che deve essere creato con ZSTUB e collegato (tramite "zloc") all'l-modulo che esporta.
Inoltre ogni l-modulo che chiama (importing l-module) procedure globali fornite da altri l-moduli deve avere a sua disposizione una serie di "librerie di interfaccia", una per ogni l-modulo dal quale importa le procedure globali.
Una "libreria di interfaccia" contiene un insieme di procedure banali che, lavorando sulle tabelle del sistema, agiscono come interfaccie fra il modulo chiamante e la procedura globale chiamata. Ciascuna libreria di interfaccia e' di nuovo un modulo oggetto "zcode" che, creato una volta per tutte con ZSTUB, e' disponibile, insieme al relativo l-modulo esportante, per il collegamento, tramite link, ad ogni l-modulo che importa le procedure interfacciate dalla libreria.

Si osservi che, mentre la tabella delle procedure e' costruita e usata una volta sola dallo stesso programmatore, la "libreria di interfaccia" e' costruita una sola volta e usata molte volte anche da piu' programmatori.

In conclusione il programmatore di un l-modulo che esporta procedure globali deve usare il programma ZSTUB per creare sia la "tabella di procedure" che la "libreria di interfaccia"; quindi egli deve collegare al proprio l-modulo la relativa p-table e rendere la interface-library disponibile ad altri utenti.
Invece il programmatore di un l-modulo che importa procedure globali deve semplicemente collegare , tramite linker, al proprio l-modulo le appropriate "librerie di interfaccia".

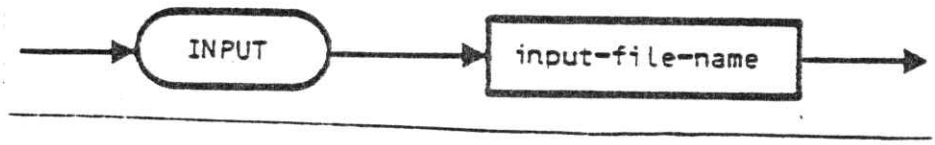
PROCEDURE OPERATIVE

Per usare opportunamente ZSTUB e' necessario fornirlo di alcuni comandi, che possono essere dati sia da tastiera (come command-line) sia da un file di comandi dove sia rediretto l'input. Sono i seguenti :

- INPUT "input-file-name"
- TABLE "table-file-name"
- INTER "inter-file-name"
- INTRA "intra-file-name"

IL COMANDO INPUT

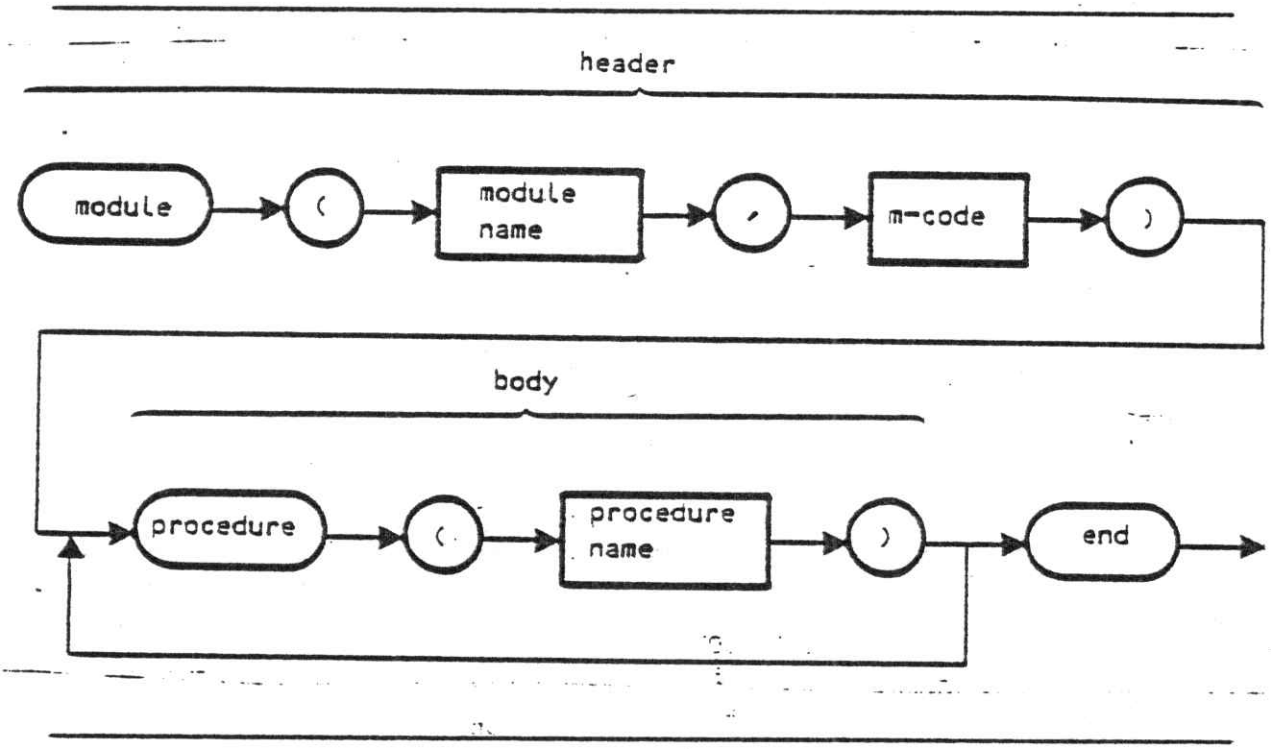
Questo comando definisce il pathname "input-file-name" del file sorgente contenente le dichiarazioni di tutte le procedure globali esportate da un singolo l-modulo.
La sintassi e' la seguente:



dove :

input-file-name e' il nome del file "input-file" avente il formato illustrato dal seguente diagramma

Input-File



dove :

module e' la parola chiave che identifica l'header dell'input-file

module-name e' il nome logico del modulo oggetto zcode contenente le procedure da dichiarare globali; questo nome ha significato solo per l'l-modulo esportante che sara' collegato tramite link alla "tabella di procedure" (p-table) degli entry point delle procedure; l'l-modulo che esporta tali procedure deve passare l'identificatore "module-name"_proctbl al Sistema Operativo per mezzo della primitiva "AttachLModule", al momento dell'inizializzazione del sistema.

m-code e' il codice globale (un numero intero) che identifica univocamente l'l-modulo contenente le procedure da dichiarare globali. Ciascun l-modulo di sistema (ad esempio file system, pmm etc) ha il suo proprio m-code, cosi' come ciascun l-modulo utente che fornisca procedure globali per uso da parte di altri l-moduli utente (librerie matematiche, transaction handler etc...).

procedure e' la parola chiave che precede il nome di ogni procedura dichiarata globale.

procedure-name e' il nome della procedura che si vuole dichiarare globale, che puo' essere cosi' esportata dal proprio l-modulo o importata da altri l-moduli.

end segnala la fine dell'"input-file".

Nota : l'ordine dei nomi delle procedure all'interno dell' "input-file" e' fondamentale poiche' esso definisce il "p-code" (procedure-code) delle procedure elencate : la prima direttiva "procedure" associa p-code = 0 al suo procedure-name, la seconda associa p-code = 1, etc... .
Poiche' il p-code **deve essere costante** durante la vita del sistema, **non e' permesso** togliere o aggiungere dei "procedure-name" nell' "input-file" a meno che questo non sia fatto alla fine dell' "input-file".
Si osservi inoltre che blank, tab e newline possono essere posti ovunque nell' "input-file" tranne che attorno al "module-name" ; infatti, per esempio, dei blank fra il "procedure-name" e le circostanti parentesi costituirebbero dei caratteri validi dello stesso identificatore "procedure-name".

Esempio di INPUT file

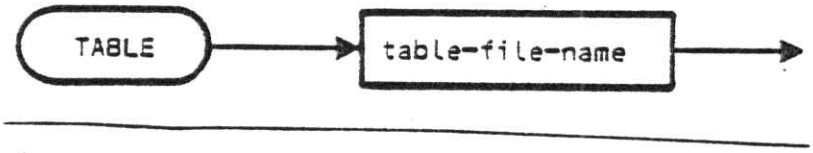
L'esempio che segue mostra una parte dell'"input-file" pm_pt.s :

```
module(pm,10)
procedure(pm-makechan)
procedure(pm_destrchan)
procedure(pm_load)
procedure(pm_unload)
...
end
```

Osservazioni : il "module-name" e' "pm", percio' la "procedure-table" e' conosciuta dall' "l-modulo" che esporta col nome di "pm-proctbl" ; l' "m-code" e' 10 ; la procedura "pm_makechan" ha codice "p-code" = 0 e il suo "g-code" (global-code) e' dunque <10,0>; etc... .

IL COMANDO TABLE

Questo comando definisce il pathname del modulo oggetto zcode che conterra' la "tabella di procedure".
Questo comando e' opzionale.

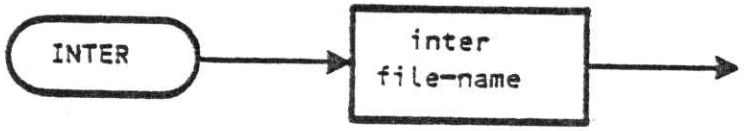


dove :

table-file-name e' il nome del file (modulo oggetto) che conterra' la "procedure-table" dell'l-modulo esportante.

IL COMANDO INTER

Questo comando definisce il pathname del modulo oggetto zcode che conterra' la "libreria di interfaccia"
Questo comando e' opzionale.



dove:

inter-file-name e' il nome del file (modulo oggetto) che conterra' la "libreria di interfaccia" dell'l-modulo relativo.

IL COMANDO INTRA

Il comando INTRA, se inserito fra i comandi di ZSTUB, permette di creare il file "intra-file-name"; questo file e' un modulo oggetto zcode che contiene una speciale "libreria di interfaccia" usata per i links dinamici fra l-moduli di sistema.
Nota : questa libreria di interfaccia non puo' essere usata per collegamenti fra l-moduli di sistema e l-moduli utente.

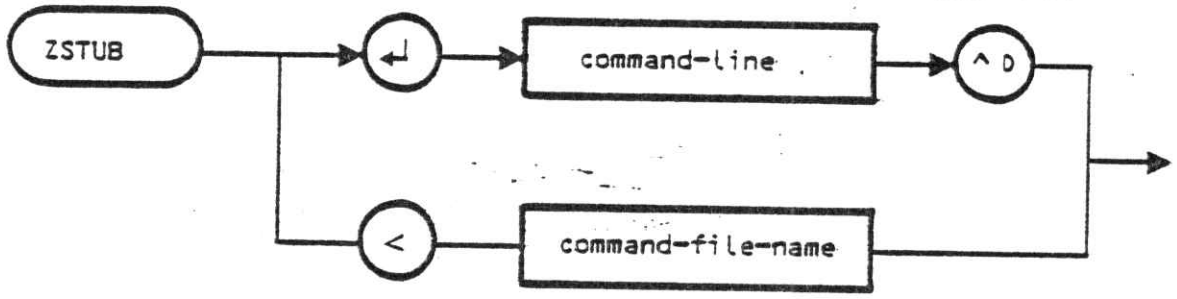
CARATTERISTICHE IMPLEMENTATIVE

Valgono per l'attuale versione di ZSTUB le seguenti limitazioni :

- i pathname del file "input-file", "table-file" e "inter-file" non possono superare i 50 caratteri
- il "module-name" nell' input-file non puo' superare i 12 caratteri
- tutti i "procedure-name" nell' input-file non possono superare i 20 caratteri
- il numero di procedure globali dichiarate nell' input-file non puo' superare 100.

COME LANCIARE ZSTUB

Il programma ZSTUB viene lanciato in ambiente Shell; i comandi possono essere dati o da tastiera (command-line) o da un file comandi dove sia stato rediretto l'input (secondo la normale convenzione Shell).



dove

command-line e' una linea di comandi, battuta da tastiera, contenente i comandi :

```
INPUT "input-file-name"
TABLE "table-file-name"
INTER "inter-file-name"
INTRA "intra-file-name"
```

descritti in precedenza. L'ordine dei comandi non ha importanza e solo il comando "INPUT ..." e' obbligatorio. Inoltre i comandi possono essere separati da blank, tab o newline. Si ricordi che ognuna delle parole chiave INPUT, INTER, INTRA e TABLE deve essere seguita dal relativo "file-name" e puo' essere scritta anche in caratteri minuscoli.

^D (/ctrl/D) indica la fine della "command-line" (si puo' considerare come un "eof")

< indica la redirezione dell'input verso il file "command-file-name" che segue

command-file-name e' il nome del file, contenente i comandi di ZSTUB su cui viene rediretto l'input. Per il contenuto di questo file si veda la precedente voce "command-line".

ESEMPIO

Si supponga che i comandi di input per ZSTUB siano contenuti nel file di "mkpmmlinks" seguente :

```
INPUT pm_pt.s
TABLE pm_table.obj
INTER pm_inter.obj
```

dove "pm_pt.s" e' il 1' "input-file" contenente la lista di procedure da dichiarare globali (si veda pag. 4.5). Allora ZSTUB puo' essere lanciato, in ambiente Shell, come segue :

```
MCL: ZSTUB < mkpmmlinks
```

Le operazioni di ZSTUB si dividono in 3 fasi :

- nella prima il programma legge e interpreta i comandi contenuti nel file "mkpmmlinks" (o nella linea di comandi data da tastiera) ;
- nella seconda legge il file "input-file-name" e memorizza le informazioni necessarie nella memoria principale;
- nella terza fase, avviata solo se non sono stati trovati errori nelle due fasi precedenti, crea i moduli di output richiesti.

Le tre fasi vengono cosi' visualizzate a terminale :

```
Olivetti ZSTUB        — Release x.x

— PHASE 1 —
                      input = pm_pt.s
                      table = pm_table.obj
                      inter = pm_inter.obj

— PHASE 2 —
                      module name : pm
                      m-code      : 10
                      global-procedure : 36

— PHASE 3 —

Making TABLE = pm_table.obj
       Section pm_p  Size 0
       Section pm_d  Size 144
       Table Global Name : pm_proctbl
Making INTER = pm_inter.obj
       Section pm_p  Size 444
       Section pm_d  Size 0

ZSTUB  Complete

MCL:
```

Dopo l'esecuzione di ZSTUB, i due moduli "pm_table.obj" e "pm_inter.obj", creati secondo la dichiarazione delle procedure globali contenuta in "pm_pt.s", sono pronti per essere collegati, tramite "zloc", agli appropriati l_moduli.

Lo stesso risultato lo si puo' ottenere con il seguente comando :

```
MCL: ZSTUB

Olivetti  ZSTUB       -- Release x.x --

-- PHASE 1 --

INPUT pm_pt.s  INTER pm_inter.obj  TABLE pm_table.obj

^D

                      input = pt_pt.s
                      table = pm_table.obj
                      inter = pm_inter.obj
   ...

MCL:
```

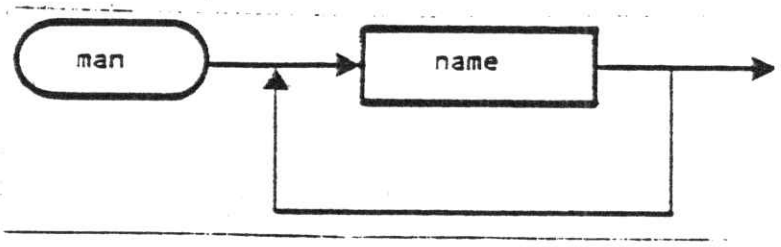
dove le parole in grassetto indicano i soli comandi da battere in input da tastiera, mentre le altre parole sono scritte su terminale dal programma ZSTUB stesso.

5. TOOLS VARI UTILI NELLA PREPARAZIONE ED ESECUZIONE DI PROGRAMMI

I seguenti programmi sono strumenti utili ad un utente che lavori in un ambiente di preparazione ed esecuzione di programmi PASCAL+:

- man : consente di leggere file-documenti
- more,page : consentono di leggere qualsiasi file di tipo testo
- zpcmesg : decodifica i codici di errore prodotti dal compilatore
- zpdis : disassembla il file .obj prodotto in output dal compilatore

Il programma "man" visualizza, una pagina alla volta, un documento , cioe' un file della libreria del sistema M30/M40. Esso e' chiamato in ambiente Shell secondo la seguente sintassi:



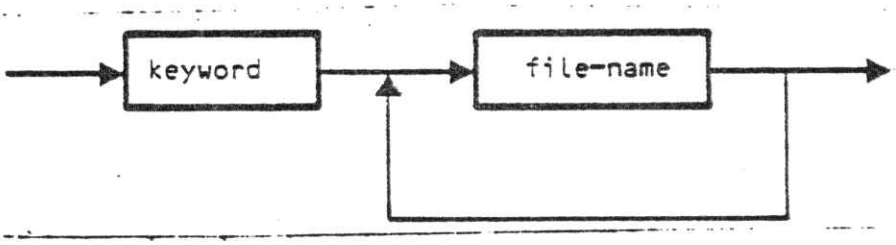
dove :

name e' il nome di un file-documento che si trova nella libreria di documenti del sistema M30/M40 (directory /Z0T/docs)

Possono essere visualizzati fino a sei file documento diversi; inoltre non e' necessario specificare il pathname del file documento scelto, poiche' il programma lo trova da se'. Il comando "man" e' lanciato in ambiente Shell, eventualmente preceduto dal pathname della directory che contiene il programma "man" (directory /Z0T/cmd).
Si osservi che "man" non visualizza invece file di tipo generale, per i quali si puo' usare il programma "more" descritto qui di seguito.
I comandi validi per il comando "man" sono gli stessi di quelli del programma "more" descritto piu' avanti : per averli visualizzati basta battere, una volta entrati nel programma "man", il comando "h" (help).

Nota : "man" accetta fino a 6 argomenti "name" separati da blank, e almeno uno deve essere presente.

I programmi "more" e "page", lanciati in ambiente Shell, visualizzano su terminale un file di tipo testo (cioe' contenente caratteri ASCII), una pagina alla volta.
La sintassi di entrambi e' la seguente :



dove :

keyword e' la parola chiave, in caratteri minuscoli, "more" o "page", a seconda del programma scelto.

file-name e' il nome del file di tipo testo da visualizzare, eventualmente preceduto dal suo pathname (se il file non si trova nella directory corrente di lavoro).
Possono essere visualizzati successivamente fino a sei file distinti, ed almeno un "file-name" deve essere presente come argomento.

I comandi validi per entrambi i programmi sono elencati qui di seguito, indicati fra apici e seguiti da una breve spiegazione :

'blank' : visualizza la successiva pagina del file

'/' /CR/ : visualizza la successiva linea del file

'd' : fa avanzare lo schermo di mezza pagina (11 linee) del file

'f' : fa saltare la prossima pagina del file (22 linee) e visualizza quella che segue

/ctrl/'L': ridisegna la pagina di file sul video (ad esempio se alcune righe della pagina mancano su video per la presenza di comandi di more, il comando ctrl-L ridisegna la pagina

completa)

'/'string: cerca la prima occorrenza della stringa "string" nella parte di file che segue la pagina presente su video. Se dopo "/" non viene battuta alcuna stringa, cerca la prima occorrenza dell'ultima stringa battuta in precedenza. Se non era stata battuta alcuna stringa, visualizza il messaggio "NO PREVIOUS REGULAR EXPRESSION". Se la stringa cercata non viene trovata, visualizza il messaggio "PATTERN NOT FOUND"

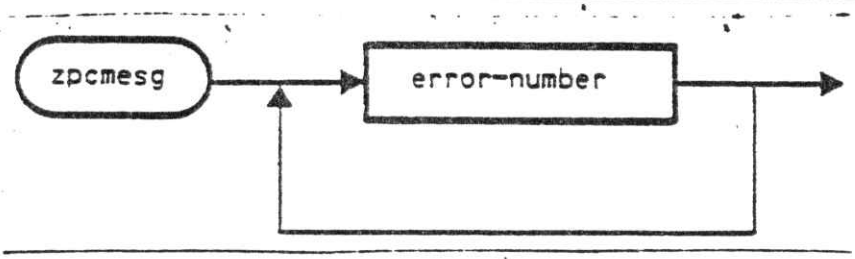
'q' : fa uscire dal programma

'h' : visualizza la lista dei comandi valevoli per i programmi man, more, page ;

'n' : va all'inizio del prossimo file ; se si e' gia' sull'ultimo file da leggere, fa uscire dal programma

'p' : va all'inizio del file precedente (quello attuale se e' il primo)

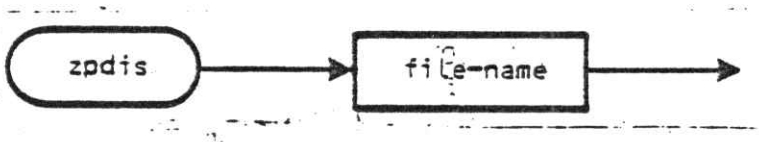
Il programma "zpcmesg" decodifica e visualizza su terminale i messaggi di errore (in lingua inglese) corrispondenti ai codici di errore prodotti dal compilatore PASCAL+ ; tali numeri (nel range 1..1000) sono visualizzati a terminale al termine di una compilazione o stampati sul listing in corrispondenza della istruzione errata.
La sintassi e' la seguente :



dove

error-number e' il numero dell'errore da decodificare.

Il programma disassemblatore "zpdis" disassembla il file oggetto prodotto in output dal compilatore "zpc" e produce un file "file-name".a (dove "file-name" e' il nome del file contenente il programma oggetto da disassemblare) combinando i file "file-name".obj e "file-name".l e convertendo in valori mnemonici(linguaggio assembler) i valori esadecimali del file "file-name".obj .
E' percio' necessario che il programmatore inserisca l'opzione '-a' all'interno della linea di comandi che chiama il compilatore e che sia inserita l'opzione (*$l"file-name".l*) nella prima linea del programma sorgente (si veda il primo capitolo) in modo che sia prodotto in output dal compilatore il file listing "file-name".l.
Il programma viene chiamato in ambiente Shell secondo la seguente sintassi:



dove :

file-name e' il nome del file da disassemblare.

Vengono riportati di seguito un esempio di listing di una parte di programma sorgente PASCAL+ e il relativo listing disassemblato.

```
 0.   0 (*$lpu01030.l*)
 1.   0 program pu01030;
 2.   0
 3.   0 { ***************************
 4.   0     system       = S3000
 5.   0     component    = pascal z8000
 6.   0     func. area   = 01
 7.   0     test unit    = pu01030
 8.   0     date         = 27/10/81
 9.   0   *************************** }
10.   0
11.   0
12.   0 $include 'main.new'
13.   0  type ALFA = packed array[1..9] of char;
14.   0
15.   0       testname=packed array[1..7] of char;
16.   0  const system   = 'PDP-11/70';
17.   0       component= 'pascal';
18.   0       lines    = 60;          {numero di linee per pagina}
19.   0       llength  = 80;          {numero di caratteri per riga}
20.   0  var   tunit : testname;      {nome del test program}
21.   0       lcount: integer;       {contatore delle righe complessivamente stampate}
22.   0       j     : integer;
23.   0       cases : array[1..20] of packed array[1..5] of char;
24.   0                              {vettore contenente i casi di check list provati}
25.   0       expect: integer;       {numero di procedure che devono essere eseguite}
26.   0
27.   0       ERR   : BOOLEAN;       {presenza o assenza di errori}
28.   0       EXEC  : INTEGER;       {numero di procedure effettivamente eseguite}
29.   0       KO    : INTEGER;       {numero di proc. che hanno avuto esito negativo}
30.   0
31.   0
32.   0 PROCEDURE START(area:integer; testprog:testname; tnum:integer);
33.   0   const needlength = 7;
34.   0   var i:integer;
35.   0
36.   0   BEGIN
37.   1     writeln(' - SYSTEM',' = ',system);
38.   2     writeln('COMPONENT',' = ',component);
39.   3     writeln('FUNC.AREA',' = ',area:2);
40.   4     writeln('TEST UNIT',' = ',testprog);
41.   5     writeln;
42.   6     writeln;
43.   7     for i:= 1 to llength do
44.   8       write('*');
45.   9     writeln;
46.  10     tunit:=testprog;
47.  11     expect:=tnum;
48.  12     lcount:=needlength;
49.  13     for i:=1 to 20 do
50.  14       cases[i]:='     ';
51.  15     EXEC:=0;
52.  16     KO:=0
53.  16   END;
54.   0
```

```
0000:  5ED8 8000 0F3C      JP      ((0))%0F3C
  0.   0 (*slpu01030.ls)
  1.   0 program pu01030;
  2.   0
  3.   0 ( ************************
  4.   0      system       = 53000
  5.   0      component    = pascal %8000
  6.   0      func. area   = 01
  7.   0      test unit    = pu01030
  8.   0      date         = 27/10/81
  9.   0    ************************ )
 10.   0
 11.   0
 12.   0 #include 'main.new'
 13.   0  type ALFA = packed array[1..9] of char;
 14.   0
 15.   0        testname=packed array[1..7] of char;
 16.   0  const system    = 'PDP11/70';
 17.   0        component = 'pascal';
 18.   0        lines     = 60;          (numero di linee per pagina)
 19.   0        llength   = 80;          (numero di caratteri per riga)
 20.   0  var   tunit : testname;        (nome del test program)
 21.   0        lcount: integer;         (contatore delle righe complessivamente stampate)
 22.   0        j     : integer;
 23.   0        cases : array[1..20] of packed array[1..3] of char;
 24.   0                                 (vettore contenente i casi di check list provati)
 25.   0        expect: integer;         (numero di procedure che devono essere eseguite)
 26.   0
 27.   0        ERR   : BOOLEAN;         (presenza o assenza di errori)
 28.   0        EXEC  : INTEGER;         (numero di procedure effettivamente eseguite)
 29.   0        KO    : INTEGER;         (numero di proc. che hanno avuto esito negativo)
 30.   0
 31.   0
 32.   0  PROCEDURE START(area:integer; testprog:testname; num:integer);
 33.   0    const needlength = 7;
 34.   0    var i:integer;
 35.   0
 36.   0    BEGIN
0006:  030F 001C            SUB     R15 #%001C
0004:  0DE3 0020            LD      @RR14 #%0020
000E:  50F9 02D5 02D6       LDM     ((2))%0006 R2.6
0014:  54FC 02DA            LDL     RR12,((2))%000A(R15)
0018:  34EA 0012            LDA     RR10,RR14(#18)
001C:  8014                 LDK     R1,#4
001E:  88C1 0140            LDIR    @R10,@R12,R1
 37.   1      writeln('    SYSTEM',' = ',system);
0022:  7602 8300 0000       LDA     RR2,((3))%0000
0028:  2104 FFFF            LD      R4,#%FFFF
002C:  8059                 LDK     R5,#9
002E:  7606 8300 0044       LDA     RR6,((3))%0044
0034:  5F00 8300 0001       CALL    ((3))%0001
003A:  7602 8300 0000       LDA     RR2,((3))%0000
0040:  2104 FFFF            LD      R4,#%FFFF
0044:  8053                 LDK     R5,#3
0046:  7606 8300 0040       LDA     RR6,((3))%0040
004C:  5F00 8300 0001       CALL    ((3))%0001
0052:  7602 8300 0000       LDA     RR2,((3))%0000
0056:  2104 FFFF            LD      R4,#%FFFF
005C:  8058                 LDK     R5,#8
005E:  7606 8300 0038       LDA     RR6,((3))%0038
0064:  5F00 8300 0001       CALL    ((3))%0001
006A:  7602 8300 0000       LDA     RR2,((3))%0000
0070:  5F00 8300 0002       CALL    ((3))%0002
 38.   2      writeln('COMPONENT',' = ',component);
0076:  7602 8300 0000       LDA     RR2,((3))%0000
007C:  2104 FFFF            LD      R4,#%FFFF
0080:  8059                 LDK     R5,#9
0082:  7606 8300 002E       LDA     RR6,((3))%002E
0088:  5F00 8300 0001       CALL    ((3))%0001
008E:  7602 8300 0000       LDA     RR2,((3))%0000
0094:  2104 FFFF            LD      R4,#%FFFF
0098:  8053                 LDK     R5,#3
009A:  7606 8300 002A       LDA     RR6,((3))%002A
00A0:  5F00 8300 0001       CALL    ((3))%0001
00A6:  7602 8300 0000       LDA     RR2,((3))%0000
00AC:  2104 FFFF            LD      R4,#%FFFF
00B0:  8057                 LDK     R5,#7
00B2:  7606 8300 0022       LDA     RR6,((3))%0022
00B8:  5F00 8300 0001       CALL    ((3))%0001
00BE:  7602 8300 0000       LDA     RR2,((3))%0000
00C4:  5F00 8300 0002       CALL    ((3))%0002
```

```
 39.  3      writeln('FUNC.AREA',' = ',area:2);
00CA:  7602 8300 0000      LDA     RR2,((3))Z0000
00D0:  2104 FFFF           LD      R4,#ZFFFF
00D4:  BD59                LDK     R5,#9
00D6:  7606 8300 0018      LDA     RR6,((3))Z0018
00DC:  5F00 8300 0001      CALL    ((3))Z0001
00E2:  7602 8300 0000      LDA     RR2,((3))Z0000
00E8:  2104 FFFF           LD      R4,#ZFFFF
00EC:  BD53                LDK     R5,#3
00EE:  7606 8300 0014      LDA     RR6,((3))Z0014
00F4:  5F00 8300 0001      CALL    ((3))Z0001
00FA:  7602 8300 0000      LDA     RR2,((3))Z0000
0100:  BD42                LDK     R4,#2
0102:  61F5 02D6           LD      R5,((2))Z0006(R15)
0106:  5F00 8300 0003      CALL    ((3))Z0003
010C:  7602 8300 0000      LDA     RR2,((3))Z0000
0112:  5F00 8300 0002      CALL    ((3))Z0002
 40.  4      writeln('TEST UNIT',' = ',testprog);
0118:  7602 8300 0000      LDA     RR2,((3))Z0000
011E:  2104 FFFF           LD      R4,#ZFFFF
0122:  BD59                LDK     R5,#9
0124:  7606 8300 000A      LDA     RR6,((3))Z000A
012A:  5F00 8300 0001      CALL    ((3))Z0001
0130:  7602 8300 0000      LDA     RR2,((3))Z0000
0136:  2104 FFFF           LD      R4,#ZFFFF
013A:  BD53                LDK     R5,#3
013C:  7606 8300 0006      LDA     RR6,((3))Z0006
0142:  5F00 8300 0001      CALL    ((3))Z0001
0148:  7602 8300 0000      LDA     RR2,((3))Z0000
014E:  2104 FFFF           LD      R4,#ZFFFF
0152:  BD57                LDK     R5,#7
0154:  34E6 0012           LDA     RR6,RR14(#18)
0158:  5F00 8300 0001      CALL    ((3))Z0001
015E:  7602 8300 0000      LDA     RR2,((3))Z0000
0164:  5F00 8300 0002      CALL    ((3))Z0002
 41.  5      writeln;
016A:  7602 8300 0000      LDA     RR2,((3))Z0000
0170:  5F00 8300 0002      CALL    ((3))Z0002
 42.  6      writeln;
0176:  7602 8300 0000      LDA     RR2,((3))Z0000
017C:  5F00 8300 0002      CALL    ((3))Z0002
 43.  7      for i:= 1 to ilength do
0182:  4DF5 021A 0001      LD      ((2))Z001A(R15),#Z0001
0188:  EB0A                JR      019E
 44.  8        write('*');
018A:  7602 8300 0000      LDA     RR2,((3))Z0000
0190:  2104 002A           LD      R4,#Z002A
0194:  5F00 8300 0004      CALL    ((3))Z0004
019A:  69F0 021A           INC     ((2))Z001A(R15),#1
019E:  4DF1 021A 0050      CP      ((2))Z001A(R15),#Z0050
01A6:  E2F2                JR      LE,018A
 45.  9      writeln;
01A8:  7602 8300 0000      LDA     RR2,((3))Z0000
01AC:  5F00 8300 0002      CALL    ((3))Z0002
 46.  10     tunit:=testprog;
01B2:  34EC 0012           LDA     RR12,RR14(#18)
01B6:  760A 8100 0008      LDA     RR10,((1))Z0008
01BC:  BD14                LDK     R1,#4
01BE:  BBC1 01A0           LDIR    @R10,@R12,R1
 47.  11     expect:=tnum;
01C2:  61F1 02DE           LD      R1,((2))Z00DE(R15)
01C6:  6F01 8100 008C      LD      ((1))Z008C,R1
 48.  12     icount:=needlength;
01CC:  4DD5 8100 0010 0007 LD      ((1))Z0010,#Z0007
 49.  13     for i:=1 to 20 do
01D4:  4DF5 021A 0001      LD      ((2))Z001A(R15),#Z0001
01DA:  EB19                JR      02DE
 50.  14       cases[i]:=' ';
01DC:  61F1 021A           LD      R1,((2))Z001A(R15)
01E0:  0801 0001           CP      R1,#Z0001
01E4:  E103                JR      LT,01EC
01E6:  0801 0014           CP      R1,#Z0014
01EA:  E2D4                JR      LE,01F4
01EC:  BD2D                LDK     R2,#13
01EE:  5F00 8300 0005      CALL    ((3))Z0005
01F4:  1900 0006           MULT    RR0,#Z0006
01F8:  760C 8300 0000      LDA     RR12,((3))Z0000
01FE:  761A 8100 000E      LDA     RR10,((1))Z000E(R1)
0204:  BD13                LDK     R1,#3
0206:  BBC1 01A0           LDIR    @R10,@R12,R1
020A:  69F0 021A           INC     ((2))Z001A(R15),#1
020E:  4DF1 021A 0014      CP      ((2))Z001A(R15),#Z0014
0214:  E2E3                JR      LE,01DC
 51.  15     EXEC:=0;
0216:  4D08 8100 0090      CLR     ((1))Z0090
 52.  16     KO:=0
021C:  4D08 8100 0092      CLR     ((1))Z0092
0222:  010F 001C           ADD     R15,#Z001C
0226:  9ED8                RET
 53.  16   END;
 54.  0
```

A. MESSAGGI DI ERRORE DI RUN TIME

In questa appendice vengono riportati i messaggi di errore che possono essere emessi a run time durante l'esecuzione di un programma PASCAL+.

Questi messaggi sono visualizzati a terminale e possono assumere una delle due seguenti forme :

- **** Pascal+ Run Time Error ****
  " Stringa esplicativa "

  dove " Stringa esplicativa " e' uno qualsiasi dei messaggi (in inglese) di errore riportati nella prossima sezione

- **** Pascal+ Run Time Error ****
  --- System Error : " Numero "

  dove " Numero " e' uno qualsiasi dei numeri della Tabella contenente i messaggi di errore di Sistema (Manuale dei Messaggi).

ELENCO DEI MESSAGGI

Vengono elencati di seguito, in ordine alfabetico, i messaggi di errore di run time che possono apparire (in lingua inglese) su terminale durante durante l'esecuzione di un programma PASCAL+.

Argument of seek is out of file bounds

L'argomento della seek e' fuori dei limiti del file

Array index out of range

L'indice dell'array va fuori del range consentito

Assignment out of range

L'assegnamento effettuato va fuori del range consentito

Attempt to read or get file open for writing

E' stato fatto un tentativo di leggere o prelevare elementi da un file aperto per scrittura

---

Attempt to read past end of file

E' stato fatto un tentativo di leggere oltre la fine di un file

---

Attempt to write or put file open for reading

E' stato fatto un tentativo di scrivere o inserire elementi in un file aperto per lettura

Can't reset file

Non si puo' reimpostare dall'inizio il file con intento di lettura

---

Can't rewrite file

Non si puo' reimpostare dall'inizio il file con intento di riscriverlo

---

Case label not found

La case label non e' stata trovata

Chr argument out of range

L'argomento chr va fuori del range consentito

Dispose pointer out of range

Rilascio del puntatore fuori del range consentito

Eoln called on a file open for writing

E' stato richiesto un fine linea (eoln) in un file aperto per scrittura

---

Error reading integer

E' stato commesso un errore nella lettura di un integer

Error reading longinteger

E' stato commesso un errore nella lettura di un longinteger

Error reading real

E' stato commesso un errore nella lettura di un real

Floating conversion overflow

Vi e' stato un overflow nella conversione in floating point

Floating overflow

Vi e' stato un overflow nella scrittura in floating point

Floating zero divide attempt

E' stato fatto un tentativo di divisione per zero (floating)

For loop termination out of range

La chiusura del for loop va fuori del range stabilito

Function arctan not yet implemented

La funzione arctan non e' stata ancora implementata

Function cos not yet implemented

La funzione cos non e' stata ancora implementata

Function exp not yet implemented

La funzione exp non e' stata ancora implementata

Function ln not yet implemented

La funzione ln non e' stata ancora implementata

Function sin not yet implemented

La funzione sin non e' stata ancora implementata

Function sqrt not yet implemented

La funzione sqrt non e' stata ancora implementata

Heap segment out of memory

Il segmento di heap del programma va fuori dello spazio di memoria consentito

---

Not proper dispose pointer

Rilascio di puntatore non corretto

Only ByteStream or Sequential files can be opened

Possono essere aperti solo file ByteStream o Sequenziali

Path name too long

Il path name indicato e' troppo lungo

Ran out of memory

Il programma e' andato fuori dei limiti concessi di memoria

Set range out of implementation limits

Il range del set oltrepassa i limiti consentiti di memoria

Singleton set out of implementation limits

Il set e' fuori dei limiti di implementazione

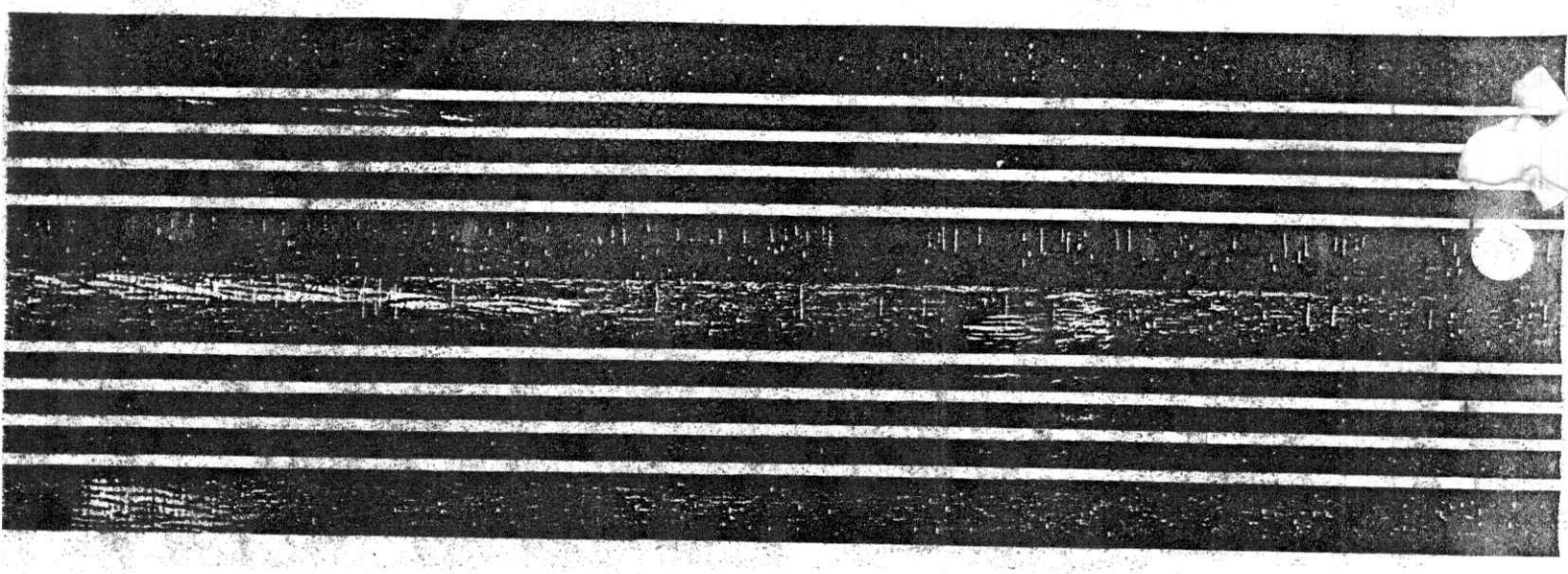
Succ or Pred result out of range

Il risultato delle funzioni Succ o Pred e' fuori del range consentito

---

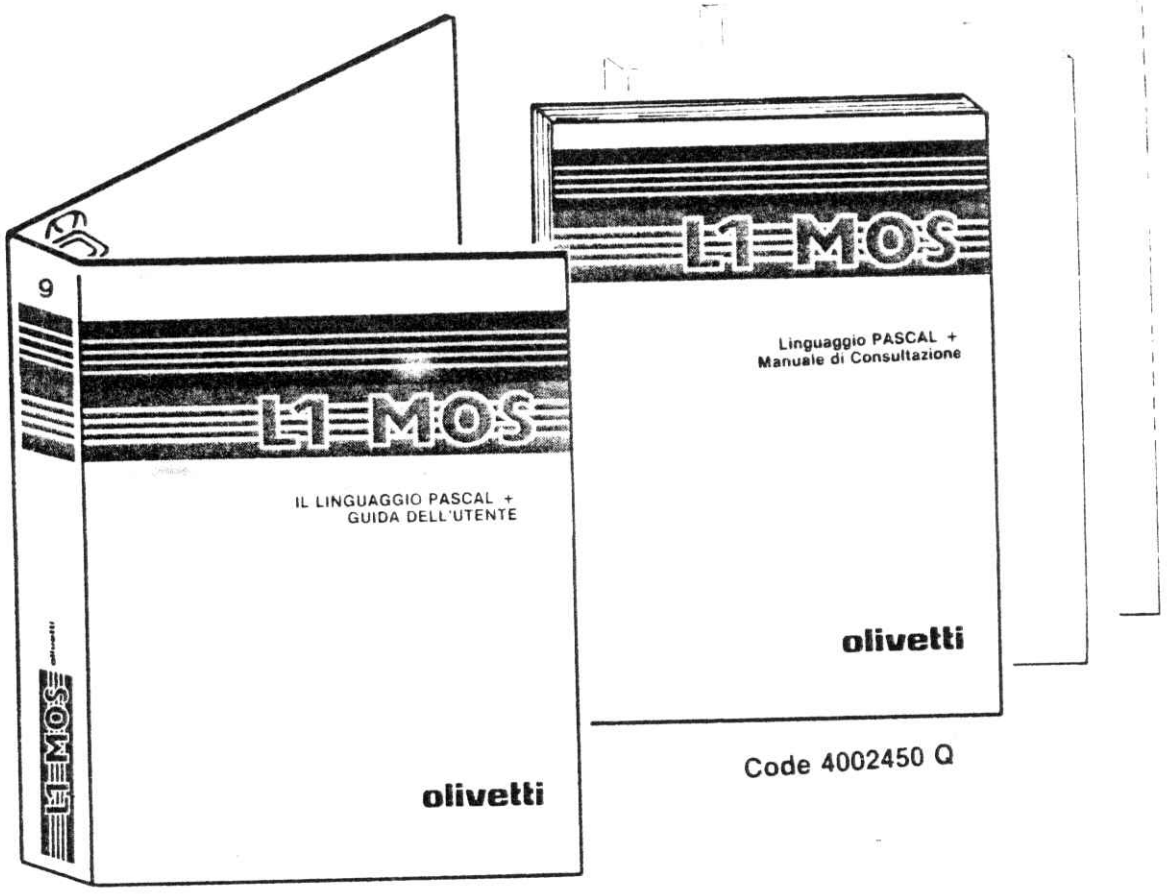
Value parameter out of range

Il value parameter e' fuori del range stabilito



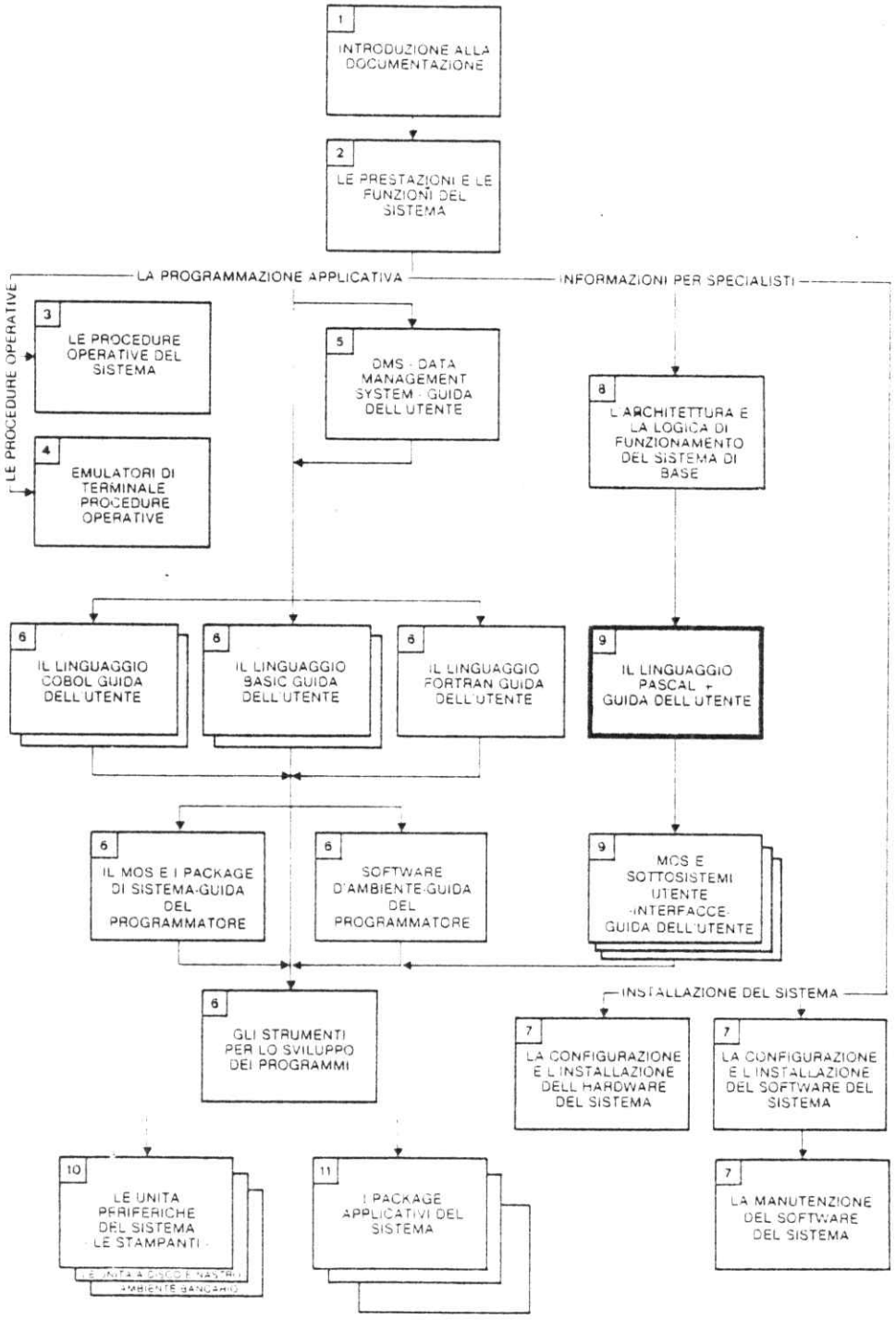
Code 4002470 J (0)

Printed in Italy



Code 4002450 Q

Code 4004980 F


1
INTRODUZIONE ALLA DOCUMENTAZIONE
2
LE PRESTAZIONI E LE FUNZIONI DEL SISTEMA
LA PROGRAMMAZIONE APPLICATIVA
INFORMAZIONI PER SPECIALISTI
LE PROCEDURE OPERATIVE
3
LE PROCEDURE OPERATIVE DEL SISTEMA
4
EMULATORI DI TERMINALE PROCEDURE OPERATIVE
5
DMS - DATA MANAGEMENT SYSTEM - GUIDA DELL'UTENTE
8
L'ARCHITETTURA E LA LOGICA DI FUNZIONAMENTO DEL SISTEMA DI BASE
6
IL LINGUAGGIO COBOL GUIDA DELL'UTENTE
6
IL LINGUAGGIO BASIC GUIDA DELL'UTENTE
6
IL LINGUAGGIO FORTRAN GUIDA DELL'UTENTE
9
IL LINGUAGGIO PASCAL + GUIDA DELL'UTENTE
6
IL MOS E I PACKAGE DI SISTEMA-GUIDA DEL PROGRAMMATORE
6
SOFTWARE D'AMBIENTE-GUIDA DEL PROGRAMMATORE
9
MOS E SOTTOSISTEMI UTENTE -INTERFACCE- GUIDA DELL'UTENTE
6
GLI STRUMENTI PER LO SVILUPPO DEI PROGRAMMI
INSTALLAZIONE DEL SISTEMA
7
LA CONFIGURAZIONE E L'INSTALLAZIONE DELL'HARDWARE DEL SISTEMA
7
LA CONFIGURAZIONE E L'INSTALLAZIONE DEL SOFTWARE DEL SISTEMA
7
LA MANUTENZIONE DEL SOFTWARE DEL SISTEMA
10
LE UNITA' PERIFERICHE DEL SISTEMA LE STAMPANTI
LE UNITA' A DISCO E NASTRO
AMBIENTE GANGRIO
11
I PACKAGE APPLICATIVI DEL SISTEMA

INDICE

PAGINA

PARTE 1. FILOSOFIA DEL LINGUAGGIO E STRUTTURA DEL PROGRAMMA

1-1 1. COS'E' IL PASCAL+ ?

1-1 NOTE STORICHE E COLLOCAZIONE DEL LINGUAGGIO

1-2 CARATTERISTICHE FONDAMENTALI DEL LINGUAGGIO

2-1 2. NOTAZIONE, VOCABOLARIO E STRUTTURA LESSICALE

2-1 NOTAZIONE DEI DIAGRAMMI SINTATTICI

2-3 L'INSIEME DEI CARATTERI

2-3 STRUTTURA LESSICALE

3-1 3. PROGRAMMAZIONE TRAMITE MACCHINE ASTRATTE

3-1 UN ESEMPIO DI MACCHINA ASTRATTA

3-3 LA STRUTTURA GERARCHICA DI UN PROGRAMMA

3-5 SEQUENZA DI SCRITTURA E DI COMPILAZIONE

4-1 4. REGOLE DI VISIBILITA'

4-1 TERMINOLOGIA RELATIVA ALLA NIDIFICAZIONE DELLE RISORSE

4-3 ENUNCIATO DELLE REGOLE DI VISIBILITA'

4-4 ROUTINE RICORSIVE

4-5 RISORSE PREDEFINITE

5-1 5. IL CONCETTO DI TIPO

5-1 COS'E' UN TIPO

5-3 TIPI SEMPLICI

5-5 TIPI STRUTTURATI

6-1 6. ESECUZIONE DI PROGRAMMI E GESTIONE DELLA MEMORIA

6-1 L'EFFETTO DI UNA CHIAMATA DI ROUTINE

6-2 IL RUN TIME STACK

PAGINA

6-6 LO HEAP

7-1 7. COMPILAZIONI SEPARATE

7-1 LIBRERIE DI RISORSE

7-1 COS'E' UNA LIBRERIA DI RISORSE

7-6 ROUTINE SEPARATE

7-7 TIPI DI DATI INCAPSULATI

7-9 UNITA' PRINCIPALE DI PROGRAMMA

7-9 DIAGRAMMA SINTATTICO

7-11 MODULI

7-12 DIAGRAMMA SINTATTICO

PARTE II. CARATTERISTICHE DEL LINGUAGGIO BASE

8-1 8. TIPI

8-1 INDICATORI DI TIPO

8-2 DESCRIZIONE DI TIPO

8-3 TIPI SEMPLICI

8-3 DEFINITI DALL'UTENTE

8-3 ENUMERAZIONE

8-4 SUBRANGE

8-5 TIPI SEMPLICI PREDEFINITI

8-8 TIPI STRUTTURATI

8-8 IL COSTRUTTORE ARRAY

8-11 IL TIPO STRINGA

8-11 IL COSTRUTTORE RECORD

8-15 IL COSTRUTTORE SET

8-16 IL COSTRUTTORE FILE

8-17 IL TIPO TEXT

8-17 TIPI PUNTATORE

8-19 COMPATIBILITA' TRA TIPI

PAGINA

8-19 COMPATIBILITA'

8-19 COMPATIBILITA' NELL'ASSEGNAZIONE

9-1 9. COSTANTI

9-1 INDICATORI DI COSTANTI

9-2 DESCRIZIONE DI COSTANTI

10-1 10. VARIABILI

10-1 DESCRIZIONE DI VARIABILE

10-2 INDICATORE DI VARIABILE

11-1 11. ESPRESSIONI

11-1 LA STRUTTURA DI UN'ESPRESSIONE

11-3 OPERANDI

11-4 OPERATORI

11-4 OPERATORI ARITMETICI

11-5 OPERATORI BOOLEANI

11-5 OPERATORI SU INSIEMI

11-6 OPERATORI RELAZIONALI

11-7 PRECEDENZA TRA OPERATORI E USO DELLE PARENTESI

12-1 12. ISTRUZIONI

12-1 GENERALITA' SULLE ISTRUZIONI

12-3 ISTRUZIONI DI ASSEGNAZIONE

12-4 ISTRUZIONE DI CHIAMATA DI PROCEDURA

12-4 SEQUENZA DI ISTRUZIONI

12-5 ISTRUZIONE COMPOSTA

12-6 ISTRUZIONE VUOTA

12-7 ISTRUZIONE IF

12-8 NIDIFICAZIONE DI ISTRUZIONI IF

12-9 ISTRUZIONE CASE

PAGINA

12-11 ISTRUZIONE REPEAT

12-12 ISTRUZIONE WHILE

12-13 ISTRUZIONE FOR

12-17 ISTRUZIONE EXIT

12-18 ISTRUZIONE RETURN

12-19 ISTRUZIONE WITH

13-1 13. PROCEDURE E FUNZIONI

13-1 DESCRIZIONE DI UNA PROCEDURA

13-4 DESCRIZIONE DI FUNZIONE

13-6 CHIAMATA ED ESECUZIONE DI PROCEDURE E FUNZIONI

13-8 PASSAGGIO DI PARAMETRI

13-17 PROCEDURE E FUNZIONI PREDEFINITE PER LA GESTIONE DI FILE

13-18 REWRITE (F)

13-18 REWRITE (F,s)

13-18 PUT(F)

13-19 RESET (F)

13-19 RESET (F,s)

13-19 GET (F)

13-19 EOF (F)

13-20 CLOSE (F)

13-20 READ (F, V1,.., Vn)

13-21 WRITE (F, E1,..,En)

13-22 READ (F,V)

13-24 READLN (F)

13-25 WRITE (F,P)

13-27 WRITELN (F)

13-28 EOLN (F)

PAGINA

13-28 PAGE (F)

13-28 PROCEDURE PREDEFINITE PER L'ALLOCAZIONE DINAMICA

13-28 NEW (P)

13-29 DISPOSE (P)

13-30 FUNZIONI ARITMETICHE PREDEFINITE

13-30 ABS (X)

13-30 SQR (X)

13-31 SIN (X)

13-31 COS (X)

13-31 EXP (X)

13-31 LN (X)

13-31 SQRT (X)

13-32 ARCTAN (X)

13-32 PROCEDURE E FUNZIONI DI CONVERSIONE PREDEFINITE

13-32 TRUNC (X)

13-32 ROUND (X)

13-33 PACK (A,I,Z)

13-34 UNPACK (Z,A,I)

13-35 FUNZIONI ORDINALI PREDEFINITE

13-35 ORD (X)

13-36 CHR (X)

13-36 SUCC (X)

13-37 PRED (X)

13-37 ODD (X)

14-1 **14. MODULI**

14-1 COSA E' UN MODULO

14-1 LISTA DI ESPORTAZIONE

PAGINA

14-3 LISTA DI IMPORTAZIONE

14-5 RIDEFINIZIONE ED USO DI RISORSE IMPORTATE

14-7 ESEMPI

PARTE III. GUIDA ALLA PROGRAMMAZIONE

15-1 15. CONSIDERAZIONI DI PROGRAMMAZIONE

15-1 L'USO DEL PUNTO E VIRGOLA

15-4 PROGRAMMAZIONE SENZA 'GOTO'

15-7 L'USO DI STRUTTURE RIPETITIVE E RICORSIVE

15-7 WHILE

15-7 REPEAT

15-8 FOR

15-9 RICURSIONE

15-11 L'USO DI PARAMETRI

15-11 INPUT

15-12 OUTPUT

15-12 INPUT/OUTPUT

15-14 DETTAGLI SULL'INPUT/OUTPUT

15-14 FILE-TESTO

15-16 FILE NORMALI

16-1 16. PARTICOLARITA' SUI MODULI

16-1 COSTRUZIONE DI STRUTTURE DINAMICHE

16-1 ARRAYS SPARSI

16-3 LISTE

16-7 ALBERI

16-10 LA METODOLOGIA BOTTOM UP

PARTE IV. IL LINGUAGGIO ESTESO

17-1 17. PROGRAMMAZIONE CONCORRENTE

PAGINA

17-1 PROCESSI PASCAL+

17-1 COSA E' UN PROCESSO PASCAL+

17-2 CREAZIONE E DISTRUZIONE DI PROCESSI

17-3 PROTEZIONE DI PROCESSI 'EFFETTIVI'

17-8 MONITOR PASCAL+

17-8 COSA E' UN MONITOR PASCAL+

17-14 IMPORTAZIONI, ESPORTAZIONI E RIDEFINIZIONI

17-14 ESEMPIO

17-18 SINCRONIZZAZIONE DI PROCESSI COOPERANTI

17-21 IL TIPO CONDITION

17-21 LA PROCEDURA WAIT

17-22 LA PROCEDURA SIGNAL

17-24 ESEMPIO

18-1 18. ESTENSIONI ALLE LIBRERIE DI RISORSE

18-1 ISTANZE DI LIBRERIE DI RISORSE

18-1 COS'E' UN'ISTANZA DI LIBRERIA DI RISORSE

18-5 COMPILAZIONE DI LIBRERIE DI RISORSE ISTANZIABILI

18-7 DIAGRAMMA SINTATTICO DI LIBRERIA DI RISORSE INSTANZIABILI

18-8 COME ISTANZIARE UNA LIBRERIA DI RISORSE

18-10 RIDEFINIZIONE ED USO DELLE RISORSE ISTANZIATE

18-11 LISTA DI IMPORTAZIONE

18-13 DEFINIZIONE DI ROUTINE

18-15 DEFINIZIONE DI VARIABILE, COSTANTE, TIPO

18-15 LISTA DI ESPORTAZIONE

PARTE V. PRESTAZIONI DEL LINGUAGGIO ESTESO

19-1 19. COSTANTI

19-1 COSTANTI OTTALI

PAGINA

19-2 COSTANTI ESADECIMALI

19-3 SPECIALI INTERPRETAZIONI DI STRINGHE

19-4 COSTANTI ISTANZIATE

20-1 20. TIPI

20-1 TIPI PREDEFINITI

20-1 LONGINTEGER

20-2 CONDITION

20-2 DYNAMIC ARRAY

20-5 TIPI DEFINITI DALL'UTENTE

20-5 MODULE

20-5 MONITOR

21-1 21. VARIABILI

21-1 DIRETTIVA AT

21-2 VARIABILI ISTANZIATE

22-1 22. PROCEDURE E FUNZIONI

22-1 ROUTINE ISTANZIATE

22-2 ROUTINE PREDEFINITE PER LA GESTIONE DEI FILE

22-2 UPDATE (F)

22-2 UPDATE (F,s)

22-3 SEEK (F,n)

22-3 SEEKEOF (F)

22-4 SENSE (F)

22-4 GET (F)

22-5 PUT (F)

22-5 READ (F,V)

22-6 WRITE (F,E)

22-6 SFRUTTAMENTO OTTIMALE DELLA MEMORIA AUSILIARIA

PAGINA

22-9 FUNZIONI PREDEFINITE DI CONVERSIONE

22-9 RECAST (X, identificatore-di-tipo)

22-10 LONGINT (X)

22-10 INT (X)

22-10 LTRUNC (X)

22-10 LROUND (X)

22-11 ROUTINE PREDEFINITE DI COMUNICAZIONE TRA PROCESSI

22-11 WAIT (X)

22-11 SIGNAL (X)

22-11 ALTRE ROUTINE PREDEFINITE

22-11 SIZE (identificatore-di-tipo)

22-12 MOVEINTO (espressione-iniziale, variabile-array, espressione-indice

22-13 MOVEFROM (var-destinazione, var-array, espressione-indice)

22-13 MOVEARRAY (in -array, in-indice, dest-array, dest-indice, lungh)

22-14 MAKEPTR (X)

22-14 ALLOC (variabile-array, limite1..limite2, limite3..limite4)

22-15 DEALLOC (variabile-array)

A-1 A. DIAGRAMMI SINTATTICI

A-1 LESSICO

A-3 PROGRAMMA E MODULO

A-4 DESCRIZIONE DI RISORSE

A-12 ESPRESSIONE

A-14 ISTRUZIONE

A-19 PROCEDURA E FUNZIONE

B-1 B. INDICE ANALITICO

PARTE 1. FILOSOFIA DEL LINGUAGGIO E STRUTTURA DEL PROGRAMMA

1. COS'E' IL PASCAL+ ?

In questo capitolo vengono riassunte le principali caratteristiche del linguaggio PASCAL+, le sue motivazioni e la sua collocazione tra altri linguaggi.

Prima viene presentata una breve collocazione del linguaggio nella storia degli altri linguaggi di programmazione.

Di seguito, sono riassunte le piu' importanti particolarita' del linguaggio Pascal standard, da cui e' derivato il Pascal+, seguite da uno schema delle principali estensioni di PASCAL+ rispetto al PASCAL standard.

NOTE STORICHE E COLLOCAZIONE DEL LINGUAGGIO

Il PASCAL+ e' una evoluzione del linguaggio PASCAL, progettato nel 1968 da Wirth e Jensen, ed in seguito sviluppato e raffinato nell'ISO PASCAL ancora in fase di sviluppo.

Il linguaggio PASCAL e' stato progettato come linguaggio "general purpose" fortemente orientato verso l'utente rivelandosi un evento chiave nella storia dei linguaggi di programmazione. In essa possono essere riconosciuti alcuni orientamenti fondamentali:

a) l'allocazione dei dati e' diventata nel tempo sempre piu' un compito affidato al sistema, anziche' all'utente;

b) l'insieme delle istruzioni e la struttura del linguaggio in generale sono stati sempre piu' orientati verso modelli umani piuttosto che verso configurazioni della macchina;

c) la rappresentazione dei dati che un tempo era legata alle configurazioni hardware interne, e' diventata sempre piu' definibile dall'utente per mezzo di dati di tipo astratto: cioe' i dati sono considerati piu' dal punto di vista del loro valore logico che della loro rappresentazione fisica in memoria;

d) il supporto run-time per l'esecuzione dei programmi e' diventato sempre piu' potente rendendo i linguaggi piu' indipendenti dalle caratteristiche del sistema operativo.

Tale evoluzione puo' essere osservata in una sommaria ricapitolazione dei piu' importanti linguaggi di programmazione:

LINGUAGGI ASSEMBLER: l'allocazione dei dati e' a carico del programmatore; scarseggiano i modelli di dati e non esistono ne' strutture di controllo ne'

	strutture di dati;
FORTRAN:	allocazione statica dei dati; limitati i tipi di dati; le strutture di controllo e di dati sono limitate;
COBOL:	allocazione statica dei dati; tipi di dati e strutture di controllo limitati; numerose strutture di dati; supporti run-time limitati;
PL1:	allocazione dinamica dei dati; tipi di dati limitati; buone strutture di controllo; numerose strutture di dati; supporti run-time estesi;
PASCAL:	allocazione dinamica dei dati; esteso insieme di tipi di dati definibili dall'utente; numerose strutture di controllo e di dati; supporti run-time estesi;
PASCAL+:	estensione del PASCAL: possibilita' di sviluppare packages e compilazioni separate;
ADA:	estensione di PASCAL: possibilita' di sviluppare packages, compilazioni separate, programmazione simultanea; facile gestione delle eccezioni; strutture di dati astratte.

CARATTERISTICHE FONDAMENTALI DEL LINGUAGGIO

Il linguaggio PASCAL dispone di un insieme di strumenti per la strutturazione dei programmi. Esso si basa sul concetto di programmazione strutturata, ed include le piu' note forme condizionali ed iterative. Si hanno due tipi di routine: le procedure e le funzioni, che possono essere utilizzate ricorsivamente ed accettano inoltre differenti tipi di parametri: valore, riferimento e routine.

La piu' importante caratteristica del linguaggio e', tuttavia la capacita' di definire tipi astratti di dati enumerando i loro possibili valori. Quindi si possono creare modelli strutturati tramite gli schemi base di strutturazione: array, record, file e set e costruire, tramite i puntatori, strutture dinamiche di dati di ogni forma.

Nonostante queste possibilita', il linguaggio e' caratterizzato da una notevole concisione e chiarezza di descrizione, che ne facilitano l'apprendimento, la scrittura e la lettura.

Inoltre, il linguaggio PASCAL+ consente compilazioni separate di programmi con condivizione delle risorse e con possibilita' di racchiudere le relative operazioni all'interno di tipi strutturati (moduli).
Si ha anche un certo numero di procedure e funzioni predefinite per consentire lo sviluppo del software di sistema.

2. NOTAZIONE, VOCABOLARIO E STRUTTURA LESSICALE

Nel presente capitolo sono illustrati i simboli con i quali e' costruito un programma PASCAL+.

Inoltre, e' spiegata la notazione utilizzata in questo manuale per la descrizione dei costrutti sintattici.

NOTAZIONE DEI DIAGRAMMI SINTATTICI

Nel presente manuale gli elementi di PASCAL+ sono descritti con l'ausilio di diagrammi sintattici.
I diagrammi sintattici mostrano come formare un costrutto sintatticamente corretto, ma non danno informazioni sulla loro semantica.
Per esempio, il diagramma sintattico della figura che segue, mostra come scrivere una sequenza di cifre ("digit") che il compilatore accetta come un numero intero corretto, ma non spiega nulla sul suo comportamento o sul contesto nel quale viene usato.

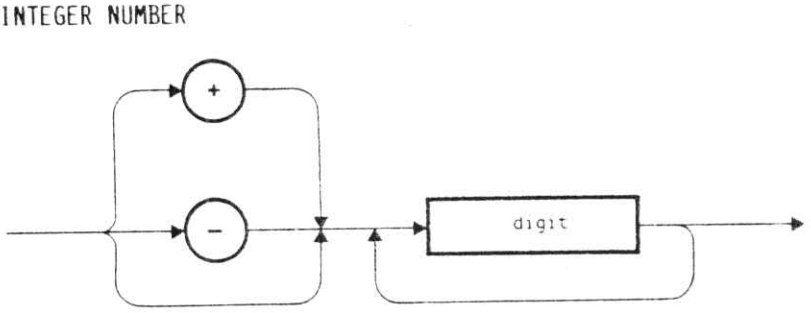


Fig. 2-1 Un Semplice Diagramma Sintattico

I diagrammi sintattici seguono le seguenti regole per la descrizione dei costrutti sintattici:

- i simboli racchiusi in cerchi sono simboli terminali, cioe' appartengono al lessico del linguaggio e devono percio' essere utilizzati nelle frasi PASCAL+ cosi' come appaiono nei diagrammi sintattici.

- i simboli racchiusi in rettangoli o quadrati sono simboli non terminali indicanti un costrutto sintattico. La maggior parte di questi costrutti e' ovviamente descritta attraverso altri diagrammi sintattici, ad esclusione di quelli che sono comuni alla maggior

parte dei linguaggi di programmazione (ad esempio "digit").

- i simboli terminali e non terminali sono connessi con frecce che indicano i corretti percorsi sintattici. Nell'intersezione di piu' linee, si puo' seguire un percorso qualsiasi.

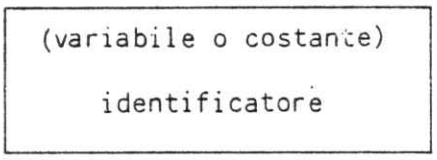
Quando si scrive un programma in PASCAL+, si possono usare i diagrammi sintattici ogniqualvolta si vuole controllare la correttezza del testo scritto; esso risulta corretto se sui diagrammi sintattici esiste almeno un cammino corrispondente al testo scritto. Percio', il diagramma sintattico della figura precedente indica che un numero intero e' costituito da una sequenza di cifre, tante quante si vuole, eventualmente precedute dal simbolo '+' o dal simbolo '-'.
In base a quanto detto, tutti i numeri seguenti sono numeri interi corretti:

```
     1
  -998
 +4725
 10546
000000
    -0
```

Al contrario, quelli che seguono sono numeri interi errati:

```
+ 1   (tra il segno e la cifra c'e' uno spazio)
 3A   (A non e' una cifra)
 5.   (il punto non e' permesso).
```

All'interno di un riquadro, un simbolo non terminale puo' essere preceduto da un commento racchiuso tra parentesi; ad esempio:



Tali commenti appaiono unicamente a scopo di chiarezza; per cui non si fa mai riferimento ad essi all'interno dei diagrammi sintattici.

Essi danno alcune informazioni sul significato dei simboli non-terminali: nell'esempio visto sopra il commento stabilisce che l'identificatore da usare deve essere un identificatore che denota una variabile o una costante precedentemente definita.
L'assenza di un commento indicherebbe che si puo' inventare un nuovo identificatore.

L'INSIEME DEI CARATTERI

Un programma PASCAL+ e' una sequenza di caratteri appartenenti ad un insieme contenente almeno i seguenti simboli:

a) le lettere maiuscole dell'alfabeto inglese (da A a Z);
b) le cifre decimali (da 0 a 9);
c) il carattere di sottolineatura (_);
d) inoltre i seguenti caratteri:

Simbolo	Nome
+	piu'
-	meno
*	asterisco
/	barra
=	uguale
>	maggiore
<	minore
(	parentesi tonda aperta
)	parentesi tonda chiusa
[	parentesi quadra aperta
]	parentesi quadra chiusa
.	punto
,	virgola
;	punto e virgola
:	due punti
^	freccia
'	apice
	spazio
(niente)	fine linea

Ad eccezione delle parentesi graffe { e }, che hanno un significato particolare, gli altri caratteri sono ammessi solo all'interno di commenti o in stringhe di caratteri (questo argomento sara' trattato in seguito). Sono ammesse anche le lettere minuscole e sono considerate equivalenti alle corrispondenti lettere maiuscole.

STRUTTURA LESSICALE

I precedenti caratteri sono utilizzati, nei programmi PASCAL+, per comporre dei simboli. Se ne illustrano alcuni nel seguente semplice esempio:

```
IF 100 >= A THEN
```

Questa parte di programma contiene simboli di diversa specie: alcune parole (IF, A, THEN), un numero (100), una coppia di caratteri (>=) e degli spazi tra i simboli.

Le parole IF e THEN, tra quelle viste, hanno un significato ben preciso, determinato dal linguaggio: esse sono dette parole chiave in quanto immediatamente riconosciuta dal compilatore.

La parola A, al contrario, ha un significato che dipende dal contesto (in questo caso e' una variabile definita dall'utente): essa e' chiamata identificatore, perche' utilizzata per identificare un ente (ad esempio routine, tipi, variabili,...).

Il numero 100 e' una costante numerica, cioe' denota un valore fissato: intero o reale, come si vedra' piu' avanti.

La coppia di caratteri >= e' un simbolo unico appartenente all'insieme di simboli speciali che hanno lo stesso ruolo delle parole chiave, costituite da uno o due caratteri speciali della lista precedente. Nell'esempio precedente, il significato di ">=" e' "maggiore o uguale".

Si fanno ora alcune considerazioni estendendo l'esempio gia' visto:

```
IF 100 >= A THEN (* ASSEGNAZIONE *)
   ST := 'ABCD'
```

I caratteri tra i simboli speciali (* e *) non hanno alcun effetto: tale costrutto e' chiamato "commento" ed e' usato a scopo di documentazione.

Si noti poi un altro costrutto costituito da una stringa di caratteri inclusa fra apici: esso e' una costante di tipo stringa.

Inoltre, c'e' la presenza di alcuni spazi tra i simboli a una "fine linea", che agiscono da simboli separatori.

Questo schema riassume le diverse specie di simboli del PASCAL+:

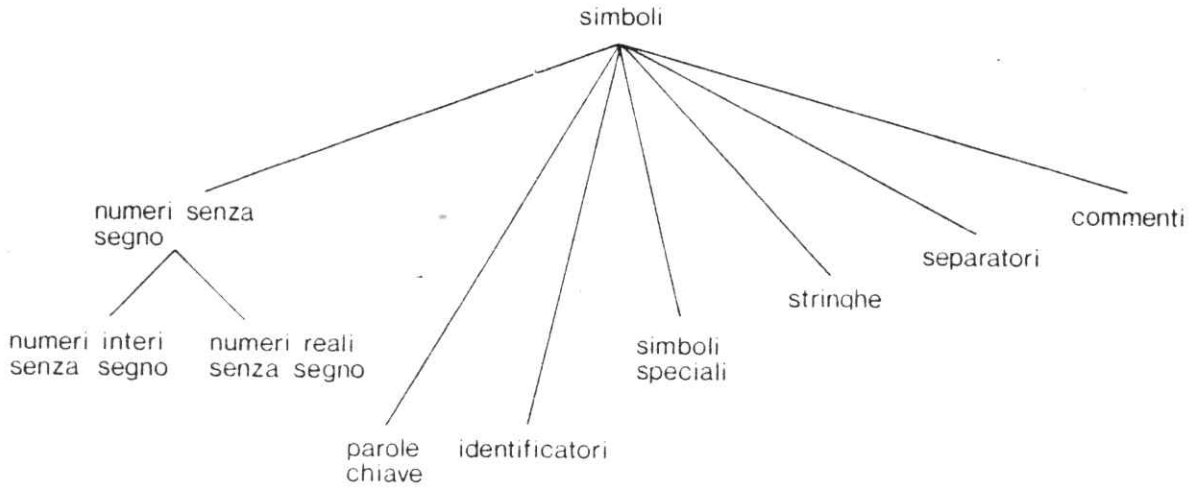


Fig. 2-2 Classificazione dei Simboli PASCAL+

Ogni clssse di simboli sara' ora trattata separatamente.

Numeri

I numeri sono il mezzo fondamentale attraverso cui vengono denotati i valori numerici. Il PASCAL+ fa distinzione fra numeri interi e numeri reali; ad esempio il numero 10 ed il numero 10.0 sono valori distinti, il primo e' di tipo intero, il secondo di tipo reale.

Numeri Interi Senza Segno

Un numero senza segno e' una sequenza di almeno una cifra ("digit").

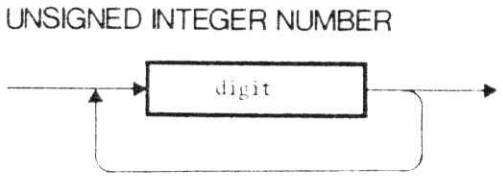


Fig. 2-3 Diagramma Sintattico di un Numero Intero Senza Segno

Un numero intero senza segno e' una costante di tipo INTEGER, ed e' soggetta alle relative limitazioni definite per ogni implementazione (vedi TIPO INTEGER).

Esempio di numeri interi:

```
3
57
0
00
05
```

Numeri Reali Senza Segno

Un numero reale senza segno e' un numero intero senza segno ("unsigned integer number") seguito da una parte decimale e/o da un fattore di scala.

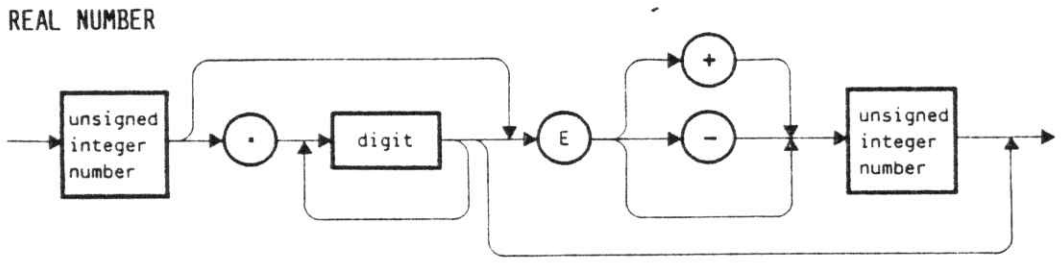


Fig. 2-4 Diagramma Sintattico di un Numero Reale Senza Segno

Si noti che il punto decimale deve essere sempre preceduto e seguito almeno da una cifra; il fattore di scala puo' contenere un segno ed esprime la potenza di dieci per il quale il numero e' moltiplicato.

Un numero reale senza segno e' una costante di tipo REAL ed e' percio' soggetto alle relative limitazioni definite per ciascuna implementazione (vedi TIPO REAL).

Esempio di numeri reali:

```
43.393
0.0
0.0E+2
193E-3
6.000
8E0
```

Parole Chiave

Talune parole hanno un significato fissato, indipendente dal contesto, che non possono essere cambiate dal programmatore. Come detto in precedenza esse sono chiamate parole chiave.

Si hanno le seguenti parole chiave:

AND ARRAY BEGIN CASE CONST DIV DOWNTO ELSE END EXIT EXPORT EXTERNAL FILE FOR FORWARD FROM FUNCTION GOTO IMPORT IF IN LABEL MOD MODULE NIL NOT OF OR OTHERWISE PACKED PROCEDURE PROGRAM RECORD REPEAT RETURN SET THEN TO TYPE UNTIL VAR WHILE WITH.

Identificatori

Gli identificatori sono costituiti da una lettera che puo' essere seguita da una sequenza di lettere e/o numeri e/o caratteri di sottolineatura.

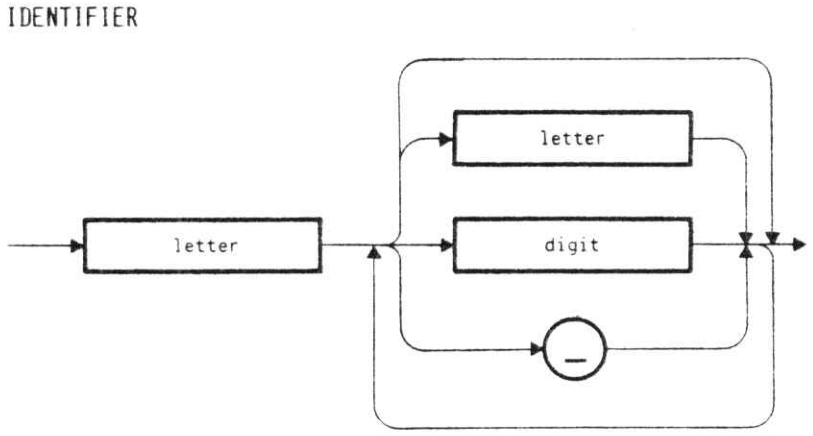


Fig. 2-5 Diagramma Sintattico di un Identificatore

Gli identificatori denotano oggetti, cioe' costanti, tipi, variabili, procedure, funzioni, programmi, moduli, parametri, estremi, campi ed etichette di campo.

Sebbene gli identificatori possano essere di qualsiasi lunghezza, solo i primi 12 caratteri sono significativi. Ogni singolo identificatore deve differire da una parola chiave o da altri identificatori nella sua parte significativa; due identificatori uguali nei primi 12 caratteri vengono considerati come lo stesso identificatore da parte del compilatore.

Esempi:

```
A23
0
TOP_OF_STACK
THIS_IS_A_VERY_LONG_IDENTIFIER
THIS_IS_A_VERY_LONG_IDENTIFIER_BUT_THE_SAME_AS_ABOVE
IFTHEN
```

Il significato di un identificatore dipende dal contesto e puo' essere definito dall'utente o predefinito: esistono infatti taluni identificatori, chiamati identificatori predefiniti che corrispondano ad oggetti predefiniti, a meno che il programmatore non ne sovvrapponga il significato.
La seguente lista comprende gli identificatori predefiniti:

ABS ARCTAN BOOLEAN CHAR CHR CLOSE COS DISPOSE EOF EOLN EXP FALSE GET INPUT INTEGER LN MAXCHAR MAXINTEGER MAXREAL MAXSETCARD MINCHAR MININTEGER MINREAL NEW ODD ORD OUTPUT PACK PAGE PRED PUT READ READLN REAL RESET

REWRITE ROUND SIN SQR SQRT SUCC TEXT TRUE TRUNC UNPACK WRITE WRITELN

Simboli Speciali

I simboli speciali sono costituiti da uno o due caratteri speciali. Quella che segue e' la lista dei simboli speciali:

simbolo speciale	nome
+	piu'
-	meno
*	asterisco
/	barra
=	uguale
>	maggiore
<	minore
(	parentesi tonda aperta
)	parentesi tonda chiusa
[	parentesi quadra aperta
]	parentesi quadra chiusa
.	punto
,	virgola
;	punto e virgola
:	due punti
^	puntatore
>=	maggiore o uguale
<=	minore o uguale
<>	diverso
..	doppio punto
:=	assegnazione

Tra due caratteri che costituiscono un simbolo speciale non puo' apparire ne' uno spazio ne' un fine linea.

Stringhe

Le stringhe sono il mezzo fondamentale per denotare una sequenza di caratteri.

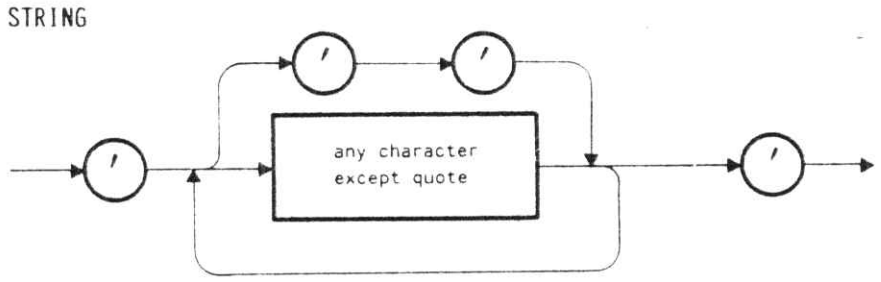


Fig. 2-6 Diagramma Sintattico di una Stringa

Una coppia di apici all'interno di una stringa indica l'apice stesso.

Le stringhe possono essere di lunghezza qualsiasi ma non contenere un fine linea, cioe' una stringa non puo' continuare su piu' linee.

All'interno delle stringhe e' soppresso ogni significato speciale dei caratteri, infatti non e' ammesso il singolo apice.

Le stringhe sono costanti di tipo stringa (vedere tipo ARRAY) con un numero di componenti uguale al numero di caratteri (o coppie di apici) che contiene.

Le stringhe costituite da un singolo carattere o da una singola coppia di apici possono anche essere di tipo CHAR, in relazione al contesto.

Separatori

I separatori sono lo spazio e il termine della riga (fine linea). L'uso dei separatori in un programma e' soggetto alle seguenti regole;

1. deve apparire almeno un separatore tra due numeri, identificatori o parole chiave

2. tra due simboli consecutivi puo' apparire un numero qualsiasi di separatori.

Nei diagrammi sintattici che seguono, la necessita' di eventuali separatori tra due simboli verra' omessa per concisione.

Commenti

I commenti, in un programma PASCAL+, sono il mezzo con cui possono essere inserite le informazioni orientate al lettore.

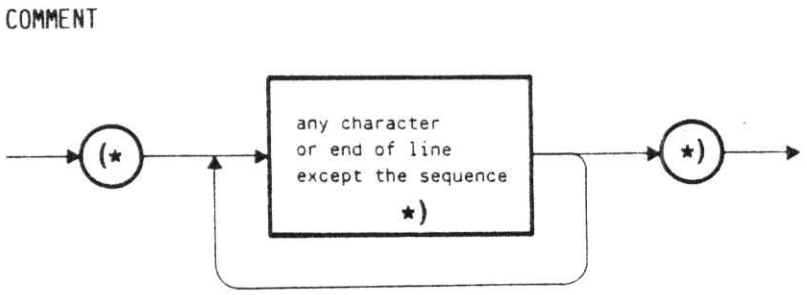


Fig. 2-7 Diagramma Sintattico per i Commenti

Se nell'insieme dei caratteri nazionali sono compresi i caratteri speciali { e } (parentesi graffe), essi risultano rispettivamente equivalenti alle sequenze (* e *) per la costruzione dei commenti, i quali possono contenere qualsiasi carattere.
Si noti che occorre fare attenzione all'uso del simbolo $ nei commenti, poiché viene utilizzato per inviare istruzioni al compilatore, si deve pertanto evitare l'uso di tale carattere nei commenti.

La seguente tabella indica, per ogni nazione, l'equivalente grafico dei codici ASCII specificati.

CODICE ASCII		EQUIVALENTE NAZIONALE															
DECIMALE	ESADECIMALE	USA	ITALIA	FRANCIA	INGHILTERRA	GERMANIA (EST)	GERMANIA (OVEST)	SPAGNA	PORTOGALLO	DANIMARCA	SVEZIA FINLANDIA	NORVEGIA	SVIZZERA FRANCESE	SVIZZERA TEDESCA	GRECIA	JUGOSLAVIA
35	23	#	£	£	£	#	#	£	#	£	#	£	£	£	£	#
36	24	$	$	$	$	$	$	$	$	$	¤	$	$	$	$	¤
64	40	@	§	à	@	§	§	§	§	´	@	´	§	§	@	Š
91	5B	[	°	°	[	Ä	Ä	¡	Ã	Æ	Ä	Æ	à	à	[	Ð
92	5C	\	ç	ç	\	Ö	Ö	Ñ	Ç	Ø	Ö	Ø	ç	ç	\	Ć
93	5D	]	é	§	]	Ü	Ü	¿	Õ	Å	Å	Å	è	è	]	Ž
96	60	`	ù	`	`	`	`	`	`	`	`	`	`	`	`	š
123	7B	{	à	é	{	ä	ä	°	ã	æ	ä	æ	ä	ä	{	đ
124	7C	|	ò	ù	|	ö	ö	Ñ	ç	ø	ö	ø	ö	ö	|	ć
125	7D	}	è	è	}	ü	ü	ç	õ	å	å	å	ü	ü	}	ž
126	7E	~	ì	¨	~	ß	ß	~	º	¨	—	¨	é	é	¨	č

* I caratteri compresi in un cerchio sono usati per funzioni in linguaggio **PASCAL+**

3. PROGRAMMAZIONE TRAMITE MACCHINE ASTRATTE

Lo scopo di questo capitolo e' di illustrare la filosofia del linguaggio ed alcuni aspetti metodologici di programmazione in PASCAL+, oltre a trattare la struttura gerarchica di un programma.

UN ESEMPIO DI MACCHINA ASTRATTA

Supponendo di avere un problema ed una macchina capace di risolverlo, si vuole usare tale macchina per tentare di risolverlo.
Per esempio, se si considera una macchina in grado di giocare a scacchi, essa puo' essere usata per una partita; mentre, se si ha una macchina che non sa giocare, ma e' capace di calcolare una mossa durante una partita di scacchi, si possono dare ad essa appropriate istruzioni per giocare una partita:

RIPETI FINCHE' PARTITA TERMINATA:

ESAMINA LA MOSSA AVVERSARIA,

CALCOLA LA PROSSIMA MOSSA,

EFFETTUA LA PROSSIMA MOSSA.

In questo modo, dal punto di vista del giocatore, le due macchine menzionate si possono considerare identiche.

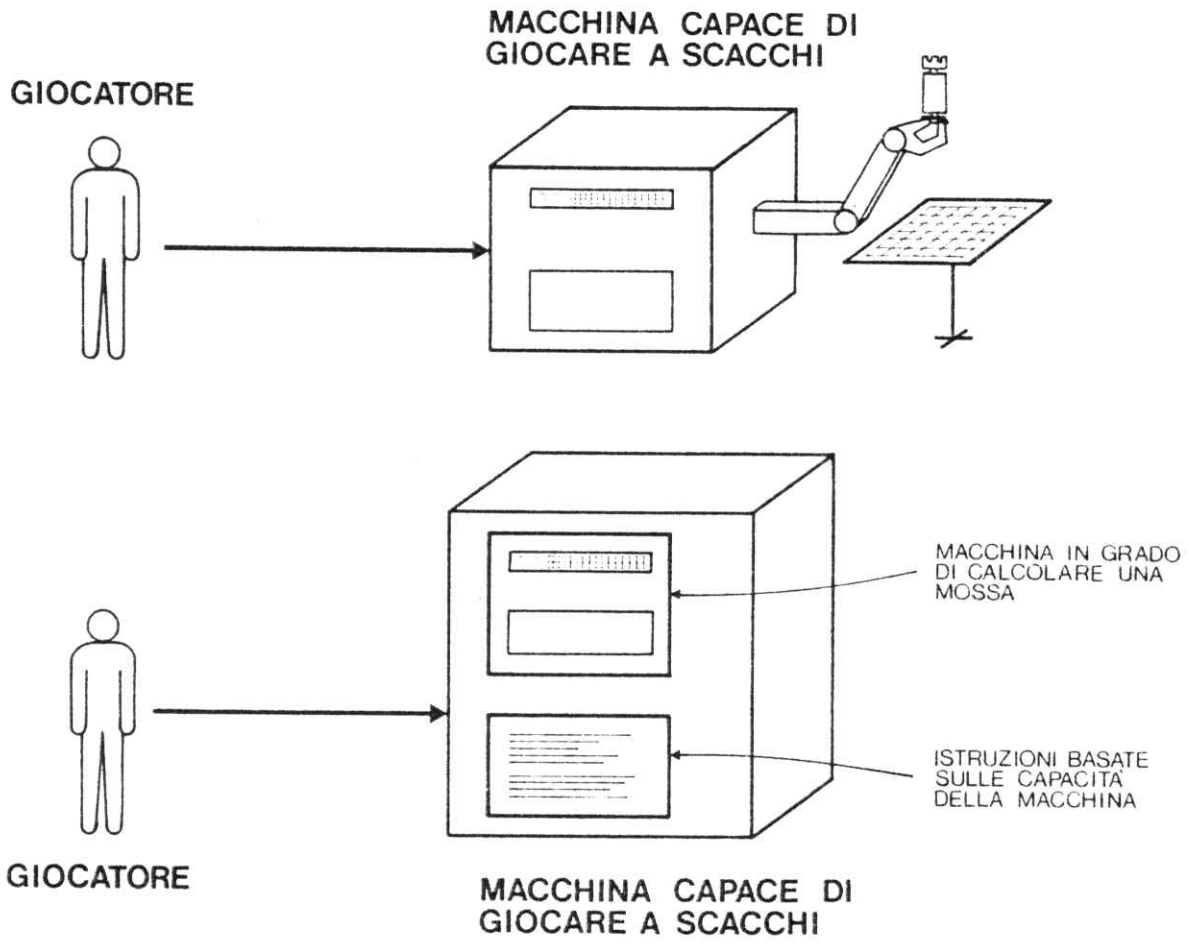


Fig. 3-1 Macchine Equivalenti

Piu' in generale si puo' dire che, avendo bisogno di una macchina che non esiste, la si puo' costruire con l'ausilio di macchine "base" e di un linguaggio che esprime il modo con cui esse devono essere connesse per eseguire l'operazione richiesta.

Naturalmente, ogni componente della macchina puo' essere costituito, in modo simile, da macchine di piu' basso livello.

Questo processo termina quando il livello delle macchine, che sono effettivamente a disposizione, viene raggiunto.

L'approccio descritto sopra costituisce il metodo fondamentale per la costruzione di un programma PASCAL+: esistono alcune caratteristiche del linguaggio PASCAL+ che costituiscono le "macchine" base, ovvero le risorse predefinite, ed altre che consentono la definizione di risorse ad alto livello (ad esempio le procedure e le funzioni).

Questo metodo e' conosciuto anche come programmazione top-down, metodo che consiste nell'analizzare il problema da risolvere in termini di sottoproblemi di minore complessita'.

Nel seguito del capitolo, sara' descritta piu' in dettaglio la struttura gerarchica di un programma PASCAL+, anche da un punto di vista sintattico.

LA STRUTTURA GERARCHICA DI UN PROGRAMMA

Come affermato precedentemente, un programma PASCAL+ e' costituito da risorse, sia fornite dal linguaggio stesso che definite dal programmatore.

Da un punto di vista strutturale il programma e' diviso in due parti: la prima descrive le risorse ad alto livello, la seconda comprende le istruzioni che utilizzano le risorse nel modo voluto.

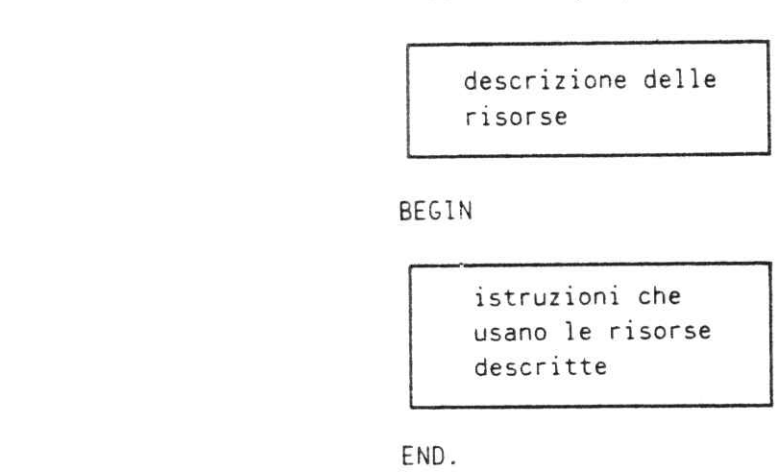


Fig. 3-2 Struttura a Grandi Linee di un Programma PASCAL+

Con il termine "risorsa" si intende definire ogni ente che faccia da supporto alla parte istruzioni, per esempio variabili, procedure e funzioni. Ciascun tipo di risorsa verra' trattata piu' avanti: tra di esse le procedure e le funzioni giocano un ruolo fondamentale poiche' costituiscono le "istruzioni" ad alto livello.

Poiche' le procedure e le funzioni possono essere specificate successivamente con l'uso di risorse di piu' basso livello, la loro struttura sara' simile a quella del programma stesso.

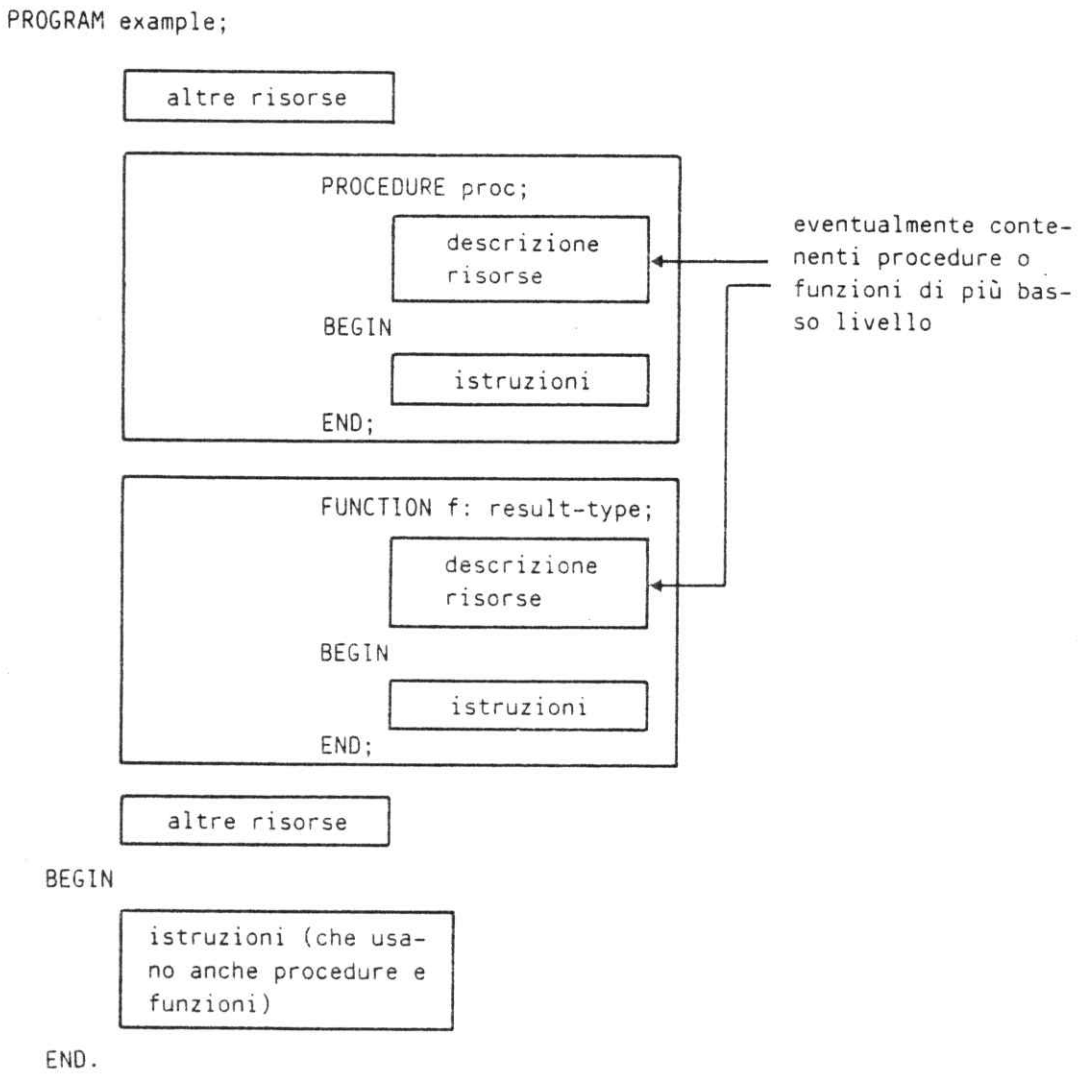


Fig. 3-3 Risorse Nidificate

Ogni procedura o funzione contiene la descrizione delle risorse di piu' basso livello ed una parte istruzioni che usa tali risorse.
Si noti che tra queste risorse di piu' basso livello ci possono essere altre procedure o funzioni strutturate, a loro volta, nello stesso modo.

La differenza tra procedure e funzioni verra' trattata nel Capitolo 13: per il momento si possono indicarle comunemente come "routine". Nella programmazione PASCAL+ e' fondamentale nidificare le risorse; nel

prossimo capitolo vengono spiegate alcune regole su tale argomento.

SEQUENZA DI SCRITTURA E DI COMPILAZIONE

Quando si scrive un programma PASCAL+ in termini di gerarchia di risorse, si inizia da quella piu' ad alto livello: cio' implica come primo passo la definizione della "parte istruzioni" che fa riferimento alle routine, ai dati ed alle altre risorse che saranno dettagliate successivamente. Questo modo di scrittura top-down e' suggerito dal linguaggio stesso.

Il compilatore PASCAL+, al contrario, scandisce solo una volta il programma, per tale motivo e' necessario che siano scritte le risorse di piu' basso livello prima che si faccia riferimento ad esse.
Quindi l'ordine temporale con cui si scrive un programma non coincide con l'ordine testuale: innanzitutto si scrive la parte istruzioni del programma principale (posta al di sotto del BEGIN) usando le variabili e le routine la cui definizione dettagliata viene specificata in seguito (situata al di sopra del BEGIN):

```
PROGRAM P;

BEGIN
  (* istruzioni del
  programma principale *)
END.
```

Poi si descrivono le risorse del secondo livello, per esempio le routine:

```
PROGRAM P;

  PROCEDURE Q;

  BEGIN
    (* istruzioni della
    procedura Q *)
  END;

  PROCEDURE R;

  BEGIN
    (* istruzioni della
    procedura R *)
  END;

BEGIN
  (* istruzioni del
  programma principale *)
END.
```

Ogni routine e' costruita con lo stesso criterio specificando prima la definizione delle risorse e poi la parte istruzioni:

```
PROGRAM P;

   PROCEDURE Q;
     (* descrizione risorse della procedura *)

   BEGIN
    (* istruzioni della procedura Q *)
   END;

   PROCEDURE R;
    (* descrizione risorse della procedura R *)

   BEGIN
     (* istruzioni della procedura R *)
   END;

  BEGIN
  (* istruzioni del programma principale *)
  END.
```

4. REGOLE DI VISIBILITA'

In questo capitolo e' mostrato l'uso, in un programma PASCAL+, delle risorse ai diversi livelli.

Vengono stabilite poche regole generali, chiamate "regole di visibilita'" (scope rules) relative a tutti i tipi di risorse, incluse quelle fornite dal linguaggio stesso, cioe' le risorse predefinite.

Inoltre, e' introdotto il concetto di procedure e funzioni ricorsive.

TERMINOLOGIA RELATIVA ALLA NIDIFICAZIONE DELLE RISORSE

Il raffinamento in modo top-down di un programma PASCAL+ richiede che si abbia una corretta visibilita' della nidificazione delle risorse.

Si deve considerare ogni risorsa come assegnata ad una specifica parte del programma. Per esempio, nello scrivere una routine per la ricerca di un valore in un array, si utilizzera' all'interno della routine una variabile indice per scandire l'array; questa variabile e' una risorsa privata della routine: sarebbe scorretto (e inutile) usarle all'esterno. La disciplina relativa all'uso delle risorse e' governata da poche e semplici regole dette "regole di visibilita'".

Poiche' tali regole sono basate sui livelli di nidificazione delle risorse, e' opportuno introdurre una semplice terminologia relativa a questo argomento.

Si consideri ora l'esempio in cui il programma P sia scritto in termini di due risorse: una variabile A ed una procedura Q.

```
PROGRAM P;
   VAR A: (* descrizione di A *);
   PROCEDURE Q;
          (* risorse di Q *)
   BEGIN
          (* istruzioni di Q *)
     END;

BEGIN
    (* istruzioni del programma principale *)
END.
```

Fig. 4-1 Risorse del Programma Principale

Poiche' A e Q sono scritte specificatamente per P, si dice che esse sono risorse "figlie" di P e P e' "padre" di A e Q; inoltre A e Q sono dette "fratelli".

La risorsa Q e' una routine e puo' contenere risorse proprie, ad esempio le due routine R e S:

```
PROGRAM P;
  VAR A: (* descrizione di A *);
  PROCEDURE Q;
    PROCEDURE R;
         (* risorse di R *)
    BEGIN
         (* istruzioni di R *)
    END;

    PROCEDURE S;
         (* risorse di S *)
    BEGIN
         (* istruzioni di S *)
    END;

  BEGIN
      (* istruzioni di Q *)
  END;
BEGIN
    (* istruzioni di P *)
END.
```

Fig. 4-2 Nidificazione di Risorse

Esistono quindi le seguenti relazioni: R ed S sono figlie di Q e P e' un antenato per esse in quanto padre del loro padre Q.

Si noti la particolare indentazione usata per le righe del testo: questo e' un modo utile e piuttosto diffuso per evidenziare la nidificazione di

risorse nel testo di un programma.

Quanto detto e' sufficiente per formulare le regole di visibilita' nei prossimi paragrafi.
Si hanno tre regole fondamentali:

- regola gerarchica
- regola sequenziale
- regola dell'ombra.

Ognuna di queste regole specifica alcune restrizioni circa la parte di testo del programma in cui una risorsa puo' essere usata; tale parte di testo e' detta "area di visibilita'" della risorsa.

ENUNCIATO DELLE REGOLE DI VISIBILITA'

La Regola Gerarchica

La regola gerarchica stabilisce che una risorsa puo' essere usata solo all'interno del proprio padre (inclusi i discendenti del padre), ma mai in una parte del testo esterna ad esso.

Con riferimento all'esempio precedente, in base a tale regola le procedure R ed S non possono essere utilizzate al di furori della procedura Q.

La Regola Sequenziale

La regola sequenziale afferma che una risorsa non puo' essere utilizzata in una parte del testo che precede la sua stessa definizione.

Nell'esempio precedente, la procedura R puo' essere utilizzata all'interno di S, mentre S non puo' apparire all'interno di R.

La variabile A puo' essere usata in Q, e percio' anche in R ed in S.

C'e' tuttavia un'eccezione riguardante la regola sequenziale, circa i tipi puntatori, tale argomento verra' trattato in seguito.

La Regola dell'Ombra

Precedentemente sono state descritte le relazioni tra risorse come relazioni all'interno di una famiglia.
Si consideri ora il problema dei nomi in una famiglia. Solitamente si cerca di non dare gli stessi nomi a piu' fratelli, cio' per evitare confusione nel chiamarli!
D'altro canto, il problema non sussiste se a due persone appartenenti a famiglie distinte sono dati gli stessi nomi, poiche' il contesto e' sufficiente per distinguerle.

Questo accade anche per gli identificatori di risorse del PASCAL+, ed e' con la regola dell'ombra che si elimina ogni possibile ambiguita'. Si aggiunga ora all'esempio precedente una variabile "A" all'interno della procedura R:

```
PROGRAM P;
   VAR A: (* descrizione di A *);
   PROCEDURE Q;
      PROCEDURE R;
        VAR A: (* altra descrizione di A diversa dalla precente *);
      BEGIN
         (* istruzioni di R *)
      END;
      PROCEDURE S;
          (* risorse di S *)
      BEGIN
          (* istruzioni di S *)
      END;
   BEGIN
      (* istruzioni di Q *)
   END;
BEGIN
  (* istruzioni di P *)
END.
```

Fig. 4-3 La Regola dell'Ombra

Queste due risorse sono completamente distinte, sebbene abbiamo lo stesso nome.

Normalmente la variabile globale A, (le risorse figlie del programma principale sono dette "globali"), sarebbe accessibile in R.
Pero', la presenza di un'altra risorsa con lo stesso nome all'interno di R "fa ombra" alla sua esistenza nella parte di R che segue la nuova definizione di A. Si tratta, comunque di una combinazione relativa solo ad R: la variabile globale A sara' dunque accessibile in S e nelle parti rimanenti di Q e P.

Quindi in base alla regola dell'ombra una risorsa non puo' essere utilizzata tutte le volte che e' accessibile una variabile piu' interna con lo stesso nome.

ROUTINE RICORSIVE

Molti problemi di programmazione sono facilmente risolvibili con l'impiego di algoritmi ricorsivi. Questo argomento non verra' trattato da un punto di vista matematico (se necessario si puo' consultare qualche testo di informatica di base), comunque si osservi solo come ogni routine PASCAL+ (procedura o funzione) possa essere ricorsiva, cioe' contenere al proprio interno, nella parte istruzioni, una chiamata a se stessa.

Non vi e' alcuna limitazione nelle chiamate di routine purche' siano rispettate le regole di visibilita'; cosicche' una routine puo'

eventualmente chiamare anche le routine del proprio padre (se stessa compresa) ed ogni altra routine antenata.

C'e' tuttavia il problema della mutua ricorsivita', che si presenta quando due routine dello stesso livello si richiamano l'un l'altra ricorsivamente, per cui una delle due puo' essere richiamata prima ancora di essere definita. Questo problema viene risolto per mezzo della direttiva FORWARD.

```
PROGRAM P;
   PROCEDURE Q;
        FORWARD;
   PROCEDURE S;
         (* risorse di S *)
   BEGIN
       (* istruzioni di S *)
       (* chiamata di Q *)
   END;
   PROCEDURE Q;
        (* risorse di Q *)
   BEGIN
        (* istruzioni di Q *)
        (* chiamata di S *)
   END;
BEGIN
    (* istruzioni di P *)
    (* chiamata di Q o S, o di entrambe *)
END.
```

Fig. 4-4 Routine Mutuamente Ricorsive

La procedura Q e' citata per la prima volta in una definizione dove compare la direttiva FORWARD anziche' la descrizione per esteso di Q: cioe' permette di compilare correttamente la chiamata di Q che compare all'interno di S.
La descrizione per esteso di Q verra' compilata dopo quella di S.

RISORSE PREDEFINITE

Si accenna, ora, brevemente alle risorse utilizzabili in un programma.

Il linguaggio PASCAL+, di per se', fornisce un certo numero di risorse predefinite.

Si consideri l'esempio:

```
PROGRAM P;
    VAR I: INTEGER;
  BEGIN
    WRITE (I)
  END.
```

Fig. 4-5 Risorse Predefinite

WRITE costituisce una procedura predefinita (descritta piu' avanti), cioe' e' una risorsa che puo' essere usata in un programma senza la necessita' di una sua precedente definizione. Lo stesso vale per l'identificatore INTEGER usato nella definizione della variabile "I", INTEGER e' il nome della risorsa predefinita che specifica il tipo di valore che I puo' assumere.

Se si immagina un ulteriore ambiente che comprenda il programma stesso, l'uso di risorse predefinite risulta in accordo con le regole di visibilita'.
Tale ambiente e' il sistema PASCAL+ avente le definizioni di tali risorse, tipo WRITE o INTEGER.
Tutte queste sono risorse allo stesso livello e precedono il programma, in modo da poter essere utilizzate in tutto il testo del programma stesso, a meno di essere 'ombreggiate' da risorse piu' interne con lo stesso nome.

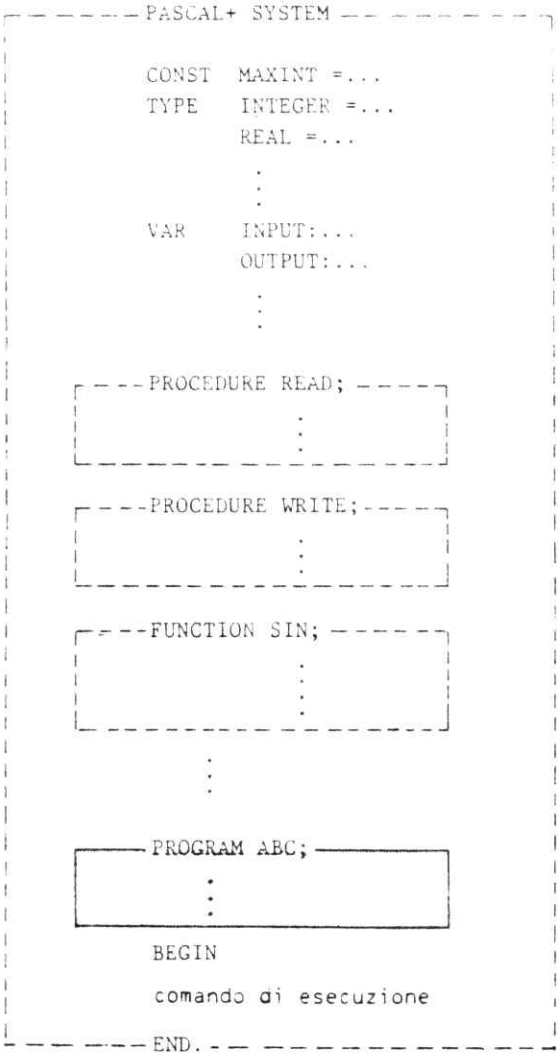


Fig. 4-6 Risorse ed Istruzioni Predefinite

Nel Capitolo 2 sono stati elencati tutti gli identificatori (identificatori predefiniti) corrispondenti alle risorse predefinite. In ogni caso e' consigliabile cercare di evitare di 'ombreggiare' tali risorse.

5. IL CONCETTO DI TIPO

In questo capitolo viene considerata un'importante caratteristica di tutti i linguaggi ad alto livello: il concetto di tipo di dato.

Si introduce inoltre una classificazione sommaria dei tipi, distinguendoli in semplici e strutturati, oltre ai piu' importanti strumenti linguistici per la loro definizione: i "costruttori" e gli "enumeratori".

COS'E' UN TIPO

E' noto che i programmi non operano direttamente sul mondo reale, ma su una sua rappresentazione simbolica. Si consideri l'esempio di un programma che gioca a scacchi, che ovviamente non opera su una scacchiera reale ma su una sua rappresentazione, localizzata fisicamente nella memoria del calcolatore.

Quindi, per scrivere un programma, si deve analizzare il problema e scegliere la migliore rappresentazione delle entita' coinvolte (le risorse). La migliore rappresentazione e' tuttavia quella piu' vicina al problema reale. Ad esempio nel caso della scacchiera, non conviene considerare 64 variabili che contengono numeri interi, ma e' certamente meglio analizzare una tabella 8 x 8 costituita da "caselle" ognuna delle quali assume uno dei seguenti valori: "vuoto", "pedone", "alfiere", "regina", ecc.

Il linguaggio PASCAL+ permette la stesura di programmi le cui entita' sono espresse negli stessi "termini" del mondo reale, ovvero, invece di modellare il problema in conformita' del programma, si puo' modellare il programma in conformita' del problema.

In PASCAL+ cio' e' possibile in quanto le variabili non sono viste come contenitori di numeri, quanto come contenitori di valori di ogni tipo.

Una variabile deve essere considerata come un oggetto contenente alcuni valori definiti nel programma e denotati da identificatori, oltre che da numeri o stringhe di caratteri.

L'insieme di tutti i valori che una variabile puo' assumere e' detto il "tipo" di quella variabile.

Percio', definire un tipo significa definire un insieme di valori.

In riferimento all'esempio precedente si puo' definire il tipo "casella" elencando i suoi valori: "re", "regina", "alfiere", "cavallo", "torre", "pedone", "vuoto".

Una variabile di tipo "casella" puo' assumere solo uno dei precedenti

valori: non ha senso, naturalmente, l'assegnazione di un valore diverso.

Inoltre, per il tipo "scacchiera", l'insieme di valori di questo tipo e' costituito dall'insieme di tutte le possibili combinazioni dei 64 valori delle sue caselle.
Ovviamente, per definire questo tipo non conviene elencare tutti i possibili valori, ma usare una definizione piu' coincisa: il tipo "scacchiera" e' una matrice 8 x 8 di elementi di tipo "casella".

Spesso e' necessario definire alcune "operazioni" associate al tipo in considerazione.

In generale la definizione di un tipo implica quella di alcune operazioni che agiscono sui suoi valori.

Per esempio, si puo' considerare il tipo INTEGER come l'insieme dei valori il cui limite inferiore e' - infinito e quello superiore e' + infinito; inoltre tale insieme e' fornito di un insieme di operazioni come la somma, il prodotto, la divisione, e cosi' via.

Come altro esempio, si consideri il tipo "colore" come l'insieme dei valori: "bianco", "giallo", "arancione", "rosso", "blu", "viola", "nero"; e possiamo immaginare di dotarlo dell'insieme di operazioni:

"piu'" cosi' definita:

```
PIU (ROSSO, BLU) = VIOLA
PIU (GIALLO, ROSSO) = ARANCIONE
  .
  .
```

"inverso" cosi' definita:

```
INVERSO (NERO) = BIANCO
INVERSO (GIALLO) = VIOLA
  .
  .
```

"tonalita'" definita come:

```
TONALITA (BIANCO) = 0
TONALITA (GIALLO) = 1
  .
  .
TONALITA (NERO) = 6
```

Una variabile di tipo colore puo' assumere solo uno dei valori elencati in precedenza, sui quali si puo' agire tramite le operazioni elencate precedentemente.
Si consideri ora una parte di un programma PASCAL+ che usa questo tipo:

```
TYPE COLORE = (BIANCO, GIALLO, ARANCIONE, ROSSO, BLU, VIOLA, NERO);
VAR X,Y : COLORE;
      I : INTEGER;
BEGIN
.
.
X:=ROSSO;
Y:=INVERSO(X);
I:=TONALITA  (PIU (Y, BIANCO));
.
.
END.
```

Gli identificatori usati X, Y I sono nomi di variabili, infatti appaiono dopo la parola chiabe "VAR". Essi possono essere usati in una istruzione di assegnazione (X:=...) oppure in una espressione (...:=INVERSO(X)).

BIANCO, GIALLO,...,NERO sono nomi di costanti, o meglio, nomi di valori del tipo colore: si deve evitare di usarli in un'istruzione di assegnazione, ma possono essere usati in un'espressione (...:=ROSSO).

COLORE ed INTEGER sono nomi di tipi: non si devono usare nella parte istruzioni, ma solo nella parte di descrizione delle risorse (VAR...:COLORE). Qualsiasi altro uso di questi identificatori diverso da quello descritto sarebbe privo di senso.

Nei successivi due paragrafi viene introdotta una classificazione di tipi, distinguendo i concetti di tipo "semplice" e tipo "strutturato".

TIPI SEMPLICI

Un tipo semplice e' un insieme di valori ognuno dei quali e' considerato un elemento informativo elementare ed indivisibile.

Ogni valore e' "costante" nel suo tipo ed e' denotato da un identificatore, da un numero o da una stringa, in relazione al modo in cui il tipo e' stato definito.

Esistono due strumenti linguistici basilari per la definizione di tipo semplice:

- enumerazione, che consiste nell'elencare tutti i valori di un tipo nuovo definito dall'utente;

- "subrange", che consiste nello specificare il primo e l'ultimo valore di un sottoinsieme compatto di un tipo semplice definito in precedenza.

Esempi di tipi enumerati:

```
TYPE CASELLA        =(VUOTO, PEDONE, CAVALLO, ALFIERE, TORRE, REGINA,
                      RE);

     COLORE         =(BIANCO, GIALLO, ARANCIONE, ROSSO, BLU, VIOLA,
                      NERO);

     MISURA         =(TEMPERATURA, PRESSIONE, UMIDITA', FUMOSITA');

     GIORNO         =(LUNEDI', MARTEDI', MERCOLEDI', GIOVEDI', VENERDI',
                      SABATO, DOMENICA);

     SEME           =(PICCHE, FIORI, QUADRI, CUORI);

     CIFRE_ROMANE   =(I, V, X, L, C, D, M);
```

Esempi di tipi "subrange":

```
TYPE COLORE_CHIARO  = BIANCO..ARANCIONE;

     GIORNI_FERIALI = LUNEDI'..VENERDI';

     SEMI_ROSSI     = QUADRI..CUORI;

     INDICE         = 1..100;

     LETTERA        = "A".."Z";
```

Inoltre, si hanno alcuni particolari tipi semplici propri del linguagggio, che sono risorse predefinite, indicate con i seguenti identificatori:

INTEGER	indica l'isieme dei numeri interi
REAL	indica l'insieme dei numeri reali
BOOLEAN	indica l'insieme dei valori: "falso" e "vero"
CHAR	indica l'insieme dei caratteri (dipendente dall'implementazione)

Da un punto di vista concettuale questi tipi non differiscono da quelli che l'utente puo' definire con l'enumerazione, poiche' essi sono formati da insiemi ordinati di valori, denotati non solo da identificatori, ma anche in altro modo (numeri, stringhe). Inoltre, l'insieme di valori definiti dai tipi INTEGER, REAL e CHAR dipendono dalla particolare implementazione. Il significato completo di questi tipi viene illustrato in dettaglio nel Capitolo 8 in "TIPI SEMPLICI".

Le operazioni applicabili ad un tipo definito da programma devono essere definite utilizzando le routine, mentre si hanno alcune operazioni applicabili ai tipi predefiniti che sono predefinite (+, -, *, /, EXP, LN, ...).

Inoltre vi sono alcune operazioni predefinite per tutte le varieta' di tipi semplici, anche definite da programma, come gli operatori

relazionali (=, >, <, >=, ...), successore e predecessore (SUCC, PRED), e cosi' via.

TIPI STRUTTURATI

Un tipo strutturato e' un insieme di valori ognuno dei quali costituito da un aggregato di valori di altri tipi.

Un tipo strutturato e' caratterizzato dai tipi dei valori che lo compongono, chiamati tipi base, e da un metodo di strutturazione.

Non e' possibile definire un tipo strutturato con l'enumerazione dei suoi valori, sia perche' non ci sono identificatori che denotano i valori, sia perche' spesso la cardinalita' e' elevata.

I valori strutturati sono trattati per lo piu' a livello di componente: il riferimento ai componenti e' effettuato attraverso le operazioni associate al tipo, dette "selezioni".

Uno strumento linguistico in base alla definizione sia del metodo di strutturazione che dell'operazione di selezione, e' detto "costruttore": esso pero' non definisce completamente il tipo strutturato, poiche' non specifica il tipo base.
Un costruttore puo' essere visto come un tipo strutturato, i cui tipi base sono parametrizzati e dotati di un'operazione di selezione.

Si considerino, ora, alcuni esempi di costruttori.
Dapprima, si osservi come puo' essere rappresentata in un programma PASCAL+, una carta di identita'. L'intera informazione e' suddivisa in un numero piu' dettagliato (e piu' elementare) di porzioni di informazione: nome, cognome, data e luogo di nascita. L'ordine di queste porzioni di informazione all'interno della carta d'identita' non e' importante; ognuna e' selezionata individualmente attraverso un identificatore: "nome", "luogo di nascita",...

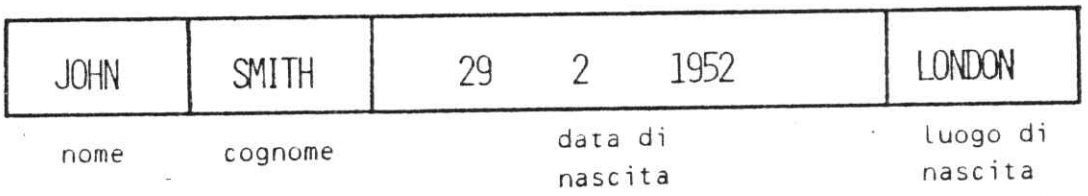


Fig. 5-1 Rappresentazione Grafica di una Carta di Identita'

In PASCAL+, questo metodo di strutturazione e di operazione di selezione costituisce il costrutture RECORD, in cui i tipi base, detti campi, sono i tipi di ciascuna porzione di informazione.
Il costruttore RECORD puo' essere utilizzato per definire il tipo

strutturato "CARTA_IDENTITA", descritto nel modo seguente:

```
TYPE CARTA_IDENTITA = RECORD
                      NOME : STRINGA;
                      COGNOME : STRINGA;
                      DATA_NASCITA : RECORD
                                        GIORNO : 1..31;
                                        MESE   : 1..12;
                                        ANNO   : INTEGER
                                     END;
                      LUOGO_NASCITA : STRINGA
                      END;
```

Naturalmente si puo' dire che l'insieme dei valori del tipo CARTA_IDENTITA e' dato dalla combinazione di tutti i possibili valori (cioe' tipi) di ogni porzione di informazione.
Si noti che il campo DATA_NASCITA e' a sua volta un tipo record.

Come secondo esempio si vuol rappresentare con un tipo PASCAL+ l'insieme dei valori che puo' assumere una scacchiera.
Si tratta di una matrice di 8 x 8 caselle, ognuna delle quali puo' contenere un pezzo di un giocatore. Inoltre si vuole selezionare ciascuna casella attraverso una coppia di coordinate, ad esempio "C7" o in modo simile.

In altri termini, si ha un certo numero di valori componenti, tutti dello stesso tipo (CASELLA) il cui ordine riveste una certa importanza ed ogni valore e' selezionato tramite indici: il costrutture che denota tale metodo di strutturazione e tale operazione di selezione e' detto ARRAY.
Quindi si puo' definire il tipo strutturato "SCACCHIERA" specificando il tipo base nel modo seguente:

```
TYPE SCACCHIERA = ARRAY [1..8, 'A'..'H'] OF CASELLA
```

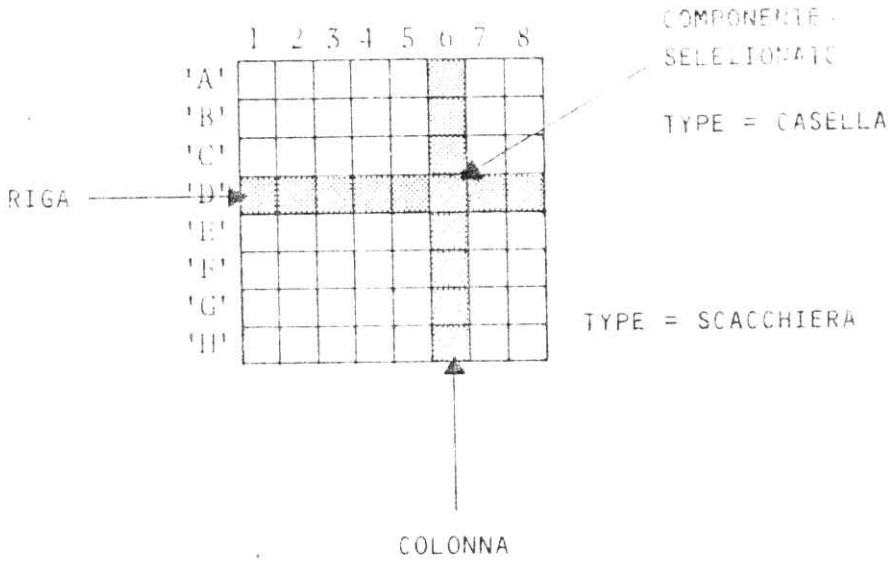


Fig. 5-2 Rappresentazione del Tipo Strutturato SCACCHIERA Definito Tramite il Costruttore ARRAY

Come terzo esempio si consideri la rappresentazione di un mazzo di carte durante una partita di poker. Esso e' costituito dalla sequenza di un numero non fissato di carte, dove solo la superiore puo' essere estratta e sono permessi possibili inserimenti solo dall'ultima.

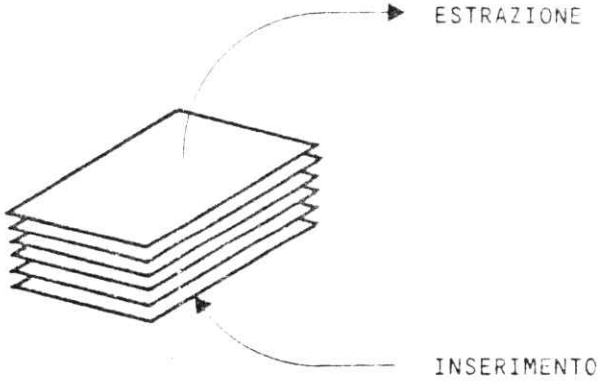


Fig. 5-3 Un Mazzo di Carte in una Partita di Poker

Questo medoto di strutturazione ed operazione di selezione e' detto costruttore QUEUE (coda) le operazioni di selezione sono solitamente dette ENQUEUE e DEQUEUE.
Il tipo strutturato MAZZO_CARTE puo' essere definito specificando il tipo base CARTA.

Un costruttore simile e' lo STACK. Esso differisce dalla coda poiche' sia l'inserimento che l'estrazione si effettuano nella stessa direzione, queste operazioni di selezione sono comunemente dette PUSH e POP.

Un esempio finale puo' essere la rappresentazione di un libro, esso

infatti e' costituito da un numero non definito ed ordinato di pagine, scritte e/o lette sequenzialmente a partire dalla prima pagina.
Il costruttore che permette la definizione di tale tipo strutturato e' detto FILE e la definizione del tipo LIBRO sara' effettuata nel modo seguente:

```
TYPE PAGINA = ...;
     LIBRO = FILE OF PAGINA;
```

Si noti che il tipo base puo' essere a sua volta un tipo strutturato: il tipo PAGINA, ad esempio, puo' essere definito come un array di righe.

```
TYPE RIGHE = ... (* eventualmente strutturato *);
     PAGINA = ARRAY [1..30] OF RIGHE;
     LIBRO = FILE OF PAGINA;
```

I costruttori piu' comunemente usati sono predefiniti in PASCAL+, ovvero l'array, il record, il file, ecc. (vedere in Capitolo 8, TIPI STRUTTURATI).

Tuttavia, costruttori particolari (stack, code, alberi,...) possono essere definiti da programma attraverso le librerie di risorse (vedere in Capitolo 7, Tipi di Dati Incapsulati).

6. ESECUZIONE DI PROGRAMMI E GESTIONE DELLA MEMORIA

In questo capitolo e' illustrato come eseguire un programma in PASCAL+; vengono trattate solo le "call" di routine che costituiscono delle chiamate a macchine astratte, non considerando ne' la sintassi ne' il formalismo di tali chiamate.
Tuttavia, poiche' le chiamate di routine implicano l'allocazione di memoria, e' necessario indicare le tecniche di gestione della memoria da PASCAL+, cioe' l'allocazione automatica nel Run-Time-Stack e l'allocazione manuale nello Heap.

L'EFFETTO DI UNA CHIAMATA DI ROUTINE

La chiamata di una routine causa il trasferimento del controllo a tale routine; al termine della sua esecuzione, il controllo torna alla istruzione che segue quella di chiamata.

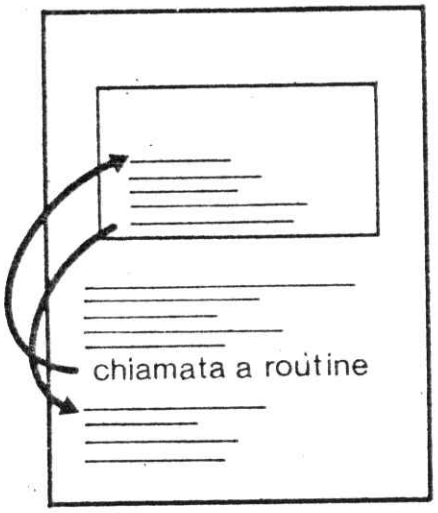


Fig. 6-1 L'Ordine di Esecuzione

Tuttavia, la chiamata di una routine non implica solo il trasferimento del controllo, ma anche la chiamata ad una macchina astratta costituita da risorse e da istruzioni.

In particolare, alcune delle risorse possono essere delle variabili. In base alle regole di visibilita', tali variabili non sono visibili al momento della chiamata, per cui non e' necessaria la loro allocazione nella memoria del calcolatore.
Pero' esse devono, naturalmente, essere disponibili durante l'esecuzione della routine, pertanto la loro allocazione avviene mentre la routine e' in esecuzione. Quando invece la routine e' stata eseguita, queste

variabili vengono deallocate, in quanto non visibili dall'ambiente esterno.

Riassumendo, le variabili interne di una routine vengono allocate dinamicamente e rimangono in memoria solo durante l'esecuzione della routine. Cio' permette una efficiente gestione della memoria in quanto vengono allocate solo le variabili necessarie, e la stessa area puo' essere usata, in tempi diversi, per le altre variabili delle varie routine chiamate.

IL RUN TIME STACK

L'area di memoria riservata alle variabili di un programma PASCAL+ e' detta Run Time Stack (RTS). Essa e' logicamente organizzata come uno stack, poiche' l'allocazione e la deallocazione agiscono come gli operatori "push" e "pop".

Viene illustrato ora un esempio di esecuzione di programma.

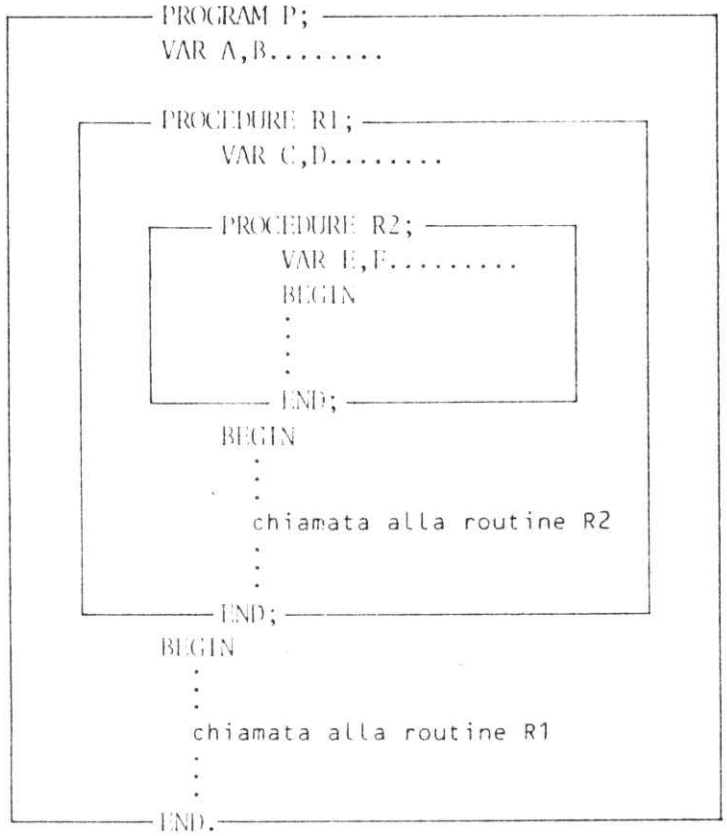


Fig. 6-2 Esempio di Allocazione delle Variabili

Prima di iniziare l'esecuzione del programma l'RTS e' vuoto; mentre si eseguono le istruzioni di P (il programma principale), l'RTS contiene solo le variabili A e B; quando si esegue una chiamata ad R1, vengono allocate le sue variabili C e D; analogamente per le chiamate di R2.

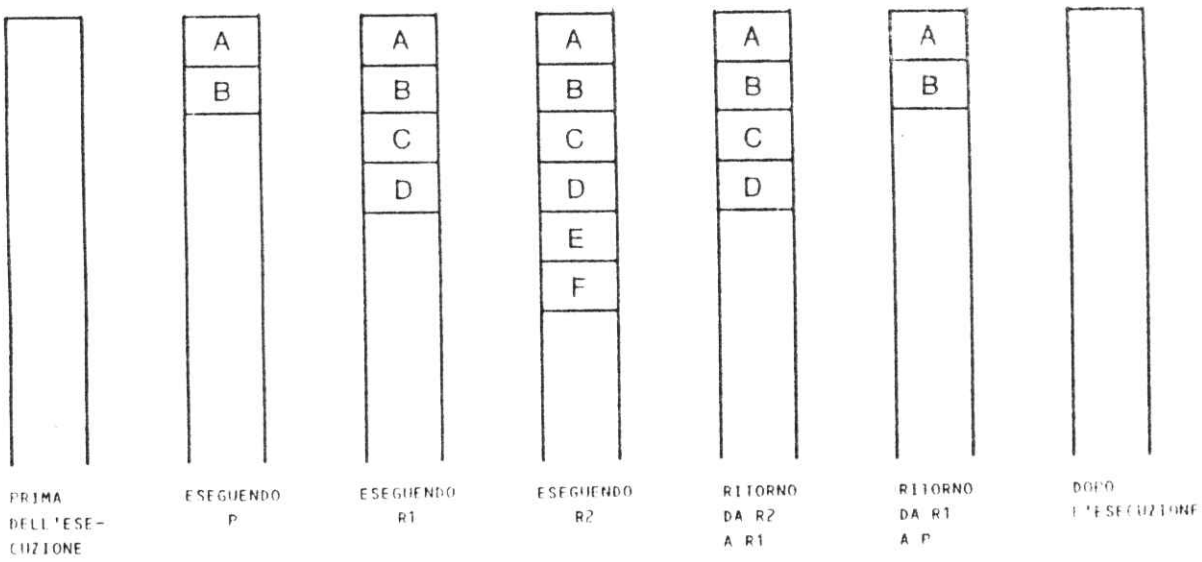


Fig. 6-3 Evoluzione Temporale del Run Time Stack

Inoltre, ciascun rientro da routine provoca la deallocazione delle variabili ad essa relative.

Si consideri, ora, il caso di routine "fratelli" chiamate sequenzialmente dalla routine "padre".

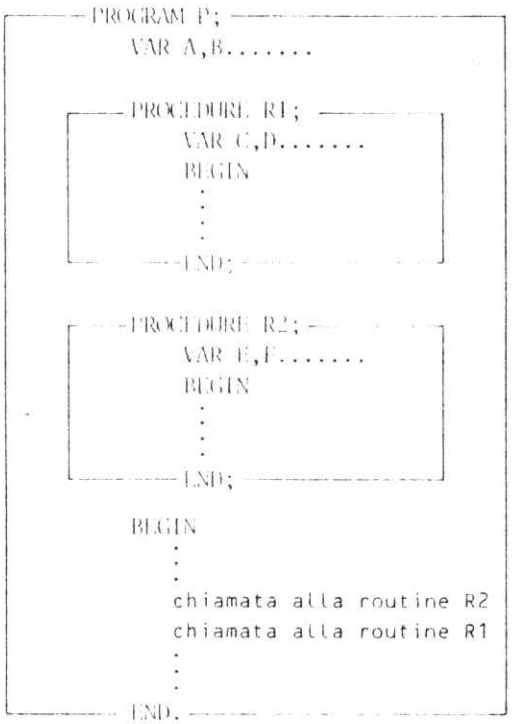


Fig. 6-4 Esempio di Allocazione delle Variabili

In questo caso, le variabili C e D di R1 sono allocate nella stessa area in cui erano precedentemente allocate le vatiabili E ed F di R2.

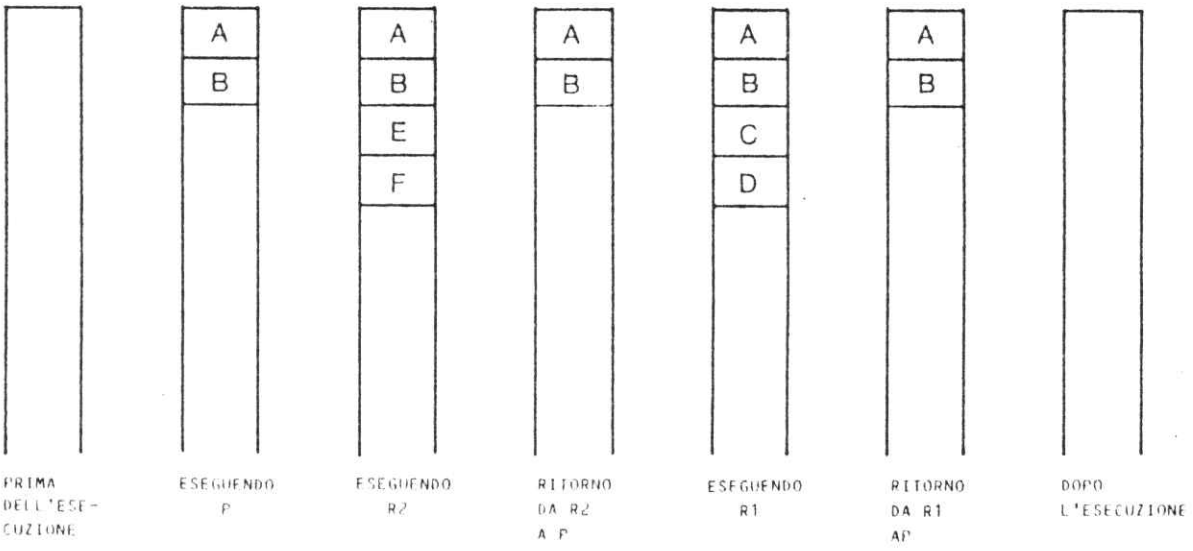


Fig. 6-5 Evoluzione Temporale del Run Time Stack

Nei due esempi appena visti si puo' notare che le due variabili globali (A e B) sono sempre allocate durante l'esecuzione del programma, mentre prima e dopo l'esecuzione non e' allocata alcuna variabile. Cio'

significa che un programma PASCAL+ non puo' comunicare con l'ambiente esterno tramite le sue variabili (si ricordi che anche l'intero programma deve essere visto come una "scatola nera").

In PASCAL+, le tecniche di input-output sono basate su un particolare tipo di variabile chiamata FILE, che non viene allocata nell'RTS. (vedere Capitolo 8, Il Costruttore FILE).

Infine, si consideri il meccanismo di allocazione nel caso di routine ricorsive; ad esempio:

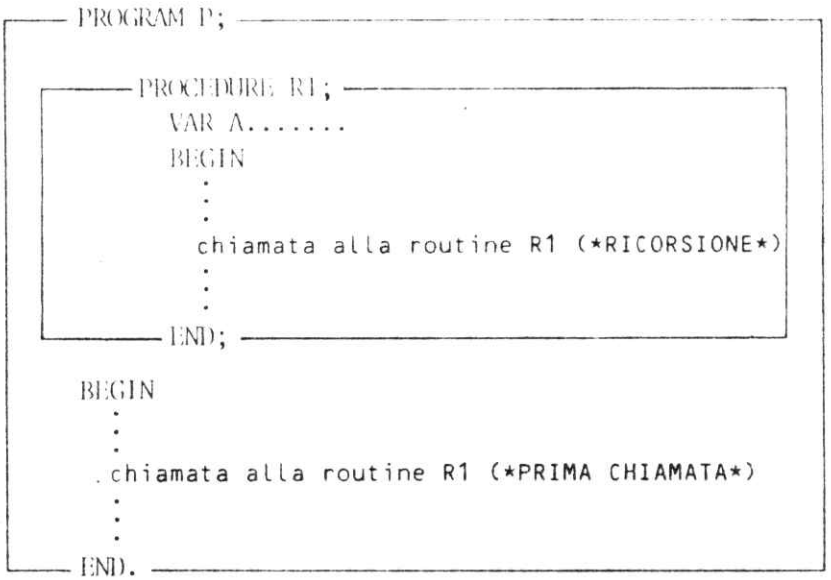


Fig. 6-6 Una Routine Ricorsiva

In tal caso viene allocata una nuova copia di A tutte le volte che viene eseguita una chiamata ad R1 e deallocata dopo ogni ritorno da essa. Si supponga che R1 chiami se stessa due volte ricorsivamente; l'evoluzione dell'RTS sarebbe la seguente:

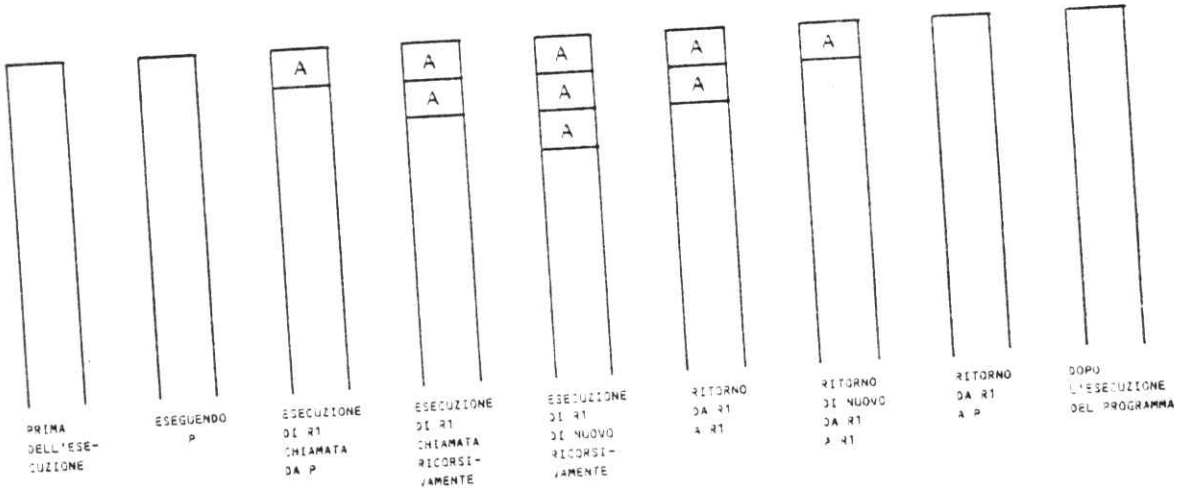


Fig. 6-7 Evoluzione Temporale dell'Area RTS durante l'Esecuzione di una Routine Ricorsiva

Si noti che la stessa variabile A e' allocata piu' volte. Questo non crea ambiguita', poiche' ogni chiamata di una routine ricorsiva implica l'allocazione di una copia privata delle proprie variabili.

LO HEAP

La tecnica di allocazione vista precedentemente, sebbene sia dinamica, non e' direttamente controllata tramite le istruzioni del programma, poiche' le variabili sono allocate e deallocate automaticamente in seguito alle chiamate di routine.

Il linguaggio PASCAL+ fornisce un altro tipo di variabili dinamiche, allocate indipendentemente dalle chiamate di routine con l'esecuzione di istruzioni specifiche.
Tali variabili non possono essere allocate nell'RTS, in quanto non devono sentire l'effetto delle chiamate di routine; quindi vengono allocate in una diversa area di memoria detta HEAP. Inoltre, non ci si puo' riferire ad esse direttamente, ma solo attraverso uno speciale tipo di variabile allocata nell'RTS detta "pointer" (vedere Capitolo 8 Tipi Puntatore).

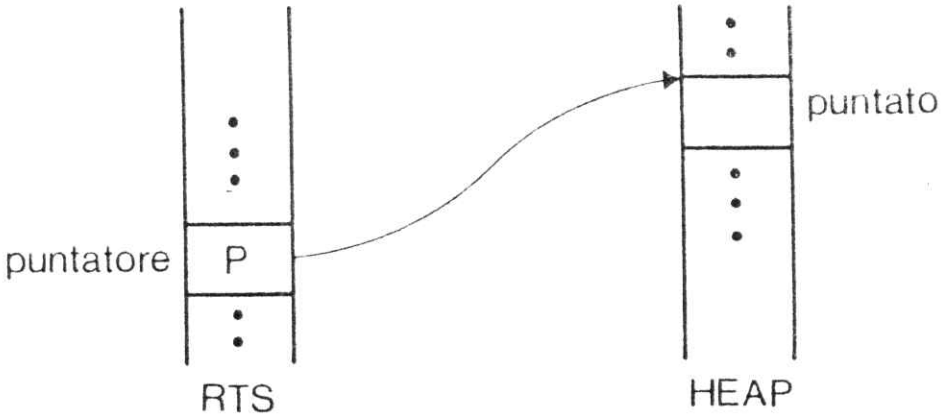


Fig. 6-8 Run Time Stack e Heap

Durante l'esecuzione del programma, e' possibile allocare nuove variabili nello heap usando un'apposita istruzione ("New", vedere in Capitolo 13 Procedure Predefinite per l'Allocazione Dinamica); tale istruzione puo' essere eseguita indifferentemente all'interno o all'esterno di una routine.

Questo tipo di variabili dinamiche permette la costruzione di strutture di dati piu' complesse, come liste, alberi, code, ecc.

Sebbene le variabili dinamiche possono rimanere nello heap anche dopo la fine dell'esecuzione del programma, non possono essere utilizzate per l'input-output, poiche' dopo la deallocazione del puntatore dal RTS, non sono piu' accessibili.

7. COMPILAZIONI SEPARATE

Nei capitoli precedenti si e' sempre considerato un programma come un'unita' compilabile ed eseguibile in modo autonomo. La manutenzione di un grosso programma risulta facilitata da compilazioni separate di alcune delle sue risorse, cosi' che la modifica di una di esse non richiede la ricompilazione dell'intero programma.

Inoltre, tali risorse esterne possono essere utilizzate da diversi programmi semplicemente "importandole" attraverso speciali parole chiave, senza richiedere l'inclusione del loro testo.

LIBRERIE DI RISORSE

COS'E' UNA LIBRERIA DI RISORSE

Un programma PASCAL+ puo' essere costruito separando fisicamente la descrizione di alcune delle sue risorse.

Le risorse descritte separatamente costituiscono la "libreria di risorse".

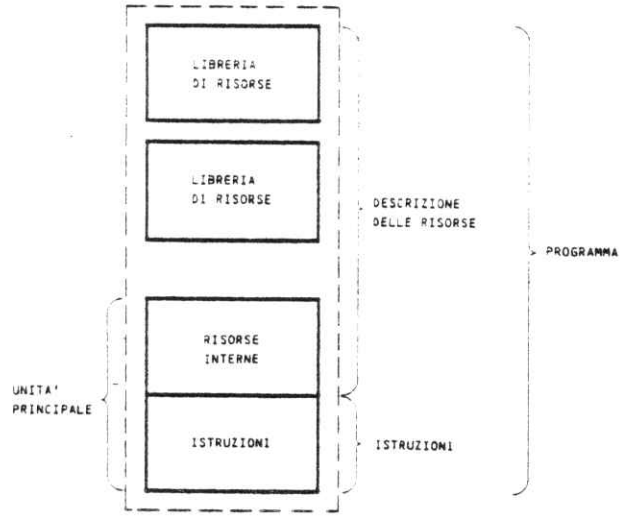


Fig. 7-1 Un Programma PASCAL+

Una libreria di risorse e' strutturalmente identica alla parte descrizione delle risorse di un programma PASCAL+: puo' contenere la descrizione di alcune costanti, tipi, variabili, procedure e funzioni.

Viene denotata con la parola chiave MODULE e non contiene la parte istruzioni.

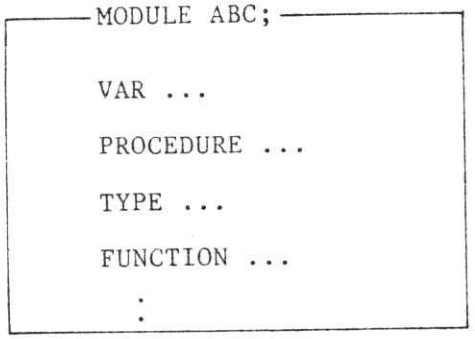


Fig. 7-2 Una Libreria di Risorse

L'"unita' principale" sara' usata per l'unita' contenente le istruzioni. Quindi un programma PASCAL+ e' costituito da una raccolta di unita' compilate che contengono alcune "librerie di risorse" ed una e una sola "unita' principale".

Le librerie di risorse possono essere compilate separatamente, ma naturalmente, non sono esequibili, devono essere linkate ad una unita' principale che le "importi".

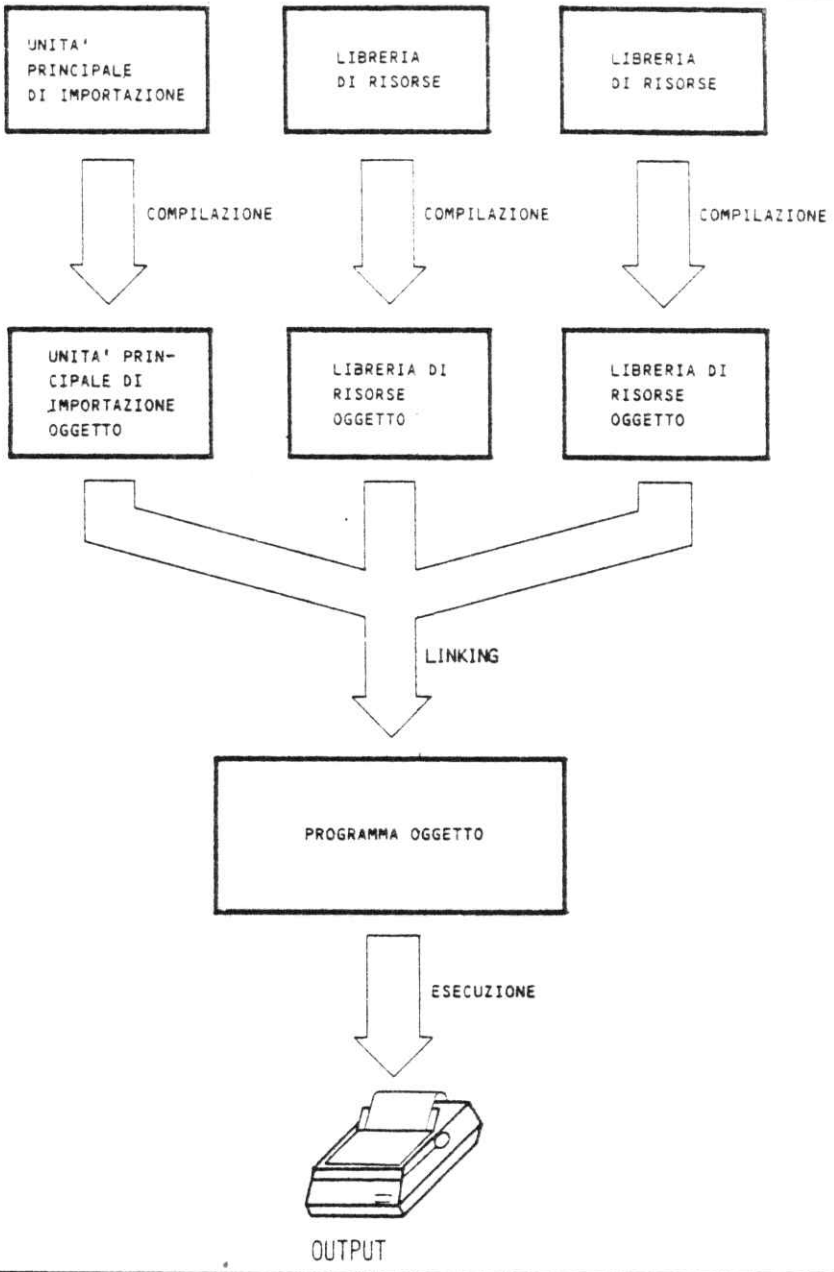


Fig. 7-3 Compilazione, Linking ed Esecuzione di un Programma PASCAL+

La stessa libreria di risorse puo' essere linkata con piu' unita' principali di importazione:

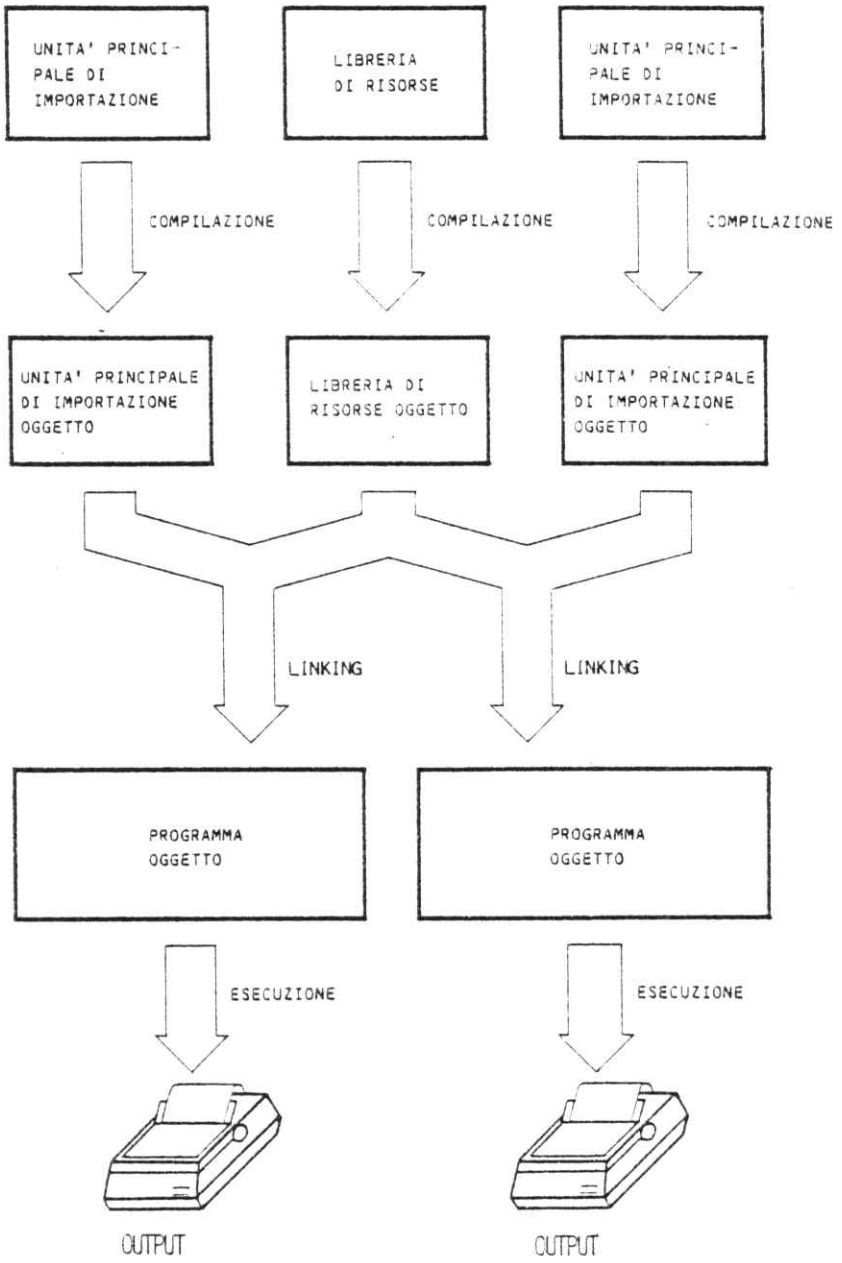


Fig. 7-4 Linking di una Libreria di Risorse con due Unita' Principali

Si considerano ora le regole di visibilita' per le librerie di risorse: un'unita' principale ed una libreria di risorse sono "fratelli", quindi ciascuna di esse non puo' usare le risorse interne dell'altra.
Cio' nonostante, alcune risorse della libreria possono essere viste dall'unita' principale se sono esplicitamente "esportate" dalla libreria ed "importate" nell'unita' principale; e viceversa.

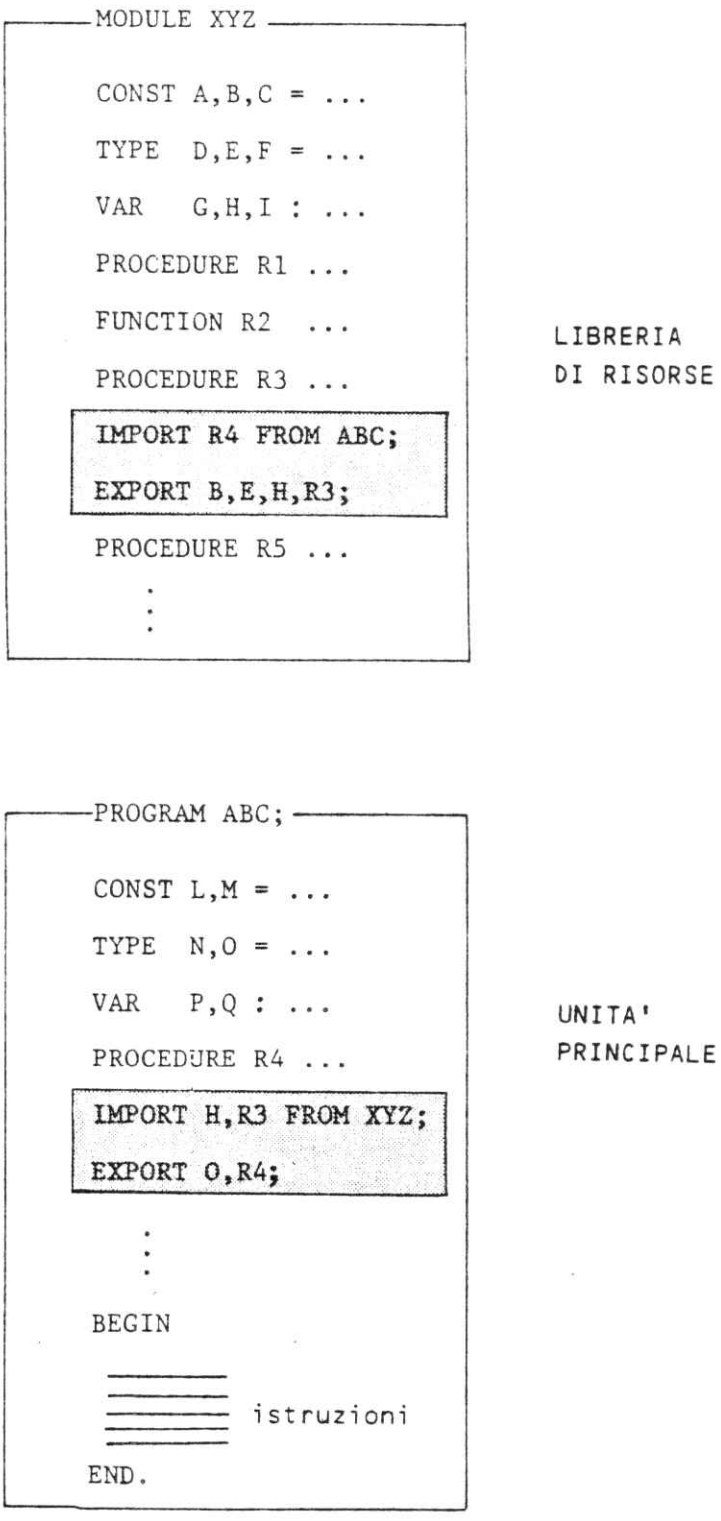


Fig. 7-5 Il Meccanismo di Import/Export

Nell'esempio precedente le istruzioni dell'unita' principale possono chiamare la routine R3 oppure usare la variabile H poiche', sebbene descritte nella libreria di risorse, esse sono esportate da XYZ ed

importate in ABC. La procedura R3 della libreria di risorse puo' chiamare R4 poiche' e' importata da ABC.
L'associazione tra risorse importate ed esportate e' basata sulla corrispondenza tra i loro nomi.

Tutte le risorse non esportate sono completamente nascoste all'ambiente esterno e si possono considerare protette da accessi illegali.

Si riconoscono due usi principali delle librerie di risorse:

- routine separate;

- tipi di dati incapsulati.

ROUTINE SEPARATE

Una libreria di risorse puo' essere usata per contenere un insieme di routine in relazione logica fra loro.

Ad esempio, e' possibile costruire una libreria matematica, che include alcune funzioni o procedure non predefinite.

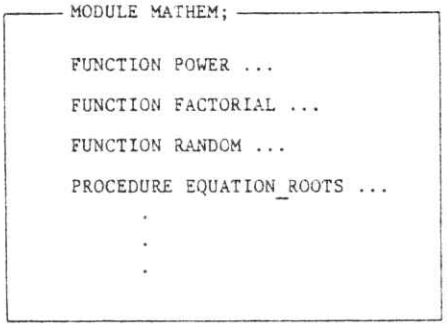


Fig. 7-6 Esempio di Routine Separate

Un altro esempio puo' essere una libreria per la gestione di periferiche.

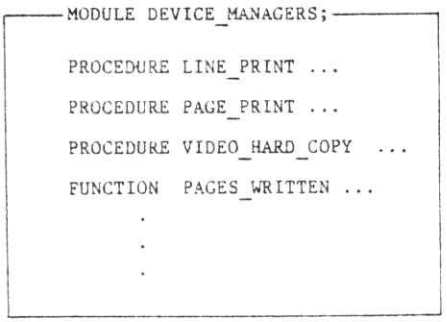


Fig. 7-7 Esempio di Routine Separate

Inoltre, per grossi progetti, che coinvolgono parecchi programmatori, le librerie di risorse permettono una separazione fisica dei testi dei programmi in modo tale da riflettere la divisione del lavoro e delle

responsabilita'. Una volta accordata e memorizzata l'interfaccia tra due qualsiasi parti, esse possono essere sviluppate e compilate separatamente.

Il vantaggio del fatto che l'interfaccia comune e' costituita da un testo fisicamente separato, risiede soprattutto nella garanzia che eventuali ricompilazioni distinte di ciascuna parte non invalidano tale interfaccia.

TIPI DI DATI INCAPSULATI

Una libreria di risorse puo' essere utilizzata come una descrizione generalizzata di un tipo strutturato.

Come visto nel Capitolo 5, un tipo strutturato e' un insieme di valori ognuno dei quali costituito da alcuni valori di un tipo base, l'operazione di selezione permette l'accesso a valori di tipi base, talvolta chiamati "elementi".

Una libreria di risorse puo' definire un tipo strutturato descrivendo semplicemente alcune variabili che possono contenere un insieme di valori, ed alcune routine che permettono l'accesso agli elementi, o altre operazioni associate.

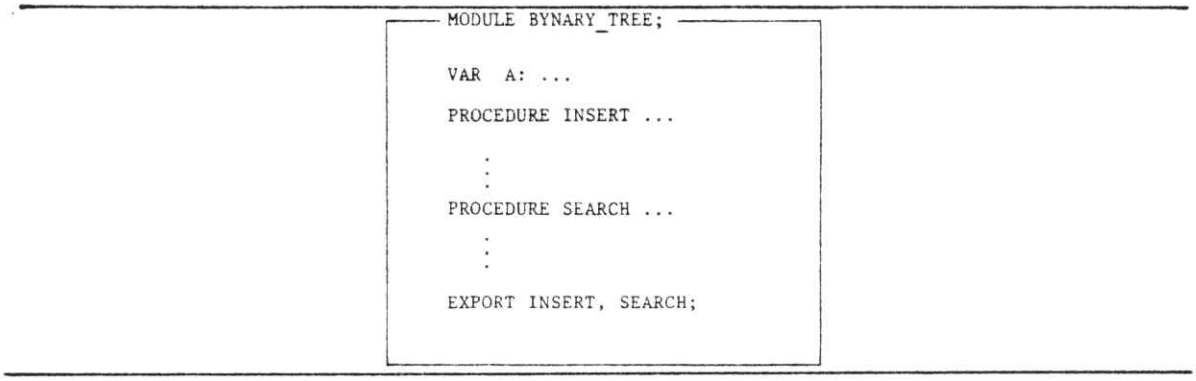


Fig. 7-8 Esempio di Tipo di Dati Incapsulato: un Albero Binario

Nell'esempio precedente, la libreria di risorse definisce il tipo strutturato albero binario ("binary_tree"), le cui operazioni di selezione sono: inserimento ("insert") e ricerca ("search") ovviamente, vengono esportate solo le operazioni e non le variabili, ci si garantisce cosi' nei confronti di possibili accessi illegali agli elementi dell'albero.

Altri possibili esempi sono i tipi strutturati: matrice ("matrix") e stack.

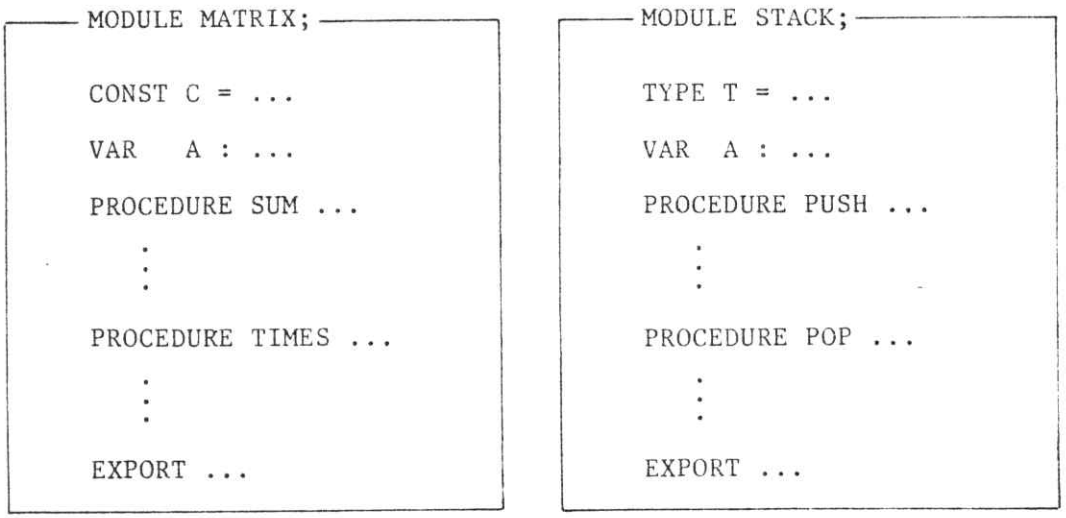


Fig. 7-9 Esempi di Tipi di Dati Incapsulati

Infine si puo' aggiungere che una libreria di risorse e' utilizzabile come una definizione di costruttore, cioe' un tipo strutturato il cui tipo base e' parametrizzato (vedere Capitolo 5, Tipi Strutturati). Nell'esempio che segue, la libreria di risorse e' usato come coda ("queue") sia di caratteri che di colori:

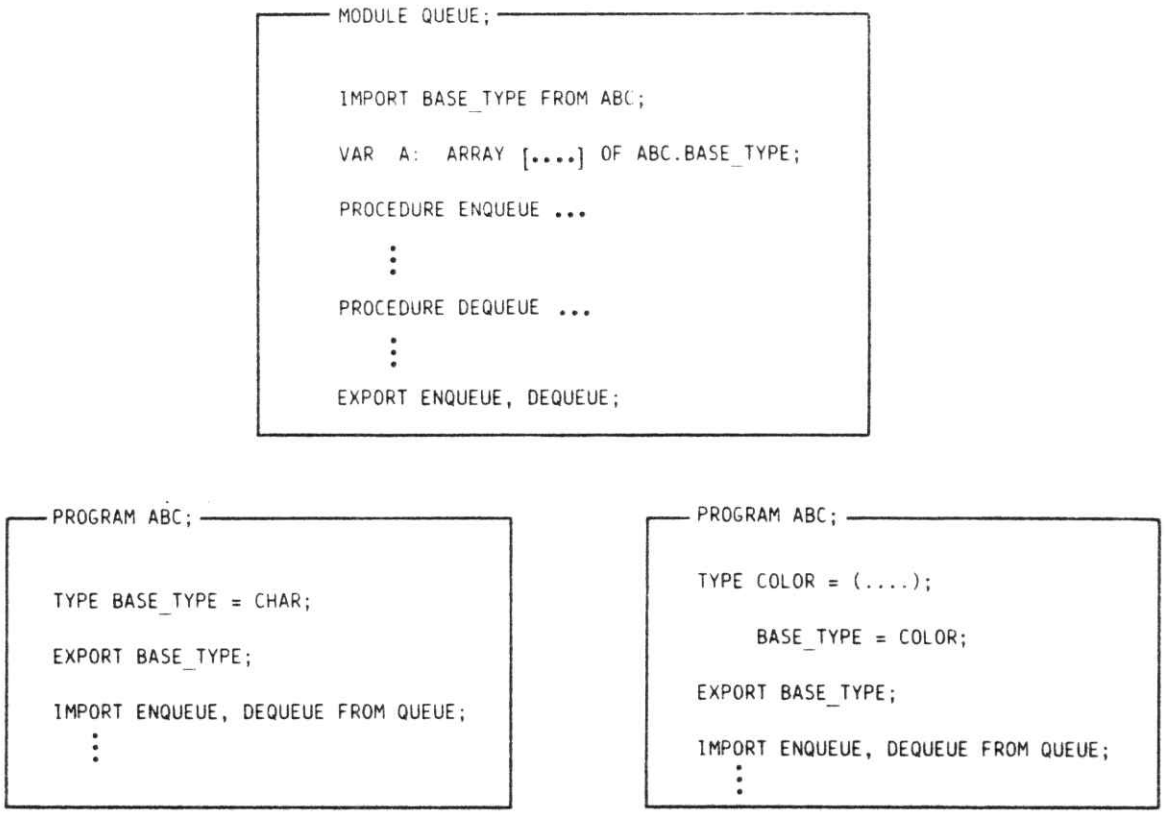


Fig. 7-10 Libreria di Risorse Vista come un Costruttore

La libreria di risorse importa il tipo base e lo usa nella descrizione della coda.

UNITA' PRINCIPALE DI PROGRAMMA

DIAGRAMMA SINTATTICO

Una unita' principale e' l'unica unita' che contiene una parte istruzioni in un programma. Il suo diagramma sintattico e' il seguente:

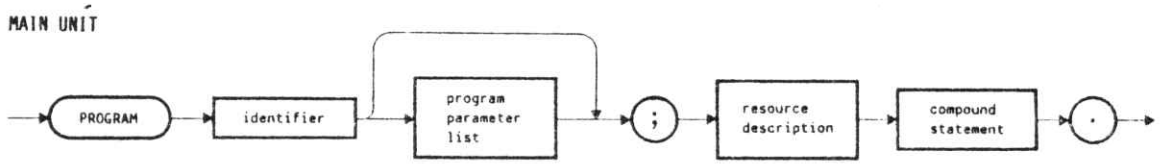


Fig. 7-11 Diagramma Sintattico per l'Unita' Principale

L'identificatore ("identifier") che segue la parola chiave PROGRAM e' il nome dell'unita' principale, la lista dei parametri di programma ("program parameter list") definisce le risorse valide prima e dopo l'esecuzione del programma; come e' stato detto alla fine del paragrafo "Il Run Time Stack" (Capitolo 6), queste risorse possono essere solo variabili file.
Il diagramma sintattico della lista dei parametri di programma e' il seguente:

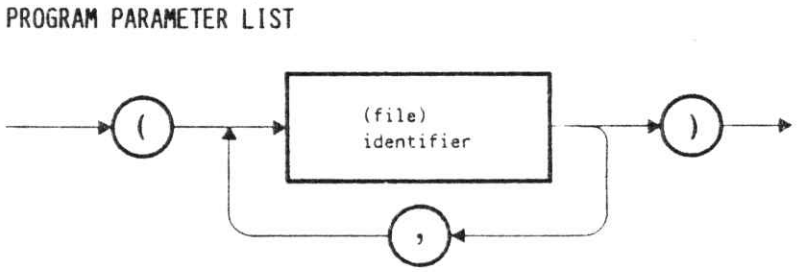


Fig. 7-12 Diagramma Sintattico per Lista dei Parametri di Programma

La descrizione di risorse ("resource description") e' una lista di risorse, usata nelle istruzioni, le quali possono essere:

- etichette ("labels description") (vedere Capitolo 12, paragrafo Istruzione GOTO);
- costanti ("costants description") (vedere Capitolo 9);
- tipi ("types description") (vedere Capitolo 8);
- procedure ("procedure description") (vedere Capitoli 4 e 13);
- funzioni ("function description") (vedere Capitoli 4 e 13).

La descrizione delle risorse puo' contenere qualsiasi numero di descrizioni, anche nessuna; il suo diagramma sintattico e':

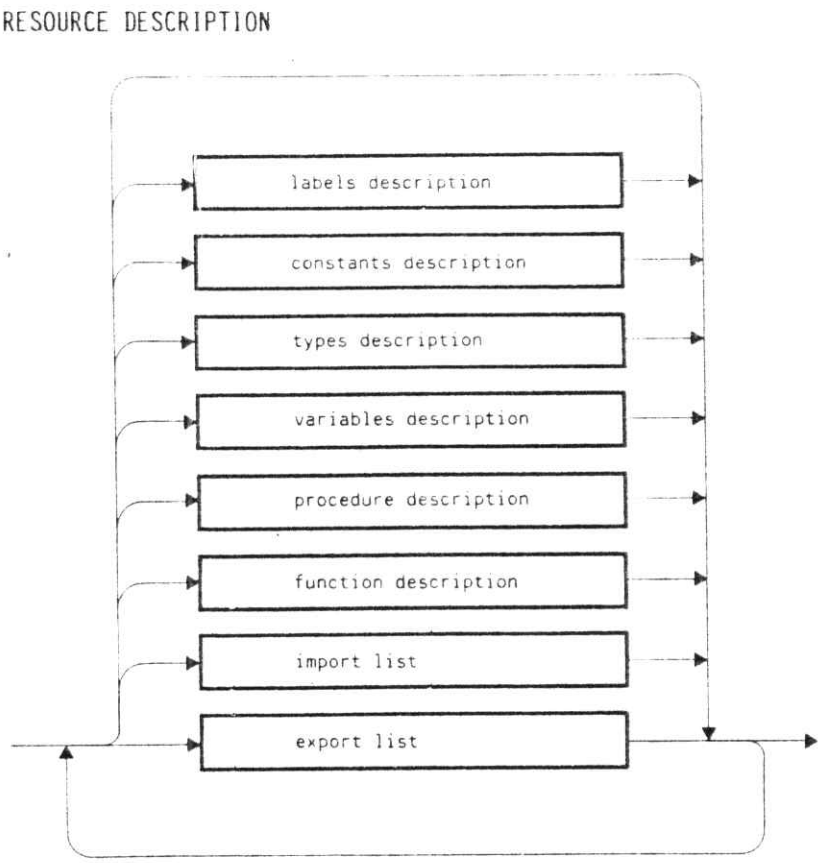


Fig. 7-13 Diagramma Sintattico per la Descrizione di Risorse

Alcune delle risorse descritte possono essere esportate in libreria di risorse, per cui devono elencarsi in una lista di esportazione ("export list").

Inoltre alcune risorse possono essere importate da una libreria di risorse, quindi devono essere elencate in una lista di importazione ("import list").

La parte istruzioni di un programma e' costituita da una istruzione composta ("compound statement") (vedere Capitolo 12).

DIAGRAMMA SINTATTICO

Modulo e' il termine che in PASCAL+ indica una libreria di risorse; la sua struttura e' simile a quella di una unita' principale, senza pero' contenere la parte istruzioni.
Il suo diagramma sintattico e' il seguente:

MODULE

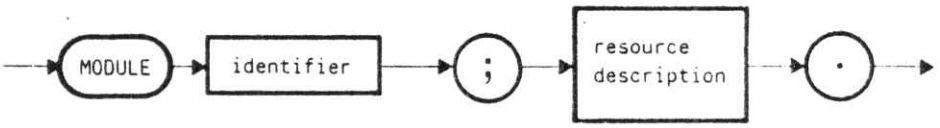


Fig. 7-14 Diagramma Sintattico del Modulo

L'identificatore che segue la parola chiave MODULE indica il nome del modulo.

La sintassi della descrizione di risorse ("resource description") per il modulo e' identica a quella che si ha per l'unita' principale.

PARTE II. CARATTERISTICHE DEL LINGUAGGIO BASE

8. TIPI

Il tipo dei dati di un programma (cioe' la descrizione dei dati) costituisce una delle sue possibili risorse.
In questo capitolo viene indicato come si definisce un tipo in PASCAL+.
Saranno considerati separatamente tipi semplici e strutturati (seguendo la classificazione introdotta nel Capitolo 5), e tipi puntatori, diversi dai precedenti.
Infine, sara' discussa la possibilita' di effettuare operazioni sui valori di tipi differenti.

INDICATORI DI TIPO

Un indicatore di tipo e' costituito dalla descrizione dell'insieme di valori che costituiscono un tipo.

Tale descrizione puo' essere usata sia in modo diretto per indicare i valori che puo' assumere una variabile, sia in modo indiretto assegnando all'indicatore di tipo un nome, cioe' un identificatore di tipo, per riferimenti successivi; e' da notarsi che a sua volta un indicatore di tipo puo' essere un identificatore.

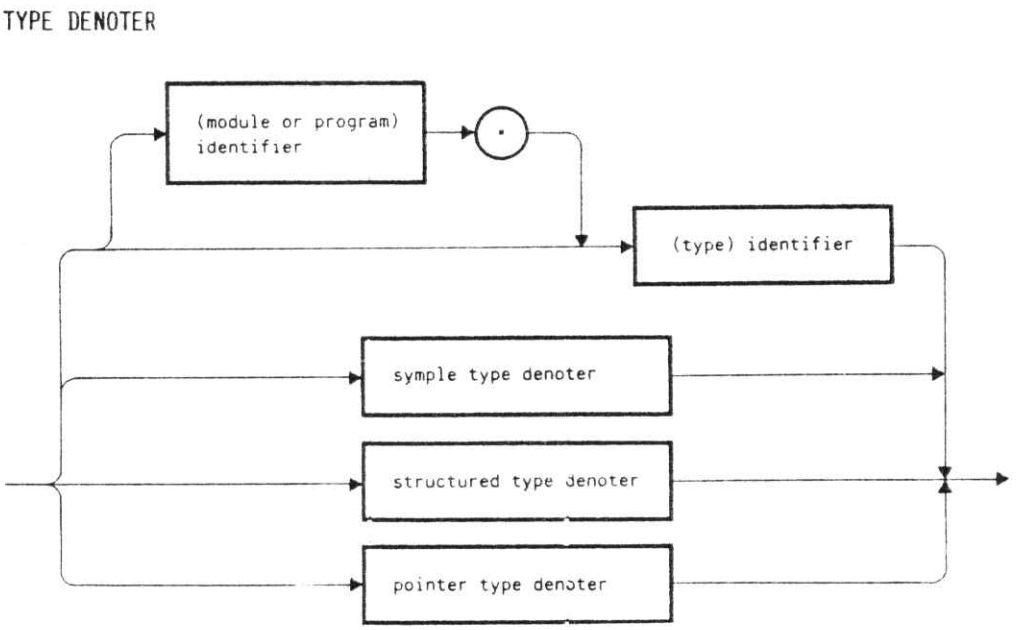


Fig. 8-1 Diagramma Sintattico per Indicatore di Tipo

I tipi predefiniti sono indicati da identificatori predefiniti (Vedere Capitolo 2, Par. Identificatori).

I tipi importati sono denotati dal nome del modulo (o del programma) seguito dal simbolo '.' e dall'identificatore di tipo. Per esempio se un programma importa il tipo T dal modulo M, sara' indicato con M.T.

Nei seguenti paragrafi vengono trattati gli indicatori di tipo semplice e strutturato, e di tipo puntatore ("symple, structured, pointer type denoter").

DESCRIZIONE DI TIPO

La descrizione di tipo costituisce il mezzo con cui assegnare un identificatore all'indicatore di tipo ("type denoter").

Il seguente diagramma sintattico fa riferimento al Capitolo 7, par. Unita' Principale di Programma.

TYPES DESCRIPTION

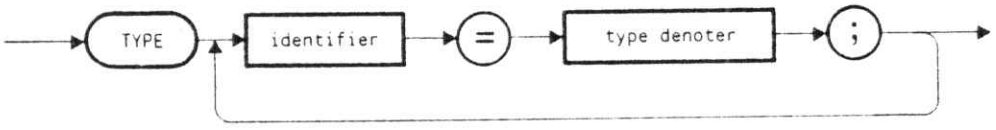


Fig. 8-2 Diagramma Sintattico per Descrizione di Tipo

La descrizione di tipo e' introdotta dalla parola chiave TYPE e puo' contenere diverse definizione di tipo, separate da un punto e virgola.

DEFINITI DALL'UTENTE

Come visto nel Capitolo 5, ci sono due strumenti di linguaggio per la definizione di un tipo semplice: enumerazione e subrange.

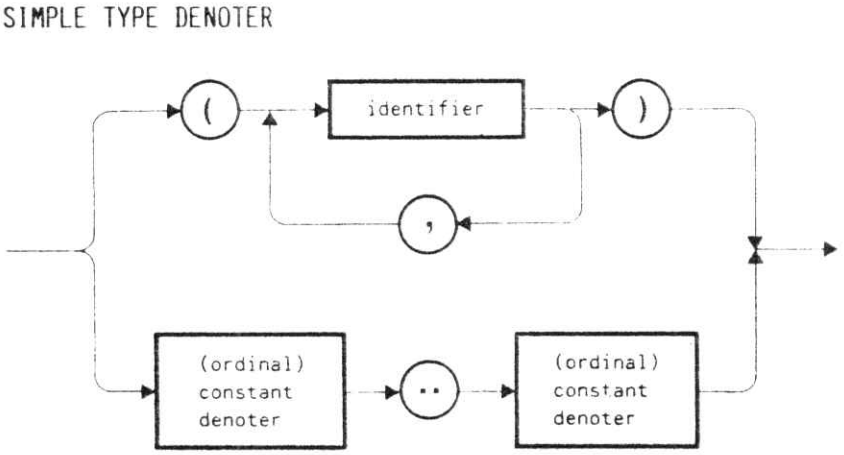


Fig. 8-3 Diagramma Sintattico per Indicatore di Tipo Semplice

ENUMERAZIONE

Un tipo enumerativo e' un insieme ordinato di valori definiti elencando gli identificatori con cui sono denotati i suoi valori.

L'ordine di enumerazione degli identificatori definisce l'ordinamento dei valori stessi.
Ogni elemento di un tipo enumerativo e' posto in corrispondenza con un numero naturale: il primo elemento e' associato a 0, il secondo ad 1, ecc. La funzione ORD (vedere Capitolo 13, par. Funzioni Ordinali Predefinite) fornisce il numero intero associato ad ogni elemento di un tipo enumerativo.

Ad un tipo enumerativo si possono applicare tutti gli operatori relazionali (vedere Capitolo 11, par. Operatori). Inoltre e' possibile valutare il successore ed il predecessore di ogni valore di un tipo enumerativo, attraverso le funzioni predefinite SUCC, PRED (vedere Capitolo 13 par. Funzioni Ordinali Predefinite).

Esempi

```
TYPE MESI = (GEN, FEB, MAR, APR, MAG, GIU, LUG, AGO, SET, OTT, NOV, DIC);
```

Il tipo MESI indica l'insieme dei codici valori indicati; ad esempio una variabile A dichiarata del tipo MESI, puo' assumere solo uno di quei valori.

```
VAR A : MESI; (*dichiarazione di variabile, vedere Capitolo 10, par.
               Descrizione di Variabili*)
  .
  .
A := FEB;      (*assegnazione, vedere Capitolo 12, par. Istruzione
                di Assegnazione*)
```

Come esempio di convenzioni di ordinamento, si hanno le seguenti relazioni:

GEN precede FEB
SET segue AGO

Il numero intero associato ad ogni valore enumerato puo' essere ottenuto tramite la funzione standard ORD:

ORD(GEN)=0
ORD(LUG)=6
ORD(DIC)=11

Altri esempi di tipi enumerativi sono dati nel Capitolo 5, paragrafo Tipi Semplici.

SUBRANGE

Un tipo subrange e' un insieme ordinato di valori, descritto come sottoinsieme compatto di un tipo semplice definito in precedenza, detto tipo ospite.
I sottoinsiemi di tipo reale non sono permessi.

Un tipo subrange e' descritto indicando i valori limite del sottoinsieme, cioe' specificando le costanti del limite inferiore e superiore ("(ordinal) constant denoter"), separate dal simbolo '..' (doppio punto, vedere Capitolo 2, 'Simboli Speciali').
La definizione di un tipo subrange e' valida solo se:

    limite inferiore <= limite superiore

Gli stessi operatori e funzioni definiti per il tipo ospite sono applicabili al tipo subrange.

E' da notare che ad ogni elemento del tipo subrange e' assegnato il valore ordinale appartenente a quell'elemento nel tipo ospite.

Esempi

```
TYPE MESI = (GEN,FEB,MAR,APR,MAG,GIU,LUG,AGO,SET,OTT,NOV,DIC);

MESI_ESTIVI = GIU..SET;
```

Il tipo MESI_ESTIVI e' costituito solo dai valori compresi tra GIU e SET (limiti inclusi); se si dichiara una variabile A di tipo MESI_ESTIVI:

```
VAR A: MESI_ESTIVI;
```

allora l'assegnazione

```
A:=LUG;
```

e' legale, ma l'assegnazione

```
A:=OTT;
```

e' illegale, poiche' OTT e' esterno ai limiti indicati dalla definizione. Il valore intero associato puo' essere ottenuto attraverso la funzione standard ORD:

```
ORD(AGO)=7
```

Durante l'esecuzione di un programma. ogni operazione effettuata sui valori del tipo subrange genera un controllo run-time dell'intervallo: percio' risulta molto utile, quando possibile, usare i tipi subrange per semplificare il testing dei programmi.

Altriesempi di tipi subrange si possono trovare nel capitolo 5, par. "Tipi semplici".

TIPI SEMPLICI PREDEFINITI

Il PASCAL+ dispone di tipi semplici predefiniti, denotati con i seguenti identificatori predefiniti:

```
INTEGER
REAL
BOOLEAN
CHAR
```

Tipo Integer

Il tipo integer rappresenta l'insieme di tutti i numeri interi. I suoi limiti sono definiti dall'implementazione, in relazione ai limiti fisici della dimensione del maggior numero che puo' essere memorizzato.
Gli identificatori predefiniti MAXINTEGER e MININTEGER rappresentano il massimo ed il minimo numero intero memorizzabile.

I valori interi sono rappresentati come nel diagramma sintattico di Figura 2.1.

Tipo Real

Il tipo real rappresenta l'insieme dei numeri reali, anche questi valori sono definiti dall'implementazione ed il massimo e minimo valore reale e' rappresentato dagli identificatori predefiniti MAXREAL e MINREAL.

I valori reali sono rappresentati come in figura 2.4.

E' importante notare che un tipo reale differisce da un tipo semplice in quanto non rappresenta un insieme discreto di valori, ma un insieme continuo, generando cosi' una mancanza di precisione per i numeri reali. Due valori reali se sono molto vicini, possono non essere distinguibili dal computer, e due valori uguali, se calcolati in modo diverso, possono essere differenti:

```
3.17842 puo' essere uguale a 3.17843
5.0 - 0.4E1 possono essere differenti da 1E2/5.0
```

Per questa ragione, il tipo reale non e' posto in corrispondenza con i numeri naturali e non deve essere utilizzato ne' per la definizione di tipi intervallo ne' nelle istruzioni FOR oppure CASE, ne' come indice di array, ne' come tipo base del tipo set.

Tipo Boolean

Il tipo boolean rappresenta i valori di verita' denotati dagli identificatori predefiniti TRUE e FALSE.

Essi non dipendono dall'implementazione, si puo' immaginare il tipo boolean definito come:

```
TYPE BOOLEAN = (FALSE, TRUE)
```

La convenzione circa l'ordine dei valori di tipo boolean e':

```
FALSE < TRUE
```

I comuni operatori booleani AND, OR, NOT insieme agli operatori relazionali sono applicabili ai valori booleani (Capitolo 11, 'Operatori').

Una variabile booleana, ad esempio B, puo' essere usata come condizione di controllo nelle istruzioni condizionali (Capitolo 12, istruzioni IF,

CASE, REPEAT .....) oppure ad essa puo' essere assegnata una espressione booleana (Capitolo 11):

```
VAR B:BOOLEAN; I:INTEGER
 .
 .
B:= I=0;
IF B THEN ....;
B:= FALSE;
```

il valore della variabile B e' TRUE se I=0, altrimenti e' FALSE; l'istruzione che segue THEN e' eseguita solo se B vale TRUE.

Tipo Char

Il tipo char rappresenta l'insieme di caratteri che dipende dall'implementazione della macchina. Alcuni caratteri sono talvolta senza rappresentazione grafica, cioe' non stampabili.

Come per ogni tipo ordinale, ad ogni valore CHAR corrisponde un valore naturale ottenuto dalla funzione predefinita ORD. La corrispondenza tra i valori char e l'intero associato dipende dalla macchina ed e' dato da una tabella di codifica (vedere documentazione dell'implementazione).

La costanti predefinite MAXCHAR e MINCHAR rappresentano i caratteri corrispondenti al massimo e al minimo valore associato.

La funzione predefinita CHR accetta un valore intero (>= 0) come parametro e fornisce il carattere associato (Capitolo 13, Funzioni Ordinali Predefinite).

I valori di tipo CHAR sono denotati da stringhe di lunghezza uno, come mostrato in Fig. 2.6.
Esempio dell'uso delle variabili CHAR.

```
VAR C: CHAR;   I: INTEGER;
 .
 .
I:=ORD('A');   (*I=193*);
C:=CHR(64);    (*C=spazio*);
C:='!';
C:='';
```

Per rappresentare il carattere apice lo si scrive due volte.

TIPI STRUTTURATI

In PASCAL+ i tipi strutturati sono descritti tramite alcuni metodi di strutturazione chiamati "costruttori". Inoltre la costruzione di strutture complesse di dati puo' essere ottenuta attraverso moduli e tipi puntatori, che sono trattati in dettaglio piu' avanti.
Nei paragrafi che seguono sono descritti i costruttori di tipo del PASCAL+, cioe' ARRAY, RECORD, SET e FILE.

Ogni costruttore di tipo puo' essere preceduto dalla parola chiave PACKED. Cio' indica che deve essere risparmiata la memoria occupata dai dati (per esempio eliminando i byte ridondanti), anche se questo comporta che l'accesso ai componenti risulta meno efficiente in termini di spazio o di tempo.

L'effetto della parola chiave PACKED e' determinato dalla specifica implementazione del linguaggio e descritto nella documentazione ad essa relativa.

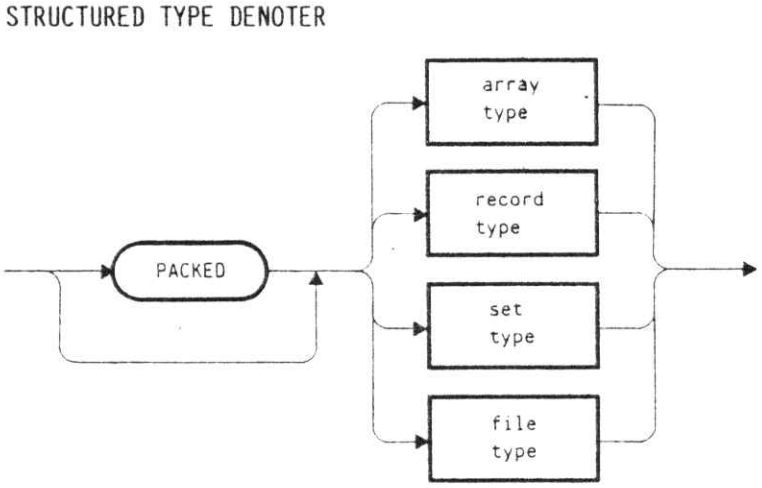


Fig. 8-4 Diagramma Sintattico per Indicatore di Tipo Strutturato

IL COSTRUTTORE ARRAY

Il costruttore ARRAY descrive un tipo strutturato composto da una raccolta di componenti, tutti del dedesimo tipo base, messo in relazione con uno o piu' tipi indice ordinali ("(ordinal) type denoter").

ARRAY TYPE

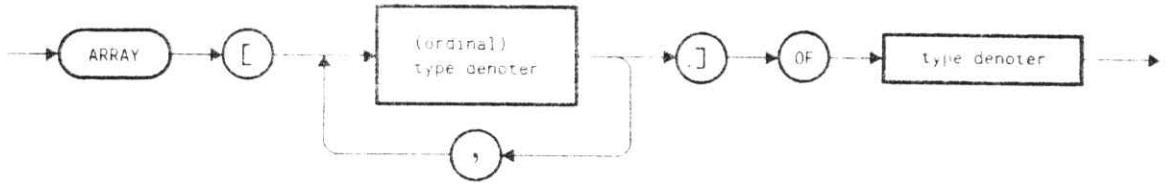


Fig. 8-5 Diagramma Sintattico per il Tipo ARRAY

Diversamente dagli altri linguaggi, cio' che e' importante nei tipi array, in PASCAL+, non e' il numero di componenti, ma la corrispondenza fra ciascun valore del tipo degli indici ed ogni singolo componente dell'array. I tipi indice devono essere ordinali, cioe' tipi semplici ad eccezione del tipo real.
Esempio:

```
TYPE GIORNI = (DOM, LUN, MAR, MER, GIO, VEN, SAB);
   LAVORO= 0..24;
   .
   .
VAR TABELLA_LAVORO: ARRAY [GIORNI] OF LAVORO;
```

Nell'esempio precedente, si ha una tabella che mette in relazione ogni giorno della settimana (cioe' ogni valore del tipo GIORNI) con il numero di ore lavorate nel giorno.
Quella che segue puo' essere la rappresentazione grafica della variabile TABELLA_LAVORO.

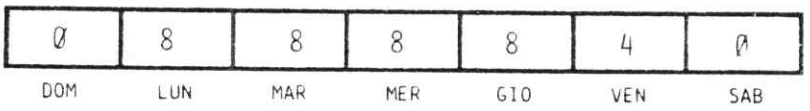


Fig. 8-6 Rappresentazione Grafica della Variabile TABELLA_LAVORO

Altro esempio:

```
TYPE FARMACIA = (JOHNS, WILLIAMS, DR_SPOCK, PERKINS);
  TABELLA_SALUTE = ARRAY [FARMACIA,GIORNI] OF BOOLEAN;
    .
    .
VAR APERTA: TABELLA_SALUTE
```

La variabile APERTA appena descritta e' una tabella settimanale che contiene, per ogni farmacia e per ogni giorno, il valore TRUE se essa e' aperta, FALSE in caso contrario.
Ne segue una possibile rappresentazione grafica:

	DOM	LUN	MAR	MER	GIO	VEN	SAB
JOHNS	true	true	false	true	true	true	false
WILLIAMS	false	true	true	true	true	true	false
DR. SPOCK	false	true	true	true	true	false	true
PERKINS	true	true	true	false	true	true	false

Fig. 8-7 Rappresentazione Grafica della Variabile APERTA

Altro esempio:

```
VAR TABELLA_ANNI: ARRAY [1..53] OF TABELLA_SALUTE;
```

Come si puo' vedere, il tipo base di un tipo array, puo' essere a sua volta strutturato. Non esistono restrizioni sul tipo base di un array.

L'accesso agli elementi di un array e' effettuato tramite subscritti, cioe' una lista di espressioni racchiuse tra parentesi quadre, una per ogni indice, ciascuna delle quali specifica un valore appartenente al corrispondente tipo indice.

Esempi:

```
VAR D: GIORNI;
     SETTIMANA: 1..53;
.
.
IF APERTA [JOHNS,VEN] THEN ...
IF TABELLA_LAVORO [D] < 0 THEN ...
TABELLA_ANNI [SETTIMANA +3] := APERTA;
IF TABELLA_ANNI [SETTIMANA+3,DR_SPOCK,DOM] THEN
```

Si noti che nell'ultimo esempio la variabile TABELLA_ANNI e' usata come singolo array tridimensionale invece che come array costituito da elementi di tipo array.
Il PASCAL+ considera le seguenti forme equivalenti:

```
ARRAY [T1] OF ARRAY [T2] OF T3
```

e

```
ARRAY [T1,T2] OF T3
```

IL TIPO STRINGA

Come anticipato nel Capitolo 2, le costanti di tipo stringa sono considerate valori di un tipo strutturato definito nel modo seguente:

```
PACKED ARRAY [1..N] OF CHAR
```

dove N e' la lunghezza della stringa, ovvero il numero di caratteri che la costituisce.
Per la ragione precedente, ogni tipo della forma:

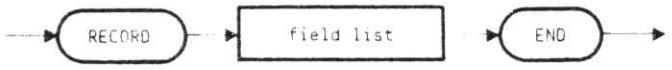
```
PACKED ARRAY [T] OF CHAR
```

dove T e' un indicatore di tipo subrange relativo ad un sottoinsieme intero con il limite inferiore uguale a 1, viene detto tipo stringa.

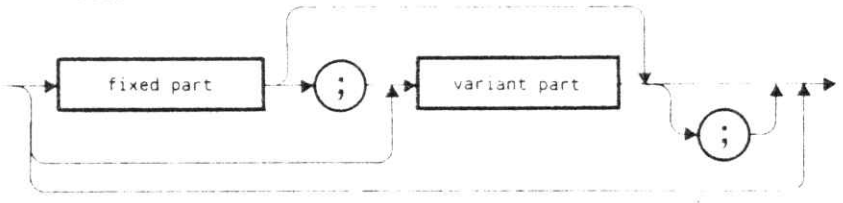
IL COSTRUTTORE RECORD

Il costruttore RECORD descrive un tipo strutturato composto da una raccolta di componenti, ciascuna di un tipo distinto, chiamati campi ("field list"), e denotati da identificatori.

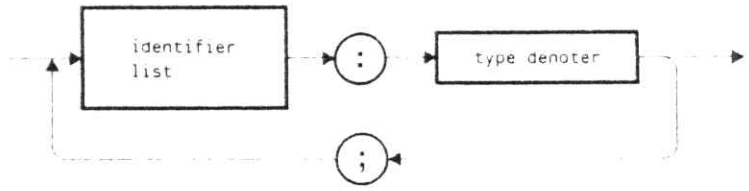
RECORD TYPE



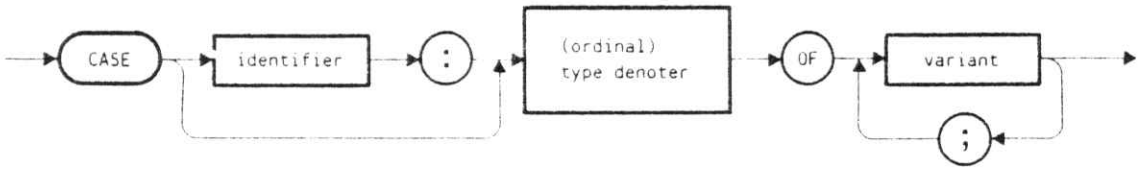
FIELD LIST



FIXED PART



VARIANT PART



VARIANT

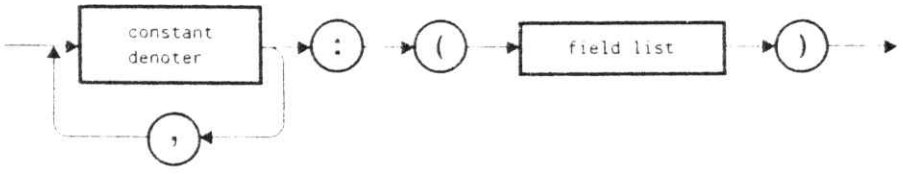


Fig. 8-8 Diagramma Sintattico per il Tipo RECORD

La presenza di una parte variante ("variant part") significa che, nel record, solo i campi associati ad una singola variante sono presenti in un dato momento accanto a quelli della parte fissa ("fixed part").

L'identificatore che segue la parola chiave CASE, se presente, indica un campo particolare (appartenente alla parte fissa) detta etichetta di campo (tag-field), comunemente usata per distinguere l'uso di una specifica variante.

Le costanti ("constant denoter") usate per definire le parti varianti devono essere distinte ed appartenenti al tipo ordinale specificato nel costrutto CASE; inoltre l'insieme usato deve essere minore o uguale oppure incluso nell'insieme di valori dell'identificatore di tipo.
Si considera ora qualche esempio:

```
TYPE = RECORD
         MESI : (GEN,FEB,MAR,APR,MAG,GIU,LUG,AGO,SET,OTT,NOV,DIC);
         GIORNO : 1..31;
         ANNO : INTEGER
       END;
```

Altro esempio:

```
TYPE STATO = (SPOSATO, CELIBE, VEDOVO);
     SESSO = (MASCHIO, FEMMINA);
VAR PERSONA : RECORD
                NOME : PACKED ARRAY [1..20] OF CHAR;
                NASCITA : DATA;
                S : SESSO;
                ST: STATO
              END;
```

Si noti che i campi in un record, a loro volta, possono essere strutturati, come NOME e NASCITA.
Un altro esempio, che include l'uso di varianti e' il seguente:

```
TYPE DICHIARANTE = (PRIVATO, DITTA);
VAR TASSA : RECORD
              CODICE,REDDITO : INTEGER;
              NOME : PACKED ARRAY [1..20] OF CHAR;
              DICH : DICHIARANTE;
              CASE DICHIARANTE OF
              PRIVATO : (NASCITA : DATA;
                         S : SESSO;
                         ST : STATO);
               DITTA : (IMPIEGATI : INTEGER)
            END;
```

Nel programma, vengono usati i campi NASCITA, S e ST, oppure il campo IMPIEGATI a seconda del tipo dichiarante. Generalmente le varianti del record sono sovrapposte nella memoria del computer.
Quando si usano i record con varianti, si ha bisogno di conoscere, di volta in volta, quale variante e' interessata: questa decisione puo' essere effettuata con un criterio noto, oppure con informazioni provenienti da altre variabili.
Nella maggior parte dei casi, comunque, la scelta della variante dipende dal valore di un campo della parte fissa.
E' il caso dell'esempio seguente.
In questo caso, il campo puo' essere associato, come etichetta di campo, al costrutto CASE stesso.

Quindi l'esempio precedente puo' essere migliorato nel modo seguente:

```
TYPE DICHIARANTE : (PRIVATO, DITTA);
VAR TASSA : RECORD
              CODICE, REDDITO : INTEGER;
              NOME : PACKED ARRAY [1..20] OF CHAR;
              CASE DICH  : DICHIARANTE OF
              PRIVATO : (NASCITA : DATA;
                         S : SESSO;
                         ST : STATO);
              DITTA : (IMPIEGATI : INTEGER)
            END;
```

Infine, si noti la forma

```
   CODICE, REDDITO : INTEGER;
```

che e' equivalente a

```
   CODICE   :  INTEGER;
   REDDITO  :  INTEGER;
```

Il prossimo esempio mostra come possono essere nidificati i record con varianti:

```
TYPE T = RECORD
          CASE 'A'..'C' OF
             'A' : ( );
             'B' : (X : INTEGER;
                    CASE I : 1..2 OF
                       1 : (A : REAL);
                       2 : (J : INTEGER));
             'C' : (B : BOOLEAN)
```

Si noti anche la "variante vuota" associata ad 'A'.

Gli identificatori di campo devono essere unici all'interno della definizione di record, e si accede a loro in relazione al valore dell'etichetta di campo, se e' presente.

I campi in una variabile record si indicano con il nome della variabile record seguita dal simbolo punto ".", ancora seguito dal nome del campo. Vediamo alcuni esempi:

```
IF TASSA.DICH = DITTA THEN
   IF TASSA.IMPIEGATI >100 THEN...
   .
   .
TASSA.NASCITA.ANNO := ANNO_ATTUALE;
    .
    .
```

Si noti il riferimento ai campi nidificati.
Pero' la notazione precedente puo' risultare non molto pratica: come viene detto piu' avanti essa puo' essere eseguita in forma contratta attraverso l'istruzione WITH.

Per ulteriori dettagli si puo' consultare il Capitolo 12.

IL COSTRUTTORE SET

Il costruttore SET descrive un tipo strutturato, di cui ciascun valore e' un sottoinsieme di un tipo scalare detto tipo base.

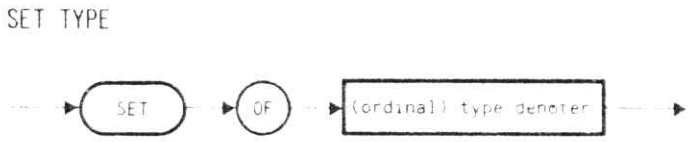


Fig. 8-9 Diagramma Sintattico per il Tipo SET

Si e' definito un tipo come un insieme di valori; in questo caso tali valori sono a loro volta insiemi di altri tipi.
Ad esempio, si consideri un tipo definito come:

```
TYPE INGREDIENTI = (ACQUA, SALE, ZUCCHERO, UOVA, FARINA);
```

Un certo numero di ingredienti puo' essere combinato per ottenere diverse ricette, come:

```
[ACQUA, ZUCCHERO]
[FARINA, ZUCCHERO, ACQUA, UOVA]
[FARINA, SALE, ACQUA]
[UOVA]
[ ]
```

Gli esempi precedenti, tra cui l'ultimo indica l'assenza di ogni ingrediente, possono essere pensati come valori di tipo RICETTA definito cosi':

```
TYPE RICETTA = SET OF INGREDIENTI;
```

Il tipo base (in questo case INGREDIENTI) puo' essere un qualsiasi tipo ordinale.
Se il tipo base e' un sottointervallo degli interi, ci possono essere restrizioni di implementazione sui limiti dell'intervallo.
La costante predefinita MAXSETCARD rappresenta la massima cardinalita' dell'insieme, cioe' il numero massimo di elementi che l'insieme puo' contenere.

Le variabili di tipo SET sono trattate tramite alcuni operatori (vedere

Capitolo 11) che eseguono alcune operazioni su insiemi, come: l'unione, l'intersezione, la differenza, l'uguaglianza, la disuguaglianza, l'inclusione e l'appartenenza.
Si considerino alcuni esempi:

```
VAR PANE, TORTA, DA_COMPERARE, COSA_HO : RICETTA;
 .
 .
PANE   := [ACQUA, FARINA, SALE];
TORTA  := [ACQUA, FARINA, ZUCCHERO, UOVA];
DA_COMPRARE  := COSA_HO - PANE + TORTA;
 .
 .
IF FARINA IN DA_COMPRARE THEN...
 .
 .
IF COSA_HO = [ ] THEN...
```

Si noti nell'ultimo esempio l'uso dell'"insieme vuoto".

IL COSTRUTTORE FILE

Il costruttore FILE descrive un tipo strutturato composto da una sequenza, di lunghezza qualsiasi, di componenti tutti dello stesso tipo base, dei quali solo uno alla volta e' accessibile in un dato istante.

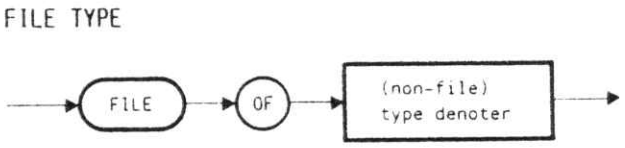


Fig. 8-10 Diagramma Sintattico per un Tipo FILE

Si noti che l'indicatore del tipo base non puo' contenere alcun componente di tipo file (nemmeno un riferimento indiretto ad uno di essi) ("(non-file) type denoter").
La dichiarazione di una variabile file F introduce automaticamente un "buffer" contrassegnato con F , che e' una variabile del tipo base della variabile file. Questo buffer non e' una parte del file, ma una specie di "finestra" attraverso la quale i valori componenti sono trasferiti dal programma al file.

Una variabile file puo' essere usata in scrittura (output) oppure in lettura (input), per mezzo di procedure e funzioni predefinite. Vedere

Capitolo 13 per esempi e dettagli.
Non e' possibile usare una variabile file contemporaneamente in lettura e scrittura, cioe' non si puo' aggiornare il contenuto di una variabile file.

Durante la lettura di una variabile file, l'ordine di comparizione dei diversi elementi riflette l'ordine della scrittura del file.
Gli elementi del file non sono ordinati, cioe' un file PASCAL+ non puo' avere chiavi; l'organizzazione del file puo' essere solo sequenziale.

Le variabili file sono le uniche variabili permesse in qualita' di parametri del programma, tali file sono, naturalmente, associati alla memoria periferica.
Si vede ora qualche esempio di tipi file:

```
PROGRAM P(F,C);
TYPE PACCO_DI_SCHEDE = FILE OF PACKED ARRAY [1..80] OF CHAR;
        .
        .
VAR F : FILE OF RECORD
                  I : INTEGER;
                  B : BOOLEAN
                END;
        C : PACCO_DI_SCHEDE;
        .
        .
```

IL TIPO TEXT

Il PASCAL+ dispone di un tipo file predefinito, TEXT, definito nel modo seguente:

```
TYPE TEXT = PACKED FILE OF CHAR;
```

Gli oggetti del tipo TEXT sono comunemente chiamati "file-testo".
I file-testo sono usati come mezzo principale di comunicazione tra programma e utente.
Tra le normali funzioni e procedure predefinite per il trattamento dei file, il linguaggio dispone di strumenti per la gestione dei file-testo.
Vedere Capitolo 13 per ulteriori dettagli ed esempi.

TIPI PUNTATORE

Il tipo puntatore rappresenta un insieme di valori che servono per accesso dinamico a variabili di un certo tipo; tali valori contengono i riferimenti agli elementi puntati.

POINTER TYPE DENOTER

Fig. 8-11 Diagramma Sintattico per l'Indicatore del Tipo Puntatore

L'identificatore di tipo che segue il simbolo " ^ " deve essere definito precedentemente oppure di seguito nella stessa procedura, funzione o blocco di programma (questa costituisce un'eccezione alle regole di visibilita').

I puntatori sono strumenti speciali forniti dal linguaggio, per permettere la costruzione di strutture dinamiche di dati come le liste, gli alberi ed array sparsi la cui struttura e dimensione non e' determinata al momento della compilazione, ed i cui componenti devono essere creati e distrutti durante l'esecuzione del programma da parte del programmatore.
I valori di tipo puntatore non si possono specificare direttamente, poiche' dipendono dall'implementazione.

Se P e' una variabile di tipo puntatore definita come

```
VAR P: ^T;
```

allora P e' detta "legata" al tipo T, nel senso che P puo' solo contenere riferimenti a variabili di tipo T.

Inoltre la variabile puntata (indicata da P^) puo' solo essere creata e distrutta per mezzo delle procedure predefinite NEW e DISPOSE (vedere Capitolo 13 per spiegazioni ed esempi).

La parola chiave NIL del linguaggio e' usata per denotare un particolare valore di puntatore che non punta ad alcun elemento.

Se il valore di una variabile puntatore P e' NIL, allora P^ non e' definito.
L'uso delle variabili puntatore e delle relative procedure predefinite per costruire struttore dinamiche di dati, e' illustrato nella parte III del manuale.

COMPATIBILITA' TRA TIPI

I "valori" non appartengono univocamente ad un determinato tipo.
Dal seguente esempio:

```
TYPE T1 = 0..20;
     T2 = 10..30
```

si puo' vedere che i valori da 10 a 20 sono comuni ai tipi T1 e T2, oltre che al tipo INTEGER.
Trattare oggetti di tipi diversi, aventi valori comuni, richiede una precisa definizione di compatibilita' tra tipi.

Il PASCAL+ considera due specie di compatibilita', cioe' la "compatibilita'" e la "compatibilita' nell'assegnazione".

Contesti diversi richiedono l'uno o l'altro tipo.
Di seguito si hanno le due definizioni distinte.

COMPATIBILITA'

Due tipi T1 e T2 sono detti compatibili se e' verificata almeno una delle seguenti condizioni:

a) T1 e T2 hanno la stessa struttura di tipo;

b) T1 e' un subrange di T2;

c) T2 e' un subrange di T1;

d) T1 e T2 sono subrange dello stesso tipo ospite;

e) T1 e T2 sono tipi set di tipi base compatibili e sono entrambi PACKED o entrambi non-packed;

f) T1 e T2 sono tipi stringa con lo stesso numero di componenti.

COMPATIBILITA' NELL'ASSEGNAZIONE

Un valore V, appartenente ad un tipo T1, e' detto compatibile nell'assegnazione con un tipo T2 se e' verificata almeno una delle seguenti regole:

a) T2 e' di tipo REAL e T1 e' di tipo INTEGER oppure un subrange di questo;

b) T1 e T2 sono tipi compatibili ed il valore V appartiene a T2. Inoltre T1 e T2 non possono essere file, ne' contenere (anche indirettamente) componenti di tipo file.

c) T1 e T2 sono tipi stringa e il numero dei componenti di T2 e' uguale o minore del numero dei componenti di T1.

9. COSTANTI

La distinzione dei valori di un tipo puo' anche essere pensata in termini di risorsa di programma. Le costanti sono il mezzo con cui sono denotati i singoli valori di un tipo.
Ad una costante puo' essere assegnato un identificatore per migliorare la leggibilita' e la manutenzione di un programma.

INDICATORI DI COSTANTI

Un indicatore di costante e' costituito dalla descrizione di un valore di un certo tipo.
Esso puo' appartenere a piu' di un tipo, pertanto non e' sempre possibile stabilirne il tipo.
La descrizione di valori e' utilizzata sia nella descrizione di tipi sia nella parte istruzioni di un programma. Ad essi puo' essere assegnato un nome, nella definizione di una costante, per successivi riferimenti.

CONSTANT DENOTER

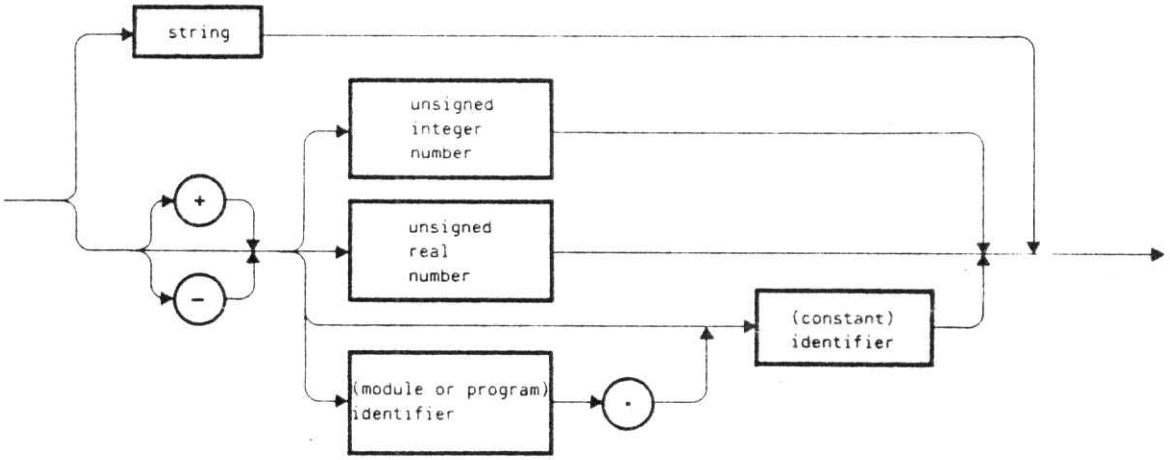


Fig. 9-1 Digramma Sintattico per Indicatore di Costante

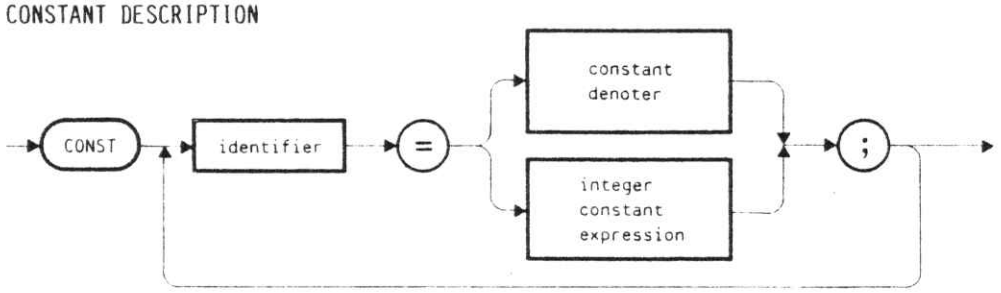
Si noti che gli identificatori di costante sono introdotti automaticamente dall'utente nella definizione di un tipo enumerato.
Inoltre il tipo predefinto BOOLEAN implica l'esistenza delle costanti predefinite FALSE e TRUE.

Le costanti importate sono distinte dal nome del modulo o del programma seguito dal simbolo "." e dall'identificatore di costante.

DESCRIZIONE DI COSTANTI

La descrizione di costanti e' il mezzo con cui si assegna un identificatore ad un valore.

CONSTANT DESCRIPTION



INTEGER CONSTANT EXPRESSION

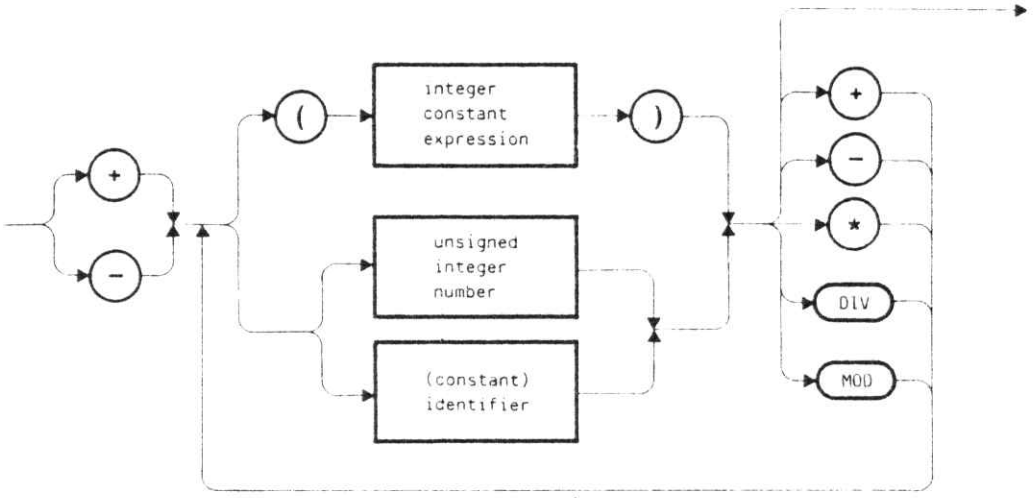


Fig. 9-2 Diagramma Sintattico per Descrizione di Costante

Si vedano alcuni esempi:

```
CONST A = -35;
      APLUS = A + 1;
      NOME = 'GIOVANNI';
      B = 3.3;
      C = 0;
      D = - (C DIV A) - APLUS + 8;
      E = MODULO_X.COSTANTE_Y;
```

Nelle espressioni costanti intere si applicano le normali regole di precedenza tra addizioni e moltiplicazioni, a meno che siano utilizzate le parentesi; gli operatori "+" e "-" hanno minore precedenza rispetto agli altri.

10. VARIABILI

Le variabili sono contenitori di dati i cui valori possono variare durante l'esecuzione del programma.
Una variabile e' sempre associata ad un tipo attraverso una dichiarazione di variabile.
Di seguito viene descritto come dichiarare ed usare le variabili in PASCAL+.

DESCRIZIONE DI VARIABILE

Una descrizione di variabile specifica il nome ed i possibili valori che essa puo' assumere.

VARIABLE DESCRIPTION

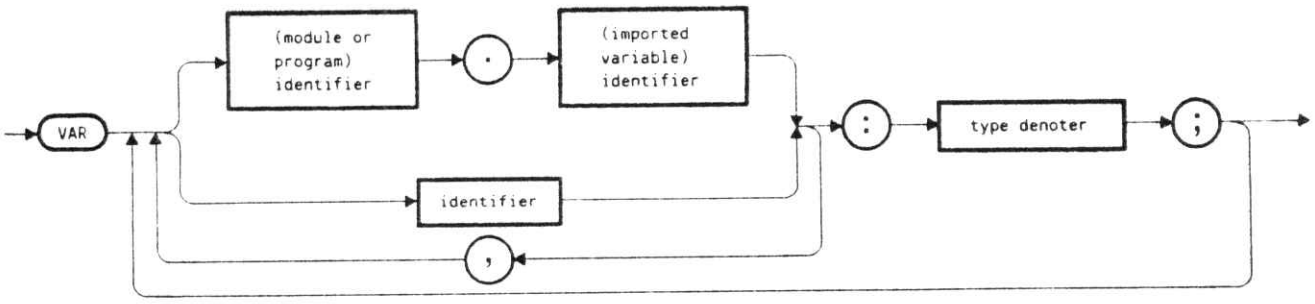


Fig. 10-1 Diagramma Sintattico per Descrizione di Variabile

Si vedano ora alcuni esempi:

```
TYPE COLORE = (ROSSO, BLU, GIALLO, NERO);
VAR A, B, C : INTEGER;
    Z : FILE OF INTEGER;
    H, X : RECORD
             X : COLORE;
             X1 : ROSSO..BLU
           END;
    M : ROSSO..BLU;
    K.V : SET OF COLORE;
```

Le variabili FILE possono essere usate anche come parametri di programma.

Nell'esempio precedente, la variabile Z potrebbe essere inclusa nella lista dei parametri di programma.

INDICATORE DI VARIABILE

Le variabili sono comunemente indicate tramite i loro identificatori. Le variabili componenti e puntate, pero', non hanno identificatori e sono indicate in modo strutturato a seconda del tipo cui appartengono.

VARIABLE DENOTER

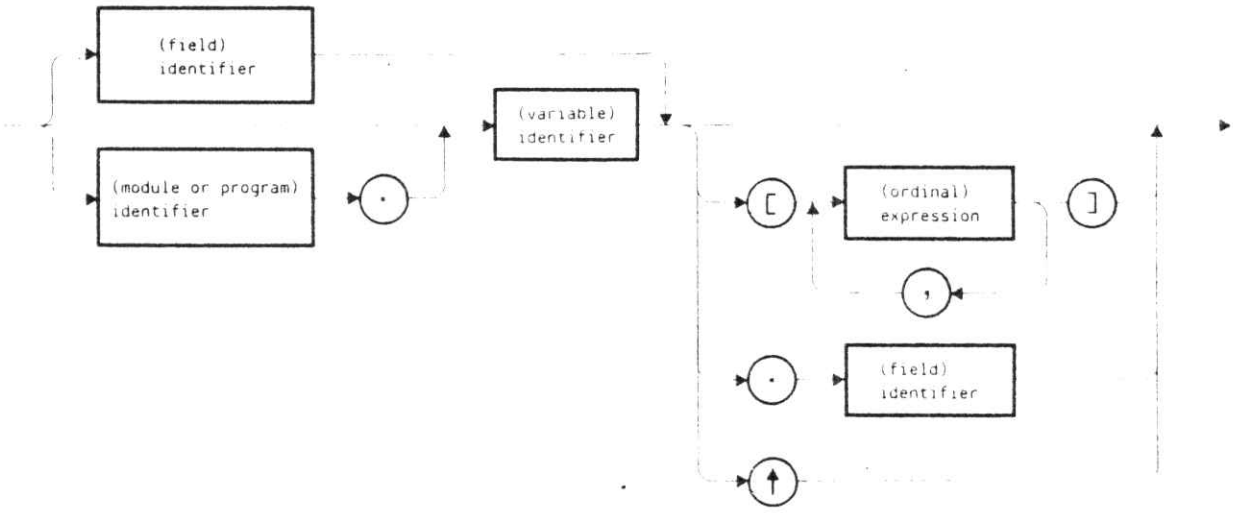


Fig. 10-2 Diagramma Sintattico per Indicatore di Variabile

Il riferimento ad una variabile puo' indicare:

a) una intera variabile;

b) un riferimento ad un elemento di array;

c) un campo di record;

d) una variabile puntata;

e) un buffer di file;

f) una variabile importata.

Riferimento ad una Variabile Intera

Ci si riferisce ad una variabile intera tramite il suo identificatore.
Esempio:

```
VAR X, Y : SET OF 0..30;
  .
  .
X:=Y
```

Riferimento ad un Elemento di Array

Ci si riferisce ad un elemento di un array specificando il nome dell'array seguito da un subscritto, cioe' una lista di espressioni racchiusa tra parentesi quadre.
L'espressione del subscritto deve essere conforme con i tipi indice dell'array.
Esempio:

```
VAR A : ARRAY  1..10, BOOLEAN  OF INTEGER;
  .
  .
A [5, K=Y] := X;
```

Riferimento ad un Campo di Record

Si fa riferimento ad un campo di un record specificando il record seguito dal simbolo "." e seguito a sua volta dall'identificatore di campo.
Esempio:

```
VAR R : RECORD
          X : INTEGER;
          Y : RECORD
              A : REAL;
              B : BOOLEAN
          END
        END;
  .
  .
R.Y.B:=TRUE
```

Si noti che all'interno dell'istruzione WITH, si puo' fare direttamente riferimento ai campi attraverso i loro identificatori.
Vedere "Istruzione WITH" nel Capitolo 12 per ulteriori dettagli.

Riferimento ad una Variabile Puntata

Si fa riferimento ad una variabile puntata specificando il puntatore seguito dal simbolo "^ ".

Esempio:

```
VAR P:^ CHAR;
 .
 .
P^ :='A';
```

Riferimento ad un Buffer di File

Si fa riferimento ad un buffer di file specificando il file seguito dal simbolo "^ ", come nelle variabili puntate.
Esempio:

```
VAR F : FILE OF CHAR;
   .
   .
F^ :='A';
```

Riferimento ad una Variabile Importata

Il riferimento ad una variabile importata (semplice o strutturata) e' effettuato tramite il nome del modulo (o programma) seguito dal simbolo "." e dal riferimento di variabile.
Esempio:

```
IMPORT I, A, R, P, F FROM M;
VAR I : INTEGER;
    A : ARRAY [CHAR] OF COLORE;
    R : RECORD
          C1 : REAL;
          C2 : BOOLEAN
        END;
    P :^ MESI;
    F : FILE OF ESTATE_MESI;
BEGIN
M.I := 3;
M.A ["!"]:=ROSSO;
IF M.R.C2 THEN...
M.P^ :=GEN;
M.F^ :=LUG;
END.
```

11. ESPRESSIONI

Le operazioni sono il mezzo fondamentale con cui valori di differenti tipi sono combinati per produrre dei risultati.
Le espressioni sono costituite da operatori, operandi e parentesi; in questo capitolo e' descritto come tali elementi sono collegati insieme in una sequenza corretta.

Separatamente vengono trattati gli operandi e gli operatori: gli operandi classificandoli come valori costanti, valori variabili, chiamate di funzioni, espressioni di tipo set, ed espressioni tra parentesi; gli operatori classificandoli sulla base dei tipi cui possono essere applicati.
Infine, vengono descritte le regole di precedenza degli operatori, insieme all'uso delle parentesi.

LA STRUTTURA DI UN'ESPRESSIONE

Le espressioni sono sequenze corrette di operandi, operatori e parentesi, valutate per avere un risultato.

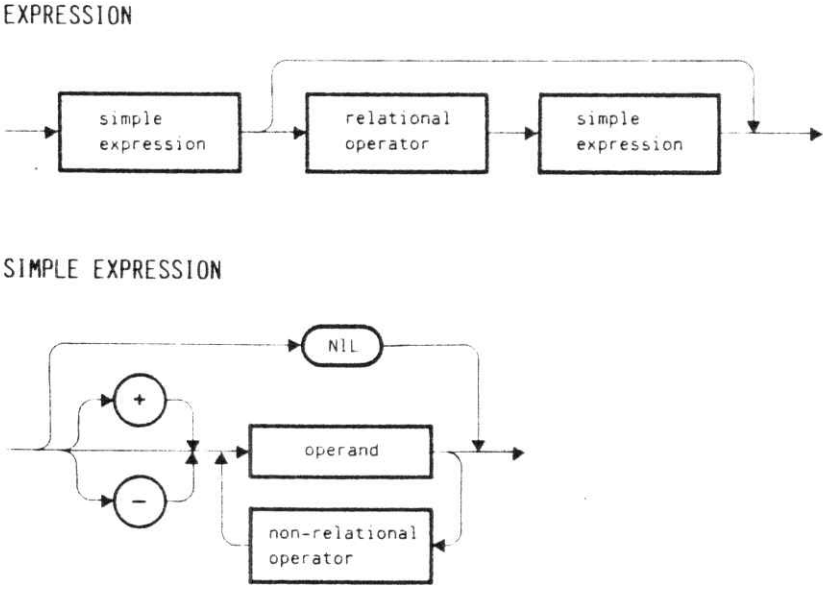
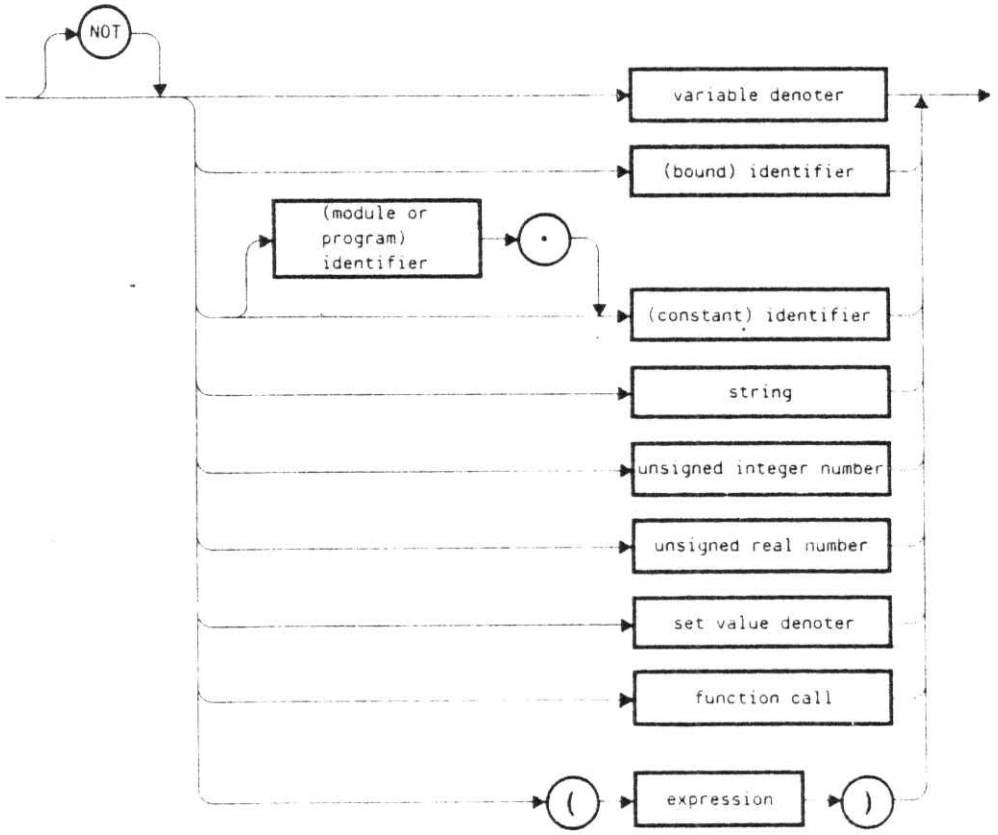
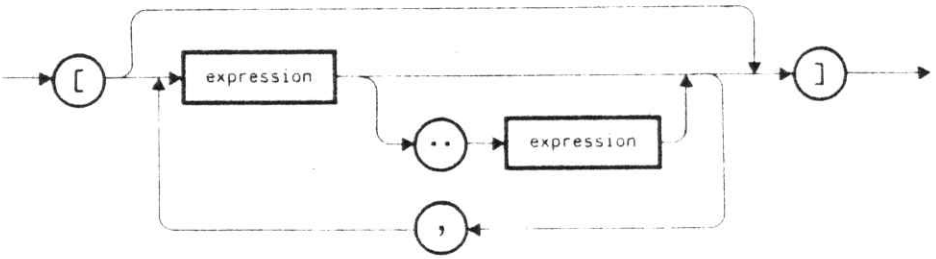


Fig. 11-1 Diagrammi Sintattici per Espressioni

OPERAND



SET VALUE DENOTER



FUNCTION CALL

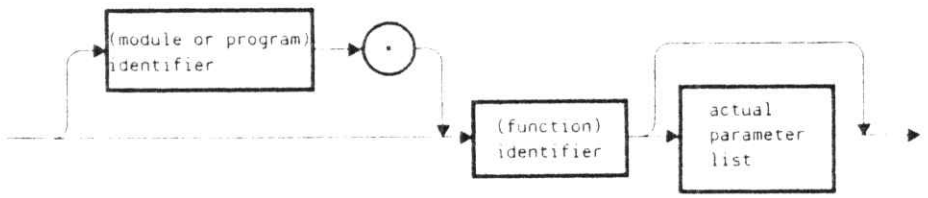


Fig. 11-1 Diagrammi Sintattici per Espressioni (Continua)

Il valore risultante di una espressione appartiene ad un tipo che dipende sia dal tipo dei suoi operandi che dagli operatori usati.

OPERANDI

L'uso di una variabile come operando indica il valore attuale della variabile. Si puo' verificare un errore se il valore della variabile e' indefinito.

L'uso di un identificatore di limite ("baund identifier") (vedere nel Capitolo 13 l'argomento riguardante i parametri) indica il valore attuale del limite.

L'uso di una chiamata di funzione ("function call") come operando indica il valore risultante della funzione.
Una chiamata di funzione e' semplicemente costituita dal nome della funzione, eventualmente seguita da una lista di parametri attuali ("actual parameter list").
Il diagramma sintattico completo della lista dei parametri attuali e' descritto nel Capitolo 13.

Il riferimento alle funzioni importate e' effettuato facendo precedere il loro nome da quello del modulo (o programma) e dal simbolo ".".

L'uso di un indicatore di valore di tipo set ("set value denoter") stabilisce che un valore (di tipo set) contiene i valori indicati da ciascuna espressione. La forma "espressione..espressione" indica l'intervallo dei valori compresi a partire dal valore dell'espressione di sinistra.

Se l'espressione di destra ha un valore inferiore a quello di sinistra, la forma precedente indica il valore "vuoto" del tipo set.

In una espressione, il valore di un operando e' trattato come il valore del tipo ospite piu' grande, se questo esiste. Per esempio, se si ha:

```
VAR X:0..10;
 .
 .
 .
```

il valore dell'operando X nell'espressione X+2 e' trattato come un valore di tipo INTEGER anziche' di tipo subrange.
Lo stesso criterio si applica ai valori di tipo SET, il cui tipo base e' possibilmente il tipo ospite piu' grande.

OPERATORI

Gli operatori si possono classificare in base alle loro applicazioni.
La precedenza tra operatori viene descritta alla fine del capitolo.

OPERATORI ARITMETICI

Il significato degli operatori aritmetici, il tipo dei loro operandi ed il tipo del risultato sono descritti nelle tabelle che seguono.

OPERATORE	SIGNIFICATO	TIPO DELL'OPERANDO SINISTRO	TIPO DELL'OPERANDO DESTRO	TIPO DEL RISULTATO
+	somma	integer o real	integer o real	integer se entrambi gli operandi sono integer, altrimenti real
−	sottrazione	integer o real	integer o real	"
*	prodotto	integer o real	integer o real	"
/	divisione	integer o real	integer o real	real
DIV	divisione con troncamento	integer	integer	integer
MOD	modulo	integer	integer	integer

Fig. 11-2 Tabella degli Operatori Aritmetici Diadici

OPERATORE	SIGNIFICATO	TIPO DELL'OPERANDO	TIPO DEL RISULTATO
+	identità	integer o real	lo stesso dell'operando
−	inversione di segno	integer o real	"

Fig. 11-3 Tabella degli Operatori Aritmetici Monadici

Esempi:

-(+4) da' come risultato -4
-(-3.0) da' come risultato 3.0
6+5 da' come risultato 11
6.0+5 da' come risultato 11.0
7/2 da' come risultato 3.5
7 DIV 2 da' come risultato 3
14 MOD 3 da' come risultato 2
(-14) MOD 3 da' come risultato -2
(-14) MOD (-3) da' come risultato -2
14 MOD (-3) da' come risultato 2

(Si noti che il segno del risultato dell'operazione modulo dipende solo dal segno dell'operando sinistro).

OPERATORI BOOLEANI

Il significato degli operatori booleani e' descritto nella seguente tabella:

OPERATORE	SIGNIFICATO	TIPO OPERANDO (1)	TIPO DEL RISULTATO
NOT (monadico)	negazione logica	boolean	boolean
AND (diadico)	AND logico	boolean	boolean
OR (diadico)	OR logico	boolean	boolean

Fig. 11-4 Tabella degli Operatori Booleani

OPERATORI SU INSIEMI

Il significato degli operatori su insiemi e' descritto nella tabella seguente:

OPERATORE	SIGNIFICATO	TIPO DEGLI OPERANDI	TIPO DEL RISULTATO
+	unione	insieme di T	insieme di T
−	differenza	insieme di T	insieme di T
*	intersezione	insieme di T	insieme di T

Fig. 11-5 Tabella degli Operatori su Insiemi

Esempi:

|ROSSO, BLU, GRIGIO| + |GIALLO, BLU| da' come risultato
|ROSSO, BLU, GRIGIO, GIALLO|
|ROSSO, BLU, GRIGIO| - |GIALLO, BLU| da' come risultato |ROSSO, GRIGIO|
|ROSSO, BLU, GRIGIO| * |GIALLO, BLU| da' come risultato |BLU|

OPERATORI RELAZIONALI

L'uso ed il significato degli operatori relazionali e' descritto nella tabella seguente:

OPERATORE	SIGNIFICATO	TIPO DELL'OPERANDO SINISTRO	TIPO DELL'OPERANDO DESTRO	TIPO DEL RISULTATO
=	uguaglianza	ogni tipo semplice, stringa, puntatura o set	come l'operando sinistro	boolean
< >	disuguaglianza	"	"	boolean
<	minore di	ogni tipo semplice o stringa	"	boolean
>	maggiore di	"	"	boolean
> =	non minore di	"	"	boolean
< =	non maggiore di	set di T	"	boolean
> =	include	set di T	"	boolean
< =	incluso	di	"	boolean
IN	appartenenza	tipo ordinale di T	set di T	boolean

Fig. 11-6 Tabella degli Operatori Relazionali

Esempi:

'GIOVANNI' = 'GIACOMO' da' come risultato FALSE (assumendo la sequenza ordinata relativa alla tabella ASCII)
BLU IN [ROSSO,NERO,BLU] da' come risultato TRUE
[ROSSO,BLU] = [ROSSO,NERO,BLU] da' come risultato TRUE
3 = 3.0 da' come risultato TRUE

Si noti che gli operandi di tipo REAL ed INTEGER possono essere messi in relazione. Inoltre le stringhe possono essere comparate solo se hanno la stessa lunghezza, cioe' se sono dello stesso tipo.

PRECEDENZA TRA OPERATORI E USO DELLE PARENTESI

Nella valutazione di una espressione, il valore risultante dipende in generale dall'ordine di esecuzione delle operazioni.
Per esempio, l'espressione

```
3 + 3 * 5
```

da' come risultato 18 se il prodotto e' eseguito per primo, 30 se invece e' la somma ad essere eseguita per prima. Le parentesi sono sempre usate nella solita notazione algebrica per determinare l'ordine di esecuzione, pero' quando le parentesi sono omesse si hanno le seguenti regole.

1. Agli operatori e' assegnato il grado di precedenza specificato dalla seguente tabella:

PRECEDENZA	OPERATORI
4 (più alta)	NOT
3	*, /, DIV, MOD, AND
2	+, -, OR
1 (più bassa)	=, <>, <, >, <=, >=, IN

Fig. 11-7 Tabella delle Precedenze tra Operatori

In presenza di un'ambiguita' tra operatori aventi diversi gradi di precedenza, quello con la piu' alta viene eseguito per primo.
Esempi:

3 + 3 * 5 significa 3 + (3 * 5)

NOT A OR B AND C = D significa ((NOT A) OR (B AND C)) = D

A + B = C significa (A + B) = C

2. Nel caso di conflitto tra operatori con la stessa priorita', l'ordine di esecuzione e' da sinistra a destra.
Esempi:

X/Y/Z significa (X/Y)/Z

3 * X DIV 3 significa (3 * X) DIV 3

Si noti, comunque, che due operatori relazionali consecutivi devono essere divisi da parentesi, cioe':

```
VAR A,B,C : BOOLEAN;
.
.
A=B=C e' errato poiche' la sintassi vuole che sia
        (A=B)=C oppure A=(B=C)
```

L'uso delle parentesi per specificare l'ordine di esecuzione delle operazioni in un'espressione e' sempre auspicabile, anche se ridondante, in quanto rende i programmi piu' leggibili e riduce la possibilita' di errori, spesso difficili da trattare.

Esempi di espressioni:

```
A=B [X,Y] DIV 5
SQRT (SIN (A))
(X   = MAXTAB) AND NOT FOUND
MODULE_X.FACTORIAL (3+Y).
```

12. ISTRUZIONI

In questo capitolo vengono trattate le istruzioni di un programma PASCAL+.

Le istruzioni PASCAL+ (statement), inducono alla costruzione di programmi strutturati, cioe' programmi in cui il flusso di esecuzione risulta particolarmente comprensibile.

Ogni istruzione viene considerata separatamente, descrivendo prima le istruzioni eseguibili, poi quelle di controllo.

GENERALITA' SULLE ISTRUZIONI

Le istruzioni costituiscono gli strumenti con i quali viene descritta la parte algoritmica, le procedure o le funzioni di un programma.

Alcune istruzioni dette eseguibili, permettono all'utente di indicare che un'operazione deve essere eseguita su dei dati (come l'istruzione di assegnazione o di chiamata a procedure).

Altre istruzioni, dette di controllo (o strutture di controllo), permettono di indicare un particolare ordine di esecuzione, in accordo con i principi della programmazione strutturata.

Inoltre e' prevista un'istruzione (WITH) per facilitare l'accesso ai campi di una variabile record.
Qualsiasi istruzione puo' avere una etichetta (label), cioe' un numero intero senza segno, che e' usata come riferimento per un'istruzione GOTO (vedere il relativo paragrafo per una completa descrizione sull'uso delle etichette).

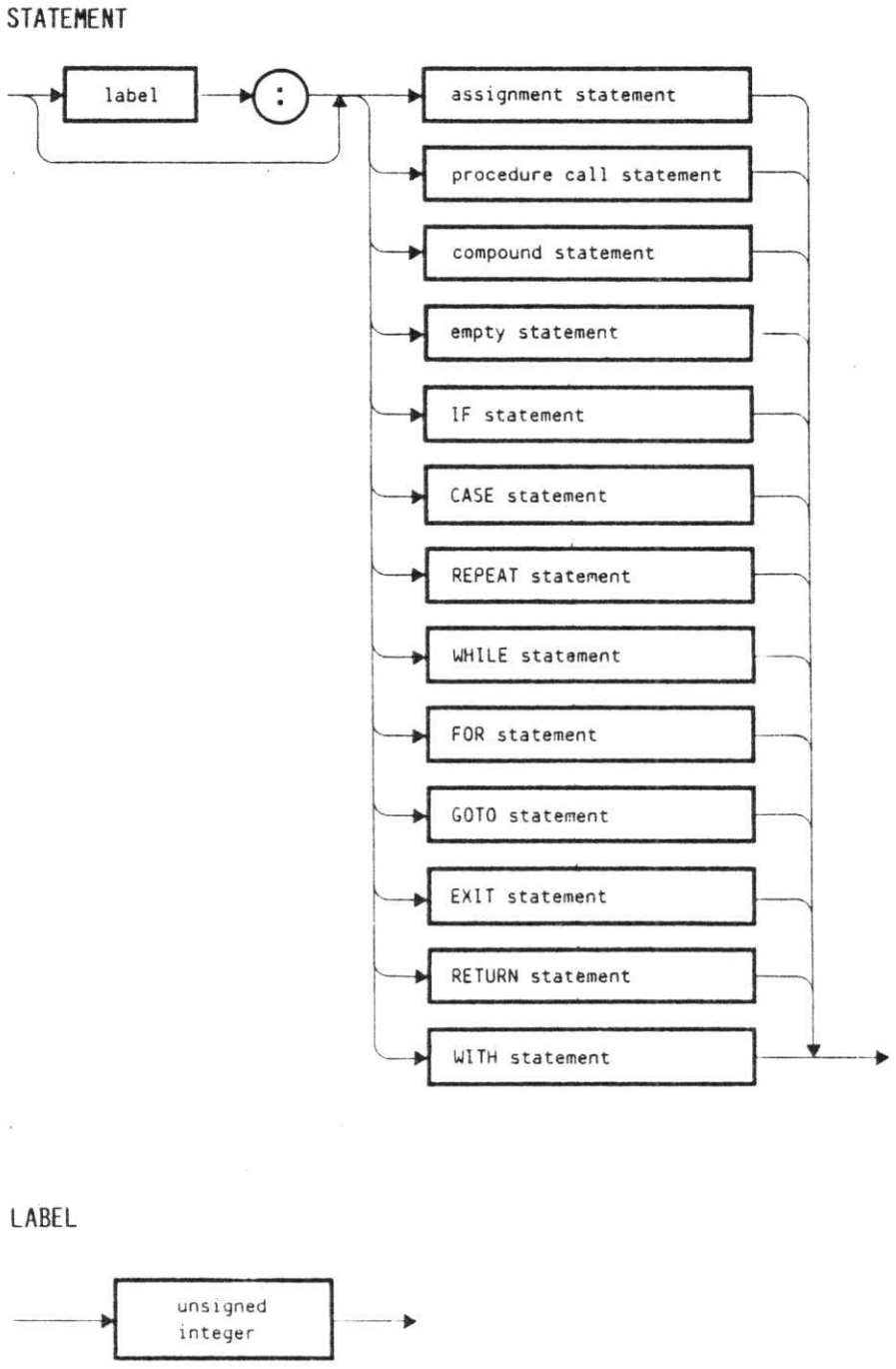


Fig. 12-1 Diagramma Sintattico per le Istruzioni

ISTRUZIONI DI ASSEGNAZIONE

L'istruzione di assegnazione (assignment statement) e' usata per sostituire l'attuale valore di una variabile (se c'e') con un nuovo valore ottenuto dalla valutazione di un'espressione di tipo compatibile.

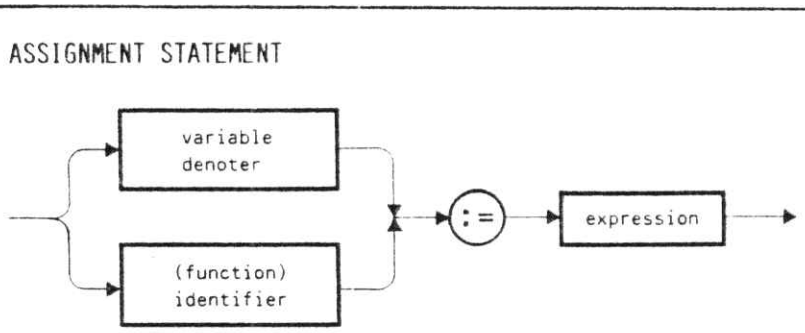


Fig. 12-2 Diagramma Sintattico per l'Istruzione di Assegnazione

Se un identificatore di funzione e' usato alla sinistra del simbolo di assegnazione ":=", allora il valore ottenuto dall'espressione e' assunto come valore di ritorno della funzione (vedere Capitolo 13, Chiamata ed Esecuzione di Procedure e Funzioni). Questa assegnazione e' permessa soltanto all'interno della funzione considerata.

Il valore ottenuto dall'espressione deve essere compatibile nell'assegnazione con il tipo della variabile (o della funzione). La descrizione delle regole di "Compatibilita' nell'Assegnazione" e' riportata nel Capitolo 8.

Esempi:

```
FUNCTION F : REAL;
  .
  .
BEGIN
  .
  .
  F:= 3.0
END;
  .
  .
VAR I,J : 1..10;
    A,B : ARRAY [1..10] OF CHAR;
      R : REAL;
  .
  .
I:=J;
R:=I-J;
A[J]:= 'X';
A:=B
  .
  .
  .
```

ISTRUZIONE DI CHIAMATA DI PROCEDURA

L'istruzione di chiamata di procedura e' usato per attivare l'esecuzione di una procedura (sia definita dall'utente che predefinita), possibilmente fornendo parametri attuali.

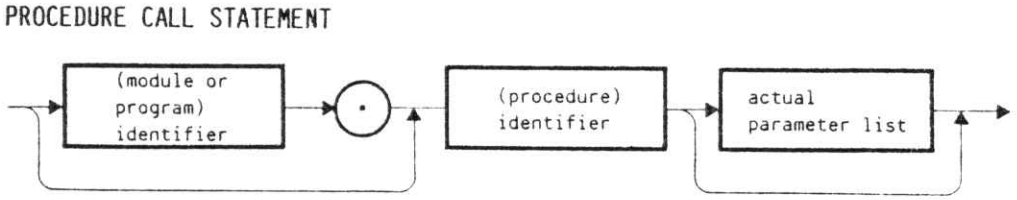


Fig. 12-3 Diagramma Sintattico di un'Istruzione di Chiamata di Procedura

L'uso di parametri, che sono comuni a procedure e a funzioni, e' descritto nel Capitolo 13. Come descritto precedentemente (Capitolo 6), l'attivazione di procedure definite dall'utente causa l'allocazione dinamica temporanea delle variabili locali alla procedura.
Quando la procedura chiamata e' terminata, l'esecuzione del programma riprende normalmente dall'istruzione (se esiste) seguente quella che ha chiamato la procedura.
La presenza di un'istruzione GOTO all'interno della procedura chiamata, puo' tuttavia far variare questa regola (vedere l'istruzione GOTO per ulteriori dettagli).

SEQUENZA DI ISTRUZIONI

Le azioni sequenziali sono espresse in PASCAL+ con l'uso del simbolo concatenatore di istruzioni ";".

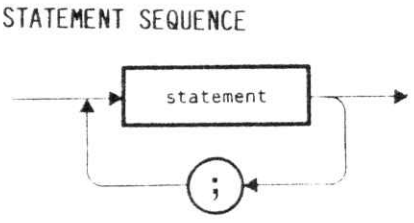


Fig. 12-4 Diagramma Sintattico di una Sequenza di Istruzioni

E' da notare che il simbolo ";" agisce da separatore di istruzioni, non come loro simbolo terminatore, cioe' l'ultima istruzione di una sequenza puo' non essere seguita dal punto e virgola. Il lettore non abituato a questa convenzione, trovera' nella Parte III alcune informazioni riguardo l'uso del punto e virgola.

Esempio:

```
(* SCAMBIO DI VALORI TRA LE VARIABILI A E B *)
PARK:=A;
A:=B;
B:=PARK
  .
  .
  .
```

Poiche' molte istruzioni strutturate (IF, CASE, WHILE, WITH) richiedono per la loro sintassi una sola istruzione, e' spesso necessario raccogliere in una sola istruzione una sequenza di istruzioni.
L'Istruzione composta BEGIN ... END, descritta successivamente, deve essere usata in questi casi.

ISTRUZIONE COMPOSTA

L'istruzione composta permette di raccogliere una sequenza di istruzioni in una singola istruzione.

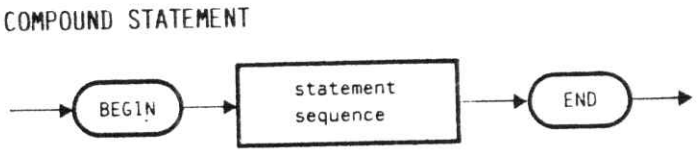


Fig. 12-5 Diagramma Sintattico di un'Istruzione Composta

Il seguente raggruppamento e' necessario nei casi in cui occorra racchiudere:

1. la parte istruzioni di un programma, procedura o funzione (vedere i relativi diagrammi sintattici nei Capitoli 7 e 13);

2. una sequenza di istruzioni relative ad un ramo THEN, ELSE o DO all'interno delle istruzioni IF, WHILE o WITH (vedere paragrafi seguenti);

3. una sequenza di istruzioni relative ad un ramo di un'istruzione CASE (vedere paragrafi seguenti);

4. una singola istruzione IF, permettendo cosi' una corretta interpretazione della condizione di ELSE (vedere paragrafi seguenti).

Inoltre un'istruzione composta puo' essere utilizzata, anche se non richiesta, per riunire una sequenza di istruzioni, allo scopo di ottenere una maggior chiarezza.
In casi di questo genere e' particolarmente utile l'uso di commenti.

Esempi:

```
BEGIN (* SCAMBIO *)
  PARK := A;
  A:=B;
  B:=PARK
END (* SCAMBIO *)
  .
  .
BEGIN
  PROC (A)
END
```

ISTRUZIONE VUOTA

L'istruzione vuota e' usata per indicare che non deve essere compiuta alcuna azione, e' un'istruzione senza operazioni.

Dal punto di vista sintattico, l'istruzione vuota e' costituita dall'assenza di simboli.

Esempi:

```
.
.
IF EOF (F) THEN (* LA VARIABILE FILE F E' VUOTA *)
ELSE
  REPEAT
    .
    .
  UNTIL EOF (F)
  .
  .
```

E' da notare l'uso di un'istruzione vuota come istruzione relativa al ramo THEN in un'istruzione IF.

```
    .
    .
 CASE CARDTYPE OF
   OPTION : ELABOPTION;
   STATEMENT : ELABSTAT;
   COMMENT :
 END
    .
    .
```

In questo caso invece l'istruzione vuota e' messa in relazione al valore COMMENT dell'istruzione CASE.

ISTRUZIONE IF

L'istruzione IF indica che si deve operare una scelta fra due istruzioni, tale scelta dipende dal valore di una espressione di tipo booleano.

IF STATEMENT

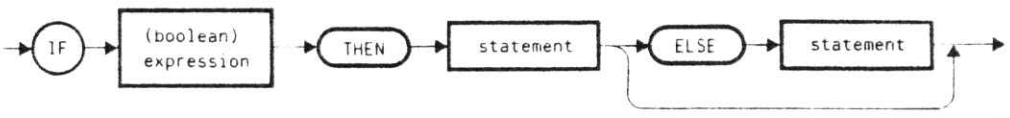


Fig. 12-7 Diagramma Sintattico per l'Istruzione IF

Se il valore dell'espressione booleana e' TRUE, allora viene eseguita l'istruzione che segue la parola chiave THEN, se invece tale valore e' FALSE viene eseguita l'istruzione che segue la parola chiave ELSE (se c'e').
In ogni caso l'esecuzione continua con l'istruzione (se c'e') successiva all'intera istruzione IF.

Come detto in precedenza, se devono essere specificate piu' istruzioni relative al ramo THEN o al ramo ELSE, e' necessario impiegare, per racchiudere tale sequenza, l'istruzione composta BEGIN ... END.

Esempi:

```
.
.
IF (A=B) OR (A=C) THEN
   WRITELN ('TRIANGOLO ISOSCELE');
WRITELN ('TRIANGOLO ANALIZZATO')
.
.
```

Se la variabile A e' uguale a B o a C viene stampata la stringa 'TRIANGOLO ISOSCELE'.
In ogni caso, cioe' anche se tale condizione non e' verificata, viene stampata la stringa 'TRIANGOLO ANALIZZATO'.

```
.
.
IF EOF (F) THEN
  WRITELN ('F E' VUOTO')
ELSE
  BEGIN
    COPYFILE (F);
    WRITELN ('F COPIATO')
  END;
WRITELN ('PROGRAMMA TERMINATO')
.
.
```

NIDIFICAZIONE DI ISTRUZIONI IF

Le istruzioni IF possono essere nidificate, cioe' l'istruzione relativa ad un ramo THEN o ELSE puo' a sua volta essere un IF.
Esempio:

```
.
.
IF C1 THEN
   IF C2 THEN
      S1
   ELSE
      S2;
S3;
.
.
```

Come si puo' notare il significato di questa istruzione e' ambiguo: infatti esso e' diverso se si associa il ramo ELSE alla condizione C1 anziche' a C2.
PASCAL+ associa sempre un ramo ELSE alla condizione piu' nidificata:

nell'esempio precedente con la condizione C2.
L'utente che volesse associare il ramo ELSE ad una condizione esterna (come C1 nell'esempio precedente) ha due possibilita':

```
IF C1 THEN
  BEGIN
   IF C2 THEN
      S1
  END
ELSE
  S2;
```

oppure:

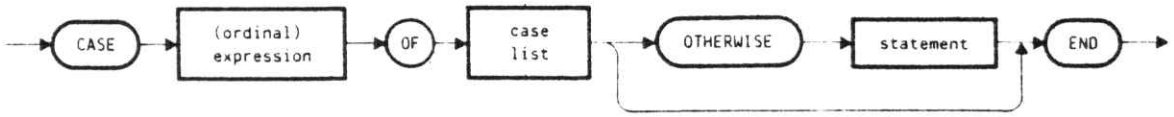
```
IF C1 THEN
  IF C2 THEN
     S1
  ELSE
ELSE
  S2;
```

Nel primo caso viene usata un'istruzione composta per racchiudere l'istruzione IF nidificata.
Nel secondo caso ci si e' serviti del ramo ELSE dell'istruzione IF interna, avente un'istruzione vuota.

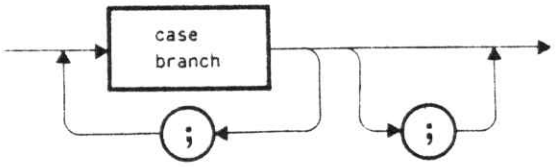
ISTRUZIONE CASE

L'istruzione CASE indica che si deve fare una scelta fra un certo numero di istruzioni, in dipendenza del valore di un'espressione chiamata "selettore".

CASE STATEMENT



CASE LIST



CASE BRANCH

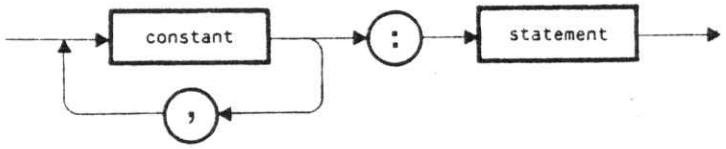


Fig. 12-8 Diagramma Sintattico per l'Istruzione CASE

Il selettore, cioe' l'espressione che segue la parola chiave CASE deve essere di tipo ordinale.

La costante (o lista di costanti) specificata per un ramo del CASE indica che l'istruzione relativa deve essere eseguita solo se il selettore assume il risultato corrispondente. Tutte le costanti devono essere dello stesso tipo del selettore e avere differenti valori.
Quando nessuna delle costanti specificate e' uguale al valore del selettore, viene eseguita l'istruzione che segue la parola chiave OTHERWISE; se tale parola chiave non e' specificata, questo caso conduce ad un errore.

Esempi:

```
.
.
CASE MONTH OF
  NOV, APR, JUN, SEP : DAYS := 30;
   FEB : IF LEAPYEAR THEN
              DAYS := 29
         ELSE
              DAYS := 28
   OTHERWISE DAYS := 31
END
.
.
CASE I+J OF
  0 : BEGIN
        PROC (P);
        A := B
  END;
1,2 : A := B
END
.
.
```

L'utente deve essere sicuro, in questo ultimo caso, che il risultato di I+J sia 0, 1 oppure 2, altrimenti si verifica un errore.

Volendo ignorare i casi rimanenti, bisogna aggiungere un ramo OTHERWISE vuoto:

```
  .
  .
CASE I + J OF
  0:BEGIN
     PROC (P);
     A:=B
  END;
1,2 : A:=B
     OTHERWISE
END
  .
  .
```

ISTRUZIONE REPEAT

L'istruzione REPEAT e' utilizzata per indicare che una sequenza di istruzioni deve essere eseguita almeno una volta, e poi ripetuta finche' l'espressione booleana seguente la parola chiave UNTIL, generi un risultato TRUE.

REPEAT STATEMENT

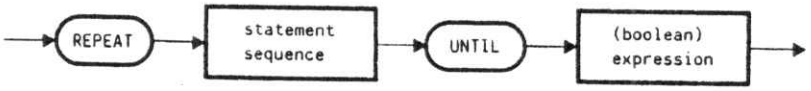


Fig. 12-9 Diagramma Sintattico per l'Istruzione REPEAT

L'espressione booleana e' valutata ad ogni iterazione. Se il risultato e' FALSE, le istruzioni della sequenza sono eseguite ancora, e l'espressione nuovamente valutata, invece se il risultato e' TRUE l'istruzione e' terminata.
Poiche' l'espressione e' valutata dopo l'esecuzione della sequenza di istruzioni, esse sono eseguite almeno una volta.
La normale iterazione puo' essere interrotta tramite le istruzioni GOTO, EXIT e RETURN, che sono descritte nei paragrafi seguenti.

Esempio:

```
.
.
RESET (F);
IF EOF (F) THEN
    WRITELN ('F VUOTO')
ELSE
    REPEAT
       ELABREC (F^ );
       GET (F)
    UNTIL EOF (F)
.
.
```

In questo esempio e' mostrato il metodo usuale di elaborare una variabile file.

ISTRUZIONE WHILE

L'istruzione WHILE e' utilizzata per indicare che un'istruzione deve essere eseguita al verificarsi di una certa condizione. Essa e' quindi ripetuta finche' tale condizione risulta vera.

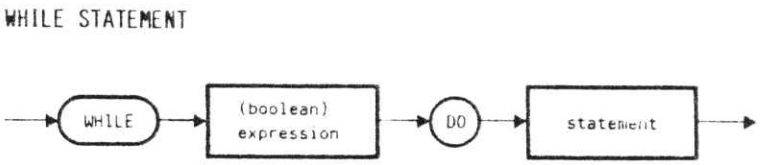


Fig. 12-10 Diagramma Sintattico per l'Istruzione WHILE

L'espressione booleana viene valutata prima di ogni iterazione. Se il valore ottenuto e' TRUE, viene eseguita l'istruzione che segue la parola chiave DO e l'espressione booleana e' valutata di nuovo; se il valore ottenuo e' FALSE, allora l'istruzione WHILE e' terminata.
Si noti che se la prima valutazione dell'espressione booleana ha come risultato il valore FALSE , l'istruzione non viene affatto eseguita.
Come per l'istruzione REPEAT, la normale iterazione puo' essere interrotta per mezzo di un'istruzione GOTO, EXIT oppure RETURN (vedere paragrafi successivi).
Esempio:

```
.
.
RESET (F);
WHILE NOT EOF (F) DO
    BEGIN
      ELABREC (F^ );
      GET (F)
    END
.
.
```

Questo esempio mostra un altro frequente modo di elaborare una variabile di tipo file, usato quando il caso in cui il file sia vuoto risulti irrilevante (in tal caso non e' eseguita alcuna operazione).
Si noti l'uso di un'istruzione composta per riunire piu' istruzioni che seguono la parola chiave DO.

ISTRUZIONE FOR

L'istruzione FOR e' utlizzata per indicare che una istruzione deve essere ripetuta per valori consecutivi di una data variabile.

FOR STATEMENT

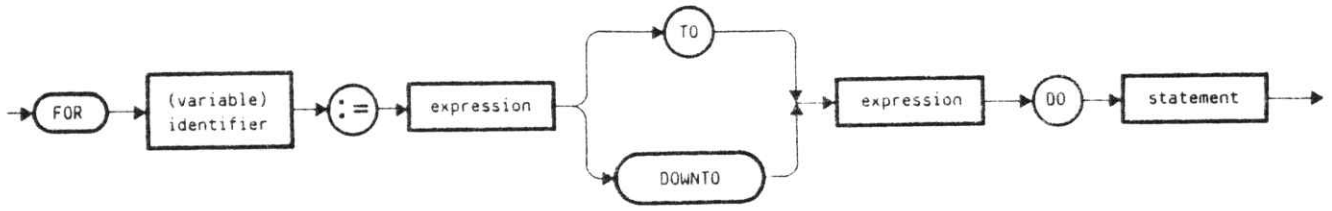


Fig. 12-11 Diagramma Sintattico per l'Istruzione FOR

L'identificatore di variabile deve fare riferimento ad una variabile dichiarata nel programma, o nella routine che propriamente contiene l'istruzione FOR, cioe' essa deve essere una variabile locale.

Questa variabile e' chiamata "variabile di controllo" dell'istruzione FOR.
La variabile di controllo deve essere di tipo ordinale. Le espressioni specificate sono chiamate rispettivamente espressione iniziale e finale, devono inoltre essere compatibili con il tipo della variabile di controllo.

Nell'eseguire un'istruzione FOR, vengono valutate le espressioni iniziale e finale.
Si possono distinguere i seguenti casi, dipendenti dall'uso della parola chiave TO oppure DOWNTO:

- uso della parola chiave TO: se l'espressione finale e' minore di quella iniziale, l'istruzione FOR termina senza che sia intrapresa alcuna azione; diversamente, l'istruzione che segue la parola chiave DO e' eseguita ripetutamente, finche' la variabile di controllo assume valori crescenti consecutivi a partire dal risultato dell'espressione iniziale a quello dell'espressione finale, estremi inclusi. A tal punto termina l'istruzione FOR.

- uso della parola chiave DOWNTO: se l'espressione finale e' maggiore di quella iniziale, l'istruzione FOR termina senza che sia intrapresa alcuna azione; in caso contrario l'istruzione che segue la parola chiave DO e' eseguita ripetutamente fino a che la variabile di controllo assume valori consecutivi decrescenti, dal risultato dell'espressione iniziale a quello dell'espressione finale, estremi inclusi.

La variabile di controllo non puo' essere alterata all'interno dell'istruzione specificata nel ramo DO. In pratica cio' significa che la variabile di controllo non puo' essere usata:

- a sinistra di un'istruzione di assegnazione;

- come parametro attuale passato per variabile (vedere Capitolo 13);

- come parametro in una chiamata di procedura READ o READLN (vedere Capitolo 13).

Come per le istruzioni REPEAT e WHILE, la normale iterazione puo' essere interrotta attraverso un'istruzione GOTO, EXIT oppure RETURN (vedere capitoli successivi).

Esempi:

```
    .
    .
FOR DAY := MON TO FRI DO
    WORKTIME[DAY] := 8;
FOR DAY := SAT TO SUN DO
    WORKTIME[DAY] := 4;
    .
    .
FOR I := J-3 DOWNTO F(X) DO
        Z := Z-J
    .
    .
```

ISTRUZIONE GOTO

L'istruzione GOTO e' utilizzata per trasferire immediatamente il punto di esecuzione del programma ad un'istruzione indicata da una etichetta (label).

GOTO STATEMENT

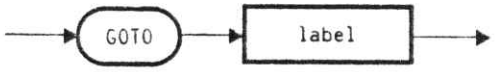


Fig. 12-12 Diagramma Sintattico per l'Istruzione GOTO

Come anticipato precedentemente, le etichette sono numeri senza segno compresi tra 0 e 9999. L'uso delle etichette e' soggetto alle seguenti regole:

- qualsiasi istruzione (anche se e' parte di un'istruzione strutturata) puo' avere una etichetta;

- le etichette devono essere uniche all'interno della parte istruzioni di un programma o di una routine;

- ogni etichetta utilizzata per identificare un'istruzione anche se non usata in un GOTO, deve essere dichiarata nella parte dichiarazione

delle etichette del programma o routine che contiene l'istruzione identificata dall'etichetta.

- un'istruzione GOTO puo' saltare ad un'etichetta solo se essa e' stata gia' dichiarate (vedere Capitolo 4).

Si noti che, a causa delle regole di visibilita' (scope rules), e' impossibile saltare dall'esterno verso l'interno della parte istruzioni di una procedura o funzione.
D'altra parte, e' possibile lasciare una routine saltando in un ambiente esterno.

Esempi:

```
PROGRAM P;
  LABEL 0,35;
  .
  .
  PROCEDURE Q;
  LABEL 35;      (* LA LABEL 35 ESTERNA E' RIDEFINITA LOCALMENTE *)
  BEGIN
   .
   .
  GOTO 35;       (* SALTO LOCALE *)
  .
  .
  35 : A := B;
  .
  .
  GOTO 0;        (* SALTO ALL'ESTERNO *)
   .
   .
END;
BEGIN            (* P *)
  Q;
  .
  .
  0 : X := Y;
  .
  .
  IF C THEN 35:Y:=X
  .
  .
  GOTO 35
  .
  .
END.             (* P *)
```

Si noti che saltando all'esterno di una routine si causa la deallocazione automatica di variabili locali.
Per maggiori chiarimenti vedere il Capitolo 6 .
L'uso dell'istruzione GOTO tende a ridurre la chiarezza del programma ed e' per tale motivo che si consiglia di evitarne l'uso.

L'utente inesperto di PASCAL+ puo' consultare la Parte III circa alcuni suggerimenti sulla programmazione senza l'uso dell'istruzione GOTO.

ISTRUZIONE EXIT

L'istruzione EXIT e' usata per interrompere prima del previsto un'istruzione iterativa, come ad esempio un REPEAT, WHILE o FOR.

EXIT STATEMENT

Fig. 12-13 Diagramma Sintattico per l'Istruzione EXIT

Un'istruzione EXIT puo' essere utilizzata solo all'interno di un REPEAT, WHILE o FOR ed ha la funzione di far terminare immediatamente l'istruzione ripetitiva piu' interna che la contiene.

Se EXIT e' usata per interrompere un'istruzione FOR, la variabile di controllo assume il valore dell'ultima iterazione effettuata.

Esempio:

```
        (* RICERCA SEQUENZIALE IN UN ARRAY *)
    .
    .
VAR A:ARRAY [1..10] OF T;
    KEY : T;
    I : 1..10;
  .
  .
FOR I := 1 TO 10 DO
    IF A[I] = KEY THEN EXIT;
.
.
```

ISTRUZIONE RETURN

L'istruzione RETURN e' utilizzata per interrompere l'attivita' di una routine, inoltre puo' essere usata anche in un programma principale.

RETURN STATEMENT

Fig. 12-14 Diagramma Sintattico per l'Istruzione RETURN

L'esecuzione dell'istruzione RETURN, all'interno della parte istruzione di una routine, provoca il ritorno immediato al punto chiamante.
Esempio:

```
PROGRAM P;
      .
      .
   PROCEDURE Q;
      .
      .
   BEGIN
      .
      .
      IF ERROR THEN RETURN;
      .
   END;
BEGIN
      .
      .
      Q;
      .
      .
   END.
```

Se si incontra l'istruzione RETURN nel programma principale, viene eseguito un ritorno al sistema.

ISTRUZIONE WITH

L'istruzione WITH e' utilizzata per evitare l'uso del nome della variabile (record, modulo, monitor, unita' principale, oggetto istanziabile), che precede la denotazione di singoli campi.

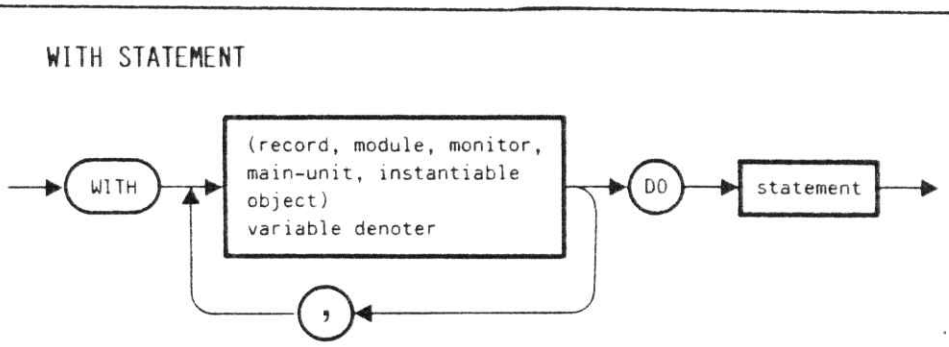


Fig. 12-15 Diagramma Sintattico per l'Istruzione WITH

All'interno dell'istruzione che segue la parola chiave DO, gli identificatori di campo possono essere usati senza prefisso per accedere a singoli campi.

Esempio:

```
.
.
VAR R : RECORD
     A : INTEGER;
     B : ARRAY[1..10] OF BOOLEAN
   END;
.
.
WITH R DO
 BEGIN
   A:=5;        (* SI ACCEDE A R.A *)
   B[10]:=TRUE (* SI ACCEDE A R.B *)
 END;
.
.
```

Qualora sorgano ambiguita' tra identificatori di campo ed altri identificatori, viene data priorita' all'identificatore di campo:

```
    .
    .
VAR R : RECORD
        A : INTEGER;
        B : ARRAY[1..10] OF BOOLEAN
      END;
  A : INTEGER;
.
.
A:=5;       (* SI ACCEDE ALLA VARIABILE A *)
WITH R DO
  A:=5      (* SI ACCEDE A R.A *)
.
.
```

Quando viene indicata piu' di una variabile record, si ha l'effetto equivalente della nidificazione di piu' istruzioni WITH:

```
    .
    .
VAR R1 : RECORD
         A : INTEGER;
         B : ARRAY [1..10] OF BOOLEAN
       END;
    R2 : RECORD
         A : CHAR;
         C : REAL
       END;
.
.
WITH R1, R2 DO
 BEGIN
   A:='Z';
   C:=3.5;
   B[1]:=TRUE
 END;
```

cio' e' equivalente a:

```
WITH R1 DO
 WITH R2 DO
  BEGIN
    A:='Z';      (* SI ACCEDE A R2.A *)
    C:=3.5;      (* SI ACCEDE A R2.C *)
    B[1]:=TRUE   (* SI ACCEDE A R1.B *)
  END;
.
.
```

Si noti che ogni ambiguita', dovuta ai campi appartenenti a differenti record con il medesimo identificatore, e superata assegnando priorita' all'istruzione WITH piu' interna (o, equivalentemente, all'ultima variabile record della lista).
Per annullare tale priorita' deve essere usata la forma estesa (come nel caso precedente R1.A).

13. PROCEDURE E FUNZIONI

Una delle piu' importanti caratteristiche del linguaggio PASCAL+ e' la possibilita' di sviluppare i programmi attraverso macchine astratte, cioe' attraverso "istruzioni ad alto livello", che eseguono operazioni complesse (vedere capitolo 3).

In PASCAL+ le istruzioni ad alto livello sono descritte tramite le procedure e le funzioni.

Le procedure e le funzioni (routine) devono essere considerate risorse piuttosto che parti di un programma.

In questo capitolo viene illustrata la struttura interna di una routine facendo distinzione tra il ruolo delle procedure e quello delle funzioni. Si ha anche una spiegazione completa sul passaggio dei parametri, in quanto essi costituiscono il mezzo di comunicazione tra una routine ed il proprio ambiente.

Inoltre viene fornita una lista completa delle procedure e delle funzioni predefinite.

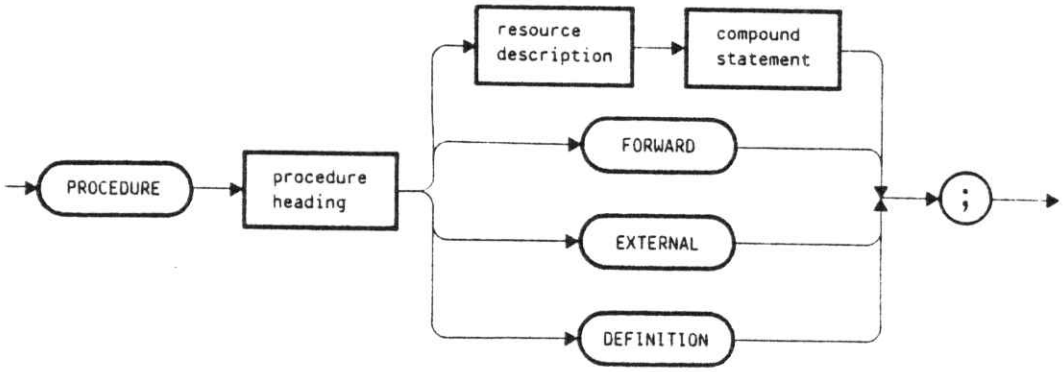
DESCRIZIONE DI UNA PROCEDURA

La descrizione di procedura e' il mezzo per assegnare un identificatore ad un algoritmo.

Poiche' le procedure sono risorse di un programma, la descrizione di una procedura e' collocata nella parte di "Descrizione delle Risorse" di un programma (vedere nel Capitolo 7, il paragrafo "Unita' Principale di Programma").

Il seguente diagramma sintattico mostra la struttura interna della descrizione di procedura:

PROCEDURE DESCRIPTION



PROCEDURE HEADING

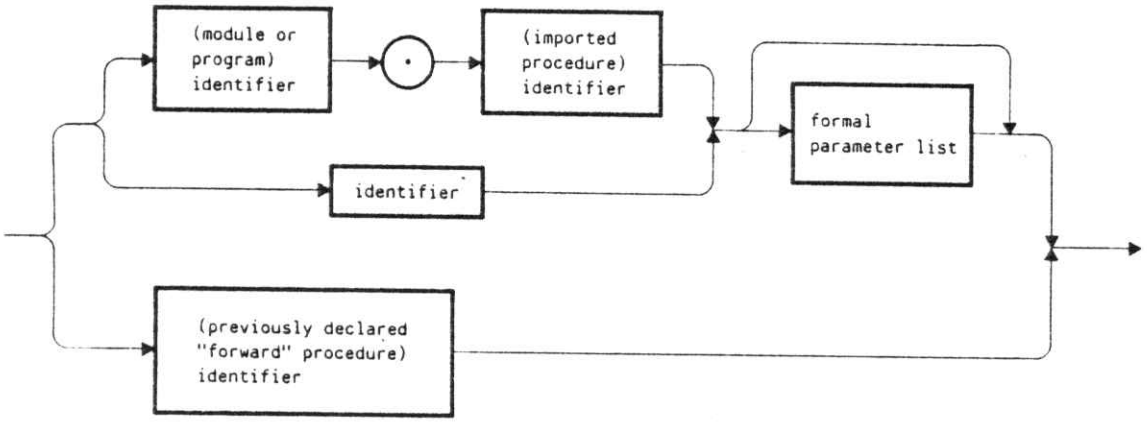


Fig. 13-1 Diagramma Sintattico di una Descrizione di Procedura

Esistono quattro tipi di descrizione di procedura:

. normale;

. futura;

. esterna;

. importata.

Nella descrizione "normale" la struttura di una procedura e' esattamente simile alla struttura dell'intero programma: viene assegnato un identificatore seguito da una lista di parametri formali (se ve ne sono) e seguita dalla descrizione delle risorse interne e dalla parte istruzioni.

Esempio:

```
.
.
PROCEDURE ESEMPIO (I : INTEGER);
  VAR A : REAL;
  BEGIN
   A := I+2
  END;
.
.
```

La descrizione di procedura "futura" e' composta solo da una intestazione seguita dalla parola chiave FORWARD. Sono assegnati il nome della procedura ed i parametri formali.
La parola chiave FORWARD asserisce che la descrizione completa apparira' nel testo piu' avanti.

Esempio:

```
.
.
PROCEDURE ARRIVEDERCI (I : INTEGER); FORWARD;
.
.
PROCEDURE ARRIVEDERCI;
  VAR A : ...
  BEGIN
  .
  .
  END;
```

La descrizione completa della procedura e' simile alla normale descrizione con l'eccezione che deve contenere lo stesso identificatore (nome di procedura) utilizzato nella descrizione "forward" e non vi si deve ripetere la lista dei parametri formali.

La descrizione "forward" e quella completa devono apparire allo stesso livello di nidificazione delle risorse, cioe' esse non possono comparire all'interno di routine differenti.
Il meccanismo "forward" permette le chiamate mutuamente ricorsive (vedere nel Capitolo 4, il paragrafo "Routine Ricorsive").

La descrizione di procedura "esterna" e' composta solo dall'intestazione seguita dalla parola chiave EXTERNAL.

Esempio:

```
.
.
PROCEDURA SALVE (I : INTEGER); EXTERNAL;
.
.
```

Una procedura esterna puo' essere scritta in qualsiasi linguaggio e deve essere collegata (tramite linker) al programma.
L'associazione tra il nome della procedura interna ed esterna ed i parametri dipende dalla implementazione ed e' descritta nella relativa documentazione.

La descrizione di una procedura "importata" e' composta dalla sola intestazione seguita dalla parola chiave DEFINITION; il nome della procedura e' preceduto dal nome del programma o del modulo e dal simbolo ".".
La parola chiave DEFINITION afferma che l'intera descrizione della procedura e' contenuta nell'unita' importata (vedere Capitolo 14, Ridefinizione ed Uso di Risorse Importate).

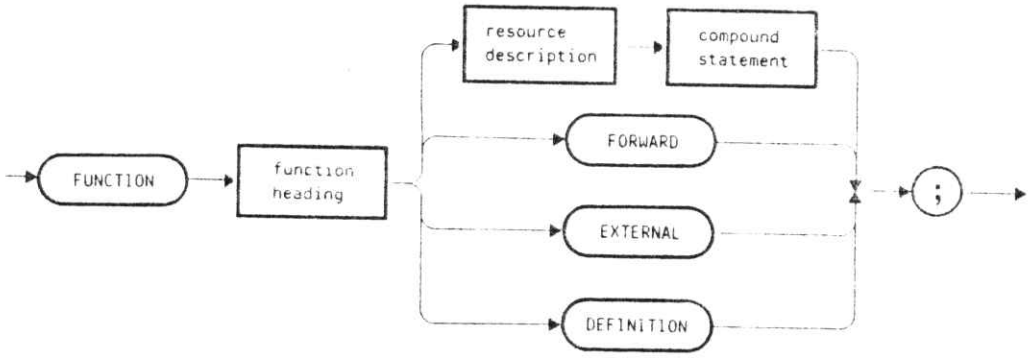
Esempio:

```
.
.
IMPORT PUSH FROM STACK;
.
.
PROCEDURE STACK.PUSH (I : INTEGER); DEFINITION;
.
.
```

DESCRIZIONE DI FUNZIONE

Una funzione PASCAL+ non differisce dal concetto matematico di funzione: e' un'entita' che produce uno ed un solo valore.
Una funzione PASCAL+ e' paragonabile sia ad una procedura che ad una variabile: e' strutturata come una procedura in quanto contiene una descrizione delle risorse interne e una parte istruzioni, ma gli e' assegnato un tipo che corrisponde al tipo del risultato.
Il diagramma sintattico di una descrizione di funzione e' del tutto analogo a quello della descrizione di procedura, ad eccezione del fatto che contiene la specificazione del tipo del risultato.

FUNCTION DESCRIPTION



FUNCTION HEADING

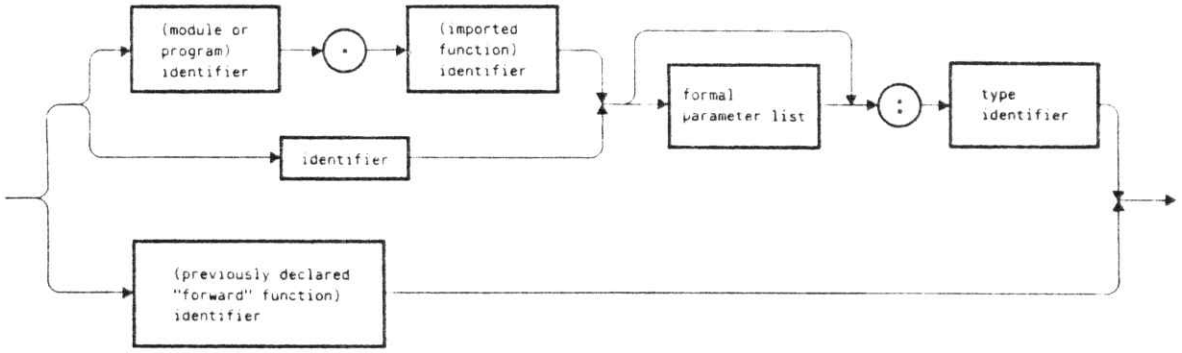


Fig. 13-2 Diagramma Sintattico della Descrizione di Funzione

Il significato delle descrizioni di funzione normale, futura, esterna, ed importata e' lo stesso visto per le procedure.

Un esempio di descrizione di funzione puo' essere il seguente:

```
.
.
FUNCTION POWER (X,Y : INTEGER) : INTEGER;
  VAR I : INTEGER;
  BEGIN
  .
  .
  END;
  .
  .
```

CHIAMATA ED ESECUZIONE DI PROCEDURE E FUNZIONI

Una procedura o funzione (routine) costituisce una risorsa per un programma (o per una routine), cosi' come lo sono le variabili, le costanti, i tipi e cosi' via.

Percio' l'invocazione di una routine non differisce sintatticamente dalla denotazione di altre risorse: si deve semplicemente citare il suo nome.

L'"invocazione" di una routine sara' detta "chiamata di routine" invece di "denotatore di routine" poiche' essa causa un salto del controllo all'esecuzione delle istruzioni interne ed al corrispondente salto di ritorno.
La chiamata di procedura e' costituita da una istruzione ed e' descritta nel Capitolo 12 (istruzione di Chiamata di Procedura); la chiamata di funzione, invece, e' costituita da un operando all'interno di una espressione ed e' descritta nel Capitolo 11 (Operandi).

A differenza di altri linguaggi di programmazione, le chiamate di routine non sono introdotte da alcuna parola chiave; questo rende il linguaggio PASCAL+ molto simile al linguaggio naturale.
Per esempio, un programma che gioca a scacchi, puo' essere formulato in PASCAL+ nel seguente modo:

```
PROGRAM SCACCHI;
PROCEDURE CONSIDERA_MOSSA_AVVERSARIA;
 .
 .
PROCEDURE CALCOLA_PROSSIMA_MOSSA;
 .
 .
PROCEDURE ESEGUI_PROSSIMA_MOSSA;
 .
 .
 .
 .
FUNCTION PARTITA_TERMINATA : BOOLEAN;
  .
  .
BEGIN
  .
  .
  REPEAT
     CONSIDERA_MOSSA_AVVERSARIA;
     CALCOLA_PROSSIMA_MOSSA;
     ESEGUI_PROSSIMA_MOSSA
   UNTIL PARTITA_TERMINATA;
    .
    .
END.
```

Si noti la somiglianza con l'esempio visto nel Capitolo 3 (Un Esempio di Macchina Astratta).

Si cerca ora di ampliare l'argomento sulle chiamate di funzione.

All'interno del corpo di una chiamata di funzione (cioe' le istruzioni interne alla descrizione di funzione), si puo' usare il nome della funzione sia come se fosse una variabile locale, sia come se fosse una chiamata di funzione.

Usare il nome della funzione come oggetto di un'assegnazione, significa assegnare un valore alla funzione; per esempio:

```
FUNCTION POWER (X,Y : INTEGER) : INTEGER;
  BEGIN
  .
  .
  POWER := .... (* assegnazione alla funzione *)
  .
  .
END.
```

Anche se una funzione puo' avere come risultato un valore di tipo strutturato, tuttavia la funzione diretta con indice cosi' come la scelta del campo della funzione sono illegali.

Usare il nome di una funzione in una espressione significa effettuare una chiamata ricorsiva a se stessa; per esempio:

```
FUNCTION POWER (X,Y INTEGER) : INTEGER;
  VAR I : INTEGER;
BEGIN
  .
  .
  I:=2 + POWER (5,3); (* chiamata di funzione *)
  .
  .
END.
```

Il valore prodotto da una funzione dopo una "chiamata" e' l'ultimo valore assegnato all'interno; il corpo di una funzione deve sempre contenere almeno un'assegnazione, altrimenti il suo valore e' indefinito.

Nell'esempio seguente:

```
FUNCTION BUON_GIORNO : CHAR;
  BEGIN
    BUON_GIORNO := 'A';
    BUON_GIORNO := 'B';
    BUON_GIORNO := 'Z'
  END;
.
.
C := BUON_GIORNO;
.
.
```

Il valore di BUON-GIORNO assegnato a C e' 'Z'.

All'esterno del corpo della funzione non sono permesse le assegnazioni di funzione, ma solo le sue chiamate.

L'esempio seguente mostra una funzione ricorsiva che calcola il fattoriale di un numero intero (si tenga presente che il fattoriale di un numero X e' definito come il prodotto tra X ed il fattoriale di X - 1):

```
FUNCTION FACT(X : INTEGER) : INTEGER;
  BEGIN
  IF X> 0 THEN
     FACT := X * FACT (X-1)
  ELSE
     FACT := 1
  END;
```

L'esecuzione di una routine termina quando si giunge alla parola chiave finale "END" oppure quando si incontra l'istruzione "RETURN".
Questo implica la deallocazione delle variabili locali (parametri inclusi) dal Run Time Stack (vedere Capitolo 6 "L'Effetto di una Chiamata di Routine"), ed il salto indietro del controllo all'istruzione di chiamata.

PASSAGGIO DI PARAMETRI

Per assicurare una vasta generalita' di applicazione, le procedure e le funzioni possono avere dei parametri.

I parametri sono un particolare tipo di risorse locali, poste in corrispondenza con alcune risorse dell'ambiente chiamante.
Possono essere pensati come "risorse finestra", attraverso cui vengono scambiate informazioni tra l'ambiente chiamante e la routine chiamata.

I parametri sono utilizzati quando la stessa azione deve essere eseguita piu' volte su risorse differenti: in tal caso viene usata una routine che esegue tali azioni con dei parametri anziche' delle normali risorse locali.

Si consideri le risorse locali nell'esempio seguente:

```
PROCEDURE GIOVANNI;
  VAR V: INTEGER;
  FUNCTION C : INTEGER;
  PROCEDURE P;
  .
  .
 BEGIN
  .
  .
 END;
```

Una chiamata a questa procedura implica l'esecuzione di alcune operazioni con l'uso delle risorse V, C, P; ma, per ciascuna chiamata, il valore della funzione C e' sempre uguale, l'azione eseguita da P e' la stessa ed

il contenuto della variabile V e' inizialmente indefinito.

Ad esempio, se si vuole che la procedura operi, a causa di differenti chiamate, su variabili diverse, con differenti valori di C, e con differenti algoritmi per P, si devono definire tali risorse come parametri anziche' come normali risorse locali:

```
PROCEDURA GIOVANNI (VAR V : INTEGER; FUNCTION C : INTEGER;
                    PROCEDURE P);
BEGIN
.
.
END;
```

La descrizione delle risorse che appare tra parentesi e' chiamata "lista dei parametri formali".
Al momento della chiamata di questa procedura, deve essere fornita una lista delle risorse corrispondenti: per esempio se si chiama la procedura nel modo seguente:

```
GIOVANNI (K,PRED(259),SALVE);
```

durante la sua esecuzione, la variabile V coincide con K, la funzione C vale 258 e la procedura P esegue la stessa azione di "SALVE".

D'altra parte se la procedura GIOVANNI viene chiamata in questo modo:

```
GIOVANNI (Z [3] .F,SUCC(1),PRINT);
```

allora, durante la sua esecuzione, V coincide con Z [3] .F, C produce 2 e P effettua una stampa.
All'interno della routine chiamata, i parametri sono utilizzati allo stesso modo delle normali variabili, procedure e funzioni, poiche' esse costituiscono risorse locali.

L'elenco che appare tra parentesi nella chiamata di routine e' detto "lista dei parametri attuali".

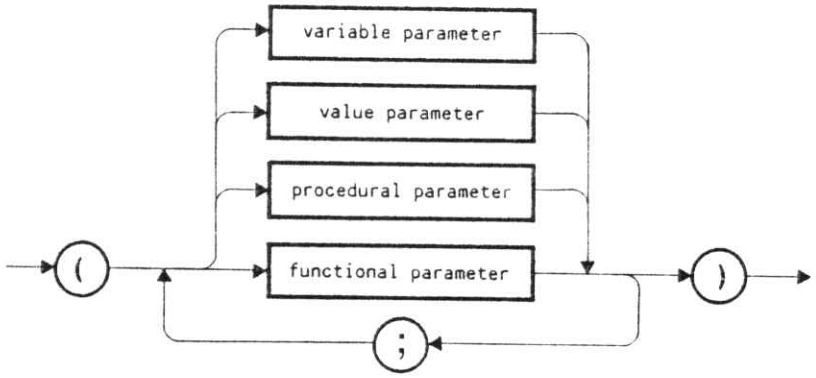
L'associazione fra parametri formali ed attuali e' effettuata considerando la loro posizione nella lista: il primo parametro attuale corrisponde al primo parametro formale e cosi' via per gli altri.
Ovviamente i due elenchi devono essere congruenti, cioe' devono contenere lo stesso numero di elementi ed i loro tipi devono essere compatibili.
Il PASCAL+ fornisce quattro tipi di parametri:

- variabile;
- valore;
- procedurale;
- funzionale.

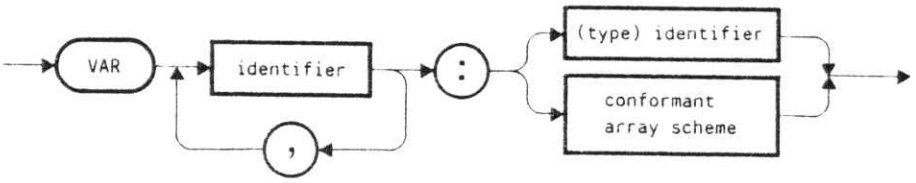
In PASCAL+ non possono essere passati tipi oppure etichette in qualita' di parametri.

Il diagramma sintattico della lista dei parametri formali e' il seguente:

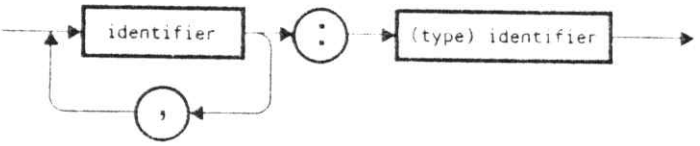
FORMAL PARAMETER LIST



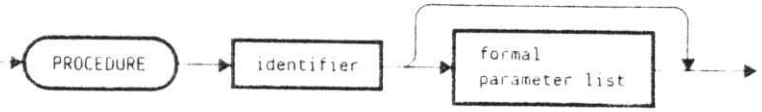
VARIABLE PARAMETER



VALUE PARAMETER



PROCEDURAL PARAMETER



FUNCTIONAL PARAMETER

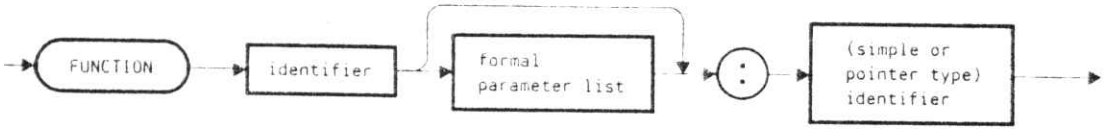


Fig. 13-3 Diagramma Sintattico della Lista dei Parametri Formali

Ogni tipo di parametro formale sara' spiegato in dettaglio piu' avanti.

Si consideri ora il diagramma sintattico corrispondente alla lista dei parametri attuali:

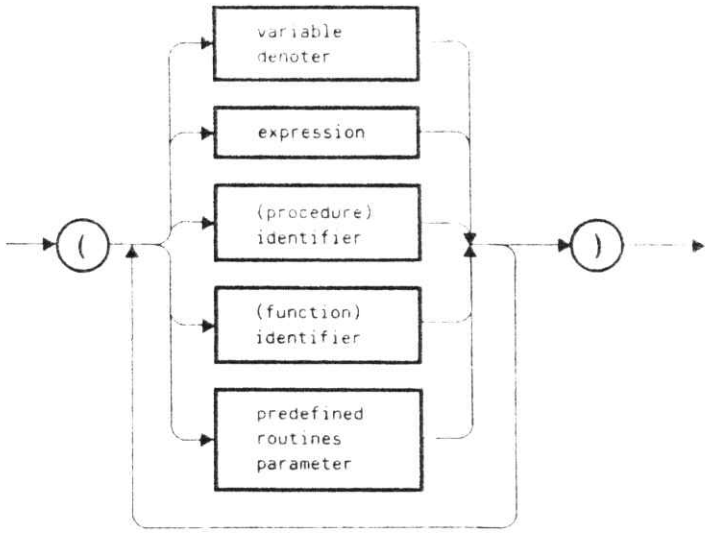


Fig. 13-4 Diagramma Sintattico della Lista dei Parametri Attuali

Questo diagramma sintattico fa riferimento ai diagrammi sintattici per la "istruzione di chiamata di procedura" e per la "chiamata di funzione".
I "parametri di routine predefinite" che appaiono nel diagramma sintattico precedente, sono descritti piu' avanti, a proposito delle routine predefinite.

I "parametri per variabile" agiscono come canale di input/output per la routine: essi permettono lo scambio di informazioni tra routine ed ambiente chiamante e viceversa.
Tutti gli altri tipi di parametri (valore, procedurale, funzionale) agiscono come solo canale di input: essi permettono solo che una informazione sia fornita alla routine ma non viceversa.

Segue ora una descrizione dettagliata dei quattro tipi di parametri.

Parametri per Variabile

Un parametro per variabile e' il mezzo con cui si assegna ad una variabile uno pseudonimo.
Il contenuto della variabile puo' essere utilizzato o modificato riferendosi ad esso con il nome formale.

Esempio:

```
PROCEDURE INCR (VAR I : INTEGER);
  BEGIN
  I := I+1
 END;
```

Quando si chiama una routine, il parametro attuale deve essere un indicatore di variabile dello stesso tipo del parametro.

Esempio:

```
VAR Z : INTEGER;
PROCEDURE INCR (VAR I : INTEGER);
  BEGIN
  I := I+1
  END;
BEGIN
 Z := 5;
 INCR (Z) (* adesso Z=6 *)
END.
```

I componenti di variabili di tipo strutturato, descritte come packed non possono essere usati come parametri attuali per variabile.

Parametri per Valore

I parametri per valore sono il mezzo con cui vengono passati dei valori alla routine chiamata dall'ambiente chiamante.
I parametri attuali corrispondenti devono essere espressioni di un tipo compatibile per assegnazione con il tipo di parametri formali.
I parametri per valore possono essere pensati come variabili locali alla routine chiamata, inizializzati al momento della chiamata con il valore delle corrispondenti espressioni.
All'interno della routine essi possono essere usati come normali variabili locali, ma, se il loro contenuto e' modificato, il corrispondente parametro attuale non subisce la modifica.

Esempio:

```
VAR Z : REAL;
PROCEDURE X (A,B : REAL);
  BEGIN
  A := A+B
  END;
BEGIN
Z := 3.5;
X (Z,10)
(* Z rimane = 3.5 *).
```

Parametri Procedurali

Un parametro procedurale e' il mezzo che permette di passare un algoritmo come un parametro dall'ambiente chiamante alla routine chiamata.
Il parametro attuale corrispondente deve essere un identificatore di una procedura descritta al livello piu' esterno (il programma).
La lista dei parametri della procedura attuale deve essere congruenta con la lista dei parametri della procedura formale, cioe' le due liste di parametri devono contenere lo stesso numero di parametri, e il tipo e la posizione devono essere corrispondenti.

Alle procedure predefinite non e' consentito essere passate come parametri.

Esempio:

```
PROCEDURE X(VAR A : T1; B,C : INTEGER);
 .
 .
PROCEDURE ESEMPIO (PROCEDURE P (VAR A1 : T1; B1, C1 : INTEGER));

  VAR Z : T1;
  BEGIN
   P (Z, 3, 2-3 * 8);
    .
    .
  END;
BEGIN
 .
 .
 ESEMPIO (X);
  .
  .
END.
```

I parametri procedurali occupano una posizione irrilevante di memoria, poiche' essi consistono fisicamente in un riferimento alla procedura passata al momento della chiamata come corrispondenti parametri attuali, e possono essere invocati, all'interno della routine chiamata, come procedure normali.

Segue ora un esempio completo:

```
PROGRAM P (F1, F2);
 TYPE CONFIGURAZIONE = ARRAY[1..10] OF INTEGER;
      SITUAZIONE : FILE OF CONFIGURAZIONE;
 VAR F1,F2 : SITUAZIONE;

 PROCEDURE STAMPA (X : CONFIGURAZIONE);
   VAR I : 1..10;
   BEGIN
    FOR I := 1 TO 10 DO WRITE (X[I],'');
    WRITELN
   END;

PROCEDURE SUM (X : CONFIGURAZIONE);
  VAR : 1..10; J : INTEGER;
  BEGIN
   J := 0;
   FOR I := 1 TO 10 DO J := J + X[I];
   WRITELN (J)
  END;

PROCEDURE MAX (X : CONFIGURAZIONE);
  VAR K : 1..10; M : INTEGER;
  BEGIN
    M := -MAXINTEGER;
    FOR K := 1 TO 10 DO
     IF A[K] > M THEN
        M := A[K];
    WRITELN (M)
  END;

PROCEDURE ELAB_FILE (VAR F : SITUAZIONE;
          PROCEDURE ELAB_CONF (X : CONFIGURAZIONE));
 BEGIN
  RESET (F);
  WHILE NOT EOF (F) DO BEGIN
  ELAB_CONF (F Ω );
  GET (F) END
 END;

BEGIN (* relativo al programma principale *)
 ELAB_FILE (F1, STAMPA);
  .
  .
 ELAB_FILE (F2, MAX);
  .
  .
 ELAB_FILE (F1, SUM);
  .
  .
END.
```

Nell'esempio precedente e' stata costruita una procedura, ELAB_FILE, che scandisce un certo file F, elaborando ciascun elemento con una procedura parametrica ELAB_CONF.
Quando e' chiamata ELAB_FILE sia il file che la procedura che devono

essere usati sono specificati come parametri attuali.

Parametri Funzionali

I parametri funzionali sono del tutto analoghi ai parametri procedurali. Il tipo del risultato dei parametri funzionali deve essere fornito nella lista dei parametri formali e deve essere uguale al corrispondente tipo del parametro attuale.
Alle funzioni predefinite non e' consentito essere passate come parametri.

Esempio:

```
FUNCTION F : BOOLEAN;
  BEGIN
  .
  .
  END;

PROCEDURE ESEMPIO (FUNCTION A : BOOLEAN);
  BEGIN
  IF A THEN...
 END;
BEGIN
 .
 .
 ESEMPIO (F);
END;
```

Si noti che un parametro funzionale attuale puo' essere sintatticamente uguale ad un valore attuale oppure ad un parametro costante:

```
FUNCTION F : T;
 .
 .
PROCEDURE A (FUNCTION F : T);
 .
 .
PROCEDURE B (X:T);
 .
 .
BEGIN
 .
 .
 A (F);
 B (F);
 .
 .
END.
```

Quando A e' chiamata, la funzione F e' passata come parametro; quando e' chiamata B, il valore risultante di F e' passato come parametro, poiche' B accetta un parametro per valore.

Schema di Conformant Array

Se un parametro per variabile e' di tipo array, la dimensione dell'array puo' essere parametrizzata.

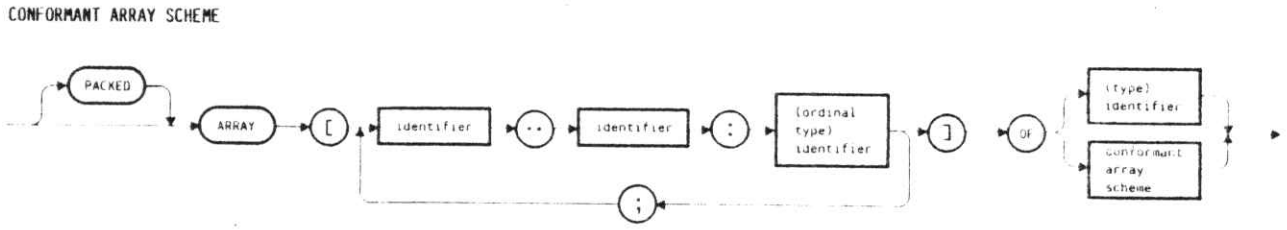


Fig. 13-5 Diagramma Sintattico dello Schema di Conformant Array

Il tipo indice di un parametro array e' preceduto da due identificatori che rappresentano il limite superiore ed inferiore dell'indice di array; per esempio:

```
ARRAY [A..B : COLOUR] OF...
```

Nell'esempio precedente, A e B possono essere pensati come un particolare tipo di parametro per valore, senza un corrispondente parametro attuale ed automaticamente inizializzati con il limite superiore ed inferiore dell'indice del parametro attuale array.

In una routine essi possono essere usati come normali parametri per valore.

Esempio:

```
PROCEDURE P (VAR A : ARRAY [M..N : INTEGER] OF CHAR);
 VAR I : INTEGER;
 BEGIN
  FOR I := M TO N DO
  ELAB (A[I])
END;
```

Il corrispondente parametro attuale deve essere un identificatore di array; se e' "packed"; il parametro formale deve essere anch'esso definito "packed".
Gli schemi di conformant array permettono l'elaborazione di array

indipendentemente dalle loro dimensioni. L'esempio che segue mostra una procedura generalizzata per copiare un file in un array:

```
PROGRAM SUSAN;
 TYPE T =...
 VAR A1 : ARRAY [1..2] OF T;
     A2 : ARRAY [2000..2030] OF T;
     A3 : ARRAY [-18..500] OF T;
     F : FILE OF T

 PROCEDURE COPY (VAR A : ARRAY [M..N : INTEGER] OF T);
  VAR I : INTEGER;
 BEGIN
  RESET (F);
  FOR I := M TO N DO
    IF NOT EOF (F) THEN
    BEGIN
     A [I] := F ^ ;
     GET (F)
     END
END;
BEGIN
  COPY (A1);
  COPY (A2);
  COPY (A3);
  .
  .
END.
```

La procedura "COPY" copia il contenuto del file F in un array parametrizzato; la prima chiamata alla procedura COPY causa la copia nell'array A1. Gli identificatori di limite M e N saranno inizializzati rispettivamente ad 1 e 2, essendo essi i limiti dell'indice di A1.

La seconda chiamata causa una copia su A2 ed M e N avranno i valori 2000 e 2030.
Analogamente, durante la terza chiamata, M e N avranno i valori -18 e 500 rispettivamente.

PROCEDURE E FUNZIONI PREDEFINITE PER LA GESTIONE DI FILE

Un file e' una struttura dati costituita da una sequenza di valori appartenenti allo stesso tipo.
Un file non puo' avere chiavi; l'accesso ad un file e' solo sequenziale.

La gestione di un file e' effettuata attraverso le seguenti procedure predefinite: REWRITE, PUT, RESET, GET, CLOSE, PAGE, READ, READLN, WRITE, WRITELN ed attraverso le funzioni predefinite EOF ed EOLN.
Le procedure READ e WRITE hanno diverso effetto a seconda che il file sia o non sia un file-testo.
Le procedure READLN e WRITELN sono usate solo per la gestione di file-testo.

Un file puo' essere elaborato:

- in "inspection mode" (INPUT)

- in "generation mode" (OUTPUT)

- in "inspection/generation mode" (UPDATE). (A tal proposito vedere Capitolo 22).

Quando un file e' elaborato in INPUT si assume che sia composto da una sequenza indefinita di valori preesistenti (eventualmente una sequenza vuota).
Quando un file e' elaborato in OUTPUT si assume che verra' generata una sequenza di valori (inizialmente vuota) con l'aggiunta di un nuovo elemento istante per istante.
Nelle descrizioni che seguono di queste procedure, F rappresenta una qualsiasi variabile file ed F^ e' la variabile buffer del file.

REWRITE (F)

La procedura REWRITE pone un file F in "generation mode" (OUTPUT).
Il file e' inizializzato a "file vuoto" ed il valore della funzione EOF(F) e' posto a TRUE.

Attenzione: l'eventuale contenuto precedente del file viene cancellato totalmente. La variabile buffer file F^ e' indefinita.

REWRITE (F,s)

Questa procedura e' molto simile alla procedura REWRITE (F), tranne che per il parametro "s".
Tale parametro indica un'espressione di tipo stringa che specifica il nome esterno del file su cui deve essere effettuata l'operazione.

PUT(F)

La procedura PUT deve essere usata su di un file posto in "generation mode".
Essa permette l'aggiunta di un nuovo elemento nel file. Il valore del nuovo elemento aggiunto e' quello della variabile F^ .
Dopo l'operazione F^ e' indefinito.

Esempio:

```
VAR NUMBER : FILE OF INTEGER;
 .
 .
REWRITE (NUMBER, 'NUM');
NUMBER ^ := 13;
PUT (NUMBER);
(* il primo elemento del file ha il valore 13 *)
```

RESET (F)

La procedura RESET pone il file F in "inspection mode" (INPUT) e posiziona F^ sul primo elemento del file, se esso e' presente.
Se il file F e' vuoto il valore di EOF(F) e' posto a TRUE, ed F^ e' indefinito.
Se il file F non e' vuoto il valore di EOF(F) e' posto a FALSE ed F^ contiene il valore del primo elemento di F.

RESET (F,s)

Questa procedura e' simile alla RESET(F), ad eccezione del parametro "s". Tale parametro indica un'espressione di tipo stringa che specifica il nome esterno del file su cui deve essere eseguita l'operazione.

GET (F)

La procedura GET e' usata per posizionarsi sul prossimo elemento del file, che e' in "inspection mode".
Se non esiste alcun prossimo elemento, il valore della funzione EOF(F) e' posto a TRUE e la variabile F^ e' indefinita.
Se esiste un prossimo elemento, il suo valore e' reso disponibile in F^ ed EOF(F) e' posto a FALSE.

Esempio:

```
VAR NUMBER : FILE OF INTEGER;
(* si supponga che il file NUM contenga i valori 1,2,3... *)
A,B : INTEGER;
  .
  .
RESET (NUMBER, 'NUM');
A := NUMBER ^ ;
(* A contiene 1 *)
GET (NUMBER);
B := NUMBER ^ ;
(* B contiene 2 *).
```

EOF (F)

La funzione predefinita EOF e' usata per verificare se si e' incontrato l'ultimo elemento della variabile file.
Il risultato di EOF e' di tipo booleano.
La funzione vale TRUE nei seguenti casi:

. dopo una procedura RESET quando il file e' vuoto;

. dopo una procedura GET qualora si incontri la fine del file

. quando il file e' in "generation mode".

Se EOF(F) e' TRUE la variabile F^ e' indefinita.
Si verifica un errore se EOF(F) e' utilizzata quando F e' indefinito (cioe' non e' effettuata ne' una RESET ne' una REWRITE).

Esempio:

```
VAR NUMBER : FILE OF INTEGER;
(* si supponga che il file NUM contenga 5 elementi *)
   .
   .
RESET (NUMBER, 'NUM');
REPEAT
 A := NUMBER ^ ;
   .
   .
 GET (NUMBER);
UNTIL EOF (NUMBER);
(* EOF diventa TRUE ed il ciclo e' interrotto dopo la quinta occor-
 renza della procedura GET *)
```

CLOSE (F)

La procedura CLOSE deve essere usata quando si vuol chiudere un file.
Non e' strettamente necessario usarla quando il file F compare nella intestazione del programma.
In tal caso, infatti, e' effettuata una implicita chiusura del file al termine del programma.
La variabile file e' chiusa implicitamente e riaperta anche quando il "file mode" e' modificato attraverso una chiamata di RESET o REWRITE.

READ (F, V1,.., Vn)

Quando una variabile F non e' un file-testo, la procedura predefinita READ e' definita come segue.
F denota una variabile di tipo FILE OF T e V1,..,Vn indicano variabili di tipo T (o compatibile).
Il file deve essere in "inspection mode", altrimenti si ha un errore.

La procedura READ assegna valori contenuti nel file alle variabili specificate.

READ (F,V) equivale a:

```
BEGIN
V := F ^ ;
GET (F)
END.
```

e READ (F, V1,..., Vn) equivale a:

```
BEGIN
READ (F,V1);
 .
 .
READ (F, Vn)
END.
```

Valgono le seguenti condizioni in accordo con le definizioni precedenti:

- se, a partire dalla posizione corrente (prima di una READ), il numero di elementi del file e' superiore ad n, allora sono assegnati n elementi ad n variabili; F^ contiene il valore dell'n+1-esimo elemento; EOF (F) e' FALSE;

- se il numero di elementi e' n, essi sono assegnati ad n variabili; F^ e' indefinito; EOF (F) e' TRUE;

- se il numero di elementi e' inferiore ad n, si ha errore.

Esempio:

```
VAR NUMBER : FILE OF INTEGER;
    A,B,C,D : INTEGER;
      .
      .
RESET (NUMBER, 'NUM');
READ (NUMBER, A,B,C,D);
(* A contiene il primo elemento di NUMBER, B il secondo, C
il terzo e D il quarto *).
```

WRITE (F, E1,..,En)

Quando la variabile F non e' un file-testo, la procedura predefinita WRITE e' definita come segue.

F indica una variabile di tipo FILE OF T ed E1,..,En indicano espressioni di tipo compatibile per assegnazione con T. Il file deve essere in "generation mode", altrimenti si ha errore.
La procedura WRITE permette di aggiungere al file i valori prodotti dalle specifiche espressioni.

WRITE (F,E) e' equivalente a:

```
BEGIN
F^ := E;
PUT (F)
END.
```

e WRITE (F,E1,...,En) e' equivalente a:

```
BEGIN
WRITE (F,E1);
  .
  .
WRITE (F,En);
END.
```

Esempio:

```
VAR NUMBER : FILE OF INTEGER;
   A : INTEGER;
      .
      .
A := 13;
REWRITE (NUMBER, 'NUM');
WRITE (NUMBER, A, 5);
(* il file NUMBER contiene due elementi: 13, 5 *).
```

File-Testo

Vengono ora descritte le procedure READ, READLN, WRITE, WRITELN, PAGE e la funzione EOLN utilizzate per la gestione dei file-testo.

READ (F,V)

F e' una variabile di tipo TEXT e V puo' essere una variabile di:

. tipo CHAR (o un subrange);

. tipo INTEGER (o un subrange);

. tipo REAL.

F deve essere in "inspection mode".
READ (V) e' una espressione equivalente a READ (INPUT, V): la procedura e' applicata al file-testo predefinito INPUT.

La procedura READ assegna a variabili specificate i valori la cui rappresentazione simbolica (sequenza di caratteri) e' ottenuta da F.
La procedura READ (F,V) opera nel modo seguente:

- se V e' di tipo CHAR essa e' equivalente a:

```
BEGIN
 V := F^ ;
 GET (F)
END;
```

- se V e' di tipo INTEGER (o subrange) oppure di tipo REAL:

  . la procedura READ (F,V) effettua la lettura di una sequenza di caratteri che formano un intero con segno oppure un numero con segno (se V e' di tipo real);

  . il valore dell'intero o numero con segno (se V e' di tipo real) e' assegnato alla variabile la quale deve essere compatibile per assegnazione con il valore letto;

  . Vengono saltati sia gli spazi che precedono che le terminazioni di linea;

- la lettura della sequenza dei caratteri ha termine quando si incontra un carattere che non fa parte dell'intero con segno o del numero con segno (se V e' di tipo real);

- si ha errore quando il primo carattere letto non fa parte dell'intero o del numero con segno.

La procedura READ puo' essere usata nella forma READ (F,V1,..,Vn). Questa forma e' equivalente a:

```
BEGIN
 READ (F,V1)
  .
  .
 READ (F,Vn)
END.
```

Esempio:

Si consideri il seguente file di input (INF):

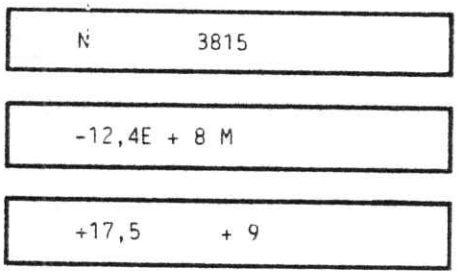


Fig. 13-6 Un Semplice File-Testo

```
VAR INF : TEXT;
   X,Y : INTEGER;
   R,S : REAL;
   A,B : CHAR;
    .
    .
RESET (INF, 'INFORM');
READ (INF, A, X);
(* A assume il valore 'N' *)
(* X assume il valore 3815 *)
   .
   .
READ (INF,R);
(* R assume il valore -12.4E+8 *)
   .
   .
READ (INF, B,B,S);
(* B assume il valore 'M' *)
(* B e' letta due volte per saltare lo spazio dopo -12.4E+8 *)
(* S assume il valore +17.5 *)
   .
   .
READ (INF,Y);
(* Y assume il valore +9 *)
   .
   .
```

READLN (F)

F e' una variabile di tipo TEXT.
READLN senza parametri equivale a READLN (INPUT): la procedura e' applicata al file-testo predefinito INPUT. La procedura READLN senza parametri permette di saltare alla prossima riga nel file.
La posizione corrente nel file dopo la READLN costituisce l'inizio della prossima riga, a meno che si incontri la fine del file.
La procedura READLN puo' essere utilizzata nella forma: READLN (F,V1,..,Vn), dove V1,..,Vn hanno lo stesso significato descritto per la READ (F,V1,..,Vn).
Usando questa forma, il salto di spazio e' effettuato dopo ogni lettura delle variabili.

READLN (F,V1,..,Vn) e' equivalente a:

```
BEGIN
 READ (F,V1,..,Vn);
 READLN (F)
END.
```

Esempio:

si consideri il seguente file di input (INF):

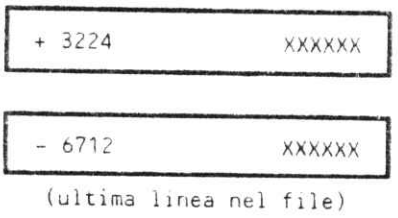


Fig. 13-7 Esempio di File-Testo

```
VAR INF : TEXT;
  X,Y : INTEGER;
      .
      .
RESET (INF, 'INFORM');
READLN (INF,X);
(* X assume il valore +3224, la parte rimanente della riga e'
saltata; INF e '-' *)
READLN (INF,Y);
(* Y assume il valore -6712, la parte rimanente della riga e'
saltata; poiche' questa e' l'ultima riga. La funzione EOF
(INF) e' posta al valore TRUE *).
```

WRITE (F,P)

F e' una variabile di tipo TEXT.
F deve essere in "generation mode".
La forma WRITE(P) e' equivalente a WRITE (OUTPUT,P): la procedura e' applicata al file-testo predefinito OUTPUT.
P puo' assumere una delle seguenti forme:
E
E : lunghezza totale
E : lunghezza totale: cifre decimali.

Dove:

- E e' una espressione di integer, real, char, boolean o stringa;

- la procedura WRITE inserisce nel file-testo determinato una sequenza di caratteri che corrisponde alla rappresentazione simbolica del valore dell'espressione specificata;

- la "lunghezza totale" definisce il numero di caratteri che sono scritti sul file-testo, con un numero appropriato di spazi alla sinistra della rappresentazione di E. Nel caso in cui E richieda piu' dei caratteri specificati per la rappresentazione, viene scritto il numero necessario di caratteri per la sua rappresentazione.

WRITE(F,E) e' equivalente alla forma WRITE(F,E: lunghezza totale), usando come "lunghezza totale" un valore di default che dipende dal tipo di E. Per i tipi real, integer e boolean, questo valore dipende dall'implementazione; per il tipo char il valore e' 1;

- le "cifre decimali" definiscono il numero di cifre che devono seguire la virgola; deve essere usato solo se E e' di tipo real.

Vengono ora considerati quattro casi differenti a seconda del tipo di E.

E di tipo CHAR

- WRITE(F,E) e' equivalente a WRITE(F,E:1) il valore di default della "lunghezza totale' e' 1; la rappresentazione sul file-testo e' il valore del carattere E;

- WRITE(F,E:N) e' equivalente a:

```
BEGIN
 FOR J := 1 TO N-1 DO
  BEGIN
   F^ := ' ';
   PUT (F)
  END;
 F^ := E;
 PUT (F)
END
```

cioe' la rappresentazione scritta sul file-testo e' la "lunghezza totale" -1 di spazi seguita dal valore del carattere E.

E di tipo INTEGER

- Sul file-testo viene scritta una rappresentazione decimale di E;

- se la "lunghezza totale" e' piu' grande del numero di cifre di cui e' costituito E, la stringa scritta sul file di output sara' formata da un numero di spazi cosi' calcolato: (lunghezza totale - numero di cifre - 1) seguito da un altro spazio (se E >= 0) o dal segno '-' (se E < 0), seguito dalla rappresentazione decimale senza segno di E.

- se la "lunghezza totale" non e' maggiore del numero di cifre di cui e' costituito E, la stringa scritta sul file di output sara' formata dal segno '-' (se E < 0) seguito dalla rappresentazione decimale senza segno di E (senza considerare l'allineamento).

E di tipo REAL

- Sul file-testo viene scritta la rappresentazione decimale del numero E.

- si possono distinguere due casi: la rappresentazione in virgola mobile e in virgola fissa.

Rappresentazione in virgola mobile:

- WRITE (F,E:lunghezza totale) causa la scrittura della rappresentazione in virgola mobile di E;

- la rappresentazione in virgola mobile di E consiste di: il segno '-' se E < 0, altrimenti uno spazio; la prima cifra del numero, seguita dal simbolo '.' e dalle cifre ricavate dal calcolo: (lunghezza totale - cifre dell'esponente di E); il carattere di esponente 'E' oppure 'e', il segno dell'esponente '+' o '-' e le cifre dell'esponente;

- se la lunghezza totale e' minore del numero di caratteri necessari, il numero viene scritto senza alcun riguardo per l'allineamento.

Rappresentazione in virgola fissa:

- WRITE (F,E: lunghezza totale: cifre decimali) causa la scrittura in virgola fissa di E;

- la rappresentazione in virgola fissa di E e' costituita da alcune cifre intere seguite dal carattere '.' e da un numero di cifre decimali equivalente al parametro "cifre decimali" e preceduto dal segno '-' se E < 0, e dal numero di spazi necessari a soddisfare il parametro "lunghezza totale" specificato;

- se la "lunghezza totale" e' inferiore al numero di caratteri necessari, il numero viene scritto senza badare all'allineamento.

E di tipo BOOLEAN

- Sul file-testo viene scritta la parola TRUE o FALSE, in accordo con il valore di E.

  Esempio:

```
.
.
A := B;
.
.
WRITE (F,A = B, ' ', A <> B)
```

  sul file F sara' scritta la seguente sequenza di caratteri: "TRUE FALSE".

WRITELN (F)

F e' una variabile di tipo TEXT.
WRITELN senza parametri equivale a WRITELN (OUTPUT): la procedura e' applicata al file-testo predefinito OUTPUT.
Il file F deve essere in "generation mode".
La procedura WRITELN permette il salto alla prossima linea del file-testo che si deve generare.
La semantica della procedura WRITELN dipende dall'implementazione e dal tipo di periferica. La procedura WRITELN puo' essere usata nella forma: WRITELN (F,P1,...Pn), dove P1,...Pn hanno lo stesso significato descritto

per la procedura WRITE (F,P1,...,Pn).
Usando questa forma il salto e' effettuato dopo la scrittura della espressione specificata.

WRITELN (F,P1,...,Pn) e' equivalente a:

```
BEGIN
 WRITE (F,P1,...,Pn);
 WRITELN (F)
END.
```

EOLN (F)

E' una variabile di tipo TEXT.
La funzione EOLN e' di tipo boolean ed assume il valore TRUE se l'ultimo carattere letto del file F e' l'ultimo della riga, altrimenti assume il valore FALSE. Si ha errore se il testo F e' indefinito oppure se EOF(F) assume il valore vero.
La funzione EOLN senza alcun parametro e' equivalente a EOLN (INPUT).

PAGE (F)

F e' una variabile di tipo TEXT. Il file F deve essere in "generation mode". PAGE senza parametri e' equivalente a PAGE (OUTPUT); la procedura e' applicata al file-testo predefinito OUTPUT.
La procedura PAGE permette il salto di una pagina nel file-testo specificato; la semantica della procedura PAGE dipende interamente dall'implementazione e dal tipo di periferica.

PROCEDURE PREDEFINITE PER L'ALLOCAZIONE DINAMICA

NEW (P)

Il parametro P e' una variabile puntatore, cioe' una variabile di tipo ^T.
La procedura NEW alloca nello HEAP (vedere Capitolo 6) una nuova variabile di tipo T, il cui valore iniziale e' indefinito.
A P e' assegnato un valore che permette l'accesso alla variabile allocata attraverso la notazione P^. Si verifica un errore qualora non ci sia spazio nello HEAP per l'allocazione di una nuova variabile.
Se il tipo T contiene varianti e' possibile allocare dello spazio adattato per le corrispondenti varianti.

Si consideri il seguente esempio:

```
TYPE T = RECORD
       CASE BOOLEAN OF
        TRUE : (R,S : REAL);
        FALSE : (X : INTEGER)
     END,
VAR P : ^T;
```

Usando la forma normale della NEW(P) viene allocato lo spazio per la variante piu' lunga, mentre specificando un valore del campo discriminante e' possibile modificare la dimensione dello spazio allocato; NEW(P,FALSE) ad esempio alloca lo spazio per un intero.
Questo tipo di allocazione puo' essere applicato anche per varianti nidificate.

Si consideri il seguente esempio:

```
TYPE T = RECORD
       CASE Z : 'A'..'B' OF
       'A' : (Q,R,S : REAL);
       'B' : (CASE X : 1..2 OF
              1 : (T,V : REAL);
              2 : (W : CHAR)
             END)
       END;
VAR P = ^ T;
```

Usando la forma normale NEW(P) viene allocato lo spazio per la variante piu' lunga.
Sono pero' permesse anche le forme seguenti:

- NEW(P,'B') per l'allocazione di spazio per la piu' lunga variante di X;

- NEW(P,'B',2) per l'allocazione di spazio per la variante piu' corta.

Le costanti del CASE devono essere elencate in ordine crescente di nidificazione delle parti varianti.
Ogni variante non specificata deve essere ad un livello piu' profondo della nidificazione specificata all'ultima posizione dell'elenco.

DISPOSE (P)

Il parametro P e' una variabile puntatore. La procedura DISPOSE rilascia l'area di HEAP allocata a P^ .
La variabile P^ non e' piu' accessibile.
Il contenuto di P ed ogni altro puntatore che si riferisce a questa variabile diventa indefinito.
L'area di HEAP rilasciata diviene accessibile per ulteriori allocazioni.
Se la variabile P^ e' stata allocata usando la forma: NEW(P,costante 1,..., costante n) la procedura DISPOSE deve essere usata nella seguente forma:
DISPOSE(P,costante 1, ... ,costante n).
Si applicano le medesime regole viste per la procedura NEW.

Esempio:

```
TYPE V = RECORD
       CASE Z : 'A'..'B' OF
       'A' : (Q,R,S : REAL);
       'B' : (CASE X : 1..2 OF
              1 : (T,U : REAL);
              2 : (W : CHAR)
             END)
       END;
VAR P : ^ V;
      .
      .
NEW (P,'B',2);
      .
      .
DISPOSE (P,'B',2).
```

FUNZIONI ARITMETICHE PREDEFINITE

Nel linguaggio PASCAL+ si hanno le seguenti funzioni aritmetiche predefinite: ABS, SQR, SIN, COS, EXP, LN, SQRT, ARCTAN.
Il valore della espressione X nella descrizione delle funzioni che seguono puo' essere sia di tipo real che di tipo integer.

ABS (X)

La funzione calcola il valore assoluto di X.
Il risultato e' dello stesso tipo del parametro X.

Esempi:

```
ABS (-5) da' 5
ABS (+4.0) da' 4.0
ABS (-3.5) da' 3.5
```

SQR (X)

La funzione calcola il quadrato di X.
Il risultato e' dello stesso tipo del parametro X.
Esempio:

```
SQR (-3) da' 9
SQR (0.32E+1) da' 10.24
SQR (3.0) da' 9.0
```

SIN (X)

La funzione calcola il seno trigonometrico di X radianti.
Il risultato e' di tipo reale.
Esempi:

```
SIN (0.5) da' 0.4794255
SIN (3.4) da' -0.2555411.
```

COS (X)

La funzione calcola il coseno trigonometrico di X radianti.
Il risultato e' di tipo reale.
Esempi:

```
COS (0.5) da' 0.8775826
COS (-2) da' -0.4161468
```

EXP (X)

La funzione calcola il valore del numero di Nepero 'e' elevato alla potenza X. Il risultato e' di tipo reale.
Esempi:

```
EXP (5) da' 148.41316
EXP (0.2) da' 1.2214028
```

LN (X)

La funzione calcola il logaritmo naturale di X, dove X deve essere maggiore di zero.
Il risultato e' di tipo reale.
Esempi:

```
LN (0.5) da' -0.6931472
LN (1) da'0.0
```

SQRT (X)

La funzione calcola la radice quadrata positiva di X, dove X non deve essere negativo.
Il risultato e' di tipo reale.
Esempi:

```
SQRT (10.24) da' 3.2
SQRT (16) da' 4.0
```

ARCTAN (X)

La funzione calcola, in radianti, l'arcotangente di X.
Il risultato e' di tipo reale.
Esempi:

```
ARCTAN (1) da' 0.7853982
ARCTAN (1.5) da' 0.9827937.
```

PROCEDURE E FUNZIONI DI CONVERSIONE PREDEFINITE

TRUNC (X)

L'espressione X deve essere di tipo reale.
Il risultato e' di tipo intero.
La funzione tronca X alla sua parte intera.

Esempi:

```
TRUNC (5.6) da' 5
TRUNC (-2.8) da' -2
TRUNC (6.0) da' 6
```

ROUND (X)

L'espressione X deve essere di tipo reale.
Il risultato e' di tipo intero.
La funzione calcola l'intero piu' prossimo ad X. Si noti che tale valore e' definito nei modi seguenti:

. se X > = 0 allora ROUND(X) = TUNC (X+0.5)

. se X < = 0 allora ROUND(X) = TRUNC (X-0.5)

Esempi:

```
ROUND (5.8) da' 6
ROUND (5.3) da' 5
ROUND (-3.8) da' -4
ROUND (-3.2) da' -3
ROUND (18.0) da' 18
ROUND (18.5) da' 19.
```

PACK (A,I,Z)

Il parametro A e' una variabile dichiarata come ARRAY [T1] di T.
Il parametro I e' una variabile di tipo T1.
Il parametro Z e' una variabile dichiarata come PACKED ARRAY [T2] di T.
La procedura trasferisce ogni elemento dell'array disimpaccato A a partire dall'I-esimo elemento nell'array impaccato Z a partire dal primo elemento di Z, finche' Z e' riempito.
Assumendo:

```
TYPE T1 = S..T;
   T2 = U..V;
.
.
VAR J : T2;
   I,K : T1;
```

l'effetto di PACK (A,I,Z) e' equivalente a:

```
K:= I;
FOR J := U TO V DO
  BEGIN
  Z[J] := A[K];
  IF J<>V THEN
     K := SUCC (K)
  END
```

Si noti che e' richiesta la condizione

ORD(V) - ORD(U) < = ORD(T) - ORD(I)

Esempio:

```
VAR A : ARRAY [1..5] OF INTEGER;
    Z : PACKED ARRAY [2..4] OF INTEGER;
```

Se A e Z sono:

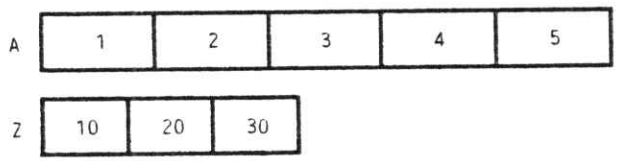


Fig. 13-8 Gli Array A e Z Prima dell'Uso della Procedura PACK

L'uso della PACK (A,2,Z) ha l'effetto:

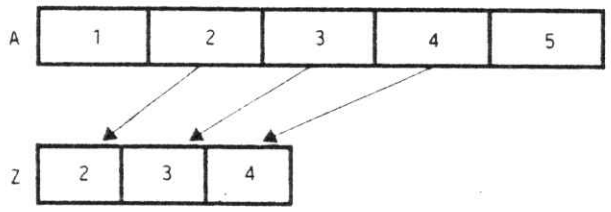


Fig. 13-9 Effetto della Procedura PACK

UNPACK (Z,A,I)

Z, A e I hanno lo stesso significato descritto per la procedura PACK.
La procedura trasferisce ogni elemento dell'array impaccato Z, a partire dal primo elemento, nell'array disimpaccato A, partendo dall'elemento I-esimo, finche' Z e' esaurito.

Assumendo:

```
TYPE T1 = S..T;
     T2 = U..V;
VAR J : T2;
    I,K : T1;
```

l'effetto della UNPACK (Z,A,I) e' equivalente a:

```
K := I
FOR J := U TO V DO
 BEGIN
  A[K] := Z[J];
  IF J <> V THEN
     K := SUCC (K)
  END
```

Facendo in modo che:

```
ORD (V) - ORD (U) < = ORD (T) - ORD (I)
```

Esempio:

```
VAR A : ARRAY [1..5] OF INTEGER;
    Z : PACKED ARRAY [2..4] OF INTEGER;
```

Se A e Z sono cosi' fatti:

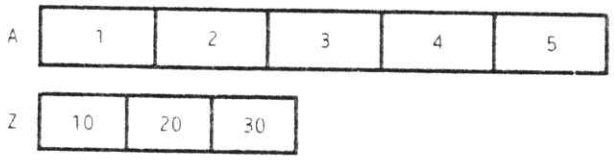


Fig. 13-10 Gli Array A e Z Prima dell'Uso della UNPACK

L'uso della procedura UNPACK (Z,A,3) ha il seguente effetto:

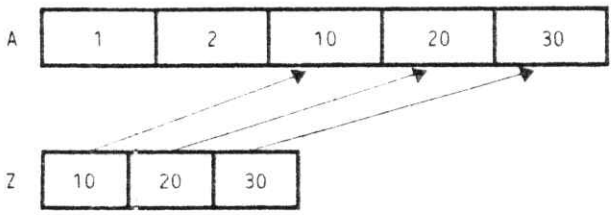


Fig. 13-11 Effetto della Procedura UNPACK

FUNZIONI ORDINALI PREDEFINITE

ORD (X)

Il parametro X deve essere una espressione di tipo ordinale (vedere Capitolo 8, Tipi Semplici).
Il risultato e' di tipo intero (> = 0).
La funzione da' un numero ordinale associato ad X appartenente all'insieme di costanti che definisce il tipo di X.

Esempi:

```
VAR Y: INTEGER;
.
.
Y:= ORD('B')
(* il valore di Y e' 66, assumendo il codice ASCII*)

TYPE GIORNI = (DOM, LUN, MAR, MER, GIO, VEN, SAB);
VAR OGGI : GIORNI; Y: INTEGER;
.
.
OGGI:=LUN;
Y:=ORD(OGGI);
(*il valore di Y e' 1*)
```

CHR (X)

Il parametro X deve essere una espressione intera (> = 0). Il risultato e' di tipo carattere.
La funzione da' come valore di ritorno il carattere associato al numero ordinale X.
La relazione tra un carattere ed il suo valore ordinale dipende dall'implementazione.

Esempio:

```
VAR Z: CHAR;
.
.
Z:=CHR(66);
(*il valore di Z e' 'B' assumendo il codice ASCII*)
```

SUCC (X)

Il parametro X deve essere una espressione di tipo ordinale. Il risultato e' dello stesso tipo del parametro X. La funzione da' un valore il cui numero ordinale e' maggiore di uno rispetto a quello dell'espressione X.
Si incorre in un errore se il successore di X non esiste.

Esempi:

```
TYPE GIORNI = (DOM, LUN, MAR, MER, GIO, VEN, SAB);
VAR OGGI: GIORNI;
.
.
OGGI:=SUCC(DOM);
(*il valore di OGGI e' LUN*).

VAR Z:CHAR;
.
.
Z:=SUCC('A');
(*il valore di Z e' 'B'*).
```

PRED (X)

Il parametro X deve essere una espressione di tipo ordinale. Il risultato e' dello stesso tipo di X.
La funzione ha come valore di ritorno quello il cui numero ordinale e' inferiore di uno rispetto a quello dell'espressione X.
Si incorre in un errore se non esiste il predecessore di X.

Esempi:

```
TYPE GIORNI = (DOM, LUN, MAR, MER, GIO, VEN, SAB);
VAR OGGI : GIORNI;
.
.
OGGI:=PRED(LUN);
(*il valore di OGGI e' DOM*).

VAR Z:CHAR;
 .
 .
Z:=PRED('N');
(*il valore di Z e' 'M'*).
```

ODD (X)

Il parametro X deve essere una espressione di tipo intero. Il risultato e' di booleano.
La funzione vale:

TRUE se X e' dispari
FALSE se X e' pari

Si noti che la funzione assegna il valore dell'espressione:
ABS (X) MOD2 = 1.
Esempio:

ODD (5) da' TRUE
ODD (-18) da' FALSE.

14. MODULI

In questo capitolo viene mostrata la struttura di un modulo ed il meccanismo di import/export.

Inoltre viene illustrata l'allocazione dell'area di memoria e la visibilita' delle risorse per i moduli "importati".

Alcuni dettagli, descritti negli altri capitoli, sono qui ripetuti al fine di dare una completa spiegazione dell'uso dei moduli.

Sono riportati esempi, che mostrano i due modi principali di usare i moduli: le routine esterne ed i tipi di dati incapsulati.

COSA E' UN MODULO

"MODULE" e' il nome PASCAL+ che denota una libreria di risorse. Un modulo e' il mezzo con cui viene isolato un gruppo di risorse dall'ambiente esterno. Le risorse contenute in un modulo sono invisibili alle altre unita' di compilazione, a meno che siano esplicitamente esportate e importate.

Se una risorsa definita nel modulo e' esportata esplicitamente, ad essa ci si puo' riferire da qualsiasi altra unita' di compilazione che la importa. La struttura interna di un modulo e' mostrata nel Capitolo 7 (Diagramma Sintattico di Moduli): essa e' costituita essenzialmente da una serie di "descrizione di risorse"

In un modulo non esiste alcuna "parte istruzioni" poiche' esso e' , un componente passivo: esso non e' eseguito, bensi' sono eseguite le sue risorse attraverso programmi attivi.

LISTA DI ESPORTAZIONE

La lista di esportazione specifica quali risorse di un modulo o di un programma sono visibili all'ambiente esterno.

Si possono importare in altre unita' (moduli o programmi) solo risorse esportate; il riferimento alle risorse non esportate e' possibile esclusivamente all'interno dell'unita' che le definisce.
Il diagramma sintattico relativo alla lista di esportazione e' il

seguente:

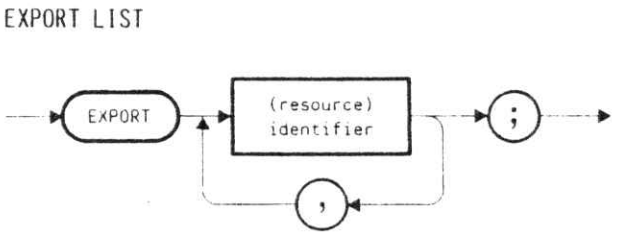


Fig. 14-1 Diagramma Sintattico per la Lista di Esportazione

Le uniche risorse esportabili sono:

- costanti
- tipi
- variabili
- procedure
- funzioni.

Non e' permessa l'esportazione di etichette (labels) o di risorse importate.

La lista di esportazione, quando e' presente, deve apparire solo al livello piu' esterno della nidificazione di risorse, cioe' non puo' apparire all'interno di una procedura o di una funzione.

La presenza di una lista di esportazione non implica interventi addizionali run-time del sistema, in quanto si tratta di un meccanismo di compile-time.

Un esempio di lista di esportazione e' il seguente (vedere Capitolo 7

"Tipi di Dati Incapsulati"):

```
MODULE STACK_OF_CHAR;
CONST C =...;
TYPE T =...;
VAR TOPMOST:...;
    HOW_MANY:...;
PROCEDURE PUSH (I:CHAR);
 .
 .
PROCEDURE POP (VAR I:CHAR);
 .
 .
PROCEDURE FAULT;
 .
 .
FUNCTION EMPTY:BOOLEAN;
 .
 .
PROCEDURE INIT;
 .
 .
EXPORT C, PUSH, POP, EMPTY, INIT, HOW_MANY;.
```

Si noti che il tipo T, la variabile TOPMOST e la procedura FAULT non sono esportate, percio' esse non sono importabili da alcuna unita'.

LISTA DI IMPORTAZIONE

La lista di importazione specifica quali risorse, descritte esternamente, sono utilizzate da un modulo o da un programma (unit).

Il diagramma sintattico e' il seguente:

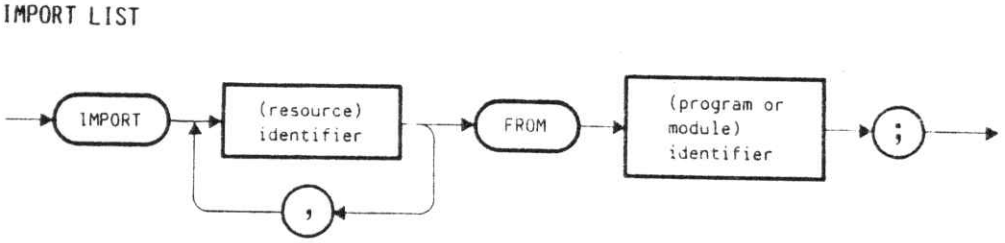


Fig. 14-2 Diagramma Sintattico della Lista di Importazione

L'identificatore che segue la parola chiave "FROM" e' il nome dell'unita' che contiene la descrizione delle risorse elencate.
Ovviamente, una risorsa che non e' esportata dall'unita' che la definisce, non puo' essere importata.

La lista di importazione, se presente, deve apparire al livello piu' esterno della nidificazione delle risorse, cioe' non puo' apparire all'interno di una procedura o funzione.

L'associazione tra risorse importate ed esportate e' effettuata tramite i nomi (a livello di compilazione collegamento o esecuzione, in dipendenza dell'implementazione): la lista di importazione deve contenere gli identificatori di risorse identici a quelli che appaiono nella lista di esportazione della unita' che li definisce.
L'ordine di apparizione degli identificatori nella lista di importazione e' irrilevante.

Se si vogliono importare delle risorse da unita' diverse, si devono citare piu' liste di importazione.

Gli esempi che seguono mostrano alcune liste di importazione: (con riferimento al modulo STACK_OF_CHAR visto in precedenza):

```
PROGRAM EXAMPLE;
  .
  .
IMPORT PUSH, INIT, HOW_MANY FROM STACK_OF_CHAR;
  .
  .
                    MODULE OTHER_EXAMPLE;
                      .
                      .
                    IMPORT INIT, POP, C FROM STACK_OF_CHAR;
                      .
                      .
```

RIDEFINIZIONE ED USO DI RISORSE IMPORTATE

Le risorse importate possono essere:

- costanti
- tipi
- variabili
- procedure
- funzioni.

Nell'unita' che effettua un'importazione, il riferimento alle risorse importate e' effettuato tramite il loro nome preceduto dal nome dell'unita' che le definisce e dal simbolo ".".
Per esempio, se si importa la procedura P dall'unita' M, il riferimento ad essa sara' M.P. (vedere diagramma sintattico per denotatore di costante, di tipo, di variabile, definizione e chiamata di procedura e funzione).

Le variabili, procedure e funzioni importate devono essere ridefinite nell'unita' che le importa; le costanti ed i tipi, al contrario, non devono essere ridefiniti e possono essere usati normalmente.

La ridefinizione di variabili importate non differisce dalla normale definizione di variabile se non nella sintassi dell'identificatore (nella forma M.V).

Esempio:

```
MODULE ESEMPIO;
 .
 .
IMPORT OGGI FROM CALENDARIO;
 .
 .
VAR CALENDARIO.OGGI: (DOM, LUN, MAR, MER, GIO, VEN, SAB);
 .
 .
BEGIN
 .
 .
IF CALENDARIO.OGGI = DOM THEN NON_LAVORO;
 .
 .
END.
```

La ridefinizione di procedure e funzioni importate differisce da quella normale in quanto e' costituita solo dalla intestazione seguita dalla parola chiave "DEFINITION".

IMPORTED ROUTINE REDEFINITION

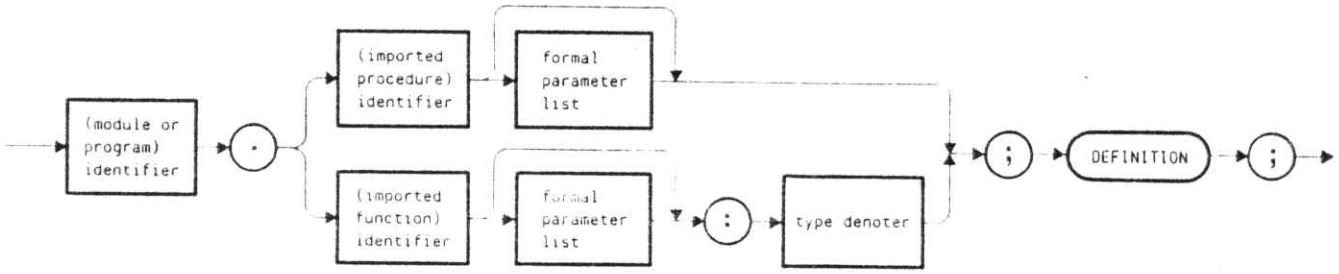


Fig. 14-3 Diagramma Sintattico per la Ridefinizione di Routine Importate

Esempio (con riferimento all'esempio STACK_OF_CHAR):

```
PROGRAM EXAMPLE;
IMPORT POP,EMPTY,HOW_MANY,C FROM STACK_OF_CHAR;
VAR STACK_OF_CHAR.HOW_MANY:INTEGER;
PROCEDURE STACK_OF_CHAR.POP(VAR I:CHAR);DEFINITION;
FUNCTION STACK_OF_CHAR.EMPTY:BOOLEAN;DEFINITION;
(*Nota: C non deve essere ridefinito*)
  .
  .
```

La ridefinizione delle risorse serve per poter effettuare un controllo sui tipi: cio' permette una compilazione separata significativa.

Attenzione: la ridefinizione delle risorse importate deve essere compatibile con la corrispondente definizione nell'unita' che le esporta. Non e' effettuato alcun controllo a livello di compilazione o di collegamento (linking).
Ad esempio, se si esporta una variabile di tipo intero, e la si ridefinisce di tipo reale nell'unita' che la importa, si incorre in un errore di esecuzione, che il sistema puo' eventualmente non segnalare.

```
MODULE M;
 VAR I : INTEGER;
 EXPORT I;

MODULE P;
  IMPORT I FROM M;
  VAR M.I : REAL; (*errore*)
```

ESEMPI

Si considerano ora due esempi di disegno di moduli, che si riferisconoa due principali modalita' di uso (vedere Capitolo 7 "Librerie di Risorse"):

- routine separate;

- tipi di dati incapsulati.

Routine separate

L'esempio seguente mostra un modulo contenente un insieme di routine per la gestione di stampa.

```
MODULE PRINT;
  CONST LPP = 50;
   HEAD = 'REPORT PRINTING; PAGE';
  TYPE LINE = PACKED ARRAY[1..80] OF CHAR;
  VAR PAGES : INTEGER;
   LINES : 0..LPP;
   PRINTER : TEXT;

  PROCEDURE HEADPRINT;
   BEGIN
    PAGES := PAGES + 1;
    WRITELN (PRINTER, HEAD, PAGES : 5);
    LINES := 0
   END;

PROCEDURE START;
  BEGIN
   PAGES := 0;
   REWRITE (PRINTER, 'printer-name');
   HEADPRINT
  END;

PROCEDURE LINER (L : LINE);
  BEGIN
   IF LINES = LPP THEN
      HEADPRINT;
   LINES := LINES + 1;
   WRITELN (PRINTER, L)
  END;

PROCEDURE STOP;
  VAR I : 0..LPP;
  BEGIN
   FOR I := LINES TO LPP DO
     WRITELN (PRINTER);
   WRITELN (PRINTER, 'END OF REPORT');
   WRITELN (PRINTER, 'PAGES PRINTED:', PAGES)
  END;
 EXPORT STOP, START, LINER, PAGES;
```

Si noti che la procedura START e' necessaria per l'inizializzazione di variabili.

Inoltre, si noti la procedura non esportata HEADPRINT: essa e' stata intenzionalmente celata all'ambiente esterno, essendo uno strumento interno per la gestione delle intestazioni.

Un possibile programma che usa questo modulo e' il seguente:

```
PROGRAM PRINT_DIV_COM (DIVINA_COMMEDIA);
TYPE LINE = PACKED ARRAY[1..80] OF CHAR;
IMPORT STOP, START, LINER, PAGES FROM PRINT;
PROCEDURE PRINT.STOP; DEFINITION;
PROCEDURE PRINT.START; DEFINITION;
PROCEDURE PRINT.LINER (L:LINE); DEFINITION;
VAR PRINT.PAGES : INTEGER;
   DIVINA_COMMEDIA : FILE OF LINE;
   SCREN : TEXT;
BEGIN
  RESET (DIVINA_COMMEDIA, 'DIVCOM');
  REWRITE (SCREEN, 'screen-name');
  PRINT.START;
  WHILE NOT EOF (DIVINA_COMMEDIA) DO
   BEGIN
    PRINT.LINER (DIVINA_COMMEDIA^);
    IF (PRINT.PAGES MOD 100) = 0 THEN
       WRITELN (SCREEN, '100 PAGES PRINTED: CHECK PAPER');
    GET (DIVINA_COMMEDIA)
   END;
 PRINT.STOP
END.
```

Il programma precedente legge il contenuto di un file di righe (DIVINA_COMMEDIA) e lo stampa usando il modulo PRINT; dopo ogni 100 pagine scritte e' mandato in output un messaggio appropriato.

Tipi di dati incapsulati

L'esempio seguente mostra un modulo che definisce una variabile di tipo strutturato stack di caratteri.

```
MODULE STACK;           (* di caratteri *)
 TYPE STACKP = ^ STACKR;
      STACKR = RECORD
               ITEM : CHAR;
               PREVIOUSITEM : STACKP
               END;
 VAR TOPMOST : STACKP; HOW_MANY : INTEGER;

  FUNCTION EMPTY : BOOLEAN;
   BEGIN
    EMPTY := (TOPMOST = NIL)
   END;

PROCEDURE INIT;
  BEGIN
   TOPMOST := NIL; HOW_MANY := 0
  END;

PROCEDURE FAULT;
  BEGIN
   WRITELN ('ILLEGAL STACK ACCESS')
  END;

PROCEDURE PUSH (I : CHAR);
  VAR P : STACKP;
  BEGIN HOW_MANY := HOW_MANY + 1;
   NEW (P);
   WITH P ^ DO BEGIN
     ITEM := I;
     PREVIOUSITEM := TOPMOST
     END;
   TOPMOST := P
END;

PROCEDURE POP (VAR I : CHAR);
  VAR P : STACKP;
  BEGIN
   IF EMPTY THEN FAULT
      ELSE
       BEGIN
        HOW_MANY := HOW_MANY - 1;
        P := TOPMOST;
        WITH P ^ DO BEGIN
         I := ITEM;
         TOPMOST := PREVIOUSITEM
         END;
       DISPOSE (P)
      END
   END;

EXPORT PUSH, POP, EMPTY, HOW_MANY, INIT;
```

Lo stack e' costituito da una lista di record, ognuno contenente un elemento dello stack (di tipo CHAR) ed un puntatore al record precedente (vedere Capitolo 16, Costruzione di Strutture Dinamiche).

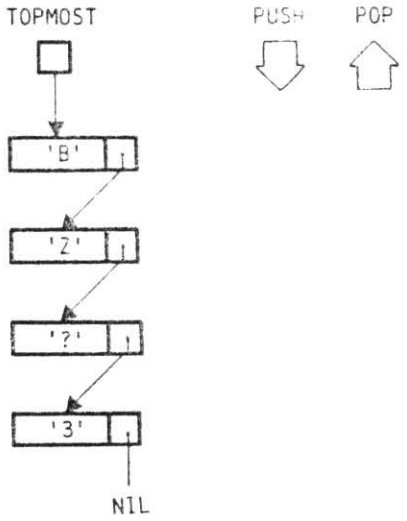


Fig. 14-4 Struttura Dinamica Usata come Stack di Caratteri

Si noti che la variabile TOPMOST non e' esportata, impedendo l'accesso diretto agli elementi dello stack dall'ambiente esterno.

Come esempio di programma importato, viene illustrato un interprete che puo' essere usato per il "testing" interattivo di moduli:

```
PROGRAM TEST_STACK;
IMPORT PUSH,POP,EMPTY,HOW_MANY,INIT FROM STACK;
PROCEDURE STACK.PUSH (I:CHAR); DEFINITION;
PROCEDURE STACK.POP (VAR I:CHAR); DEFINITION;
PROCEDURE STACK.EMPTY : BOOLEAN; DEFINITION;
PROCEDURE STACK.INIT ; DEFINITION;
VAR C, X : CHAR;
   STACK.HOW_MANY : INTEGER;
BEGIN
 STACK.INIT; READ (C);
 WRITELN ('COMMANDS ARE:');
 WRITELN ('P(PUSH),O(POP),H(HOW MANY),E(EMPTY).');
 WHILE NOT EOF DO
  BEGIN
   CASE C OF
   'P' : BEGIN WRITE ('INSERT CHARACTER:');
         READ(X); WRITELN;
         STACK.PUSH(X)
         END;
   'O': BEGIN STACK.POP(X);
        WRITELN ('CHARACTER POPPED:',X)
        END;
   'H' : WRITELN ('NUMBER OF ELEMENTS:', STACK.HOW_MANY);
   'E' : IF STACK.EMPTY THEN
           WRITELN ('STACK EMPTY')
         ELSE
           WRITELN ('STACK NOT EMPTY');
   OTHERWISE WRITELN ('WHAT ?!?')
   END;
   READ (C)
 END;
WRITELN ('GOOD BYE!')
END.
```

PARTE III. GUIDA ALLA PROGRAMMAZIONE

15. CONSIDERAZIONI DI PROGRAMMAZIONE

Lo scopo di questo capitolo e' di essere di aiuto per i programmatori meno esperti di PASCAL+ nella soluzione dei problemi pratici piu' comuni. Infatti anche i piu' semplici linguaggi di programmazione richiedono una certa esperienza per essere utilizzati efficientemente.
Il problema, quindi, non e' solo quello di studiare come usare il linguaggio per risolvere un dato problema, ma di decidere tra un certo numero di soluzioni equivalenti, quella piu' efficace, chiara, mantenibile ecc.
Gli esempi che vengono qui illustrati non vanno presi come soluzioni assolute, ma vanno interpretati criticamente ed estrapolati per ogni caso specifico.

L'USO DEL PUNTO E VIRGOLA

Il simbolo punto e virgola ";" e' spesso usato nei programmi PASCAL+ come simbolo separatore; per esempio, tra due dichiarazioni consecutive, o tra due istruzioni. Come si puo' ricordare dal Capitolo 3, il PASCAL+ e' un linguaggio a formato libero, nel senso che in un programma un fine linea ha lo stesso significato del carattere spazio (blank).

Il programmatore inesperto in PASCAL+ incontra normalmente qualche difficolta' nell'uso del punto e virgola; il senso di questo paragrafo e' di aiutarlo nei casi piu' pratici.

Innanzitutto bisogna distinguere l'uso del punto e virgola all'interno della parte istruzioni (tra BEGIN ... END di un programma, procedura o funzione) da quello nelle rimanenti parti del programma, ossia dichiarazioni ed intestazioni.

L'Uso del Punto e Virgola all'Interno di una Parte Istruzioni

L'uso del punto e virgola nella parte istruzioni segue una semplice regola: un punto e virgola e' usato solo per separare due istruzioni consecutive.

Il semplice programma che segue mostra l'uso di questa regola:

```
PROGRAM P;
.
.
BEGIN (* BEGIN non e' una istruzione, ma una parte di una istruzione
    composta, pertanto non deve essere seguita dal punto e virgola*)
A := B;
C := D;
IF A=E THEN (* L'istruzione IF non e' completa *)
   PROCP
ELSE (* ELSE e' l'unica parola chiave che non deve essere preceduta
     da un punto e virgola *)
   BEGIN
   S1;
   S2;
   S3  (* L'ultima istruzione di un gruppo puo' non essere seguita
        da un punto e virgola *)
   END; (* Questo punto e virgola separa l'istruzione IF dalla
        istruzione di assegnazione che segue *)
   C := E;
   CASE C OF
     1 : S1; (* Ogni istruzione di un ramo del CASE e' separato *)
     2 : S2;
   OTHEWISE S3
  END (* Non segue alcun altra istruzione *)
END.
```

Qualora vi siano dubbi, si consideri che il simbolo punto e virgola non deve essere usato prima della parola chiave:

```
ELSE
```

L'Uso del Punto e Virgola nelle Dichiarazioni

Ogni etichetta (label), costante, tipo, procedura, funzione, che importa o esporta una dichiarazione deve essere terminata con il simbolo punto e virgola.
Inoltre nella descrizione di un RECORD, ogni lista di dichiarazione dei campi deve essere separata dalle successive da un punto e virgola.
L'esempio seguente mostra l'uso del punto e virgola nelle dichiarazioni:

```
PROGRAM P;
   LABEL 1,9,9999,12b; (* Un solo punto e virgola per una lista *)
   IMPORT X,Y FROM M1; (* Un  punto  e  virgola  per ogni lista di
                          importazione *)
   IMPORT Z FROM M2;
   CONST K=1;
       NAME = 'JOE'; (* Ogni  dichiarazione  di  costante termina
                        con un punto e virgola *)
   TYPE T=1..10;
      RISPOSTA = (SI,NO); (* Ogni dichiarazione  di  tipo  termina
                             con un punto e virgola *)
   PROCEDURE P;
      .
      .
   BEGIN
      .
      .
   END; (* Una dichiarazione di procedura termina con punto e
        virgola *)
   FUNCTION F : REAL;
     FORWARD;          (* Anche quando e' usata una direttiva *)
      .
      .
   EXPORT K,F; (* Una lista di esportazione termina con un punto e
                  virgola *)
   VAR A : REAL; (* Ogni lista di variabili termina con un punto e
                    virgola *)
     R : RECORD
         F1 : INTEGER; (* All'interno  di  un  record,  un punto e
                          virgola *)
         F2,F3,F4 : REAL; (* per ogni lista di campi *)
         CASE BOOLEAN OF
         TRUE: (AD : INTEGER; (* anche all'interno di varianti *)
                Z : REAL);
         FALSE: (C : INTEGER);
     END;
     X1, X2 : INTEGER;
  .
  .
BEGIN
  .
  .
END.
```

L'Uso del Punto e Virgola nelle Intestazioni

Ogni intestazione di programma, modulo, procedura o funzione deve terminare con un punto e virgola.

Esempi:

```
PROGRAM P(F1,F2);
MODULE M;
PROCEDURE Q;
FUNCTION F:INTEGER;
```

Inoltre, all'interno della parte dichiarativa dei parametri (se e' presente) di una procedura o funzione, il punto e virgola deve essere utilizzato per separare ogni gruppo di parametri da quello che lo segue.

Esempio:

```
PROCEDURE P (VAR R : REAL;
                X1, X2 : REAL;
             PROCEDURE Q (Z : REAL));
```

Si noti che l'ultimo gruppo di parametri (B) non e' seguito da punto e virgola.

PROGRAMMAZIONE SENZA 'GOTO'

L'uso dell'istruzione GOTO e' da evitarsi nei programmi, poiche' verrebbe annullata la corrispondenza tra la sequenza del testo del programma e l'ordine di esecuzione: questo fatto riduce la chiarezza e la leggibilita' del programma, creando difficolta' di manutenzione per lo stesso.

Quando si programma in PASCAL+, l'uso dell'istruzione GOTO non e' effettivamente necessario, poiche' la struttura di controllo delle istruzioni permette agevolmente la creazione di algoritmi equivalenti.

Lo scopo di questo paragrafo e' di mostrare come l'istruzione GOTO puo' essere evitata.

Si supponga di usare un'istruzione GOTO.
Il GOTO puo' essere:

- un salto all'indietro;
- un salto in avanti;
- un salto all'esterno di una procedura o funzione.

Come Evitare il Salto all'Indietro

Un salto all'indietro indica la ripetizione di un gruppo di istruzioni. Questo effetto puo' essere ottenuto in modo migliore con un'istruzione ripetitiva PASCAL+ come WHILE, REPEAT o FOR (vedere Capitolo 12 per il loro uso). Il loro uso puo' sembrare apparentemente troppo complesso, poiche' esso risultera' molto piu' semplice dell'uso del GOTO con la sua dichiarazione delle etichette.

Se e' richiesto un salto all'indietro all'interno di un' istruzione REPEAT, si ha la ripetizione di tutto il gruppo di istruzioni come nell'esempio seguente:

```
REPEAT;
 1 : S1;
   IF C1 THEN
      GOTO1;
   S2
UNTIL C2
```

La soluzione corretta e' la seguente:

```
REPEAT
 S1;
 IF NOT C1 THEN
    S2
 UNTIL C2 OR NOT C1
```

Come Evitare il Salto in Avanti

Un salto in avanti indica che un gruppo di istruzioni deve essere saltato. Questo naturalmente non significa che queste istruzioni non devono essere eseguite, ma che devono essere eseguite sotto condizioni diverse.

Se l'istruzione GOTO in esame ha lo scopo di far terminare una struttura ripetitiva, puo' essere sostituita con l'istruzione EXIT (vedere Capitolo 12), o forse eliminate completamente usando una differente istruzione ripetitiva.

Se l'istruzione GOTO non e' usata per uscire da una struttura di ripetizione, significa che esprime semplicemente l'esecuzione condizionale di un gruppo di istruzioni.
In questo caso deve essere usata una struttura condizionale (vedere IF e CASE nel cap. 12).

Come Evitare il Salto all'Esterno di una Procedura o Funzione

Un salto verso l'esterno di una procedura o funzione e' usato per uscire da essa senza tornare al punto in cui e' stata chiamata.
Questa azione, comunque, puo' sempre essere effettuata al livello del punto di chiamata, passando alcune informazioni della procedura (o funzione) e' informando l'ambiente chiamante sul da farsi dopo che la procedura (o funzione) e' terminata.

Si consideri ad esempio il programma seguente:

```
PROGRAM P;
  LABEL 555;
  .
  .
 PROCEDURE Q;
  .
  .
 BEGIN
    .
    .
    IF COND THEN
        GOTO 555;
    S1
    END;
 .
 .
 BEGIN
 .
 .
 Q;
 S2;
 555 : S3;
 .
 .
END.
```

Un programma equivalente (e migliore) senza il GOTO sarebbe:

```
PROGRAM P;
 VAR SKIP : BOOLEAN;
 .
 .
 PROCEDURE Q;
  .
  .
 BEGIN
  .
  .
 IF COND THEN
    SKIP := TRUE
 ELSE
    S1
 END;
.
.
BEGIN
 .
 .
 Q;
IF NOT SKIP THEN
   S2;
S3;
.
.
END.
```

Talvolta, comunque, un'istruzione GOTO verso l'esterno ha lo scopo di far uscire da piu' livelli di funzioni o procedure nidificate; ad esempio quando si incontra un errore irrecuperabile ed il programma deve terminare.
In tali casi, l'introduzione di test, e parametri addizionali per ogni procedura o funzione e' decisamente ingombrante. L'uso dell'istruzione GOTO e' riservato solo per questi casi estremi.

L'USO DI STRUTTURE RIPETITIVE E RICORSIVE

Le azioni ripetitive influenzano moltissimo l'efficienza di un programma, per questo e' molto importante saper scegliere la migliore struttura ripetitiva.

Il PASCAL+ mette a disposizione tre strutture ripetitive, oltre al meccanismo di ricorsivita'. Il programmatore puo' scegliere tra le seguenti soluzioni possibili, ognuna adatta per uno specifico modello di programmazione.

WHILE

Durante l'esecuzione dell'istruzione WHILE, la condizione di uscita e' valutata prima di ogni ripetizione.
E' una iterazione protetta, nel senso che se una condizione di uscita e' vera in partenza, non e' eseguita alcuna istruzione.
La struttura e' particolarmente utilizzata per l'elaborazione dei file in input, poiche' RESET mette a disposizione il primo elemento del file.

Esempio:

```
RESET (F, 'nome-esterno');
WHILE NOT EOF(F) DO
 BEGIN
  .
  . (* Elaborazione dell'elemento *)
 GET (F)
END;
```

REPEAT

Nell'esecuzine di un'istruzione REPEAT, la condizione di uscita e' valutata dopo ogni ripetizione. L'istruzione e' quindi eseguita almeno una volta.

Questa istruzione e' molto comoda poiche' non richiede di racchiudere tra BEGIN-END le istruzioni che devono essere ripetute.

Attenzione: se si vuole convertire un'istruzione WHILE in REPEAT, si ricordi che la condizione di uscita deve essere invertita.

Esempio:

```
RESET (F, 'nome-esterno');
WHILE NOT EOF(F) DO
  BEGIN
   .
   .
  GET (F)
END;
```

```
RESET (F, 'nome-esterno');
REPEAT
 .
 .

GET (F)
UNTIL EOF(F);
```

FOR

L'istruzione FOR permette una iterazione "controllata", ossia il numero di ripetizioni deve essere consentito prima dell'esecuzione.
E' particolarmente adatta quando una variabile deve essere incrementata o decrementata durante il ciclo iterativo, come nelle operazioni di scansione di un array o nell'ispezione di un set.

Esempio:

```
VAR S : SET OF 1..20;
  I : 1..20
.
.
FOR I := 1 TO 20 DO
 IF I IN S THEN WRITELN (I);
```

Nell'esempio precedente l'istruzione FOR e' usato per scrivere gli elementi contenuti nella variabile set S.

L'istruzione FOR e' essenziale per la scansione dei tipi enumerativi poiche' l'ultimo incremento della variabile di controllo non e' effettuato.

Esempio:

```
VAR GIORNO : (DOM,LUN,MAR,MER,GIO,VEN,SAB);
 .
 .
FOR GIORNO := DOM TO SAB DO...
```

Si noti che l'iterazione equivalente con l'uso dell'istruzione WHILE (o REPEAT) conterra' un errore:

```
GIORNO := DOM
WHILE GIORNO <= SAB DO
 BEGIN
 .
 .
 GIORNO := SUCC (GIORNO) (* SUCC(SAB) non esiste *)
 END;
```

RICORSIONE

L'uso della ricorsione e' preferibile in tutti i casi in cui sono richieste operazioni organizzate a "stack".
La maggior parte di questi problemi possono comunque essere risolti con strutture ripetitive: questo e' il caso delle routine ricorsive dove, dopo che e' completata la chiamata piu' interna, tutte le istanze in attesa sono rilasciate.

Ad esempio:

```
FUNCTION FACT (N:INTEGER) : INTEGER;
  BEGIN
    IF N <= 1 THEN
       FACT := 1
    ELSE
      FACT := N * FACT (N-1)
END;
```

questo algoritmo puo' essere riscritto nel modo seguente:

```
FUNCTION FACT (N:INTEGER): INTEGER;
  VAR I,F : INTEGER;
  BEGIN
    F := 1;
    FOR I := 2 TO N DO
     F := F * I;
     FACT := F
END;
```

Quest'ultima soluzione e' migliore sia in termini di occupazione di memoria che in tempo di esecuzione.
Di fatto come detto nel Capitolo 6 (il RUN TIME STACK), ogni chiamata ricorsiva implica l'allocazione delle variabili interne e dei valori dei parametri e questo comporta ovviamente un certo sforzo.

Comunque, taluni problemi, che richiedono uno stack di dati si possono risolvere facilmente con routine ricorsive, sfruttando l'allocazione ripetuta dei parametri per valore.

Esempio:

```
PROGRAM EXPRESSION;
  VAR C : CHAR;
  FUNCTION EXPR : INTEGER;
   VAR N : INTEGER;
   FUNCTION TERM : INTEGER;
    VAR N : INTEGER;
    FUNCTION FACT : INTEGER;
     FUNCTION NUMBERREAD : INTEGER;
      VAR N : INTEGER;
      BEGIN
        N := 0;
         WHILE C IN ['0'..'9'] DO
          BEGIN
           N := N * 10 + (ORD(C)-ORD('0'));
           READ(C)
          END;
         NUMBERREAD := N
        END;
       BEGIN (* FACT *)
        IF C = '(' THEN
          BEGIN READ(C);
           FACT := EXPR;
           READ(C)
          END
        ELSE
         FACT := NUMBERREAD
      END;
         BEGIN (* TERM *)
          N := FACT;
          WHILE C = '-' DO
           BEGIN
             READ (C);
             N := N*FACT
           END;
          TERM := N
         END;
       BEGIN (* EXPR *)
        N := TERM;
        WHILE C = '+' DO
         BEGIN
           READ(C);
           N := N + TERM
         END;
        EXPR := N
       END;
       BEGIN (* MAIN *)
        REPEAT
         READ (C);
         WRITELN (EXPR)
        UNTIL C = ' ';
        PAGE (OUTPUT)
       END.
```

In questo esempio ogni elaborazione parziale e' salvata nel Run Time Stack, mentre viene elaborata una espressione piu' interna e poi restituita.

L'USO DI PARAMETRI

Nel Capitolo 13 si e' parlato dei quattro tipi di parametri che possono essere usati in PASCAL+.
Ora viene mostrato l'uso di tali parametri quando e' richiesto uno scambio di informazioni tra l'ambiente chiamante e la routine chiamata.

Possono essere formulati tre casi differenti:

- informazioni (ad esempio un valore) devono essere trasmesse dall'ambiente chiamante alla routine chiamata;

- informazioni devono essere ottenute dalla routine chiamata;

- una variabile deve essere gestita nella routine chiamata;

Di seguito viene considerato ogni caso separatamente.

INPUT

Un input ad una routine chiamata puo' essere effettuato usando:

- un parametro per valore;

- un parametro per variabile;

- una variabile globale.

La prima possibilita' e' la piu' corretta poiche' assicura che il parametro attuale non e' modificato.
Comunque l'uso di questo parametro puo' essere molto dispendioso in termini di allocazione di memoria e di tempi di esecuzione, poiche' e' fornita una copia locale dei dati di input.
In caso di routine ricorsive o di parametri strutturati cio' puo' essere svantaggioso.

La seconda possibilita', parametri per variabile, e' molto efficiente, ma il programmatore deve tener presente che eventuali modifiche ai parametri avranno influenza anche per l'ambiente esterno alla routine.
Questa opzione e' costosa in termini di tempo di accesso alla memoria, poiche' e' richiesto l'indirizzamento indiretto.

La terza possibilita', variabili globali, e' la piu' efficiente in quanto non vi sono ne' copie locali ne' riferimenti indiretti ai dati.
Comunque l'uso delle variabili globali non permette la parametrizzazione delle informazioni in input.

La scelta dell'opzione migliore dipendera', come sempre, dallo specifico problema.

OUTPUT

Un output da una routine chiamata puo' essere effettuato usando:

- un risultato di funzione;

- un parametro per variabile;

- una variabile globale.

La prima opzione viene usata quando le informazioni in output sono costituite da un valore di tipo semplice o puntatore.
Ad esempio, le funzioni aritmetiche (predefinite o no) generalmente accettano parametri per valore in input e restituiscono il valore risultante come output.

La seconda opzione, parametri per variabili, e' la piu' usata poiche' le informazioni in output possono consistere di piu' valori, di qualsiasi tipo, anche strutturato.
Questa opzione e' piuttosto costosa in termini di tempo d'accesso ai dati, poiche' e' richiesto l'indirizzamento indiretto.

La terza opzione, variabili globali, e' la piu' efficiente ma non permette la parametrizzazione delle variabile dove i valori di output sono memorizzati.

INPUT/OUTPUT

La modalita' di comunicazione input/output con una routine puo' essere fornito usando:

- una variabile globale;

- un parametro per variabile.

La prima possibilita' e' la piu' efficiente ma non permette parametrizzazioni.

La seconda, invece, e' usata quando devono essere eseguite le stesse operazioni su differenti variabili: in questo caso si puo' sviluppare una procedura che usi nomi simbolici delle variabili interessate.

Per esempio, si consideri la seguente parte di programma:

```
VAR A,B,C,D,TEMP : T;
 .
 .
TEMP := A;
 A := B;
 B := TEMP;
   .
   .
TEMP := C;
 C := D;
 D := TEMP;
 .
 .
```

Si noti che sono effettuati due scambi: il primo tra i valori delle variabili A e B, il secondo tra C e D.
In questo caso si potrebbe usare una procedura di scambio che agisce sui parametri per variabile:

```
VAR A,B,C,D : T
PROCEDURE EXCH (VAR X,Y : T);
  VAR TEMP : T;
  BEGIN
   TEMP := X;
   X := Y;
   Y := TEMP
  END;
.
.
EXCH (A,B);
.
.
.
EXCH (C,D);
.
.
```

In questo caso i parametri per variabile X e Y sono usati come variabili simboliche rese attuali durante la chiamata di procedura.

Il Problema degli Pseudonimi

Quando si usa un parametro per variabile, puo' sorgere qualche problema dovuto al fatto che due nomi facciano riferimento alla stessa entita'. Nell'esempio precedente, per esempio, mentre si esegue la procedura EXCH la prima volta, il nome X e' uno pseudonimo per la variabile A.

Si consideri una procedura che agisce sia su di un parametro per variabile che su una variabile globale:

```
VAR A,B : T;
PROCEDURE P(VAR X : T);
  BEGIN
  X := 1;     (* Parametro per variabile *)
  A := 30;    (* Variabile globale *)
  .
  .
 END;
.
.
P(A);
.
.
```

Chiamando questa procedura con A come un parametro attuale si possono causare effetti collaterali indesiderati, poiche' il valore di X non sara' 1 ma 30. Questo puo' essere evitato introducendo una variabile locale sulla quale vanno effettuate le operazioni e che sara' copiata da/al parametro:

```
VAR A,B : T;
PROCEDURE P(VAR PARAM_X : T);
 VAR X : T;
 BEGIN
  X := PARAM_X; (* Se e' una informazione di input *)
  X := 1;
  A := 30;
  :       (* Ora X e' comunque 1 *)
  PARAM_X := X (* Se e' un'informazione di output *)
  END;
```

DETTAGLI SULL'INPUT/OUTPUT

Il PASCAL+ permette che solo le variabili file siano il mezzo di comunicazione fra i programmi e l'ambiente esterno.
I due tipi di variabili file vengono trattati separatamente qui di seguito:

- i file-testo

- i file normali.

FILE-TESTO

Le procedure predefinite READ e READLN permettono di leggere un valore da un file-testo (ad esempio la tastiera) in una variabile intera, reale o di tipo carattere.
Leggendo in una variabile intera o reale si puo' causare una interruzione del programma se la sintassi del numero letto non e' corretta.
Ad esempio si consideri la seguente situazione:

```
VAR I:INTEGER;
 .
 .
READLN(I);
 .
 .
```

Se si premono i tasti "1", "3", "A", RETURN si incorre in un errore poiche' "A" non e' una cifra ed il programma viene interrotto.

Una possibile soluzione per evitare l'interruzione e' la seguente:

```
VAR I: INTEGER; C : CHAR;
 :
BEGIN
I := 0;
READ (C);
WHILE NOT EOLN DO
   BEGIN
   IF C IN ['0'..'9'] THEN
      I := I * 10 + (ORD (C) - ORD ('0'))
   ELSE
      (* messaggio *);
   READ (C)
   END
END;
```

Si noti che i caratteri, distinti dalle cifre, sono saltati, ma il programma non e' interrotto.

Un altro problema puo' essere quello di leggere particolari caratteri. Ad esempio se si vuole leggere una lettera o il simbolo "!", una possibibile soluzione puo' essere:

```
VAR C : CHAR;
REPEAT
  READ (C)
UNTIL C IN ['A'..'Z', '!'];
```

Si noti che la lettura puo' essere effettuata piu' volte finche' e' fornito il carattere corretto.

Relativamente alla procedura WRITE, si ricordi che taluni dispositivi di output, ad esempio stampanti in linea, scrivono effettivamente solo quando viene fornito un fine linea.

Si consideri il seguente esempio:

```
.
.
WRITE ('AAA');
.
.
WRITE ('BBB');
.
.
WRITE ('CCC');
.
.
WRITELN;
.
.
```

in tal caso viene effettuata un'unica stampa e cioe':

```
AAABBBCCC.
```

Se si vuole saltare qualche riga, una migliore soluzione e' la seguente:

```
FOR I: = 1 TO N DO WRITELN;
```

FILE NORMALI

Il PASCAL+ permette una associazione tra nome interno ed esterno del file durante l'esecuzione.
Infatti, tale associazione e' effettuata mentre e' eseguita la procedura predefinita RESET(F,s) oppure REWRITE(F,s). Se si vuole parametrizzare il file elaborato dal programma, si puo' scrivere una procedura interattiva che chiede il nome del file:

```
VAR F : FILE OF...
  .
  .
PROCEDURE ASK_AND_RESET;
 VAR EXT_NAME : PACKED ARRAY [1..10] OF CHAR;
  I : INTEGER;
  C : CHAR;
BEGIN
 FOR I := 1 TO 10 DO EXT_NAME [I] := '';
 I := 0;
 REPEAT
  READ (C);
  I := I+1;
  IF I <= 10 THEN EXT_NAME [I] := C;
 UNTIL EOLN;
 RESET (F,EXT_NAME)
END;
```

Questa procedura apre il file F in lettura il cui nome esterno e' fornito da tastiera (o da qualsiasi dispositivo di input usato).

L'elaborazione di un file in PASCAL+ e' effettuata attraverso routine

predefinite (vedere Capitolo 13, Procedure e Funzioni Predefinite per la Gestione dei File) ma sono fornite solo operazioni di basso livello, poiche' la gestione di dati complessi e' affidata al programma.

Il miglior modo di effettuare un'efficiente gestione di file sequenziali e' di copiare l'intero file in una lista (vedere Capitolo 16, Costruzione di Strutture Dinamiche). Si puo' allora lavorare sulla lista leggendo, scrivendo o aggiornando con velocita' considerevole gli elementi e, dopo l'elaborazione, copiare la lista aggiornata sul file.

Ovviamente questo metodo puo' essere usato solo se il file non e' troppo lungo, cioe' se puo' essere memorizzato nello HEAP.

16. PARTICOLARITA' SUI MODULI

In questo capitolo si illustra la costruzione dei principali tipi di dati incapsulati, ovvero le strutture dinamiche di dati.
E' fatto largo uso di variabili puntatore in quanto esse costituiscono i "mattoni" per la costruzione delle strutture dinamiche dei dati.

Inoltre e' illustrata una utile metodologia per lo sviluppo del sistema informativo, ossia la metodologia "bottom-up".

COSTRUZIONE DI STRUTTURE DINAMICHE

Le strutture dinamiche sono disegnate con l'utilizzo delle variabili puntatore, poiche' esse costituiscono i soli strumenti che permettono l'allocazione e la deallocazione manuale della memoria (vedere Capitolo 6, Lo Heap).
Vengono descritti tre tipi di strutture dinamiche:

- array sparsi;
- liste;
- alberi.

ARRAY SPARSI

Accade di frequente che una variabile array ha solo pochi elementi significativi, mentre i rimanenti hanno valori di default (ad esempio zeri o spazi).
La velocita' di accesso agli elementi, in questo caso, e' costosa in termini di occupazione di memoria, specialmente quando gli elementi dell'array sono di grandi dimensioni. Una buona soluzione in molti casi e' l'uso di un "array sparso", cioe' un array di puntatori agli elementi, in cui l'elemento in esame viene allocato attraverso la procedura NEW ed a cui si accede tramite l'operatore di riferimento "^".

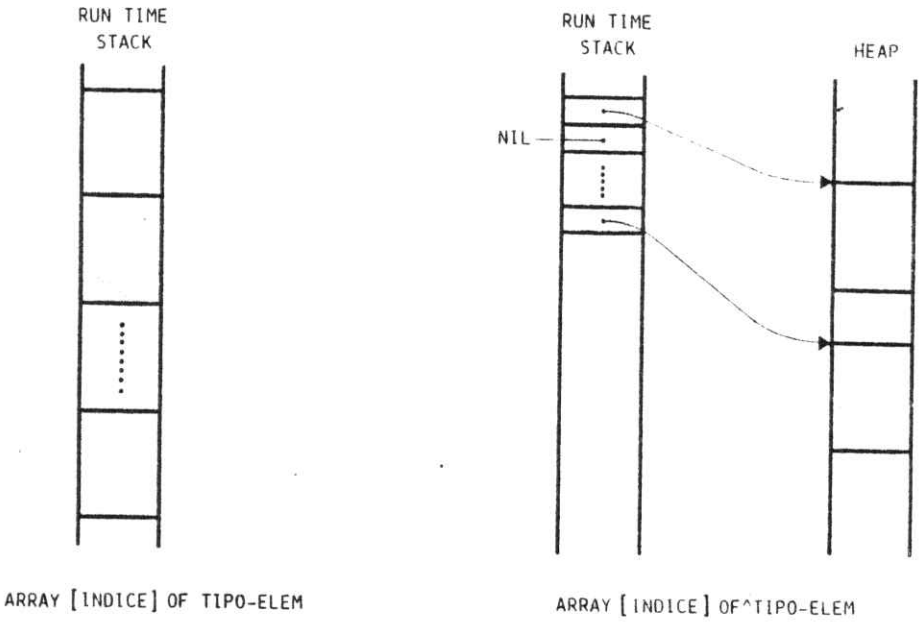


Fig. 16-1 Allocazione in Memoria di un Array Normale e di un Array Sparso

Si consideri ora un modulo che rappresenta il tipo di dati incapsulati "array sparso":

```
MODULE ARRAY_SPARSO;
TYPE TIPO_ELEM=...;
     INDICE=...;
VAR A:ARRAY [INDICE] OF ^ TIPO_ELEM;

PROCEDURE INSERIMENTO (I:INDICE; X:TIPO_ELEM);
 BEGIN
  IF X= (* valore di default *) THEN
     BEGIN
     IF A[I] <> NIL THEN
        BEGIN
         DISPOSE (A[I]); A[I] := NIL
        END
        END
     ELSE
      BEGIN
       IF A[I] = NIL THEN
          NEW (A[I]);
       A [I]^ := X
       END
     END;

 PROCEDURE OTTENERE_ELEM (I:INDICE; VAR X:TIPO_ELEM);
   BEGIN
    IF A [I] = NIL THEN
    X := (* valore di default *)
    ELSE
    X := A [I]^
  END;
EXPORT INSERIMENTO, OTTENERE_ELEM;
```

Ovviamente l'uso di questo modulo e' piuttosto dispendioso in termini di tempo, poiche' ogni accesso agli elementi e' effettuato tramite chiamate di routine (INSERIMENTO, OTTENERE_ELEM). Tuttavia si puo' usare un array sparso direttamente nel programma.

LISTE

Una lista e' una sequenza di elementi aggiornabili, chiamati nodi. Ogni nodo e' costituito da una variabile record contenente una informazione ed un puntatore al prossimo nodo.

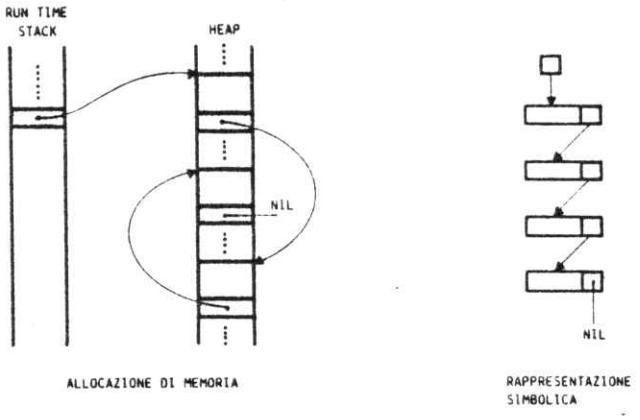


Fig. 16-2 Allocazione di Memoria di una Lista e sua Rappresentazione Simbolica

Si consideri ora un modulo che rappresenta il tipo di dato incapsulato "lista":

```
MODULE LISTA;
TYPE TIPO_INF=...;
    CIMA_LISTA = ^ NODO;
       NODO = RECORD
       INF : TIPO_INF;
       PROSSIMO : CIMA_LISTA
    END;
VAR L, ATTUALE, PRECEDENTE : CIMA_LISTA;

PROCEDURE INIZ;
 BEGIN
  L := NIL;
  ATTUALE := NIL;
  PRECEDENTE := NIL
 END;

PROCEDURE RICERCA (CHIAVE : TIPO_INF);
       (* POSIZIONA "ATTUALE" AL NODO
       CHE CONTIENE INF = CHIAVE E POSIZIONA "PRECEDENTE"
       AL NODO PRECEDENTE *)
VAR TROVATO : BOOLEAN;
BEGIN
 TROVATO := FALSE;
 ATTUALE := L; PRECEDENTE := NIL;
WHILE NOT TROVATO AND (ATTUALE <> NIL) DO
  IF ATTUALE ^.INF = CHIAVE THEN
    TROVATO := TRUE
  ELSE BEGIN PRECEDENTE := ATTUALE;
    ATTUALE := ATTUALE ^. PROSSIMO END
    END;

    PROCEDURE INSERIMENTO (X : TIPO_INF);
     (* INSERISCE UN NUOVO NODO DOPO IL
     NODO ATTUALE *)
     VAR P : CIMA_LISTA;
     BEGIN
        NEW (P);
        WITH P ^ DO
         BEGIN INF := X;
          IF L = NIL THEN
            BEGIN PROSSIMO := NIL; L := P END
          ELSE
           IF ATTUALE = NIL THEN
             BEGIN PROSSIMO := L;
             L := P END
           ELSE
             BEGIN PROSSIMO := ATTUALE ^.PROSSIMO;
             ATTUALE ^.PROSSIMO := P END
        END
END;                                          (CONTINUA)
```

```
PROCEDURE CANCELLAZIONE;
        (* CANCELLA IL NODO ATTUALE *)
 BEGIN
   IF ATTUALE <> NIL THEN BEGIN
      IF PRECEDENTE = NIL THEN
        L := L ^.PROSSIMO
      ELSE
        PRECEDENTE ^.PROSSIMO := ATTUALE ^.PROSSIMO;
      DISPOSE (ATTUALE);
      ATTUALE := PRECEDENTE END
 END;
EXPORT RICERCA, INSERIMENTO, CANCELLAZIONE;.
```

Si noti che il tipo CIMA_LISTA e' definito ricorsivamente in quanto il campo PROSSIMO del nodo e' esso stesso del tipo CIMA_LISTA.
L'effetto delle procedure RICERCA, INSERIMENTO e CANCELLAZIONE e' mostrato nella figura seguente:

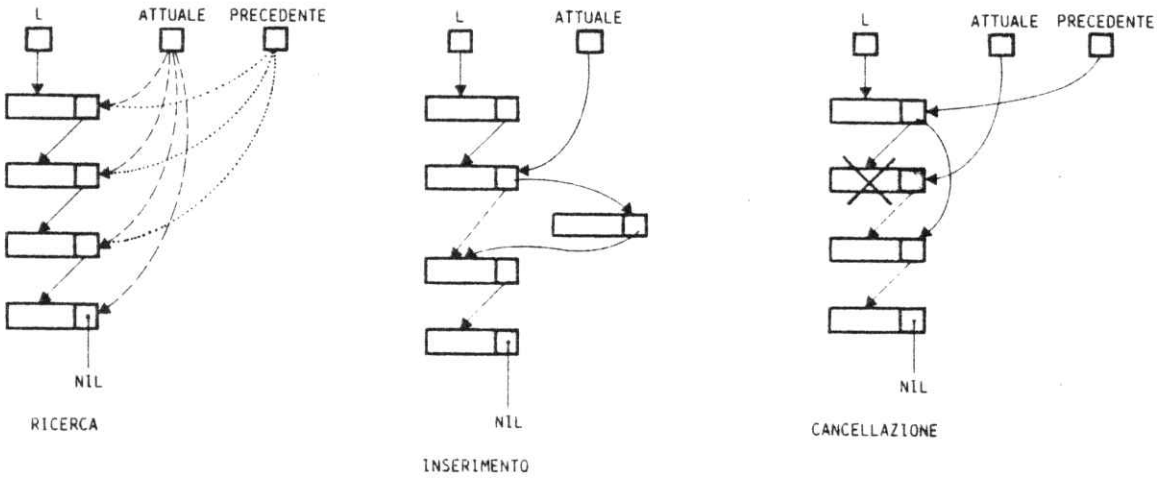


Fig. 16-3 Principali Operazioni Effettuate su una Lista

Una lista puo' essere fornita di particolari operatori (PUSH, POP, ENQUEUE, ...) che permettono l'accesso a particolari tipi di dati: stack, code, ecc. (vedere l'esempio di stack nel Capitolo 14).

ALBERI

Un albero binario e' un insieme ordinato di elementi, chiamati nodi, ognuno dei quali punta ad un sottoalbero sinistro e ad uno destro.
Ogni nodo e' costituito da una variabile record che contiene una chiave di ricerca, alcune informazioni e due puntatori ai sottoalberi.

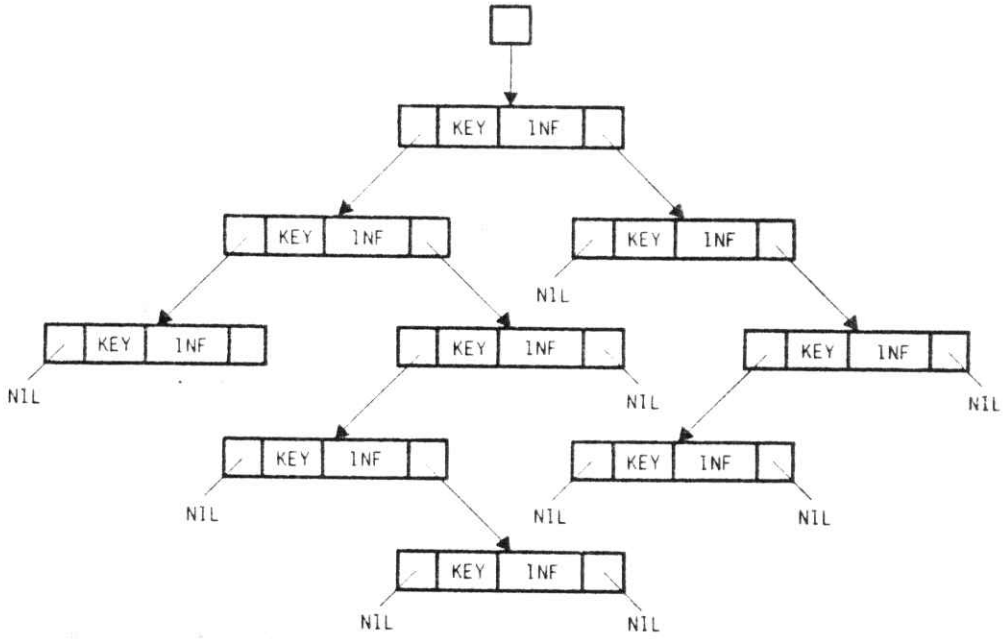


Fig. 16-4 Rappresentazione Simbolica di un Albero Binario

Il vantaggio principale ottenuto con l'uso di un albero binario e' la considerevole velocita' di ricerca, poiche' il tempo medio richiesto per l'accesso ad un elemento e' proporzionale al logaritmo in base due del numero degli elementi.

Si consideri un modulo che rappresenta il tipo di dato incapsulato "albero binario".

```
MODULE ALBERO_BINARIO;
  TYPE TIPO_CHIAVE=...;
  TIPO_INF=...;
  RADICE_ALBERO = ^NODO;
  NODO = RECORD
           KEY : TIPO_CHIAVE;
           INF : TIPO_INF;
           SINISTRA, DESTRA : RADICE_ALBERO
         END;
  VAR T, ATTUALE : RADICE_ALBERO;

  PROCEDURE INIZ;
   BEGIN
   T := NIL;
   ATTUALE := NIL
  END;                                        (CONTINUA)
```

```
PROCEDURE INSERIMENTO (K:TIPO_CHIAVE; I : TIPO_INF);
  PROCEDURE INS (VAR SUB_ALBERO : RADICE_ALBERO);
   BEGIN
   IF SUB_ALBERO <> NIL THEN
    WITH SUB_ALBERO ^ DO
     IF K = KEY THEN
        WRITELN ('CHIAVE DOPPIA')
     ELSE
       IF K < KEY THEN
          INS (SINISTRA)
       ELSE
          INS (DESTRA)
     ELSE
      BEGIN
        NEW (SUB_ALBERO);
        WITH SUB_ALBERO^DO
         BEGIN
           INF := I;
           KEY := K;
           L := NIL;
           R := NIL
         END
        END
      END;
     BEGIN
       INS (T)
     END;

PROCEDURE RICERCA (K : TIPO_CHIAVE);
 PROCEDURE POSIZIONA (VAR SUB_ALBERO : RADICE_ALBERO);
  BEGIN
  IF SUB_ALBERO <> NIL THEN
    WITH SUB_ALBERO ^ DO
    IF K = KEY THEN
       ATTUALE := SUB_ALBERO
    ELSE
     IF K < KEY THEN
        POSIZIONA (SINISTRA)
    ELSE
       POSIZIONA (DESTRA)
   END;
 BEGIN
   POSIZIONA (T)
 END;

PROCEDURE AGGIORNAMENTO (I : TIPO_INF);
 BEGIN
   IF ATTUALE <> NIL THEN
      ATTUALE ^.INF := I
 END;

PROCEDURA OTTIENI_INF (VAR I : TIPO_INF);
 BEGIN
 IF ATTUALE <> NIL THEN
 I := ATTUALE^.INF
END;                                        (CONTINUA)
```

```
PROCEDURE STAMPA;
  PROCEDURE STM (VAR SUB_ALBERO : RADICE_ALBERO);
   BEGIN
    IF SUB_ALBERO <> NIL THEN
    WITH SUB_ALBERO ^ DO
     BEGIN
     STM (SINISTRA);
     WRITELN (INF);
     STM (DESTRA);
     END
   END;
  BEGIN
   STM (T)
  END;
EXPORT INSERIMENTO, RICERCA, AGGIORNAMENTO, OTTIENI_INF,
       STAMPA, INIZ;.
```

Si noti l'uso di apposite routine locali per la scansione dell'albero: la scansione di un albero e' costituita da un confronto e da una scansione in un sottoalbero.

Si noti, inoltre, l'uso dei parametri per variabile (VAR SUB_ALBERO : RADICE_ALBERO) che permettono operazioni sulle variabili appropriate (NEW(SUB_ALBERO)) indipendentemente dal livello di nidificazione della ricursione.

Notare che, se richiesto, possono essere aggiunti dei puntatori addizionali ai nodi precedenti.

La costruzione di un albero n-ario e' molto simile: ogni nodo dovra' contenere una lista dei sottoalberi.

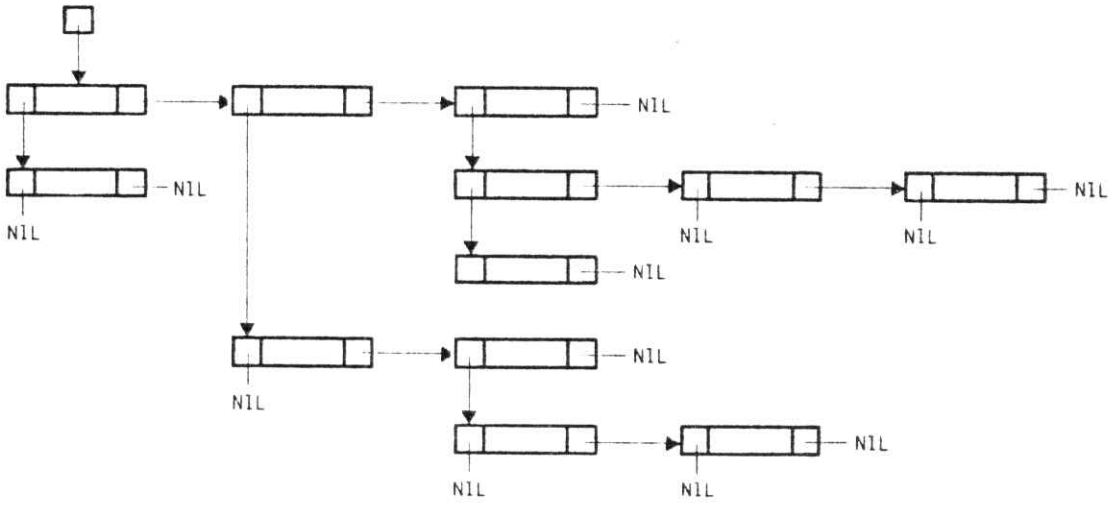


Fig. 16-5 Rappresentazione Simbolica di un Albero n-ario

Un albero n-ario e' cosi' descritto in PASCAL+:

```
TYPE RADICE_ALBERO =^NODO;
    NODO = RECORD
             INF := TIPO_INF;
             FRATELLO, FIGLIO : RADICE_ALBERO
           END;
```

La scansione di un albero n-ario dipende generalmente dal problema in esame.

LA METODOLOGIA BOTTOM UP

Si consideri la costituzione ex-novo di un sistema informativo che implichi lo sviluppo di un insieme di programmi.
La metodologia bottom-up puo' essere vantaggiosa in quanto richiede una analisi generale preventiva del problema e la costruzione di un insieme di strumenti software con caratteristiche generali, quindi lo sviluppo dei programmi applicativi, sfruttando tali strumenti, diventa molto agevole.

Il sistema di sviluppo bottom-up puo' essere realizzato, in PASCAL+, attraverso i moduli: si puo' disegnare un insieme di moduli di utilita' generale ed importare alcuni di questi in unita' principali che risulteranno percio' piuttosto semplificate.

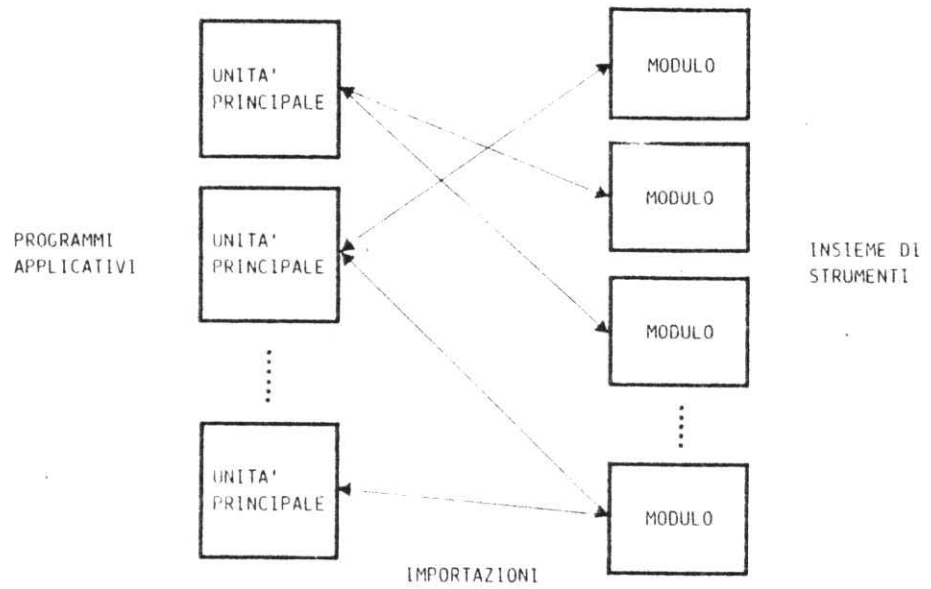


Fig. 16-6 Sviluppo Bottom-Up di Programmi

Seguendo questa metodologia, la costituzione delle unita' principali e' molto semplice, poiche' la maggior parte delle loro risorse sono descritte esternamente.

Si noti che questa metodologia e' esattamente il contrario di quella di tipo top-down accennata nel Capitolo 3 (Un Esempio di Macchina Astratta).

PARTE IV. IL LINGUAGGIO ESTESO

17. PROGRAMMAZIONE CONCORRENTE

L'implementazione del software richiede spesso lo sviluppo di programmi che gestiscono molte attivita' che devono essere svolte nello stesso istante.
Il PASCAL+ permette la costruzione di programmi che effettuano diverse azioni tra loro concorrenti.

In questo capitolo viene illustrato come sviluppare tali programmi, dando la definizione di un processo in PASCAL+ ed introducendo i monitor PASCAL+.

Inoltre vengono trattati processi cooperanti, mostrando alcune risorse predefinite adottate per la loro sincronizzazione.

PROCESSI PASCAL+

COSA E' UN PROCESSO PASCAL+

Un processo PASCAL+ e' una occorrenza di un programma che puo' essere eseguito in concorrenza con altre occorrenze dello stesso programma.

Ogni occorrenza di un programma e' fornita di una copia personale di dati (Stack Run Time e Heap), in modo tale che l'esecuzione puo' essere interrotta e ripresa senza influire sull'esecuzione CONCORRENTE delle altre occorrenze, cioe' di altri processi.

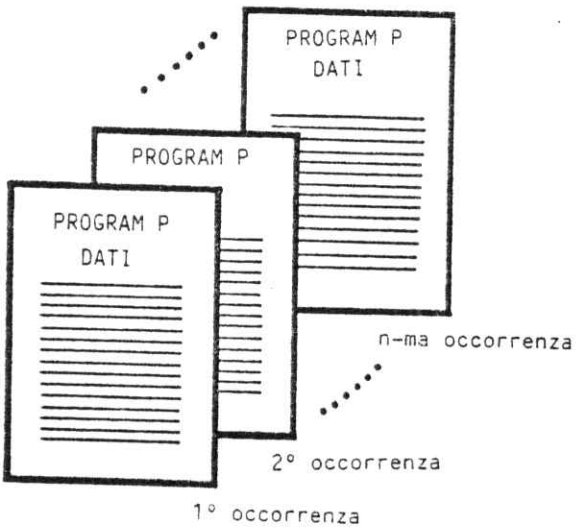


Fig. 17-1 Una Rappresentazione Grafica dei Processi PASCAL+

CREAZIONE E DISTRUZIONE DI PROCESSI

Un processo PASCAL+ e' generato in run-time chiamando una funzione esterna detta FORK.
Questa funzione crea un'altra occorrenza del programma in esecuzione, fornendo la relativa copia personale di dati; se la creazione ha successo, il processo creato entra in esecuzione con l'uso dello stesso codice oggetto (codice rientrante).

La funzione FORK e' di tipo intero e restituisce il valore 0 se la generazione del processo ha avuto successo, oppure un valore diverso da zero in caso di insuccesso.
La funzione accetta un puntatore come parametro per variabile che vale NIL nel processo creato ed ha un valore diverso da NIL nel processo creante.
Dopo l'invocazione della funzione FORK, i due processi PASCAL+ sono concorrenti e possono essere distinti solo dal valore del puntatore.

Un processo PASCAL+ avra' termine (muore) quando chiama la funzione esterna DIE oppure JOIN (vedere la documentazione relativa al sistema operativo per avere maggiori dettagli sulla funzione FORK, DIE e JOIN)

Si cosidera ora, come esempio, un programma che crea un processo; questo come i seguenti, e' un esempio non completo e quindi non eseguibile poiche' serve solo per descrivere come operano le funzioni appena menzionate.

I due processi (uno e' il programma stesso) scrivono, su differenti periferiche, due messaggi concorrentemente.

```
PROGRAM PROCESS_EXAMPLE;
TYPE NULLREC = RECORD END;
    NULLPOINT =^NULLREC;
FUNCTION FORK (VAR P: NULLPOINT): INTEGER; EXTERNAL;
FUNCTION DIE; EXTERNAL;
FUNCTION JOIN; EXTERNAL;
VAR P: NULLPOINT;
BEGIN
 IF FORK (P) = 0 THEN
    IF P = NIL THEN
      BEGIN
       WRITELN (first-device 'HELLO');
       DIE
      END
 ELSE
      BEGIN
       WRITELN (second-device, 'HOW DO YOU DO?');
       JOIN
      END
 ELSE
  (*ERRORE NELLA CREAZIONE DEL PROCESSO*)
END.
```

Si noti la definizione di un puntatore che punta a nulla come parametro per la funzione FORK.

PROTEZIONE DI PROCESSI 'EFFETTIVI'

Si usa la funzione FORK per creare due (o piu') processi PASCAL+ quando si vogliono due (o piu') azioni da svolgere contemporaneamente.
A scopo di chiarezza e' meglio separare logicamente tali azioni, a questo proposito si possono racchiudere (proteggere o incapsulare) le due elaborazioni entro due procedure.

In questo caso il programma deve essere strutturato in modo che un processo chiami una procedura e l'altro processo chiami l'altra procedura.

Come esempio si consideri un programma che simula un ascensore. Vi sono due processi: "prenotazione dei piani" e "movimento dell'ascensore". Questi processi devono essere eseguiti contemporaneamente in quanto un piano puo' essere prenotato mentre l'ascensore e' in movimento.

```
PROGRAM LIFT;
TYPE NULLREC = RECORD END;
     NULLPOINT = ^NULLREC;
FUNCTION FORK (VAR P : NULLPOINT) : INTEGER; DEFINITION;
FUNCTION DIE; EXTERNAL;
FUNCTION JOIN; EXTERNAL;
VAR P : NULLPOINT;

PROCEDURE GET_BOOK;
 .
 .
 DIE
END;

PROCEDURE LIFT_MOVE;
 .
 .
JOIN
END;

BEGIN
 IF FORK (P) = 0 THEN
   IF P = NIL THEN
     GET_BOOK
   ELSE
     LIFT_MOVE
 ELSE
  (* ERRORE NELLA CREAZIONE DEL PROCESSO *)
END.
```

Si noti che i due processi chiamano due procedure differenti e non effettuano mai la stessa azione.
Per questa ragione i due processi si possono considerare come "processi effettivi" e la parte istruzioni come un meccanismo incaricato unicamente della creazione ed esecuzione dei processi.

Si puo' rappresentare tale struttura nel modo seguente:

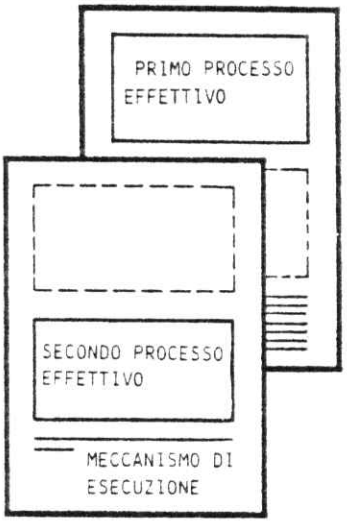


Fig. 17-2 Protezione di Processi 'Effettivi'

Se piu' di due processi PASCAL+ devono essere eseguiti in parallelo, la migliore struttura e' la seguente:

```
PROGRAM SEVERAL_PROCESSES;
CONST N = (* numero di processi *);
TYPE NULLREC = RECORD END;
     NULLPOINT =^NULLREC;
FUNCTION FORK (VAR P : NULLPOINT) : INTEGER; EXTERNAL;
FUNCTION DIE; EXTERNAL;
FUNCTION JOIN; EXTERNAL;
VAR P: NULLPOINT;

PROCEDURE PROCESS1;
 .
 .
PROCEDURE PROCESS2;
 .
 .
PROCEDURE PROCESSN;
 .
 .
BEGIN
 FOR I := 1 TO N-1 DO
  BEGIN
  IF FORK (P) = 0 THEN
   IF P = NIL THEN
     CASE I OF
       1 : PROCESS1;
       2 : PROCESS2;
       .
       .
       (* N-1 : PROCESSN-1 *)
      END
  END;
  PROCESSN
  END.
```

Un altro esempio di processi PASCAL+ puo' essere un "gestore di periferiche", cioe' un programma che tratta differenti operazioni che devono essere eseguite in concorrenza su diverse periferiche.
In tal caso si possono sviluppare appositi "processi effettivi", ognuno dei quali e' incaricato della gestione di una singola periferica.

Si consideri ora il caso particolare in cui si vogliono sviluppare e compilare separatamente i "processi effettivi"; si puo' racchiudere ogni processo effettivo entro un modulo e costruire una unita' principale generalizzata cui sia affidata soltanto l'importazione e la creazione del processo.

Si esamini di nuovo l'esempio dell'ascensore:

```
MODULE M1;
  PROCEDURE GET_BOOK;
      FUNCTION DIE; EXTERNAL;
  .
  .
  EXPORT GET_BOOK;


  MODULE M2;
   PROCEDURE LIFT_MOVE;
      FUNCTION JOIN; EXTERNAL;
   .
   .
  EXPORT LIFT_MOVE;.


  PROGRAM LIFT;
  TYPE NULLREC = RECORD END;
       NULLPOINT =^NULLREC;
  FUNCTION FORK (VAR P : NULLPOINT) : INTEGER; EXTERNAL;
  VAR P : NULLPOINT;
  IMPORT GET_BOOK FROM M1;
  IMPORT LIFT_MOVE FROM M2;
  PROCEDURE M1.GET_BOOK; DEFINITION;
  PROCEDURE M2.LIFT_MOVE; DEFINITION;
  BEGIN
    IF FORK (P) = 0 THEN
       IF P = NIL THEN
          M1.GET_BOOK
       ELSE
          M2.LIFT_MOVE
    END.
```

In questo caso si puo' rappresentare la struttura del programma nel modo seguente:

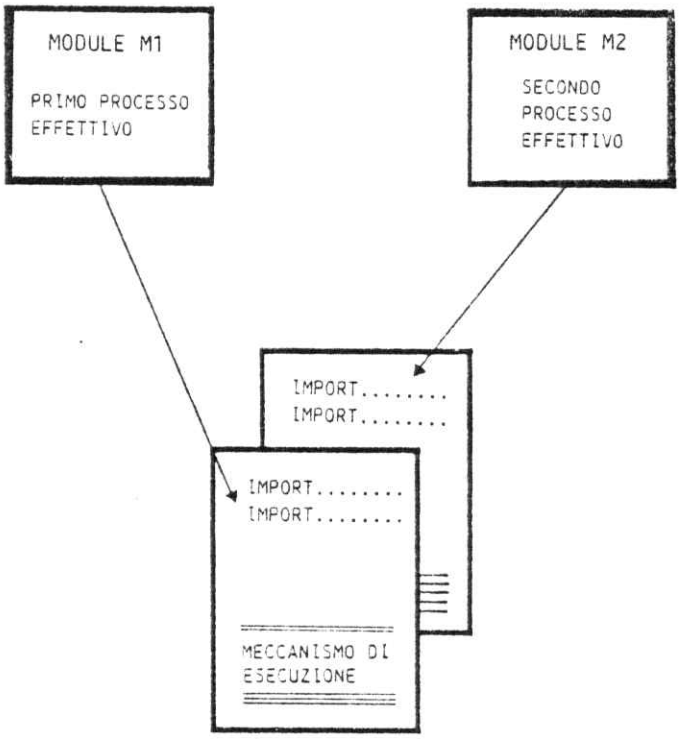


Fig. 17-3 Processi Effettivi Racchiusi in Moduli

Si noti, che ad eccezione dei nomi dei moduli e dei nomi delle procedure importate, la struttura dell'unita' principale e' molto generale e puo' essere usata per la creazione di ogni processo PASCAL+.

MONITOR PASCAL+

COSA E' UN MONITOR PASCAL+

Un monitor PASCAL+ e' una libreria di risorse.

Come per un modulo, essa e' una descrizione di risorsa compilata separatamente che deve essere collegata tramite linker all'unita' principale che la importa.

MONITOR

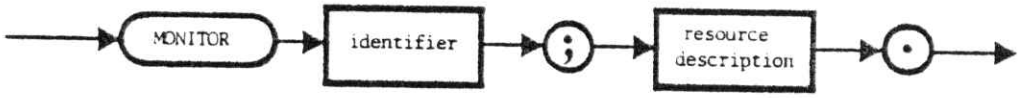


Fig. 17-4 Diagramma Sintattico di un Monitor

A differenza di moduli, ad ogni monitor PASCAL+ e' associata una coda ed un semaforo che permettono la mutua esclusione di processi PASCAL+ concorrenti.

Si considerino due (o piu') processi che importano il medesimo monitor e che sono in esecuzione contemporaneamente. Se tali processo tentano di chiamare ruotine dello stesso monitor, allora vengono accodati nella "coda" del monitor (trasparente per l'utente); quindi, le azioni eseguite da un processo sulle risorse del monitor non possono essere disturbate dalle azioni effettuate dall'altro processo.

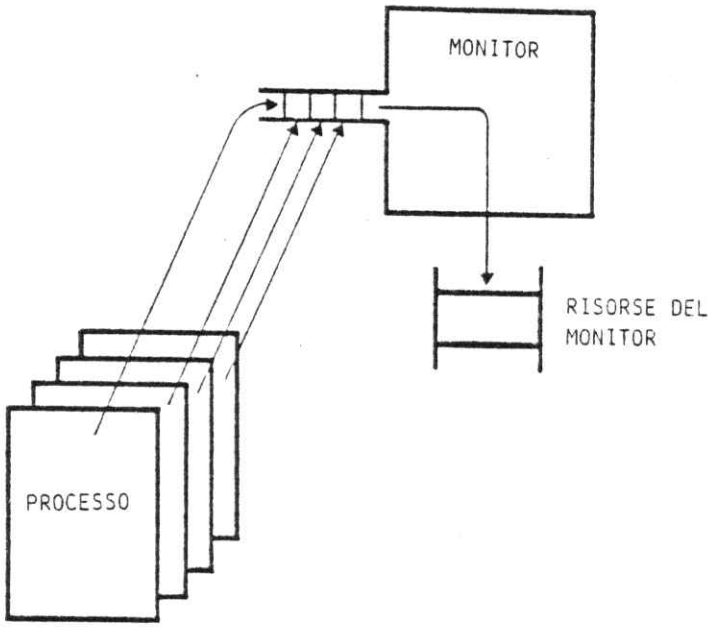


Fig. 17-5 Mutua Esclusione nell'Accesso alle Risorse del Monitor

Attenzione: non e' garantita la mutua esclusione tra processi che accedono direttamente alle costanti o variabili di un monitor.

I monitor PASCAL+ sono il solo mezzo che consente la comunicazione tra processi, poiche' solo essi vengono acceduti correttamente dai processi concorrenti PASCAL+; essi rappresentano dunque una risorsa comune tra processi paralleli.

Si consideri l'esempio dell'ascensore, visto nei paragrafi precedenti.
Si hanno due processi in esecuzione contemporaneamente: GET_BOOK che provvede all'acquisizione ed alla memorizzazione delle prenotazioni e LIFT_MOVE, che provvede al movimento dell'ascensore ed all'annullamento delle prenotazioni.
Si osservi che una "prenotazione" e' una risorsa comune per i due

processi, percio' dovra' essere racchiusa in un monitor (vedere dettagli nei prossimi paragrafi).
Il monitor assicurera', per esempio, che la "memorizzazione di una prenotazione" non sara' disturbata dall'"annullamento di una prenotazione", poiche' le suddette operazioni saranno serializzate attraverso la coda del monitor.

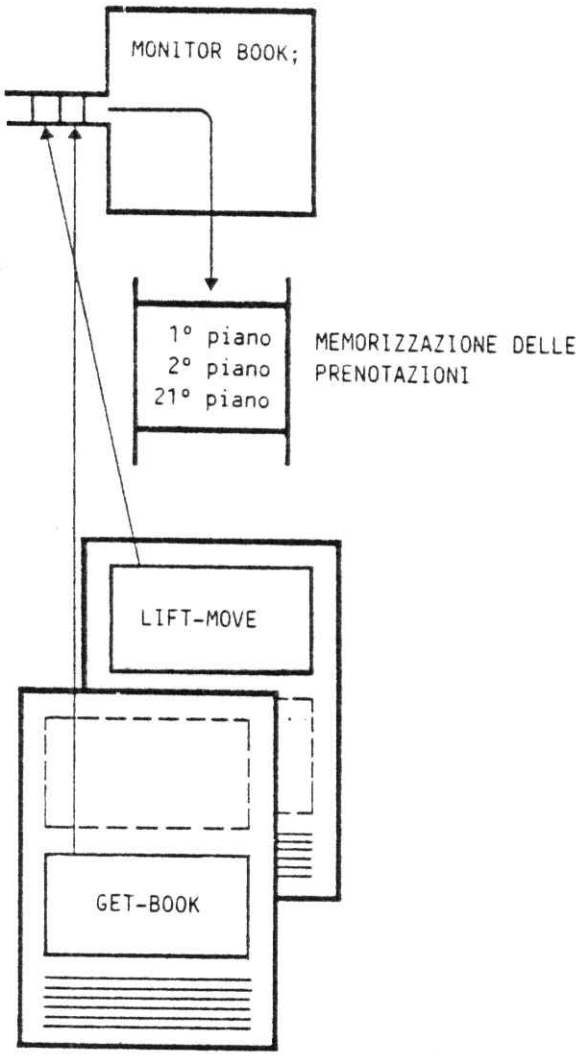


Fig. 17-6 Prenotazioni dei Piani Racchiuse entro un Monitor

Come altro esempio, si consideri un gestore di periferiche. Si supponga che alcuni processi effettivi agiscano sulla stessa periferica, ad esempio un disco.
Il disco costituira' una risorsa comune tra i processi, percio' puo' essere rappresentabile con un monitor PASCAL+.
Ogni processo operera' sul disco avendo la sicurezza che nessun altro processo possa interferire.

Se alcuni processi importano piu' monitor, questi ultimi saranno

provvisti di code separate; l'accesso a differenti monitor da differenti processi puo' quindi essere contemporaneo.

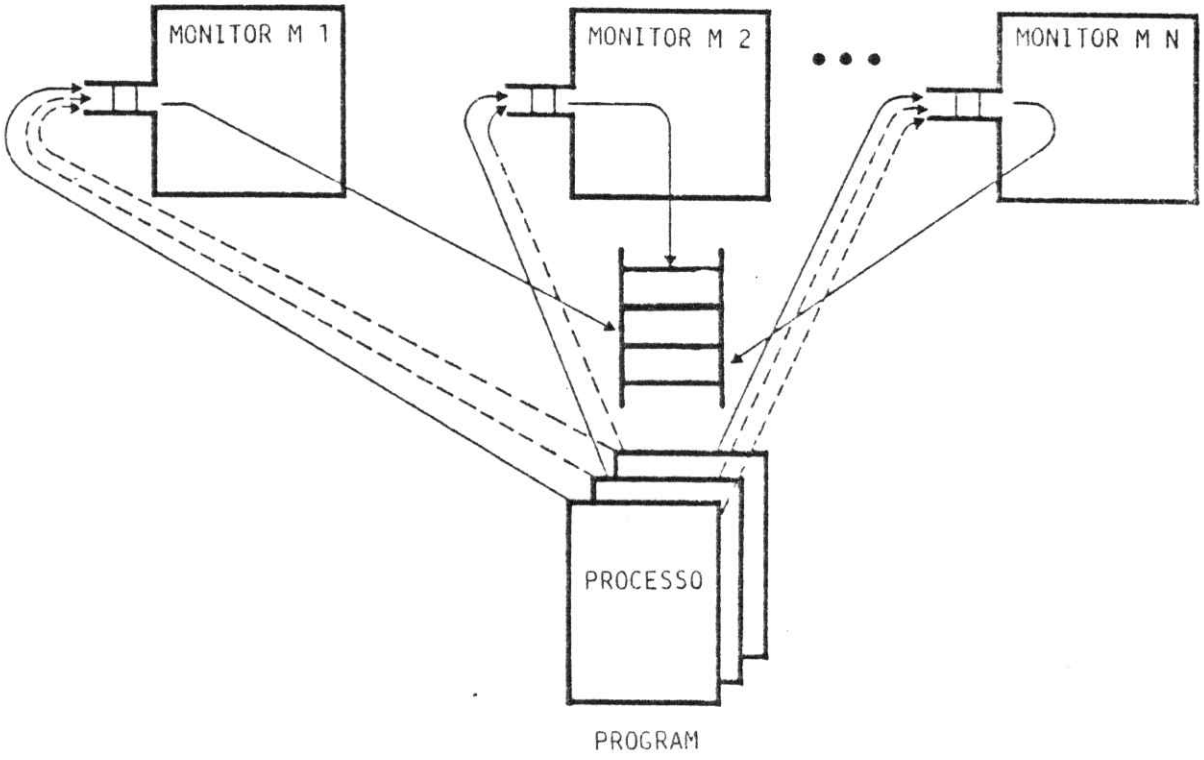


Fig. 17-7 Importazione di piu' Monitor

Se processi che appartengono a differenti programmi importano lo stesso monitor, vengono fornite code differenti; percio' programmi differenti possono accedere al medesimo monitor contemporaneamente.

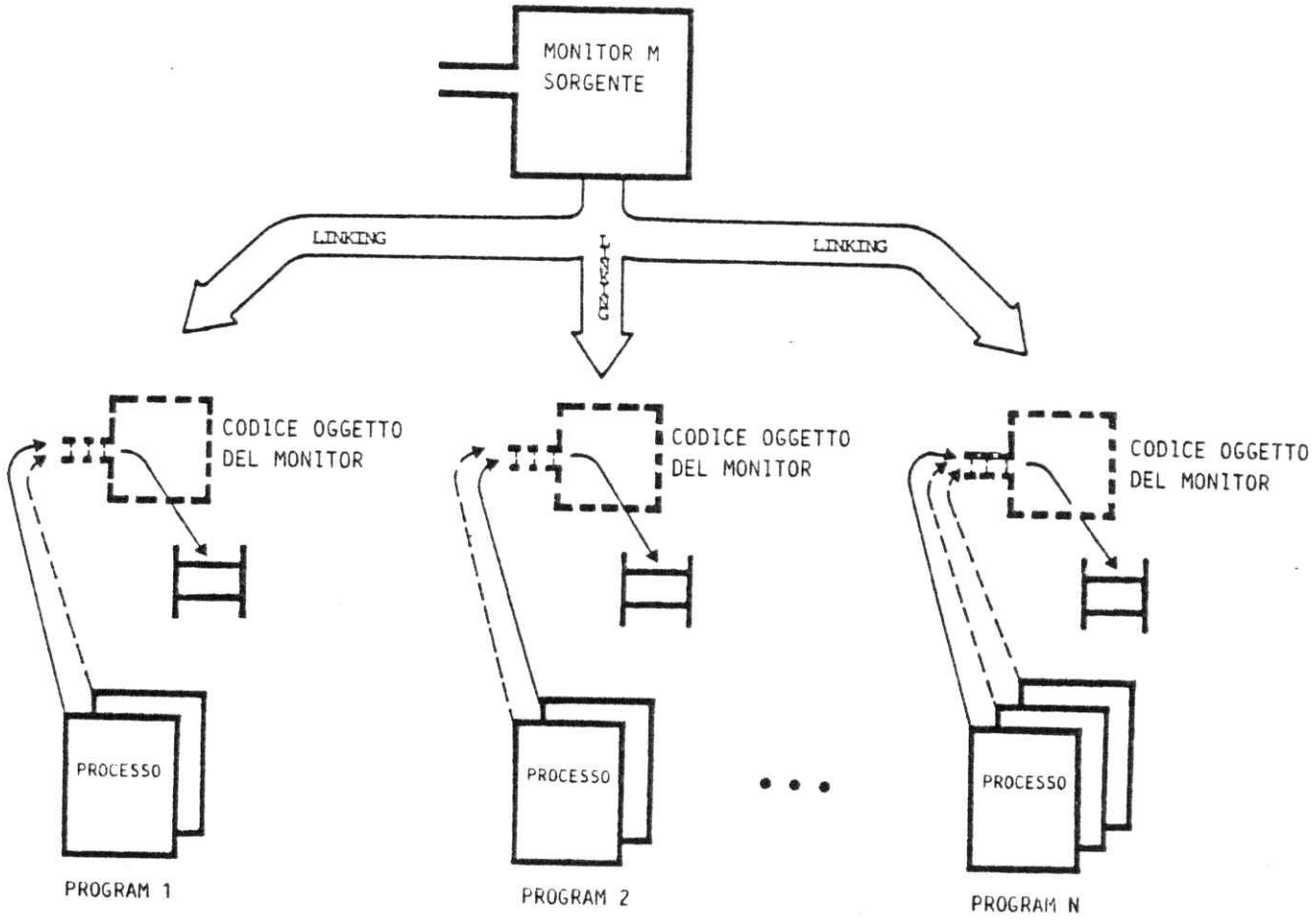


Fig. 17-8 Importazione dello Stesso Monitor da piu' Programmi

IMPORTAZIONI, ESPORTAZIONI E RIDEFINIZIONI

La struttura interna di un monitor e' identica a quella di un modulo; in particolare le sue risorse possono essere esportate e conseguentemente importate all'interno di altre unita' (programmi, moduli o monitor).

A causa del meccanismo di bloccaggio (semaforo), le routine esportabili non possono essere chiamate entro il monitor; in particolare non e' consentita la ricursione per routine esportabili.

Quando una unita' importa alcune risorse di monitor, e' necessaria una ridefinizione, con l'applicazione delle stesse regole spiegate per i moduli.

ESEMPIO

Si consideri un esempio completo riguardante la simulazione di un ascensore.
I due processi sono visti come incapsulati entro moduli, come detto nel precedente paragrafo (Fig. 17.3).
Il monitor "prenotazione" e' una risorsa comune cui si puo' accedere in modo concorrente dai due moduli.

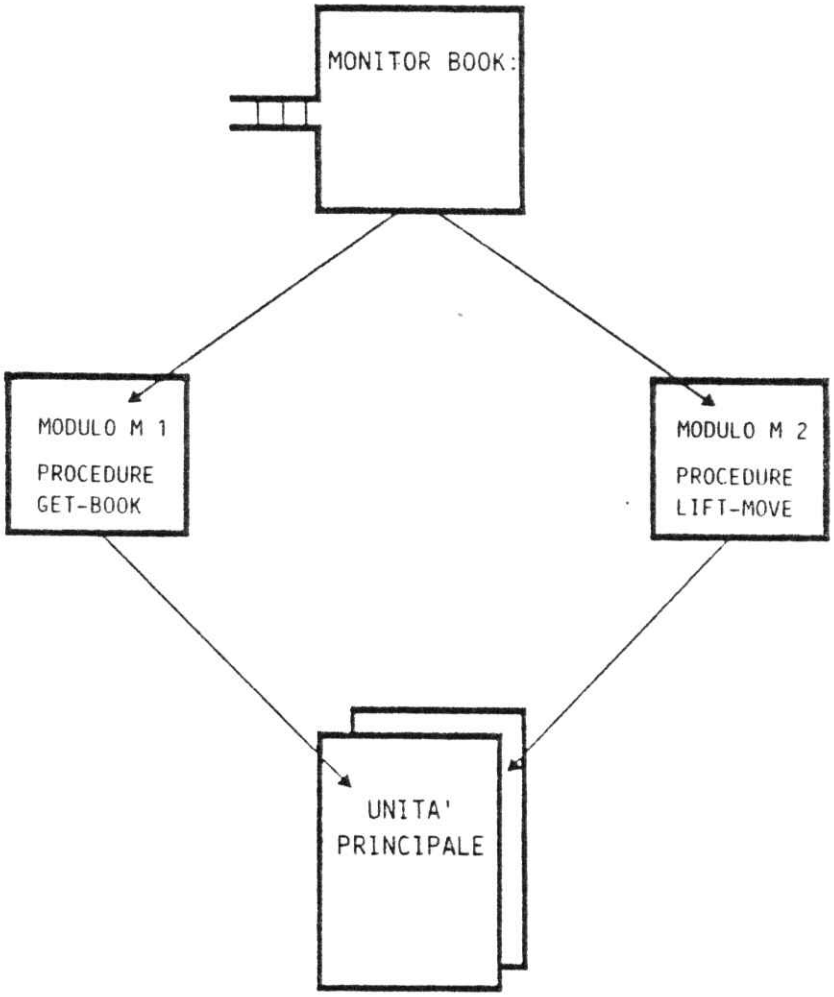


Fig. 17-9 Struttura del Programma che Simula un Ascensore

```
MONITOR BOOK;
 CONST MAXFLOORS = 9;
 TYPE UPDOWN = (UP, DOWN);
      FLOORS = 0..MAXFLOORS;
 VAR BOOKED: ARRAY [UPDOWN] OF SET OF FLOORS;
      HOWMANY : ARRAY [UPDOWN] OF INTEGER;
      ACTUAL_FLOOR : FLOORS;


 PROCEDURE INIT;
   BEGIN
    BOOKED [UP] := [];
    BOOKED [DOWN] := [];
    ACTUAL_FLOOR := 0;
    HOWMANY [UP] := 0;
    HOWMANY [DOWN] := 0
   END;


PROCEDURE GETB (N:FLOORS;DIR:UPDOWN);
   BEGIN
    IF NOT (N IN BOOKED [DIR]) THEN
    BEGIN
     BOOKED [DIR] := BOOKED [DIR] + [N];
     HOWMANY [DIR] := HOWMANY [DIR] + 1
    END
   END;


FUNCTION EMPTY (DIR:UPDOWN) : BOOLEAN;
  BEGIN
   EMPTY := HOWMANY [DIR] = 0
  END;


PROCEDURE ANNUL (N:FLOORS;DIR:UPDOWN);
  BEGIN
   IF EMPTY (DIR) THEN
    WRITELN ('ERROR!')
   ELSE
   BEGIN
    BOOKED [DIR] := BOOKED [DIR] - [N];
    HOWMANY [DIR] := HOWMANY [DIR] -1
   END
END;



FUNCTION PRESENT (N:FLOORS;DIR:UPDOWN) : BOOLEAN;
  BEGIN
   PRESENT := N IN BOOKED [DIR]
  END;



EXPORT ACTUAL_FLOOR,INIT,GETB,EMPTY,ANNUL,PRESENT;.
```

```
MODULE M1;

IMPORT GETB,ACTUAL_FLOOR FROM BOOK;
 .
 .       (* Ridefinizione di risorse importate*)
 .
PROCEDURE GET_BOOK;
 PROCEDURE DIE; EXTERNAL;
 CONST MAXFLOORS = 9;
 TYPE FLOORS = 0..MAXFLOORS;
      UPDOWN = (UP, DOWN);
      INOUT = (INSIDE, OUTSIDE);
  VAR C:CHAR;
      FLO : FLOORS;

 PROCEDURE GETANDPRINT (N:FLOORS;DIR:UPDOWN;SIDE:INOUT);
   BEGIN
    BOOK.GETB(N,DIR)
    (* STAMPA DEL PIANO PRENOTATO SUL DEVICE DI OUTPUT*)
   END;

BEGIN
 READ (C);
 WHILE NOT EOF DO
  BEGIN
   IF C IN ['0'..'9'] THEN
   BEGIN
    FLO := ORD (C) - ORD ('0');
    IF FLO < BOOK.ACTUAL_FLOOR THEN
      GETANDPRINT (FLO,UP,INSIDE)
    ELSE
      GETANDPRINT (FLO,DOWN,INSIDE)
   END
 ELSE
   CASE C OF
   'P' : GETANDPRINT (0, UP, OUTSIDE);
   'Q' : GETANDPRINT (1, UP, OUTSIDE);
    .
    .
   'M' : GETANDPRINT (9, DOWN, OUTSIDE)
   OTHERWISE
  END;
 READ (C)
 END;
 DIE
END;
EXPORT GET_BOOK;.
```

```
MODULE M2;

IMPORT EMPTY,ANNUL,PRESENT,
       ACTUAL_FLOOR,INIT FROM BOOK;

VAR BOOK.ACTUAL_FLOOR:FLOORS;
 .
 .       (* Ridefinizione delle routine importate*)
 .
PROCEDURE LIFT_MOVE;
 CONST MAXFLOORS = 9;
 TYPE FLOORS = 0..MAXFLOORS;
      UPDOWN = (UP,DOWN);

 PROCEDURE MOVE (DIR:UPDOWN);
   . (* CALCOLA IL PIANO DESIDERATO CHIAMANDO
   . "BOOK.PRESENT", E STAMPA IL MOVIMENTO SUL
   . DEVICE DI OUTPUT *)

 BEGIN
  BOOK.INIT;
  (* STAMPA IL DISEGNO DELLA CASA SUL DEVICE DI OUTPUT*)
  REPEAT
   IF NOT BOOK.EMPTY (UP) THEN
     MOVE (UP);
   IF NOT BOOK.EMPTY (DOWN) THEN
     MOVE (DOWN);
  UNTIL FALSE
 END;
 EXPORT LIFT_MOVE;.
```

SINCRONIZZAZIONE DI PROCESSI COOPERANTI

Si definiscono "cooperanti" due (o piu') processi concorrenti che, essendo a conoscenza dell'esistenza reciproca, operano insieme su di una risorsa comune, per risolvere globalmente un certo problema.

La cooperazione e' spesso effettuata con la sincronizzazione dei processi, cioe' bloccando e riattivando la loro esecuzione al momento opportuno; in particolare, se un certo processo acquisisce una risorsa di cui avrebbero bisogno altri processi, e' opportuno che la rilasci dopo un opportuno intervallo di tempo.

Il PASCAL+ fornisce alcune risorse predefinite, che permettono il controllo manuale dell'esecuzione del processo, cioe' consentono ad un processo di attendere che un altro processo invii un segnale, oppure ad un processo di far partire l'esecuzione di un altro processo.
Questo meccanismo e' realizzato tramite code addizionali di processi entro un monitor, definite dall'utente. Queste code sono variabili del tipo predefinito "CONDITION" (vedere il prossimo paragrafo).

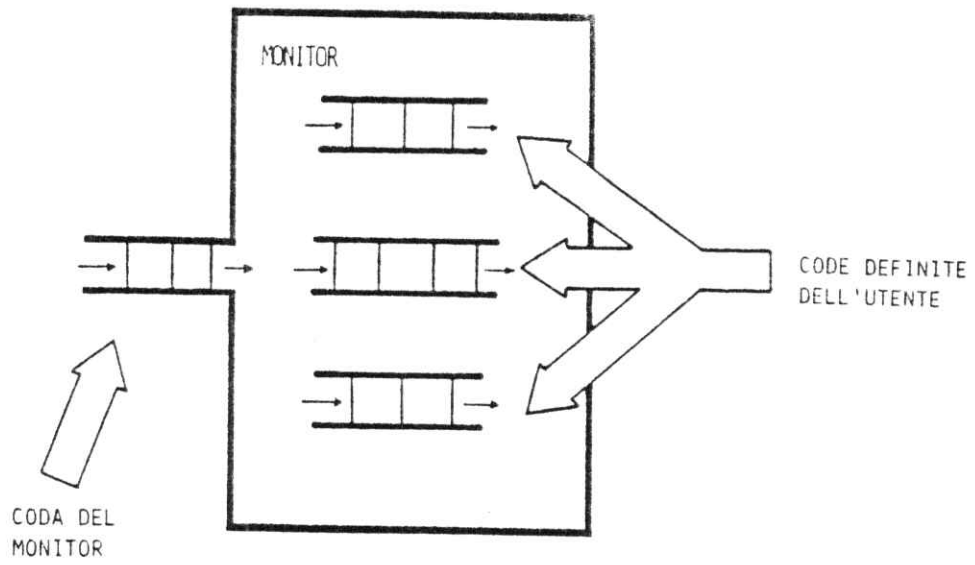


Fig. 17-10 Rappresentazione Grafica delle Variabili del Tipo Condition

La "coda del monitor" e' invisibile all'utente; essa e' gestita dal sistema automaticamente e fornisce la mutua esclusione.
Le "code definite dall'utente" sono variabili definite entro un monitor e gestite attraverso routine predefinite.
Il controllo dell'esecuzione di un processo e' realizzato chiamando, dal processo stesso, apposite procedure definite dall'utente, incaricate di "accodare" il processo attuale, o di "deaccodare" un certo processo.

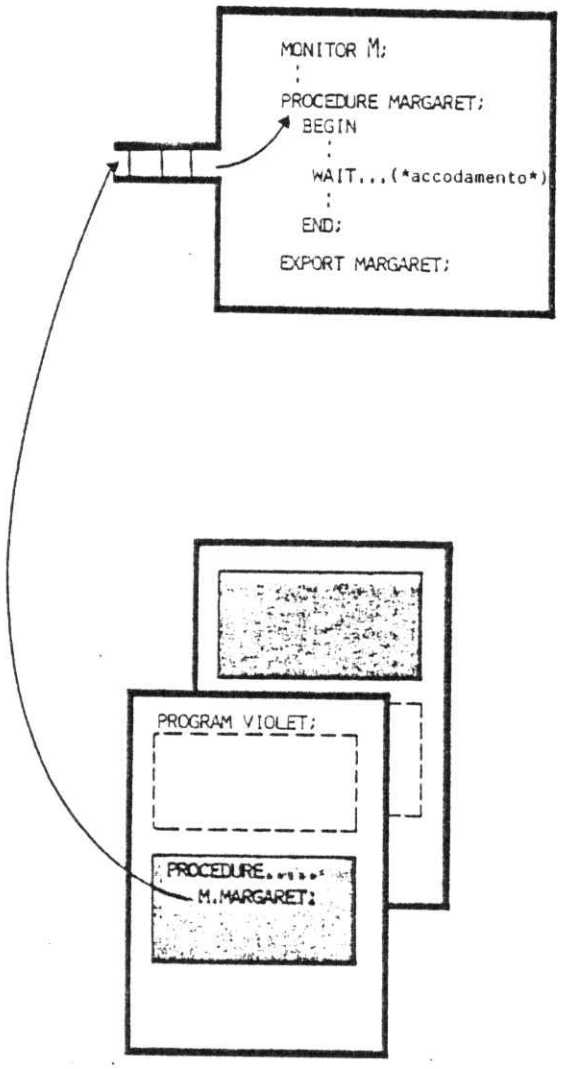


Fig. 17-11 Come Far Accodare un Processo in una Coda Definita dall'Utente

Ovviamente, e' responsabilita' dell'utente assicurare l'uso corretto di queste code.

IL TIPO CONDITION

Il tipo CONDITION e' un tipo predefinito in PASCAL+; puo' essere usato solo nei monitor, mai nei moduli o nei programmi.

Una variabile di tipo CONDITION e' una coda di processi; puo' essere gestita solo dalle routine predefinite WAIT e SIGNAL, e puo' essere passata come parametro solo per variabile, non per valore.
E' automaticamente inizalizzata al valore "vuota" all'inizio dell'esecuzione del programma.
Le variabili di tipo CONDITION non possono essere esportate e non possono essere usate come parametri di routine esportate.

LA PROCEDURA WAIT

La procedura WAIT e' una procedura predefinita che puo' essere chiamata solo in un monitor, e non nei programmi e nei moduli.

Tale procededura accetta un singolo parametro per variabile di tipo CONDITION ed "accoda" in esso il processo attuale; dopo di cio' "deaccoda" il primo processo disponibile nella coda di monitor e lo manda in esecuzione.

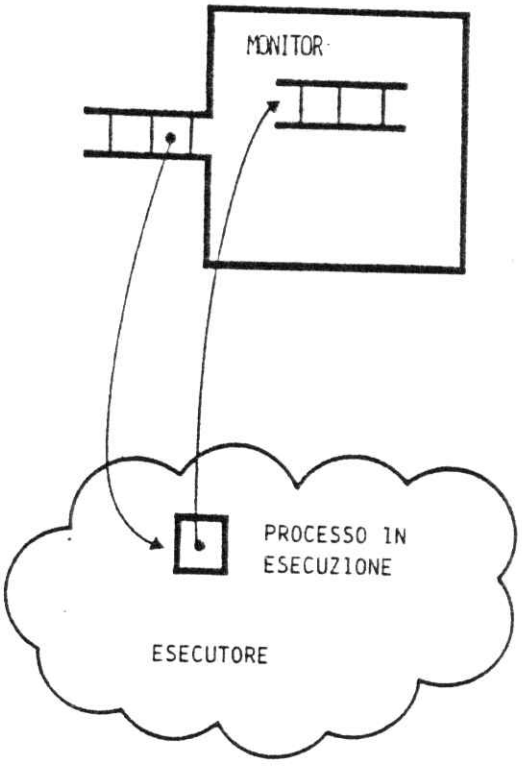


Fig. 17-12 Effetto di una Chiamata alla Procedura WAIT

Se la coda di un monitor e' vuota, il semaforo del monitor e' resettato. Non viene fatto alcun controllo per evitare situazioni di stallo, ossia

non esiste alcuna prevenzione per le situazioni in cui due o piu' processi si aspettano a vicenda per un tempo indefinito. E' l'utente che deve preoccuparsi della corretta definizione e gestione dei semafori.

Esempio:

```
MONITOR HOSPITAL;
VAR HALL : CONDITION;
    NURSE : (STOPPING, FORWARDING); (* SEMAFORO *)
PROCEDURE MEDICAL_EXAMINATION;
 BEGIN
  IF NURSE = STOPPING THEN WAIT (HALL);
  (* entra nello studio medico *)
  .
  .
END;
```

La procedura WAIT non puo' essere esportata.

LA PROCEDURA SIGNAL

SIGNAL e' una procedura predefinita in PASCAL+; puo' essere chiamata solo entro un monitor, non e' possibile chiamarla in un programma o in un modulo. Accetta un unico parametro variabile di tipo CONDITION e "accoda" il processo attuale nelle code del monitor, dopo di cio' "deaccoda" il primo processo disponibile nella coda passata come parametro e lo manda in esecuzione.

Se la condizione e' vuota, SIGNAL non ha alcun effetto.

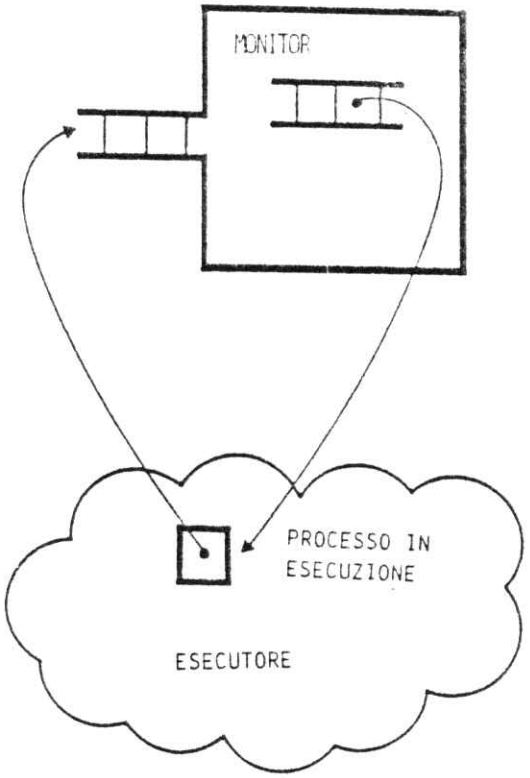


Fig. 17-13 Effetto della Chiamata della Procedura SIGNAL

Se la chiamata alla SIGNAL precede immediatamente la "END" della routine ...

```
PROCEDURE MEDICAL_EXAMINATION;
  BEGIN
    .
    .
    SIGNAL (HALL)
    END;
```

allora viene effettuato un ritorno implicito dalla routine ed il processo in atto non e' posto nella coda del monitor; il controllo passera' direttamente dal monitor al processo chiamante. In questo caso il processo in atto e quello deaccodato vengono eseguiti contemporaneamente.

La procedura SIGNAL non puo' essere esportata.

ESEMPIO

Si consideri ora un esempio riguardante la gestione di un buffer.

Si hanno due processi cooperanti: di nome PRODUTTORE e CONSUMATORE.
Il processo PRODUTTORE pone una informazione in un buffer, il processo CONSUMATORE prende l'informazione dal buffer.
La risorsa comune ai due processi e' il buffer, pertanto viene racchiuso in un monitor.

Se questi due processi hanno velocita' di esecuzione diverse (dovuta ad una qualsiasi ragione), possono nascere alcune anomalie; infatti, si supponga che il processo PRODUTTORE sia piu' veloce del processo CONSUMATORE: in questo caso una parte di informazione puo' essere sovrapposta nel buffer prima che il CONSUMATORE possa accedervi.
D'altro canto se il processo CONSUMATORE e' piu' svelto del PRODUTTORE, l'informazione memorizzata nel buffer puo' essere reperita due (o piu') volte prima che il PRODUTTORE possa sostituirla con una nuova.

Percio', e' necessaria una sincronizzazione tra i due processi, in modo che un processo attenda finche' l'altro abbia terminato la sua funzione, e viceversa.

Si supponga che il programma sia strutturato come in Fig. 17.2.

```
PROGRAM SYNCHRONIZATION_EXAMPLE (INPUT,OUTPUT);
 .
 .
IMPORT SEND, RECEIVE, START FROM BUFFER;
 .
 .      (* Ridefinizione delle routine importate *)
 .
 .
 .      (* Definizione delle routine esterne *)
 .
PROCEDURE PRODUTTORE; (* PRIMO PROCESSO EFFETTIVO *)
 BEGIN
  WHILE NOT EOF (INPUT) DO
   BEGIN
    BUFFER.SEND (INPUT^);
    GET (INPUT)
   END
  END;

PROCEDURE CONSUMATORE; (* SECONDO PROCESSO EFFETTIVO *)
 BEGIN
  WHILE TRUE DO
   BEGIN
    BUFFER.RECEIVE (OUTPUT^ );
    PUT (OUTPUT)
   END
 END;

BEGIN
 BUFFER.START;
  .
  .    (* Meccanismo di creazione dei processi *)
  .
END.


MONITOR BUFFER;

VAR BUF : CHAR;                    (* BUFFER *)
    PRESENT, FREE : CONDITION; (* CODE *)
    EMPTY : BOOLEAN;               (* SEMAFORO *)

PROCEDURE SEND (C:CHAR);
 BEGIN
  IF NOT EMPTY THEN WAIT (FREE);
  EMPTY := FALSE;
  BUF := C;
  SIGNAL (PRESENT)
 END;

PROCEDURE RECEIVE (VAR C : CHAR);
 BEGIN
  IF EMPTY THEN WAIT (PRESENT);
  EMPTY := TRUE;
  C := BUF;
  SIGNAL (FREE)
 END;
```

```
 PROCEDURE START;
  BEGIN
   EMPTY := TRUE
  END;

EXPORT SEND, RECEIVE, START;.
```

18. ESTENSIONI ALLE LIBRERIE DI RISORSE

Scopo di questo capitolo e' di illustrare alcune estensioni che riguardano i moduli ed i monitor.

Viene considerata la istanziazione delle librerie di risorse, cioe' la possibilita' di usare piu' di una volta le risorse racchiuse in un modulo o in un monitor.
Il modo di operare e la sintassi di ciascuna risorsa istanziata puo' essere reperita in altre parti di questo manuale, ma viene qui ripetuta a scopo di completezza.

ISTANZE DI LIBRERIE DI RISORSE

COS'E' UN'ISTANZA DI LIBRERIA DI RISORSE

Se una libreria di risorsa definisce un tipo di dati incapsulato, questo puo' essere utilizzato come "denotatore di tipo strutturato" nell'unita' che lo istanzia.
Cio' significa che, all'interno di una unita' che importa, puo' essere assegnato un identificatore di tipo ad un modulo o ad un monitor, usato per definire alcune variabili.

Si consideri, ad esempio, un modulo che definisca il tipo dati stack di colori:

```
MODULE STACK_OF_COLOURS ...
  .
  .
  .
EXPORT ....;.
```

Nella unita' che importa si possono definire alcune variabili, ovvero S1, S2, S3 ed S4 di tipo stack di colori:

```
PROGRAM RASPBERRY;
TYPE T = MODULE STACK_OF_COLOURS;
VAR S1, S2, S3, S4 : T;
IMPORT ...FROM  T;
 .
 .
 .
```

In altri termini si puo' dire che il programma definisce quattro stack; o meglio il programma "istanzia" il modulo quattro volte, dando quattro nomi: S1, S2, S3 ed S4.
Da ora in poi, questi nomi verranno chiamati "denotatori di istanza".
Ognuna di queste istanze comporta l'allocazione di una copia distinta

delle variabili del modulo (non importa se le variabili sono esportate oppure no).

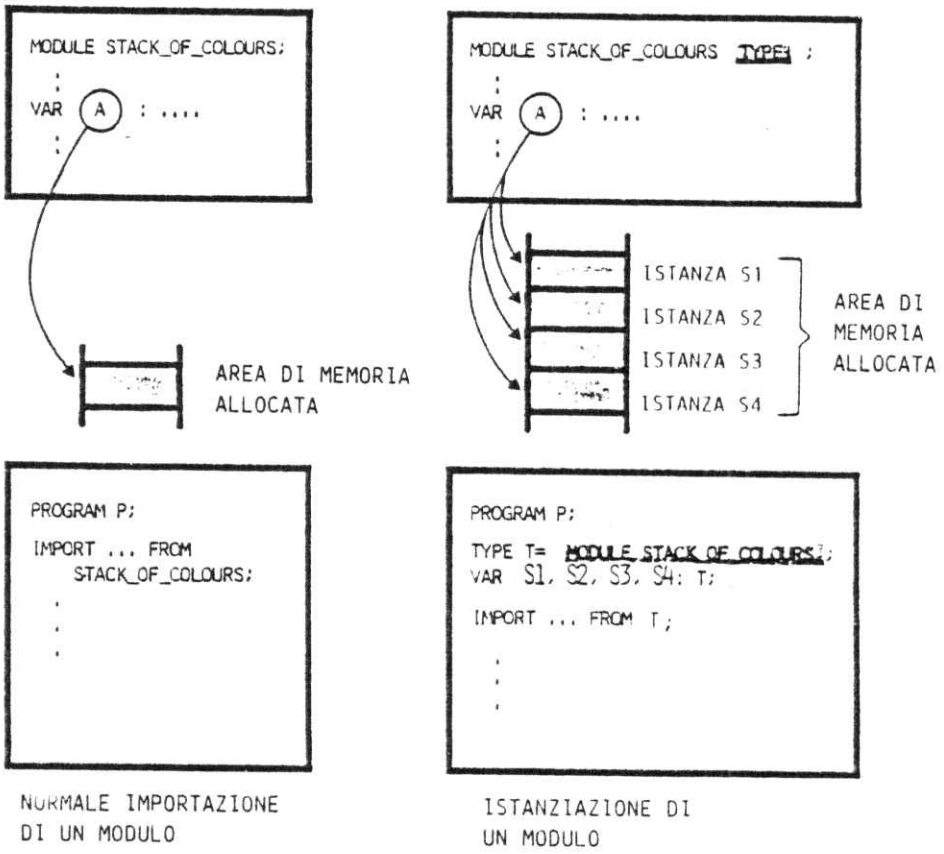


Fig. 18-1 Allocazione dell'Area di Memoria per Moduli Istanziati

Si noti che una libreria di risorse istanziabili e' contrassegnata dalla parola chiave TYPE che segue l'identificatore della libreria di risorse; questo dettaglio viene spiegato in seguito.

All'interno del programma che istanzia, si possono usare questi stack di colori in modo indipendente, riempiendoli con dati differenti ed accedendo ad essi separatamente.

Si consideri, ora, il caso di una libreria di risorse di nome M, importata da unita' differenti, dette U1, U2, ...Un, ed istanziate in ciascuna unita'.

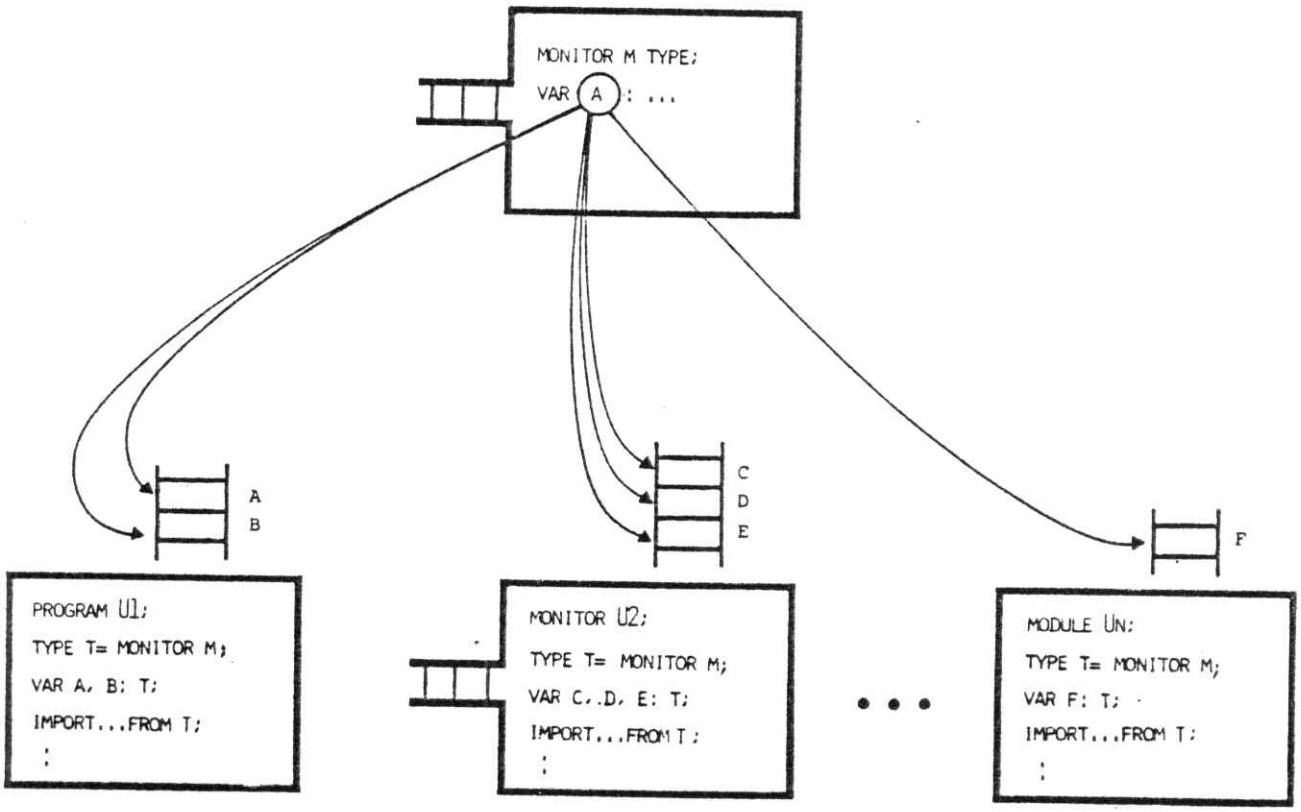


Fig. 18-2 Diverse Unita' che Istanziano una Libreria di Risorse

Infine, si consideri il caso di librerie di risorse diverse istanziate da un unico programma.

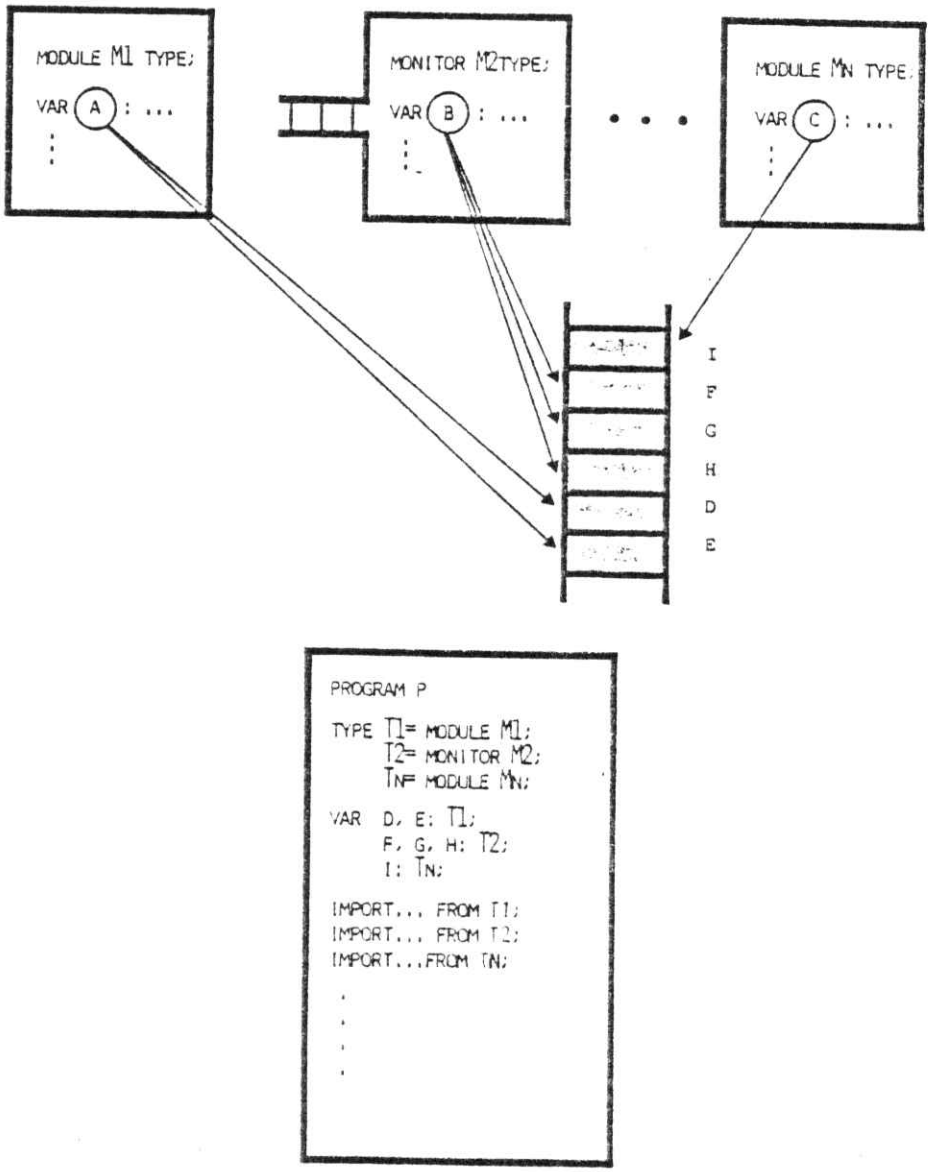


Fig. 18-3 Allocazione dell'Area di Memoria in Istanziazione Multipla

COMPILAZIONE DI LIBRERIE DI RISORSE ISTANZIABILI

Una libreria di risorse istanziabili deve essere sempre compilata prima della compilazione dell'unita' che la istanzia.

La compilazione di una libreria di risorse istanziabile implica la creazione di un file di informazioni, utilizzato automaticamente durante la compilazione dell'unita' che la istanzia.

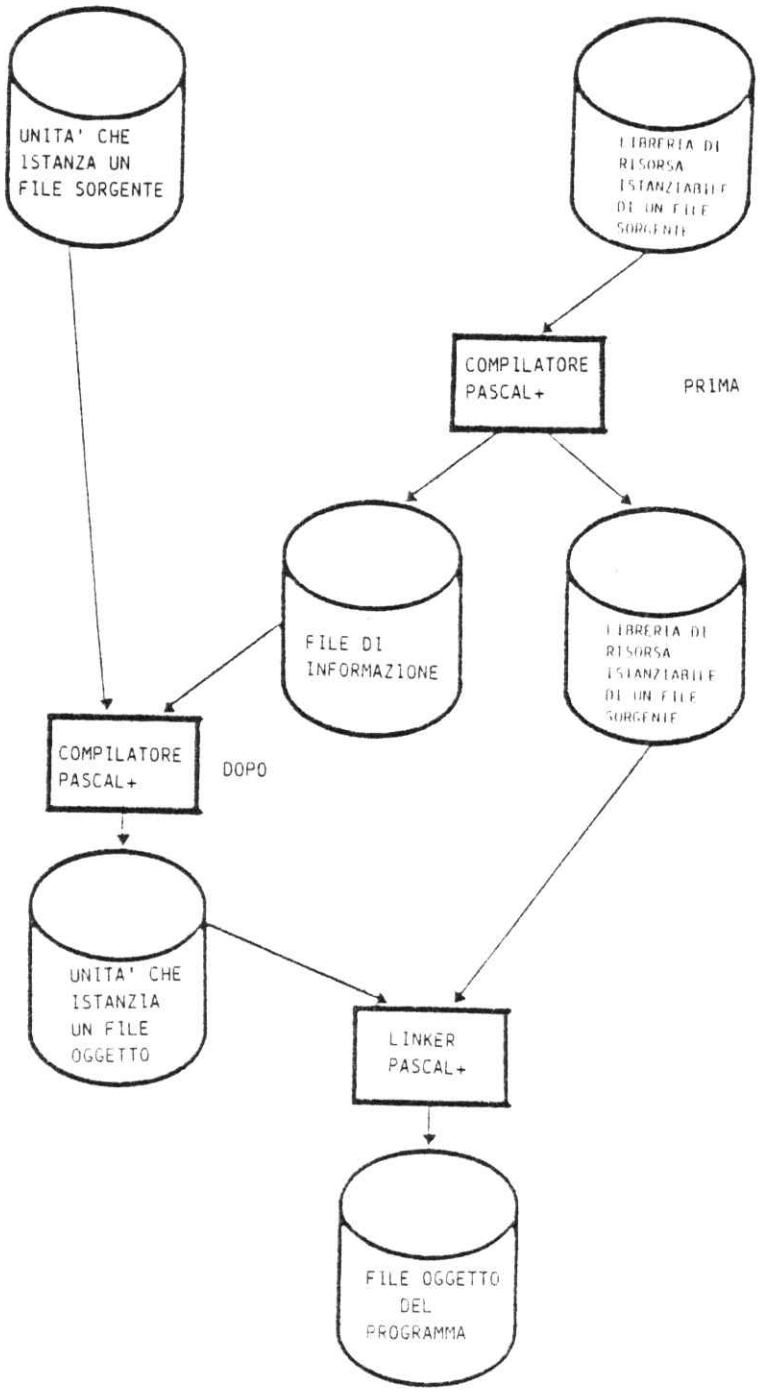


Fig. 18-4 Compilazione e Collegamento di una Libreria di Risorsa Instanziabile

Il file di informazioni contiene dati sulla dimensione del tipo strutturato definito dalla libreria di risorse stessa.
Cio' e' indispensabile per poter compilare la dichiarazione:

```
TYPE T : MODULE M;
```

oppure

```
TYPE T : MONITOR M;
```

Qualora una libreria di risorse istanziabile sia modificata e ricompilata, si e' obbligati a ricompilare (tramite linker) tutte le unita' che la importano.

D'altro canto, librerie di risorse non istanziabili (dette "librerie di risorse statiche") possono essere compilate dopo la compilazione delle unita' che le importano; in questo caso una modifica alle librerie di risorse non richiede la ricompilazione dell'unita' che effettua l'importazione, ma solo il ricollegamento.

DIAGRAMMA SINTATTICO DI LIBRERIA DI RISORSE INSTANZIABILI

Una libreria di risorse istanziabili e' strutturalmente identica ad una libreria di risorse statiche, ad eccezione che essa viene contrassegnata con la parola chiave "TYPE" che segue l'identificazione di modulo o monitor.

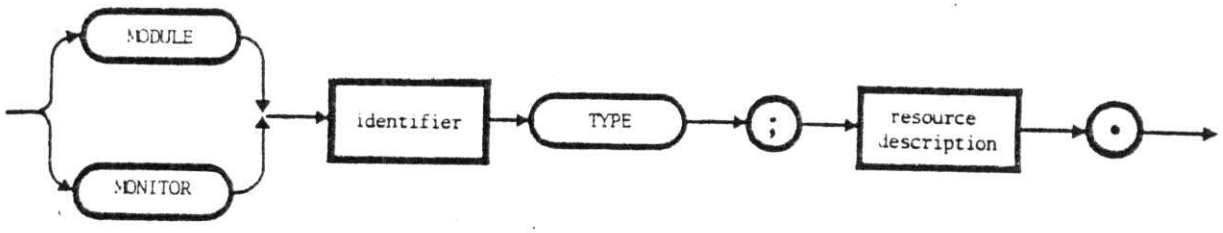


Fig. 18-5 Diagramma Sintattico di Libreria di Risorse Istanziabili

La parola chiave TYPE informa il compilatore che deve essere creato il "file di informazioni".

COME ISTANZIARE UNA LIBRERIA DI RISORSE

Una libreria di risorse viene istanziata in una unita' definendola come denotatore di tipo strutturato.
Il seguente e' il diagramma sintattico del "denotatore di tipo strutturato", contenente in questo caso l'opzione per un modulo o monitor istanziabile.

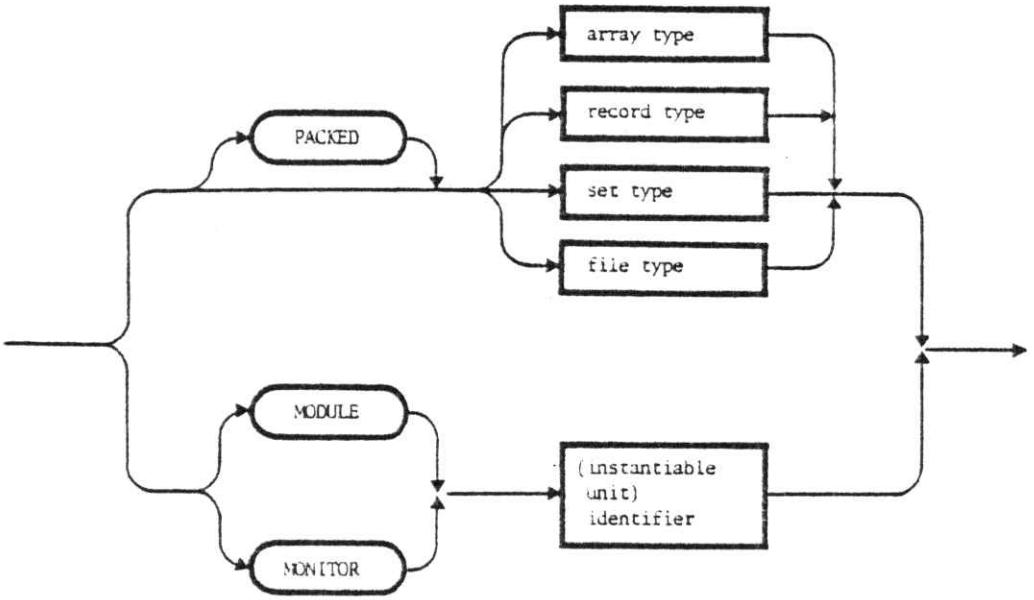


Fig. 18-6 Diagramma Sintattico per Denotatore di Tipo Strutturato

Una libreria di risorse puo' essere istanziata in tre modi:

1) usando una descrizione di tipo:

```
.
.
TYPE T = MODULE M;
VAR A,B,C : T; (* A,B,C sono i denotatori di istanze *)
.
.
```

2) direttamente in una descrizione di variabile:

```
.
.
VAR A,B,C : MODULE M; (* A,B,C  sono i denotatori di istanze *)
```

3) usando l'istruzione NEW:

```
.
.
TYPE T = MODULE M;
VAR P =^T;
.
.
NEW (P); (* P^ e' il denotatore di istanza *)
.
.
```

Nell'ultimo caso il numero di istanze non e' prefissato ed il denotatore di istanza e' "P^".

Una libreria di risorse istanziabili, cioe' contrassegnata dalla parola chiave TYPE, deve sempre essere istanziata nell'unita' che la importa: se si ha bisogno di una unica occorrenza la si istanziera' una volta sola.

Esempio:

```
PROGRAM P;
VAR A : MONITOR M;
.
.
```

Se la libreria di risorse istanziate e' un monitor, non e' garantita la mutua esclusione tra i processi che accedono a diverse istanze, cioe' due (o piu') processi PASCAL+ che chiamano contemporaneamente due (o piu') routine appartenenti ad istanze diverse non sono posti in sequenza.

Se la logica del programma richiede una mutua esclusione tra le istanze, si puo' usare un monitor "auto-istanziabile" ossia un monitor non istanziato da programma ma istanziato da se stesso.

Esempio:

```
MONITOR M TYPE;
TYPE T = MONITOR M; (*AUTOISTANZA*)
VAR I = ARRAY [1..5] OF^ T; (*5 ISTANZE RAGGRUPPATE IN UN ARRAY*)

PROCEDURE AUTO_INSTANCE; (*CREA LE CINQUE ISTANZE*)
 VAR N : 1..5;
 BEGIN
  FOR  N := 1 TO 5 DO NEW (I [N])
 END;

VAR A,B,...:...; (*RISORSE USATE DA PROGRAMMA*)

PROCEDURE P;
FUNCTION F :...;
  .
  .
EXPORT I, AUTO_INSTANCE, A,B,P,F,...;
```

Si noti che le autoistanze sono create in run-time, ovvero usando un puntatore al monitor.
Si vede ora una parte di un programma che usa alcune risorse di alcune istanze del monitor:

```
PROGRAM CHERRY;
  .
  .
IMPORT I, AUTO_ISTANCE,A,B,P,F,...FROM M;
VAR MM : MONITOR M;
  .
  .
BEGIN
MM.AUTO_INSTANCE; (*CREAZIONE DELLE CINQUE ISTANZE*)
  .
  .
IF FORK... (*CREAZIONE DEI PROCESSI*)
  .
  .
MM.I [3]^.P  (*CHIAMATA DELLA PROCEDURA P DELLA TERZA ISTANZA DI M*)
  .
  .
MM.I [1]^.A  (*UTILIZZO DELLA VARIABILE A DELLA PRIMA ISTANZA DI M*)
  .
  .
END.
```

RIDEFINIZIONE ED USO DELLE RISORSE ISTANZIATE

Le risorse che sono importate da una libreria di risorse istanziate, dette "risorse istanziate", devono essere ridefinite all'interno dell'unita' che le istanzia.

A differenza delle librerie di risorse statiche, la denotazione delle risorse istanziate all'interno della ridefinizione e' diversa dalla denotazione delle stesse all'interno dalla parte istruzioni.
Nella ridefinizione, le risorse istanziate sono denotate

dall'identificatore della libreria di risorse seguito dal simbolo "." e dall'identificatore di risorsa.

Esempio:

```
PROGRAM CIGARET;
IMPORT POP,EMPTY,HOW_MANY,C FROM STACK;
VAR S1,S2,S3,S4: MODULE STACK;
    STACK.HOW_MANY:INTEGER;
PROCEDURE STACK.POP (VAR I:COLOUR); DEFINITION;
FUNCTION STACK.EMPTY : BOOLEAN; DEFINITION;
(*NOTARE : C NON DEVE ESSERE RIDEFINITA*)
  .
  .
```

Le risorse istanziate sono usate, negli statement, denotandole con il denotatore di sistema seguito dal simbolo "." e dal denotatore di risorsa.

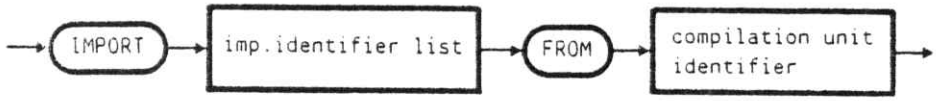
Esempio:

```
  .
  .
BEGIN
  .
  .
IF NOT S1.EMPTY THEN S1.POP (X);
  .
  .
A := S3.HOW_MANY + S4.HOW_MANY;
  .
  .
REPEAT
  S1.POP (X);
  S3.POP (Y);
  S2.POP (Z)
UNTIL FOUND;
  .
  .
A := S3.C - 18 + SUCC (S2.C);
  .
  .
END.
```

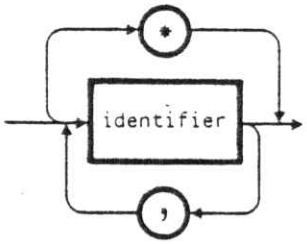
LISTA DI IMPORTAZIONE

La lista di importazione ("imports list") specifica tutte le procedure, funzioni, variabili, tipi e costanti che vengono importati dall'ambiente globale a quello attuale.

IMPORTS LIST



IMP. IDENTIFIER LIST



COMPILATION UNIT IDENTIFIER

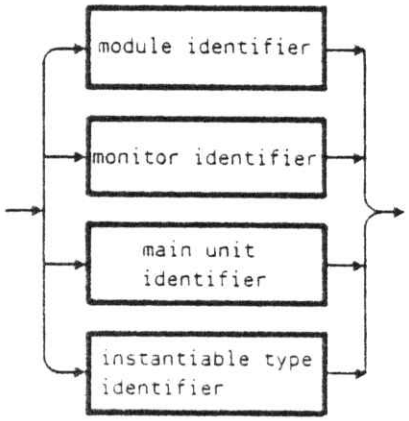


Fig. 18-7 Diagramma Sintattico della Lista di Importazione

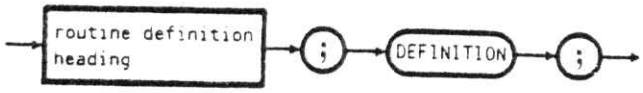
I tipi e le costanti non possono essere importate da un modulo o monitor instanziabile.

Se un oggetto P (procedura, variabile, tipo o costante) e' importato dal modulo M, allora nella lista di importazione adesso si fa riferimento come P (importazione di P da M), mentre in seguito ci si riferisce a P come M.P, includendo la definizione di P (se presente).

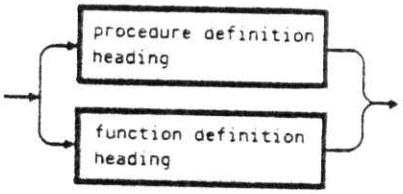
DEFINIZIONE DI ROUTINE

Qualsiasi routine importata puo' essere definita nell'ambiente di importazione con una "definizione di routine", che definisce il tipo e la posizione dei parametri per la routine e possibilmente il tipo del risultato.

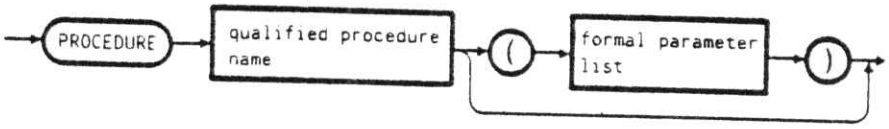
ROUTINE DEFINITION



ROUTINE DEFINITION HEADING



PROCEDURE DEFINITION HEADING



FUNCTION DEFINITION HEADING

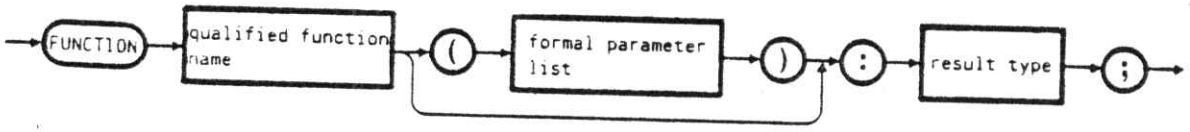
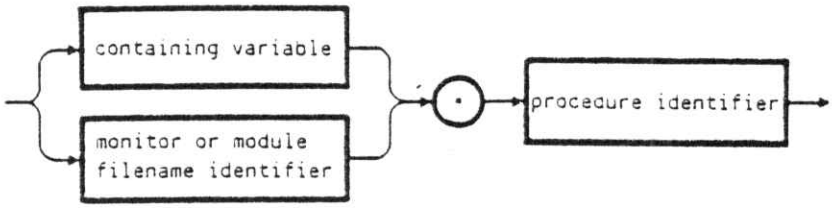
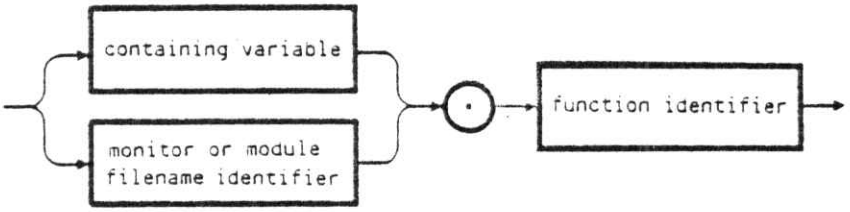


Fig. 18-8 Diagramme Sintattico per le Definizione di Routine (Cont).

QUALIFIED PROCEDURE NAME



QUALIFIED FUNCTION NAME



CONTAINING VARIABLE

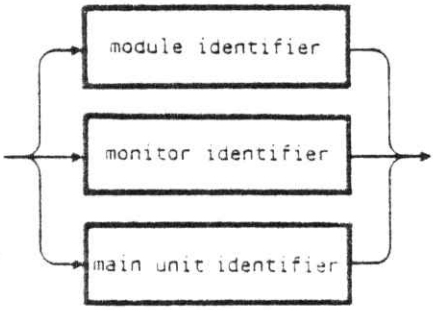


Fig. 18-8 Diagramma Sintattico per la Definizione di Routine

DEFINIZIONE DI VARIABILE, COSTANTE, TIPO

Qualsiasi variabile, costante o tipo importato, puo' essere definito nell'ambiente di importazione con una dichiarazione di variabile.
Lo scopo di effettuare tale ridefinizione e' di eseguire un controllo sui tipi.

LISTA DI ESPORTAZIONE

La lista di esportazione ("exports list") specifica le procedure, funzioni, variabili, tipi e costanti che devono essere esportate dall'ambiente attuale a quello globale.
E' possibile solo l'esportazione da un'unita' principale, un modulo o un monitor e solo al livello piu' esterno. Non e' possibile esportare tipi o costanti da un modulo o monitor instanziabile.

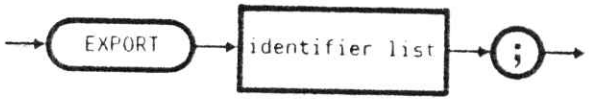


Fig. 18-9 Diagramma Sintattico della Lista di Esportazione

Un modulo che esporta alcuni oggetti deve essere compilato, in una directory di libreria, prima dei moduli che importano alcuni oggetti da essa, affinche' sia possibile l'accesso al file di informazione.
Quando un oggetto e' esportato da un modulo A e importato da un modulo B, la dichiarazione dell'oggetto in A e la sua definizione in B (se presente) deve essere identica, altrimenti il risultato sara' errato.

PARTE V. PRESTAZIONI DEL LINGUAGGIO ESTESO

19. COSTANTI

Il linguaggio PASCAL+ esteso permette la denotazione di costanti con l'uso di basi differenti.
Nei programmi e' possibile usare la rappresentazione ottale o esadecimale dei valori interi.

Inoltre, nelle costanti di tipo stringa si possono usare caratteri non stampabili rappresentandoli in un modo particolare.

Infine viene mostrato l'uso delle "costanti istanziate", cioe' costanti importate da unita' istanziate.

COSTANTI OTTALI

Nel linguaggio PASCAL+ una costante ottale e' rappresentata dal simbolo "#" seguito da una serie di cifre ottali, cioe' cifre dell'intervallo 0 .. 7.

Esempi:

```
#05371
#6
#0006
#2222222
#0
#000
```

Negli esempi precedenti la prima costante rappresenta il numero 2809; la seconda e terza rappresentano il numero 6; la quarta costante il numero 599186 e gli ultimi due rappresentano il numero 0.

Le costanti ottali sono di tipo INTEGER: esse possono essere usate nelle espressioni come normali costanti intere.

Esempio:

```
CONST C = #32;
VAR I: INTEGER;
 .
 .
 .
I := 99 * (I + #6) DIV C;
 .
 .
 .
```

Le costanti negative ottali sono precedute dall'operatore meno "-".

Esempi:

```
- #333
- #0
```

L'operatore meno non e' permesso all'interno delle costanti ottali; un esempio di errore e' il seguente:

```
# -671      (* errore *)
```

Se si conosce l'occupazione di memoria di un numero intero (per tale argomento vedere la documentazione relativa all'implementazione), si puo' rappresentare un numero negativo come un numero memorizzato nell'elaboratore con il bit di segno posizionato ad uno.
Ad esempio, se un numero intero e' memorizzato in 24 bit, si puo' rappresentare il numero -32 come #77777740 (assumendo l'interpretazione in complemento a due).

COSTANTI ESADECIMALI

In PASCAL+ una costante esadecimale e' rappresentata dal simbolo "%" seguito da una serie cifre esadecimali, cioe' cifre appartenenti all'intervallo 0 .. 9 e lettere contenute nell'intervallo A .. F.

Esempi:

```
%3B9
%E
%00E
%0D333
%0
%0000
```

Negli esempi precedenti la prima costante rappresenta il numero 953; la seconda e terza costante rappresentano il numero 14; la quarta costante il numero 906035 e le ultime due sono la rappresentazione di 0.

Le costanti esadecimali sono di tipo INTEGER e possono essere usate nelle espressioni come normali costanti intere.

Esempio:

```
CONST C = %B0;
VAR I : INTEGER;
 .
 .
 .
I = 288 * (I+ %2C) MOD C;
 .
 .
 .
WRITE (%D);
.
.
```

Nell'esempio si noti che l'ultima istruzione fa' stampare il numero intero 13 sulla periferica standard di output.

Le costanti esadecimali negative sono precedute dall'operatore meno "-", e seguono la regola vista per le costanti ottali:

```
-%3BF      (* numero negativo *)

%-A        (* errore *)
```

Comunque, si puo' sfruttare il bit del segno della rappresentazione di memoria, per ottenere i numeri negativi:

```
%FFE0      = -32
```

assumendo una occupazione di memoria di 16 bit per i numeri interi e l'interpretazione in complemento a due.

SPECIALI INTERPRETAZIONI DI STRINGHE

In una stringa si puo' rappresentare un carattere (anche non stampabile) con il simbolo slash "\" seguito dal numero intero corrispondente nella tabella di codifica.

Si consideri la tabella del codice ASCII: si puo' rappresentare il carattere "A" nei seguenti modi:

```
'A'
'\65'
'\#101'
'\%41'
```

Ovviamente l'intervallo consentito per i numeri che seguono il simbolo slash " \" dipende dall'implementazione.

Questo metodo di rappresentare i caratteri puo' essere effettuato anche con l'uso della funzione predefinita CHR; per esempio la rappresentazione di "A" puo' essere:

```
'A'
CHR (65)
CHR (#101)
CHR (%41)
```

L'uso del simbolo slash nelle stringhe e' utile poiche' permette la manipolazione dei caratteri di controllo.
Per esempio si considerino i casi seguenti:

```
WRITELN('AAA');
WRITELN('BBB');
WRITELN('CCC');
WRITELN
```

Questa sequenza di istruzioni puo' essere sostituita da:

```
WRITE ('AAA\10\13BBB\10\13CCC\10\13\10')
```

Dove la sequenza " \10" causa un salto linea e "\13" causa un ritorno a capo.

Inoltre si possono ottenere effetti speciali sovrapponendo due caratteri: ad esempio la seguente istruzione:

```
WRITE (printer, '\13 =\13/')
```

fa' stampare il simbolo "#" nella prima colonna della riga di stampa.

COSTANTI ISTANZIATE

Se una unita' istanzia una libreria di risorse ed importa da essa alcune costanti, tali costanti saranno indicate dal denotatore di istanza, seguito dal simbolo "." e dall'identificatore di costante.

Esempio:

```
MONITOR LION TYPE;
CONST C1 = 13;
      C2 = %9B;
 .
 .
 .
EXPORT C1, C2;


PROGRAM TIGER;
IMPORT C1, C2 FROM LION;
TYPE T = MONITOR LION;
VAR X,Y,Z : T;
 BEGIN
 .
 .
 I := X.C1;
 WRITE (Z.C2);
 IF Y.C1 <> 0 THEN...
 .
 .
 END.
```

Si veda il capitolo 18 per maggiori dettagli circa le istanze di monitor.

Si noti che se il monitor non fosse stato istanziato, le due costanti sarebbero state denotate con LION.C1 e LION.C2.

20. TIPI

Il linguaggio PASCAL+ esteso dispone di alcuni tipi addizionali.

In questo capitolo sono trattati separatamente i tipi predefiniti, che sono tipi semplici, ed i tipi definiti dall'utente, che sono tipi strutturati definiti tramite librerie di risorse istanziabili.

TIPI PREDEFINITI

LONGINTEGER

Il tipo LONGINTEGER rappresenta l'insieme dei numeri interi.
Come per il tipo INTEGER, il suo intervallo dipende dall'implementazione ma e' certamente piu' ampio dell'intervallo del tipo INTEGER. Le costanti predefinite MAXLONGINT e MINLONGINT rappresentano il massimo ed il minimo numero LONGINTEGER memorizzabile.

I valori LONGINTEGER, come i valori INTEGER, sono indicati da "numeri con segno".

Le operazioni permesse sui valori del tipo LONGINTEGER sono le seguenti:

+	addizione
-	sottrazione
*	moltiplicazione
DIV	divisione con troncamento
MOD	modulo
+ monadico	identita'
- monadico	inversione di segno
=, <>, <, >, <=, >=	operatori relazionali

Compatibilita'

Due tipi, T1 e T2 si dicono "tipi compatibili" se e' rispettata almeno una delle seguenti regole:

a) T1 e T2 hanno la stessa struttura di tipo;
b) T1 e' un subrange di T2;
c) T2 e' un subrange di T1;

d) T1 e T2 sono subrange delle stesso tipo ospite;
e) T1 e T2 sono tipi set di tipi base compatibili e sono entrambi PACKED o entrambi non-PACKED;
f) T1 e T2 sono tipi stringa aventi lo stesso numero di componenti.

Compatibilita' nell'Assegnazione

Un valore V appartenente ad un tipo T1 si dice compatibile nell'assegnazione con un tipo T2 se e' rispettata almeno una delle seguenti regole:

a) T1 e' di tipo LONGINTEGER e T2 e' di tipo REAL oppure T2 e' di tipo INTEGER;
b) T1 e T2 sono tipi compatibili ed il valore V appartiene a T2, a condizione che T1 e T2 non siano di tipo file ne' contengano (anche indirettamente) componenti di tipo file;
c) T1 e T2 sono di tipo stringa ed il numero di componenti di T2 e' minore o uguale al numero di componenti di T1.

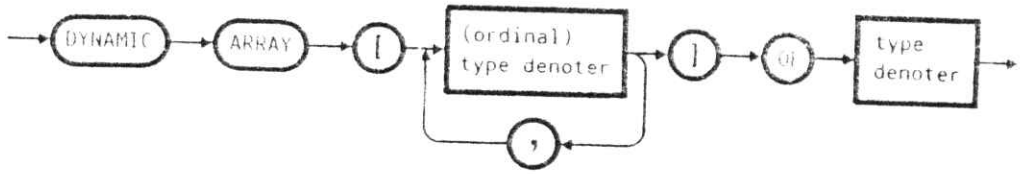
CONDITION

Il tipo CONDITION puo' essere usato solo nei monitor, mai all'interno di programmi o di moduli.
Nel capitolo 17 e' stata data una spiegazione completa sul tipo CONDITION e sul modo di usare le variabili di questo tipo.

DYNAMIC ARRAY

Il tipo DYNAMIC ARRAY consente di specificare, in run-time, i limiti ("bound") reali di un array.

DYNAMIC ARRAY SPECIFIER



(ORDINAL) TYPE DENOTER

Fig. 20-1 Diagramma Sintattico per il Tipo Dynamic Array

E' possibile dichiarare una variabile di tipo DYNAMIC ARRAY, pero' cosi' facendo tale variabile non viene allocata.
Tutti i dynamic array devono essere allocati esplicitamente chiamando la procedura standard "ALLOC" nella parte istruzioni di una routine. (Per ulteriori dettagli sulla procedura ALLOC si veda il Capitolo 22).

Quando l'array e' allocato allora puo' essere considerato come un array ordinario. Finche' la procedura ALLOC non e' chiamata, l'array non e' memorizzato e non puo' avere riferimenti.

Esempio:

```
TYPE DYNAM = DYNAMIC ARRAY [N..M : INTEGER, S..T : INTEGER] OF REAL;
VAR ALFA : DYNAM;
ALLOC (ALFA, EXPR1..EXPR2, EXPR3..EXPR4);
```

dove il tipo delle espressioni EXPR1-EXPR4 e' INTEGER.
Quando le espressioni sono state valutate, la precedente dichiarazione e' la stessa della dichiarazione "statica":

```
VAR ALFA : ARRAY [LB1..LB2, LB3..LB4] OF REAL;
```

dove i valori di LB1-LB4 sono uguali ai valori di EXPR1-EXPR4.

Dopo l'allocazione, si puo' far riferimento alle variabili dei limiti dei dynamic array come per le costanti in run-time.

La notazione sara':
nome dell'array "." nome dei limiti;
nell'esempio precedente i limiti sono chiamati:
ALFA.N,...ALFA.T

E' possibile deallocare un dynamic array chiamando la proceduta standard "DEALLOC". (Per ulteriori dettagli sulla procedura DEALLOC si veda il Capitolo 22).
Dopo essere stata deallocata, una variabile di tipo dynamic array viene considerata come se non fosse stata mai allocata; per cui una successiva chiamata della procedura ALLOC puo' essere effettuata sulla stessa variabile di tipo array.
Inoltre dopo una deallocazione, e' privo di significato far riferimento alla variabile dei limiti cosi' come alle costanti poiche' diventano non definite.

Infine un array non puo' essere allocato di nuovo, se esso non e' prima deallocato; cosi' pure un array non puo' essere deallocato due volte se l'ultima deallocazione non e' preceduta da una chiamata di allocazione.

Per esempio:
```
ALLOC (BETA, E1..E2, E3..E4);
DEALLOC (BETA);
ALLOC (BETA, EXP1..EXP2,EXP3..EXP4);
DEALLOC (BETA);
```
e' questo il giusto modo per procedere.
Inoltre si noti che i valori costanti in run-time delle variabili dei limiti, si riferiscono sempre all'ultima chiamata di allocazione, e che non viene effettuato alcun controllo run-time sui dynamic-array.

Compatibilita'

Due variabili di tipo dynamic array sono compatibili nel tipo se e solo se sono identici i "dynamic array specifier" (identici nel senso di compatibilita' di tipo del Pascal standard) e sono uguali i loro limiti ("bound").

Inoltre una variabile di tipo dynamic array non e' mai compatibile con un array statico, neppure se i loro limiti sono compatibili nel tipo ed hanno lo stesso valore in run-time.
Un dynamic array non puo' contenere altri dynamic array, a qualsiasi livello di nidificazione.
Un file non puo' avere come suoi elementi un dynamic array o un tipo contenente un dynamic array.

TIPI DEFINITI DALL'UTENTE

MODULE

Il tipo MODULE e' il mezzo con cui viene istanziato un modulo.

Un modulo istanziabile puo' essere pensato come un costruttore definito dall'utente, percio' esso puo' essere utilizzato per la definizione di tipi strutturati.

Il tipo modulo e' denotato dalla parola chiave MODULE seguito dall'identificatore del modulo istanziabile, come specificato nella Fig. 18.6.

Il tipo modulo, come ogni altro tipo, puo' essere usato per la costruzione di tipi piu' complessi.

Ad esempio, considerando il modulo "albero binario" (visto nel Capitolo 16), si puo' sviluppare il seguente programma:

```
PROGRAM COMMERCIAL_OFFICE;
  .
  .
IMPORT INSERT,...FROM B_TREE;
TYPE ARCHIVES = RECORD
                  YEAR:1900..1999;
                  INFORM : MODULE B_TREE
                END;
VAR  CARD_INDEX : ARRAY ['A'..'Z'] OF ^ ARCHIVES
.
.
CARD_INDEX ['S'] ^.YEAR := 1982;
CARD_INDEX ['S'] ^. INFORM.INSERT ('SMITH JOHN');
.
.
```

Per una completa spiegazione dell'uso delle variabili di tipo Module, si puo' consultare "Ridefinizione ed Uso delle Risorse Istanziate" nel Capitolo 18.

MONITOR

Il tipo MONITOR e' il mezzo con cui e' istanziato un monitor.

Come per i moduli, un monitor istanziabile puo' essere pensato come un costruttore definito dall'utente, percio' puo' essere usato per la definizione di tipi strutturati.
Il diagramma sintattico relativo al tipo monitor e' mostrato in Fig. 18.6.

Il tipo monitor differisce dal tipo modulo in quanto gli accessi contemporanei alle routine del monitor sono poste in sequenza assicurando la mutua esclusione tra processi concorrenti.
Al contrario accessi concorrenti a routine che appartengono ad istanze

differenti non sono serializzate: ad istanze differenti si puo' accedere in parallelo.

Maggiori dettagli circa la programmazione concorrente e le istanze di monitor sono date rispettivamente nei Capitoli 17 e 18.

21. VARIABILI

Il linguaggio esteso PASCAL+ fornisce alcune possibilita' addizionali nelle variabili.

Viene considerata la direttiva AT, che permette all'utente di definire l'allocazione di memoria delle variabili, e le variabili importate da una libreria di risorse istanziate.

DIRETTIVA AT

Il PASCAL+ permette l'uso di variabili che non sono ne' allocate nello Stack Run Time ne' nello Heap.

Nella descrizione di una variabile si puo' specificare l'indirizzo della sua allocazione attraverso la direttiva "AT".

Il diagramma sintattico per la descrizione di variabile e' il seguente:

VARIABLE DESCRIPTION

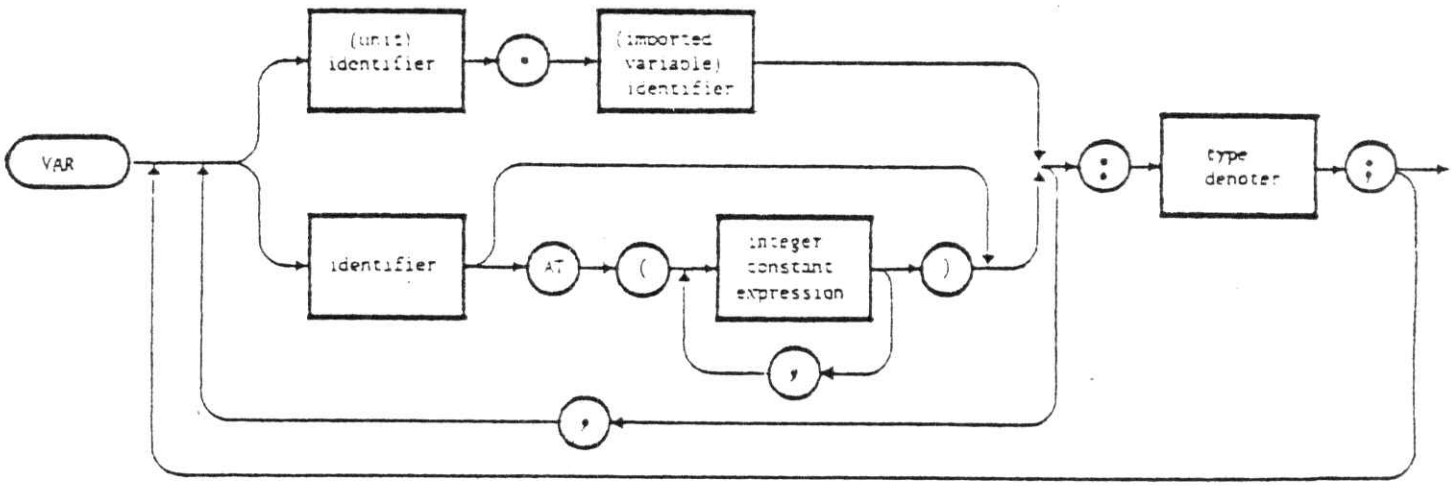


Fig. 21-1 Diagramma Sintattico per Descrizione di Variabile con Direttiva AT

L'indirizzo, specificato dalla lista di espressioni costanti intere nella direttiva AT, e' relativo alla macchina locale ed e' interpretato in modo dipendente dall'implementazione.

Questa direttiva e' applicabile solo alle variabili globali di un modulo, monitor, o di una unita' principale, e non per variabili locali ad una routine.

E' compito del programmatore assicurare che l'indirizzo specificato non si sovrapponga a porzioni di memoria gia' occupate.

Esempio:

```
MODULE ICE_CREAM;
  .
  .
  .
VAR A AT (1200) : INTEGER;
CONST BUFFER_ADDR = 100;
VAR BUF AT (BUFFER_ADDR + 300) : ARRAY [1..100] OF CHAR;
  CHANNEL_1 AT (0), CHANNEL_2 AT (5), CHANNEL_3 AT (10): LONGINTEGER;
```

VARIABILI ISTANZIATE

Se una unita' istanzia una libreria di risorse ed importa alcune variabili da essa, queste ultime devono essere ridefinite una sola volta, facendo uso dell'identificatore dell'unita' istanziata e sono indicate tramite il denotatore di istanza seguito dal simbolo "." e dall'identificatore di variabile.

Esempio:

```
MODULE CAT TYPE;
VAR A,B : INTEGER;
      C : BOOLEAN;
 .
 .
EXPORT A,B,C;


MONITOR PIG;
IMPORT A,B,C FROM CAT;
VAR X,Y,Z : MODULE CAT;
    CAT.A, CAT.B : INTEGER;
    CAT.C : BOOLEAN;
      I : INTEGER;
 .
 .
X.A := 50;
IF Z.C THEN...
I := (Y.B*18) DIV Z.A
```

Si veda il capitolo 18 per maggiori dettagli sulle istanze di modulo.

22. PROCEDURE E FUNZIONI

Il linguaggio PASCAL+ esteso dispone di molte procedure e funzioni predefinite addizionali, particolarmente adatte per la gestione della memoria ausiliaria e per il trattamento diretto della memoria centrale.

Nel presente capitolo vengono prima trattate le routine importate da una libreria di risorse istanziabili ed in seguito le routine predefinite.

Inoltre vengono dati alcuni consigli sul modo migliore di utilizzare la memoria ausiliaria.

ROUTINE ISTANZIATE

Se un'unita' istanzia una libreria di risorse ed importa alcune routine da essa, queste routine devono essere ridefinite con l'uso dell'identificatore della libreria di risorse e vengono chiamate indicandole con il denotatore di istanza seguito dal simbolo "." e dall'identificatore di routine.

Esempio:

```
MODULE ROSE TYPE;
PROCEDURE P (VAR V : COLOUR);
 .
 .
FUNCTION F (CONST C : REAL) : BOOLEAN;
 .
 .
EXPORT P,F;.


MODULE LAUREL;
IMPORT P,F FROM ROSE;
VAR X,Y,Z : MODULE ROSE;
PROCEDURE ROSE.P (VAR V : COLOUR); DEFINITION;
FUNCTION ROSE.F (CONST C : REAL): BOOLEAN; DEFINITION;
 .
 .
X.P (RED);
WHILE NOT Z F (3.8) DO ...
 .
 .
Y.P (YELLOW);
 .
 .
```

Per maggiori dettagli sulle istanze di modulo si veda il Capitolo 18.

ROUTINE PREDEFINITE PER LA GESTIONE DEI FILE

Il concetto di file nel PASCAL+ esteso e' piuttosto differente da quello precedentemente descritto.
Il file e' visto, in questo caso, come un insieme ordinato di elementi appartenenti allo stesso tipo, ed associati all'insieme dei numeri naturali.
Non c'e' limite al numero di elementi.

Ogni elemento del file puo' essere reperito sia direttamente, cioe' specificando il numero intero associato, sia in modo sequenziale accedendo a tutti gli elementi che lo precedono, ovvero agli elementi associati ai numeri naturali precedenti.

Inoltre ogni elemento del file puo' essere letto o aggiornato e si possono aggiungere nuovi elementi, ossia elementi associati ai successivi numeri naturali (cioe' i numeri naturali piu' bassi non gia' associati ad altri elementi del file).
Non e' possibile cancellare un elemento del file, e non e' possibile accedervi specificando una porzione del suo contenuto ovvero un file PASCAL+ non puo' avere chiave.

Il trattamento dei file e' effettuato tramite un insieme di routine predefinite: alcune di esse sono identiche a quelle del linguaggio base ed altre differiscono. Inoltre sono aggiunte le seguenti routine: UPDATE, SEEK, SEEKEOF, SENSE.

UPDATE (F)

La procedura UPDATE pone il file in stato di "aggiornamento" senza alterarne il contenuto.

Se il file e' vuoto, il valore di EOF(F) e' posto a TRUE ed F^ e' indefinito; altrimenti EOF(F) e' posto a FALSE ed F^contiene il valore del primo elemento.
Un file aperto con la procedura UPDATE invece che con la RESET o REWRITE, puo' essere manipolato sia con la procedura GET e READ che con la procedura PUT, WRITE: puo' cioe' essere considerato sia in stato di "ispezione" che in stato di "generazione".

Se il file e' di tipo TEXT la procedura UPDATE aggiunge automaticamente un fine linea come ultimo elemento, se non e' gia' presente.

UPDATE (F,s)

Questa procedura e' molto simile alla precedente UPDATE (F). Il parametro s e' un'espressione di tipo stringa che specifica il nome esterno del file sul quale deve essere effettuata l'operazione.

SEEK (F,n)

La procedura SEEK e' usata per l'accesso diretto ad un elemento del file. Questa procedura rende disponibile l'elemento associato al numero naturale n che e' specificato come parametro.

Il parametro di tipo file F non puo' essere indefinito, ma puo' essere sia in stato di ispezione, che di generazione che di aggiornamento.
Il parametro n e' una espressione di tipo integer oppure longinteger; inoltre deve essere maggiore o uguale a 1 e non maggiore del numero di elementi del file.
Se il file F e' in stato di ispezione o aggiornamento, la variabile buffer F^ conterra' il valore del n-esimo elemento del file. Si ricordi che il primo elemento del file e' denotato con il numero 1.
Se il file F e' in stato di generazione, la variabile buffer F^ e' completamente indefinita.
Il contenuto di EOF(F) e' in ogni caso posto al valore FALSE.

Esempio:

```
VAR NUMBER : FILE OF INTEGER;
(* si supponga che il file NUMBER contenga i valori: 3,8,5,2,15 *)
VAR A,B : INTEGER;
    C : LONGINTEGER;
 .
 .
 .
UPDATE ( NUMBER, 'NUM1');
SEEK (NUMBER, 4);
A := NUMBER^; (* A contiene 2 *)
C := 3;
SEEK (NUMBER, C-2);
B := NUMBER^; (* B contiene 3 *)
```

SEEKEOF (F)

La procedura SEEKEOF (F) rende disponibile la posizione successiva all'ultimo elemento del file F.
Questa procedura viene utilizzata quando si vogliono aggiungere elementi al file F (eseguendo la procedura PUT) senza riscrivere gli elementi esistenti.

IL parametro F non deve essere indefinito, ossia deve essere stata eseguita una delle seguenti procedure: RESET, REWRITE o UPDATE.

In ogni caso il valore di EOF(F) e' posto a TRUE.

SENSE (F)

La funzione SENSE da' come valore un longinteger, che corrisponde alla posizione corrente della variabile buffer F^, cioe' essa fornisce il numero naturale associato all'elemento attualmente disponibile.

Il parametro file F non deve essere indefinito.

Esempio:

```
VAR NUMBER : FILE OF INTEGER;
    A : LONGINTEGER;
 .
 .
 .
RESET (NUMBER, 'NUM');
SEEK (NUMBER, 13);
 .
 .
 .
GET (NUMBER);
 .
 .
 .
GET (NUMBER);
A := SENSE (NUMBER); (* A contiene 15 *)
```

La funzione SENSE puo' essere usata per posizionamenti relativi nel file: la forma

```
SEEK(F,SENSE(F)+n)
```

e

```
SEEK(F,SENSE(F)-n)
```

rendono disponibile l'elemento che segue o precede di n posizioni l'elemento attuale.

GET (F)

La procedura GET e' essenzialmente identica a quella del linguaggio PASCAL+ base (vedere la relativa documentazione nel presente manuale). L'unica differenza e' che il parametro file F puo' essere sia in stato di ispezione che di aggiornamento.

PUT (F)

La procedura PUT e' sostanzialmente uguale a quella del linguaggio base (vedere la documentazione relativa). L'unica differenza e' che F puo' essere sia in stato di generazione che di aggiornamento.

Se esiste l'elemento attuale, cioe' se EOF(F) e' FALSE, allora la procedura PUT inserisce il contenuto di F^ nell'elemento attuale sovrapponendo il contenuto precedente.
Dopo l'operazione, se il file F e' in stato di aggiornamento, la variabile buffer F^ contiene il valore dell'elemento successivo a quello attuale (se c'e'), altrimenti F^ e' indefinito.

Se l'elemento attuale non esiste, cioe' se il valore di EOF(F) e' TRUE (vederei procedura SEEKEOF), allora la procedura PUT aggiunge un nuovo elemento al file che contiene il valore di F^.
Dopo l'operazione la variabile F^ risulta indefinita.

Si consideri ora un esempio di scansione di un file con aggiornamento di un elemento.

```
TYPE STRINGS = PACKED ARRAY [1..10] OF CHAR;
VAR CHILDREN : FILE OF STRINGS;
.
.
UPDATE (CHILDREN, 'CHILDR');
WHILE (CHILDREN ^ <> 'TOM') OR EOF (CHILDREN) DO GET (CHILDREN);

CHILDREN^ := 'JAMES';
PUT (CHILDREN);
 .
 .
 .
```

Nell'esempio precedente, quando si incontra l'elemento che contiene 'TOM' il suo valore viene sostituito con 'JAMES', se tale elemento non viene trovato viene aggiunto un nuovo elemento che contiene 'JAMES'.

READ (F,V)

La procedura READ e' essenzialmente identica a quella descritta precedentemente nel linguaggio base. La differenza e' simile a quella vista per la procedura GET, poiche' READ(F,V) e' equivalente alla forma:

```
BEGIN
V := F^;
  GET (F)
END
```

WRITE (F,E)

La procedura WRITE e' sostanzialmente identica a quella del linguaggio base. La differenza e' simile a quella vista per la procedura PUT, in quanto WRITE(F,E) equivale alla forma:

```
BEGIN
 F^ := E;
 PUT (F)
END
```

SFRUTTAMENTO OTTIMALE DELLA MEMORIA AUSILIARIA

In questo paragrafo vengono dati alcuni consigli su come dotare una variabile file di una "chiave di ricerca", cioe' come sviluppare una struttura di dati in memoria ausiliaria dove gli elementi possono essere reperiti specificando una porzione del proprio contenuto.

La forma generale di un file con chiave e' la seguente:

```
TYPE KEYED_ELEM = RECORD
                    KEY : KEY_TYPE;
                    INF : INF_TYPE
                  END;
VAR F : FILE OF KEYED_ELEM;
```

Tale file puo' essere elaborato sia copiando tutto il suo contenuto nella memoria principale dell'elaboratore, sia accedendo direttamente agli elementi nella memoria ausiliaria.

Nel secondo caso la soluzione piu' efficiente e' quella di costruire una struttura ad albero usando come puntatori le posizioni relative degli elementi.

```
TYPE KEYED_ELEM=RECORD
                  KEY : KEY_TYPE;
                  INF : INF_TYPE;
                  LEFT, RIGHT : LONGINTEGER
                END;
VAR F : FILE OF KEYED_ELEM;
```

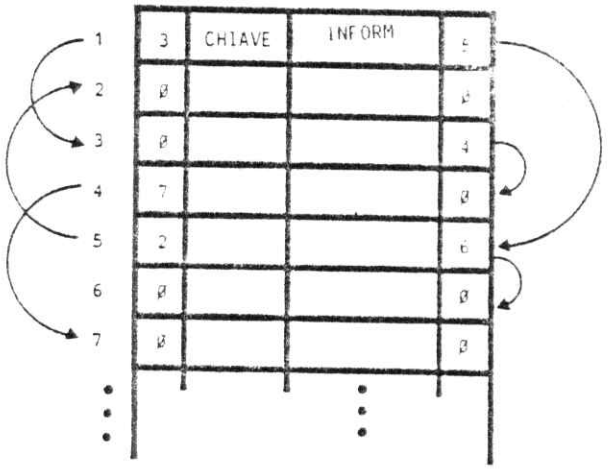


Fig. 22-1 Un File con Chiave

Una routine di ricerca per tale file potrebbe essere:

```
VAR FOUND : BOOLEAN;
.
.
PROCEDURE SEARCH (K:KEY_TYPE; SUB_T:LONGINTEGER);
 (*SUB_T e' un "sub-tree", la prima volta e' l'intero albero*)
 BEGIN
   IF SUB_T <> 0 THEN
      BEGIN
        SEEK (F, SUB_T);
        IF F^.KEY=K THEN
           FOUND:=TRUE
        ELSE
          IF F^.KEY > K THEN
             SEARCH (K,F^.LEFT)
          ELSE
             SEARCH (K,F^.RIGHT)
      END
    ELSE
      FOUND:=FALSE
END;
```

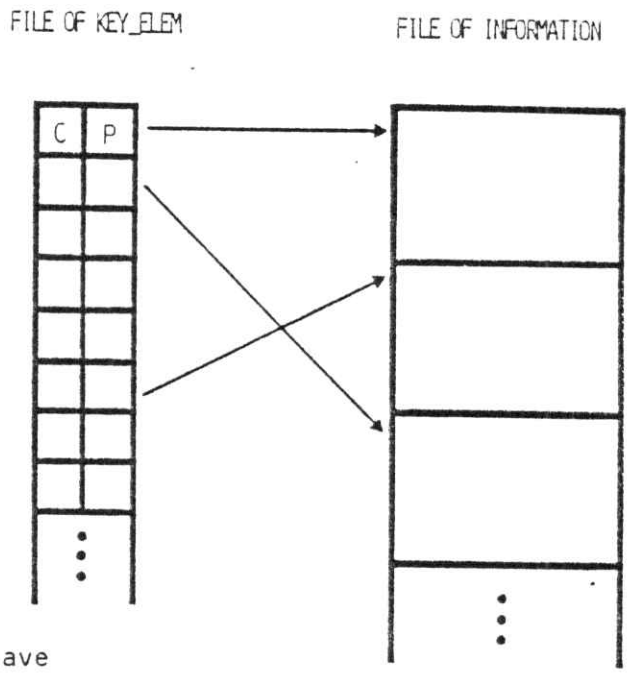
La precedente procedura puo' essere usata in questo modo (supponendo KEY_TYPE=STRING):

```
   .
   .
UPDATE (F, 'nome-esterno');
SEARCH ('JAMES', 1);
IF FOUND THEN
   (* elaborazione di F^.INF *)
.
.
```

Notare che, essendo la routine SEARCH ricorsiva, la prima chiamata ad

essa richiede il numero 1 come parametro.
L'elaborazione dei file con chiave puo' essere piu' efficiente se le chiavi vengono memorizzate separatamente dalle informazioni.

```
TYPE KEY_ELEM = RECORD
                KEY : KEY_TYPE;
                POINTER_TO_INF : LONGINTEGER
                END;
     INFORMATION =...
VAR K_F : FILE OF KEY_ELEM;
    I_F : FILE OF INFORMATION;
```



C) chiave
P) puntatore all'informazione

Fig. 22-2 Struttura di Dati con Chiave

Prima dell'elaborazione di tale file, le chiavi devono essere copiate in un albero binario (allocato in memoria principale) come quello descritto qui di seguito:

```
TYPE KEY_TREE =  ^NODE;
     NODE = RECORD
              KEY_AND_POINTER : KEY_ELEM;
              LEFT, RIGT : KEY_TREE
            END;
VAR T : KEY_TREE;
  .
  .
  .
PROCEDURE COPY;
     (* copia di K_F in T *)
          .
          .
          .
```

Si consideri ora come poter elaborare un elemento di questa struttura di dati (supponendo KEY_TYPE = STRING):

```
VAR P : KEY_TREE;
  .
  .
  .
RESET (K_F, 'nome-esterno');
COPY; (* copia K_F in T *)
UPDATE (I_F; 'nome-esterno');
SEARCH ('JAMES') (* cerca in T il nodo con chiave 'JAMES';
                    il nodo trovato e' puntato da P *)
SEEK (I_F, P^.KEY_AND_POINTER.POINTER_TO_INF);
  (* elaborazione di I_F^ *)
   .
   .
   .
```

Puo' essere vantaggioso racchiudere tutte queste operazioni in un modulo, ottenendo cosi' il tipo di dati "file con chiave".

FUNZIONI PREDEFINITE DI CONVERSIONE

RECAST (X, identificatore-di-tipo)

L'espressione X e' di qualsiasi tipo. La funzione restituisce il valore di X come se il suo tipo fosse quello specificato dall'"identificatore-di-tipo".
La dimensione dell'identificatore di tipo deve essere uguale alla dimensione di X (vedere la procedura SIZE). Non e' effettuata alcuna alterazione del contenuto di X e neppure alcuna conversione logica o matematica.

Esempio:

```
TYPE GIORNI = (DOM,LUM,MAR,MER,GIO,VEN,SAB);
     T1 = PACKED ARRAY [-3..0] OF ^GIORNI;
     T2 = PACKED RECORD C1,C2,C3,C4: CHAR END;
VAR A : T1; B : T2;
RECAST (TRUE,GIORNI) vale LUN
RECAST ('A',INTEGER) vale 65, assumendo la codifica ASCII
A := RECAST (B,T1);
```

LONGINT (X)

Il parametro X deve essere una espressione di tipo intero; la funzione converte il valore di tale espressione in tipo longinter.

Esempio:

```
VAR I : INTEGER;
    L : LONGINTEGER;
    F : FILE OF...;
L := LONGINT (I+8);
SEEK (F,LONGINT (I-14).
```

INT (X)

Il parametro X deve essere una espressione di tipo longinteger. La funzione converte il valore di tale espressione in tipo intero.

Esempio:

```
VAR I : INTEGER;
    L : LONGINTEGER;
    F : FILE OF...;
I := INT (L+33);
I := INT (SENSE(F))
```

LTRUNC (X)

LROUND (X)

Eseguono rispettivamente le stesse operazioni delle funzioni TRUNC(X) e ROUND(X) viste nel Capitolo 13, con la differenza che queste funzioni restituiscono un risultato di tipo longinter.

ROUTINE PREDEFINITE DI COMUNICAZIONE TRA PROCESSI

Le seguenti routine sono spiegate in dettaglio con esempi, nel Capitolo 17; qui vengono ripetute le principali caratteristiche di tali routine.

WAIT (X)

Il parametro X deve essere una variabile di tipo CONDITION. La procedura pone il processo in corso nella coda di condizione X. La procedura puo' essere richiamata solo all'interno dei monitor.

SIGNAL (X)

Il parametro X deve essere una variabile di tipo CONDITION. La procedura immette il processo in corso nella coda del monitor e risveglia il primo processo disponibile nella coda di condizione X.
Se la coda di condizione X e' vuota non e' effettuata alcuna operazione.
Questa procedura puo' essere richiamata solo all'interno dei monitor.

ALTRE ROUTINE PREDEFINITE

SIZE (identificatore-di-tipo)

Questa funzione ritorna il numero massimo di byte necessari per memorizzare una variabile del tipo specificato dall'"identificatore-di-tipo". La funzione SIZE puo' essere usata nella forma:

```
SIZE (identificatore-di-tipo, t1, ... , tn)
```

dove l'identificatore di tipo deve contenere un record con varianti e t1, ..., tn sono le costanti che identificano una configurazione del record. La funzione SIZE in questo caso restituisce il numero massimo di byte richiesti per memorizzare quella variante del record.

Esempio:

```
TYPE GIORNI = (DOM,LUN,MAR,MER,GIO,VEN,SAB);
     T1 = PACKED ARRAY [GIORNI] OF RECORD
                                   C1 : BOOLEAN;
                                   C2 : CHAR
                                 END;

     T2 = RECORD
          A : CHAR;
          CASE  BOOLEAN OF
          TRUE : (B : GIORNI);
          FALSE : (C : REAL)
          END;

SIZE (GIORNI)     vale 1
SIZE (INTEGER)    vale 2
SIZE (T1)         vale 14
SIZE (T2,FALSE)   vale 5
```

Questi risultati devono essere presi solo come esempi in quanto la dimensione di un tipo dipende dall'implementazione.

MOVEINTO (espressione-iniziale, variabile-array, espressione-indice)

Questa procedura sposta l'area di memoria occupata dalla espressione iniziale nell'area occupata dalla variabile array unidimensionale, a partire dalla posizione specificata tramite l'espressione indice.
Il tipo dell'espressione iniziale puo' essere diverso dal tipo degli elementi di array; tuttavia il tipo dell'espressione indice deve essere compatibile con il tipo indice dell'array.
Questa procedura puo' essere usata per particolari conversioni di tipo.

Esempio:

```
TYPE STRING = PACKED ARRAY [1..20] OF CHAR;
    PERSONA = RECORD
              NOME, COGNOME : STRING;
              ETA : INTEGER;
              SPOSATO : BOOLEAN
              END;
VAR GIOVANNI : PERSONA;
           A : ARRAY [1..100] OF REAL;
           I : INTEGER;
 .
 .
MOVEINTO (GIOVANNI,A,25);
 .
 .
MOVEINTO (I+3 DIV 18, A, I MOD 14);
```

MOVEFROM (var-destinazione, var-array, espressione-indice)

Questa procedura sposta l'area di memoria della variabile array unidimensionale, a partire dalla posizione specificata dall'espressione indice, nell'area di memoria occupata dalla variabile destinazione, finche' quest'area e' riempita.

Il tipo della variabile destinazione puo' essere differente da quelle degli elementi dell'array.
La procedura MOVEFROM puo' essere usata per particolari conversioni di tipo.

Il tipo dell'espressione indice deve essere compatibile con il tipo indice dell'array.

Esempio:

```
VAR A : ARRAY [1..50] OF BOOLEAN;
    B, C : ^INTEGER;
 .
 .
MOVEFROM (B,A, 3+C^)
```

MOVEARRAY (in -array, in-indice, dest-array, dest-indice, lungh)

Questa procedura sposta il contenuto di un array unidimensionale in un altro array unidimensionale. In-array e dest-array sono rispettivamente l'array di inizio e di destinazione, in-indice e dest-indice sono espressioni che indicano gli elementi di array da cui parte lo spostamento.

Lungh e' una espressione intera positiva che indica il numero di byte che deve essere spostato.

I tipo degli elementi di array possono essere diversi mentre i tipo degli indici debbono essere compatibili con indici degli array; non e' effettuato alcun controllo sull'intervallo degli indici.

Esempio:

```
VAR A : ARRAY [1..100] OF BOOLEAN;
    B : ARRAY [CHAR] OF REAL;
    I : INTEGER;
 .
 .
MOVEARRAY (A, I+18, B, '!', 33 DIV I);
 .
 .
```

MAKEPTR (X)

Questa funzione e' di tipo puntatore e restituisce un valore che punta alla variabile specificata come parametro X.
Il parametro X deve essere un denotatore di variabile di qualsiasi tipo ad eccezione di un conformant array.

Si noti che questa funzione permette ad un puntatore di puntare all'esterno degli heap.

Esempio:

```
VAR A,B : REAL;
    P : ^REAL;
 .
 .
 .
P := MAKE_PTR (A);
PΩ := 10.5 (* equivalente ad A := 10.5 *)
WRITE (RECAST (MAKEPTR(B), INTEGER));
          (* stampa l'indirizzo di B *)
```

ALLOC (variabile-array, limite1..limite2, limite3..limite4)

Questa procedura viene chiamata per l'allocazione in memoria dei dynamic array.
Ha i seguenti parametri:
- la variabile-array che deve essere allocata
- una lista di limiti reali da usarsi nell'allocazione dell'area.

I limiti reali sono espressioni che devono essere compatibili con i tipi dichiarati, e tutti i limiti superiori devono essere maggiori o uguali ai limiti inferiori.

Le espressioni sono associate con i limiti nell'ordine in cui appaiono.

Esempio:

```
TYPE DYNAM = DYNAMIC ARRAY [N..M:INTEGER, S..T:INTEGER] OF REAL;
VAR  ALFA : DYNAM;
 .
 .
ALLOC (ALFA, EXPR1 .. EXPR2, EXPR3 .. EXPR4);
 .
 .
```

L'array dinamico ALFA puo' essere allocato chiamando la procedura ALLOC, in cui il tipo della espressione EXPR1-EXPR4 e'intero.

DEALLOC (variabile-array)

Questa procedura e' chiamata per la deallocazione in memoria dei dynamic array. Possiede solo un parametro: la variabile-array da deallocare.

La procedura DEALLOC deve essere chiamata prima della procedura ALLOC, altrimenti non produce alcun effetto.

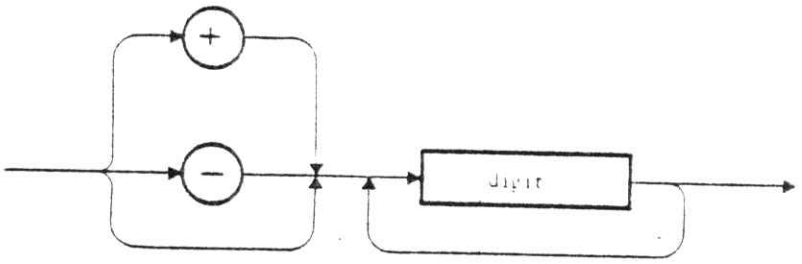
Esempio:

```
ALLOC (BETA, E1..E2,E3..E4);
DEALLOC (BETA);
```

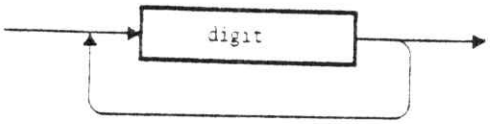
A. DIAGRAMMI SINTATTICI

LEXICON

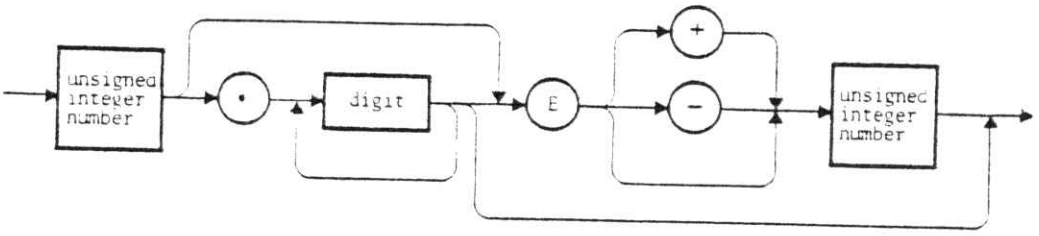
INTEGER NUMBER



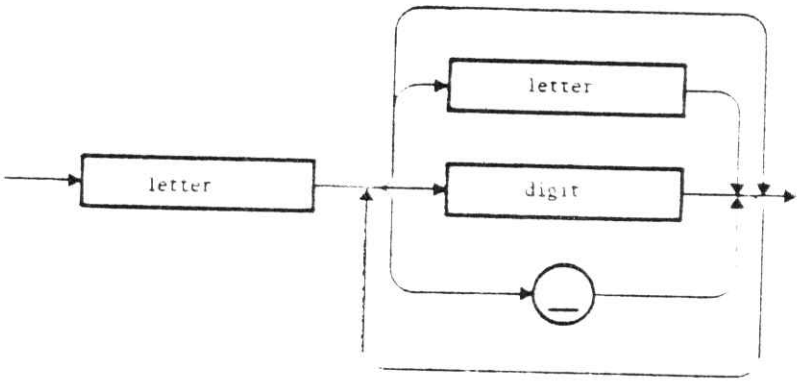
UNSIGNED INTEGER NUMBER



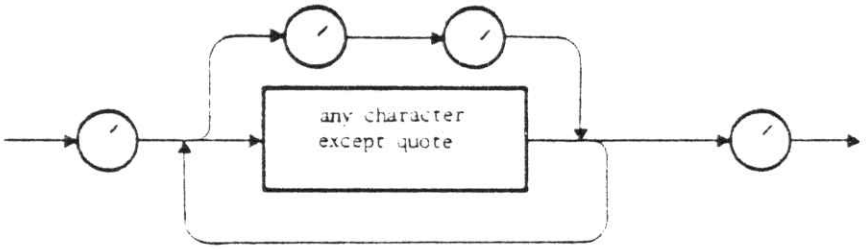
REAL NUMBER



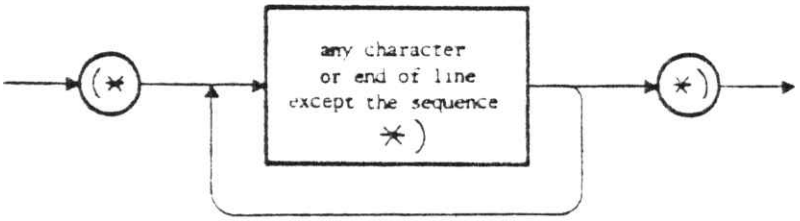
IDENTIFIER



STRING

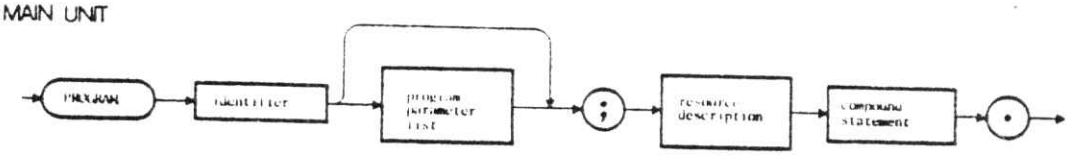


COMMENT

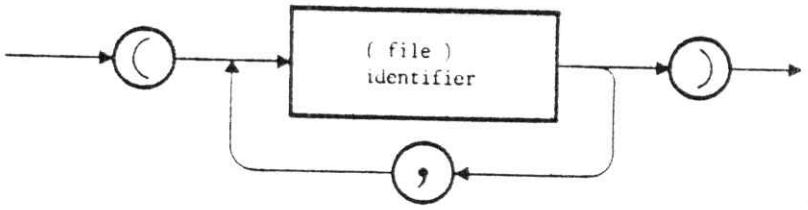


PROGRAM AND MODULE

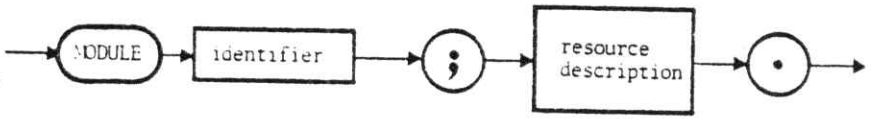
MAIN UNIT



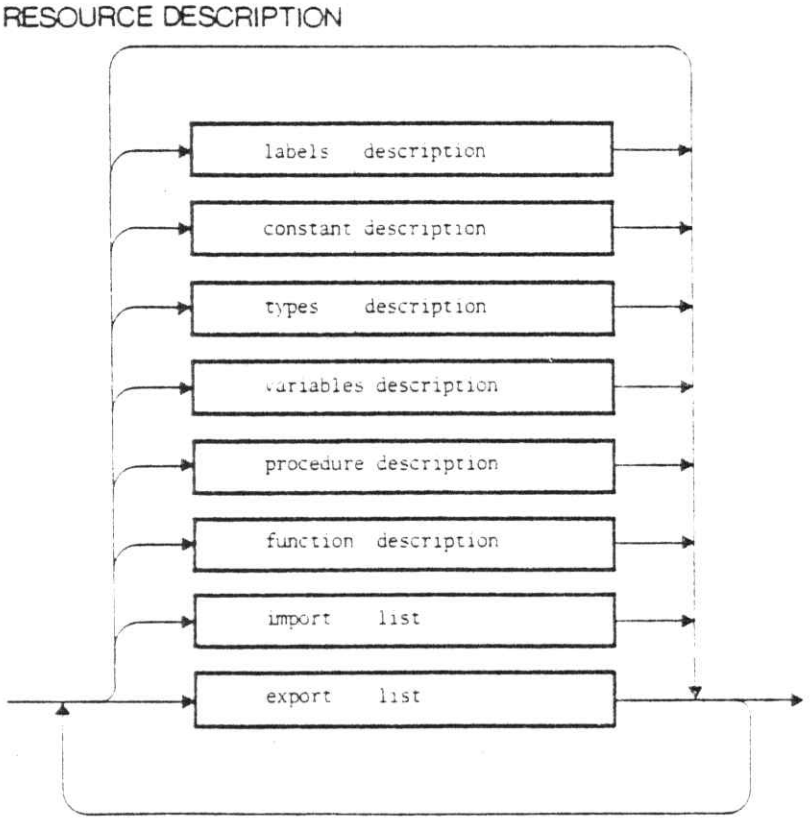
PROGRAM PARAMETER LIST



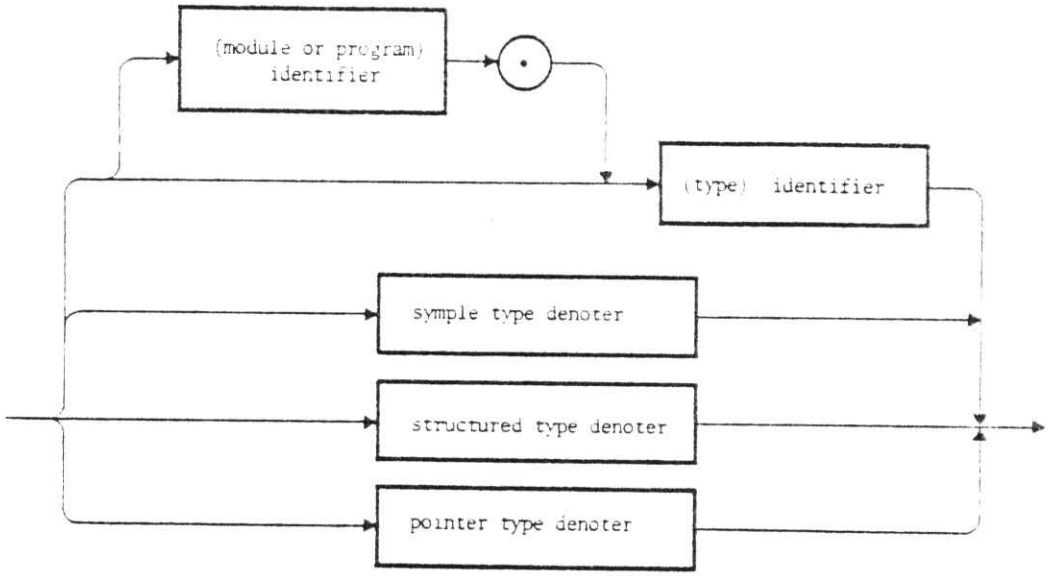
MODULE



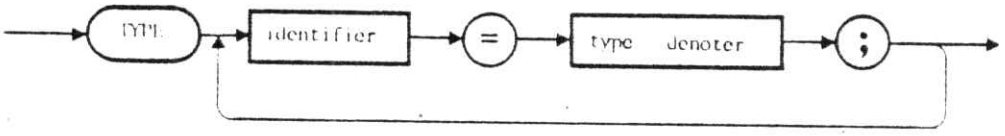
RESOURCE DESCRIPTION



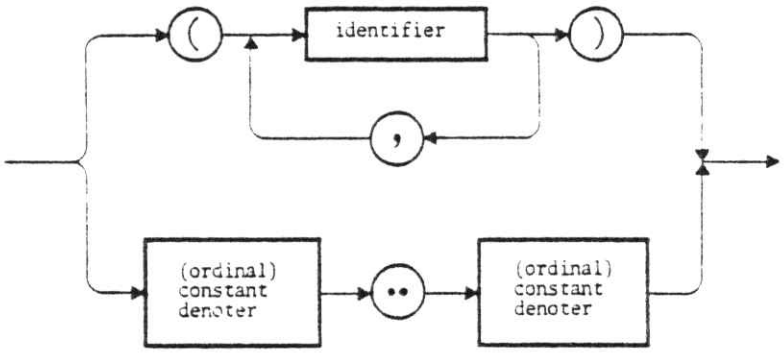
TYPE DENOTER



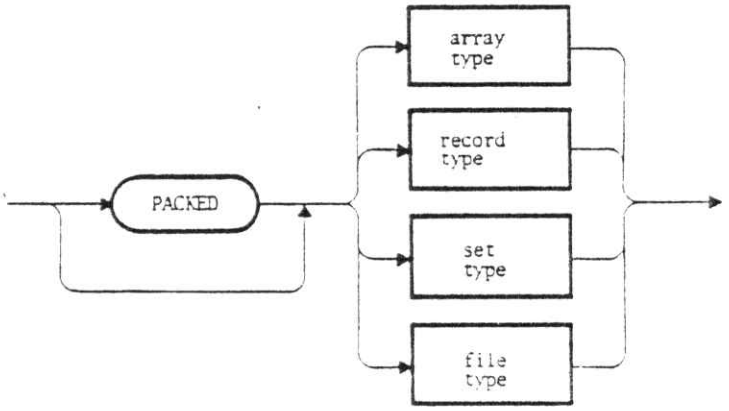
TYPES DESCRIPTION



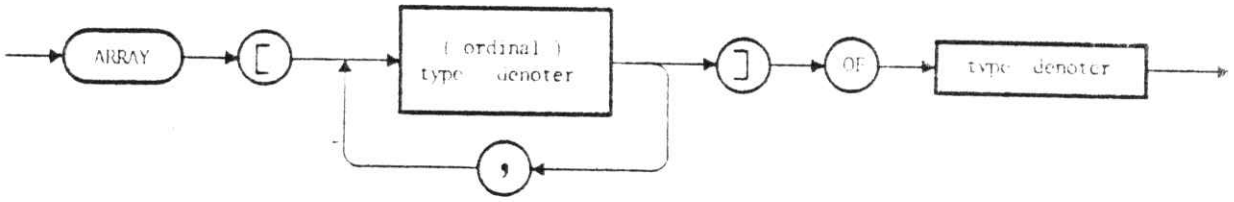
SIMPLE TYPE DENOTER



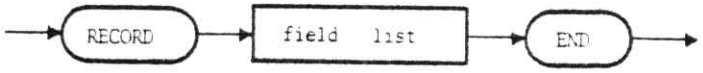
STRUCTURED TYPE DENOTER



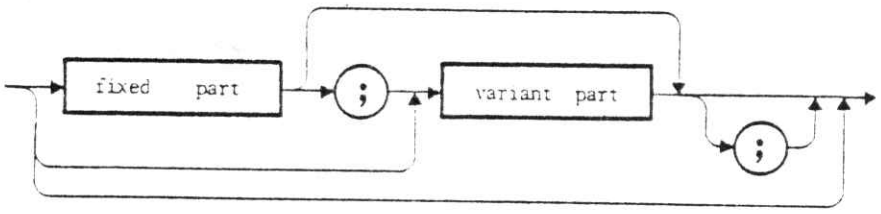
ARRAY TYPE



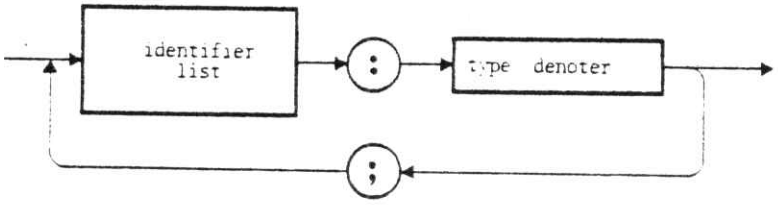
RECORD TYPE



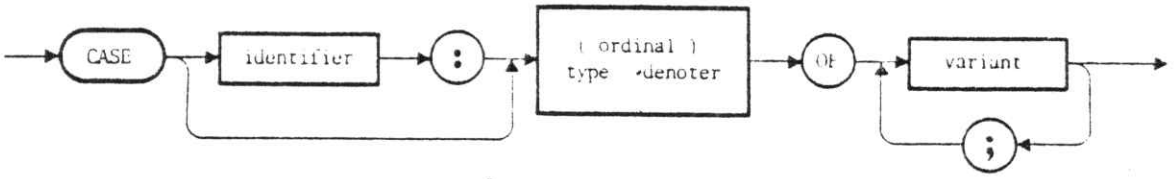
FIELD LIST



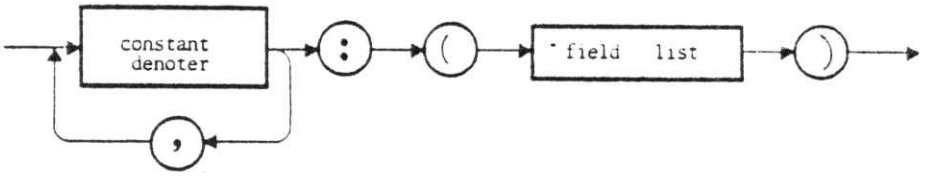
FIXED PART



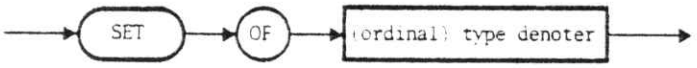
VARIANT PART



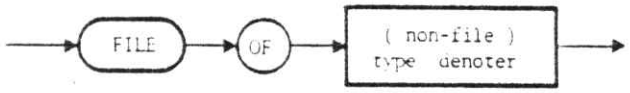
VARIANT



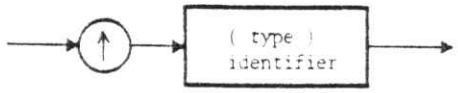
SET TYPE



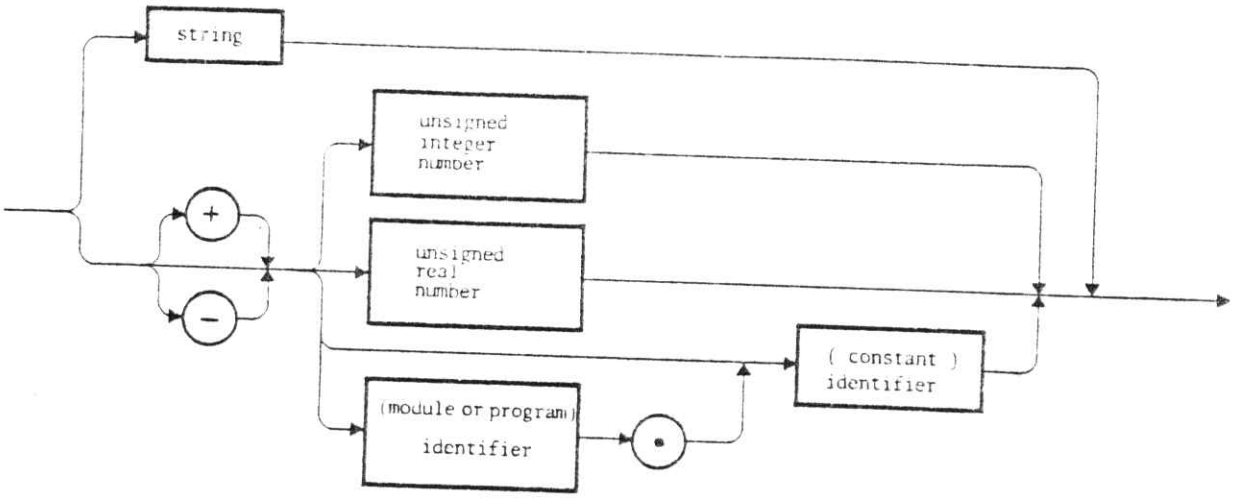
FILE TYPE



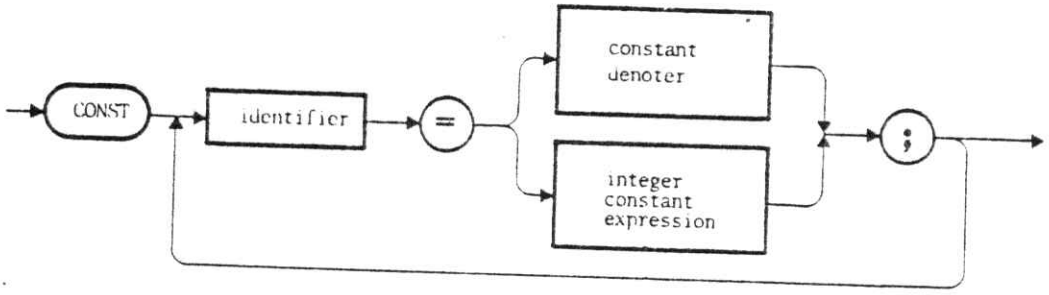
POINTER TYPE DENOTER



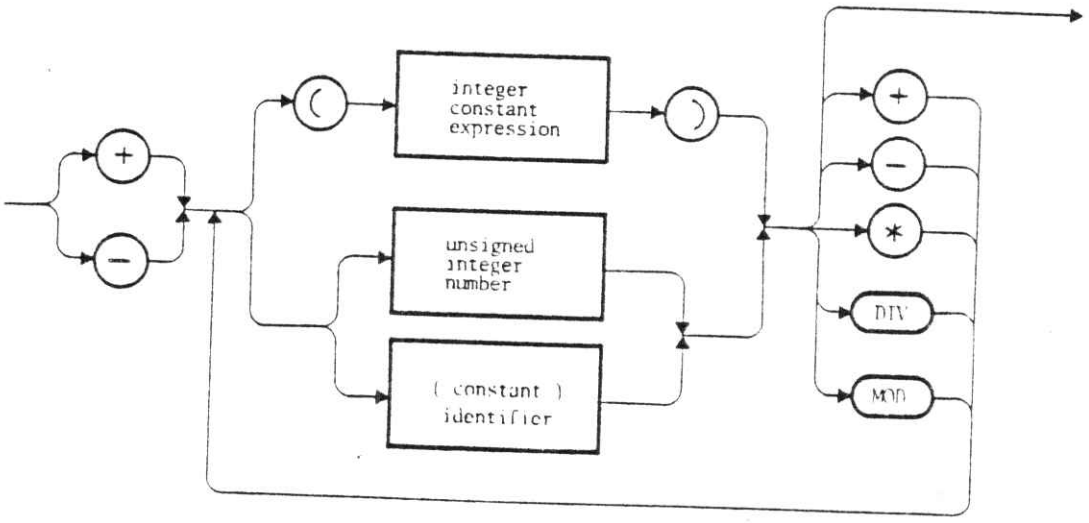
CONSTANT DENOTER



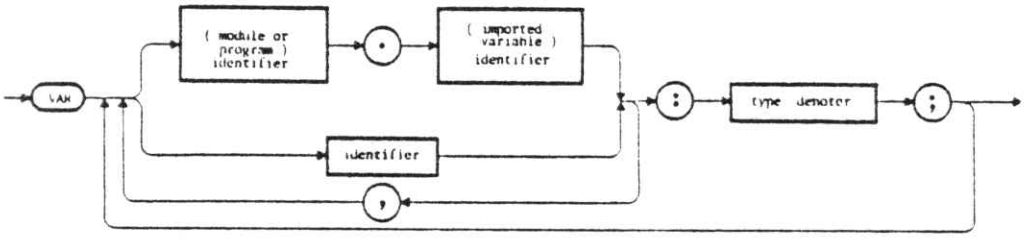
CONSTANT DESCRIPTION



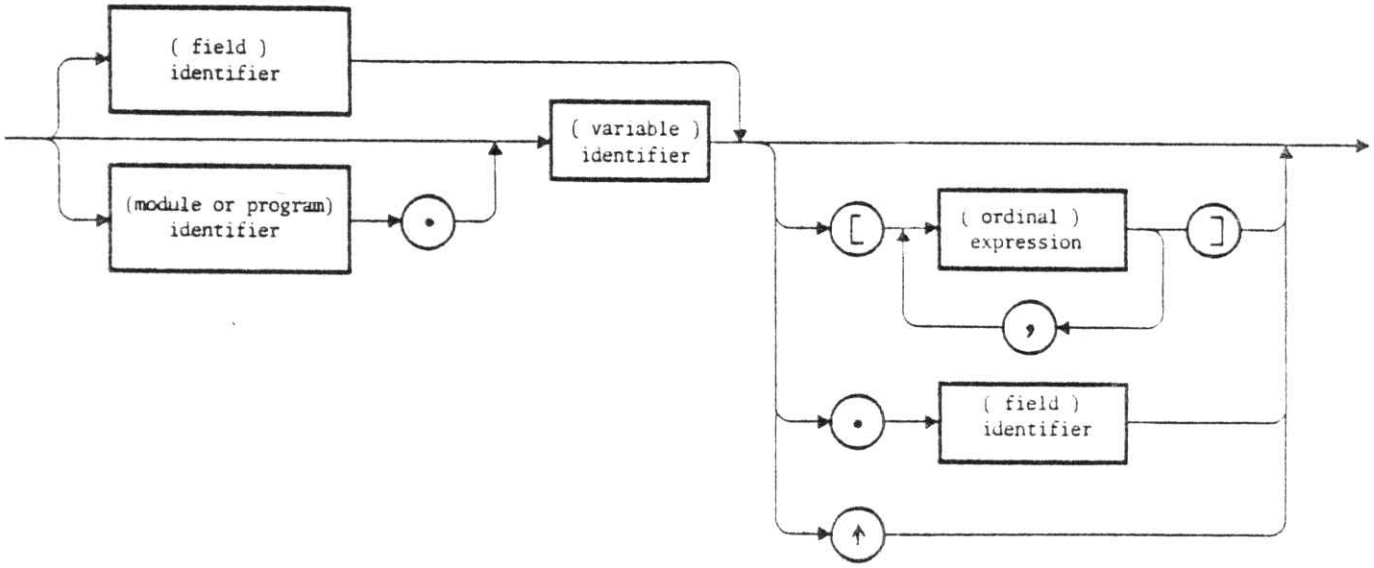
INTEGER CONSTANT EXPRESSION



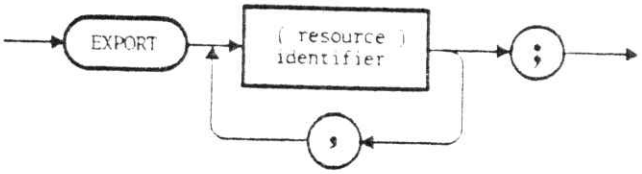
VARIABLE DESCRIPTION



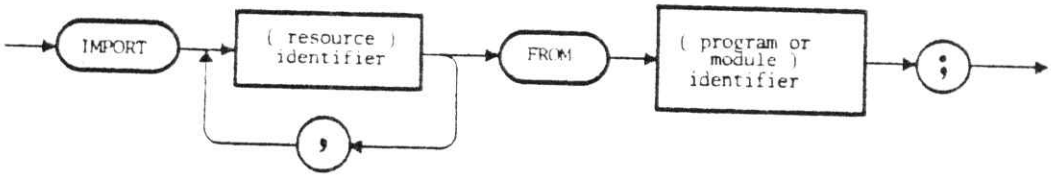
VARIABLE DENOTER



EXPORT LIST

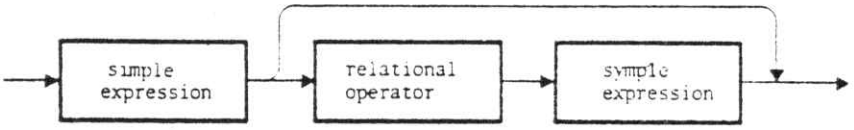


IMPORT LIST

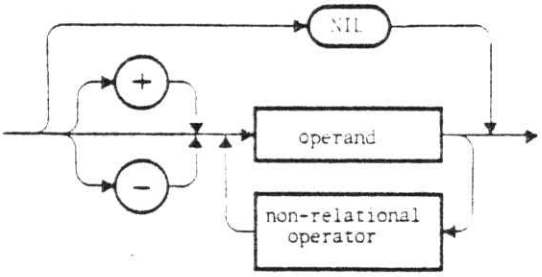


EXPRESSION

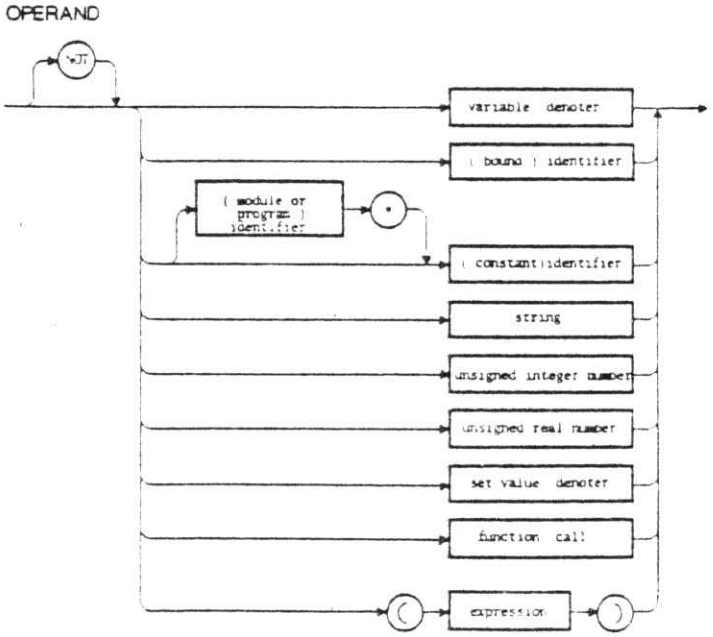
EXPRESSION



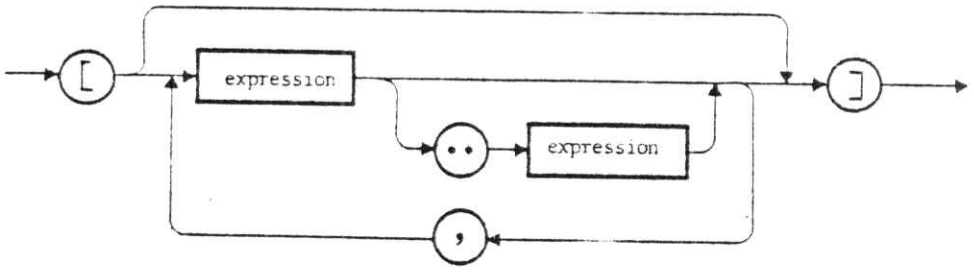
SIMPLE EXPRESSION



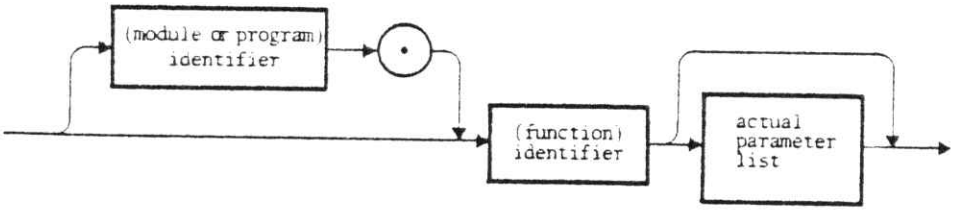
OPERAND



SET VALUE DENOTER

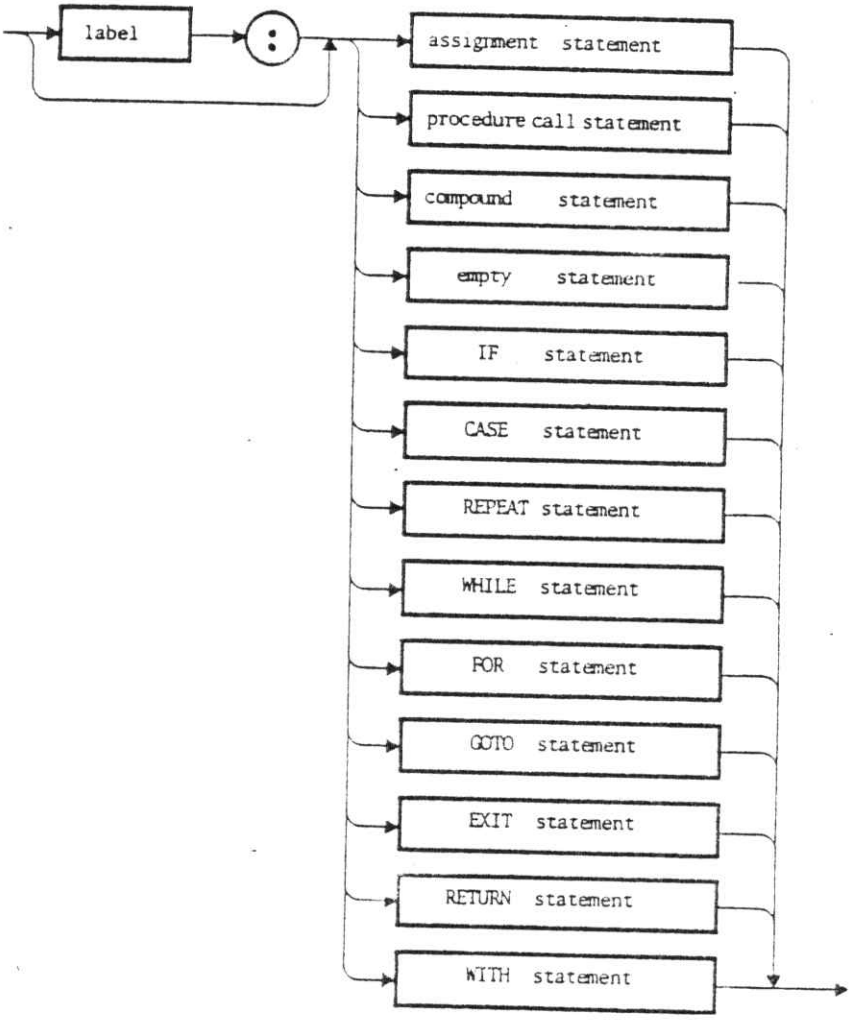


FUNCTION CALL

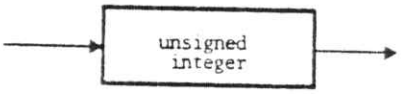


STATEMENT

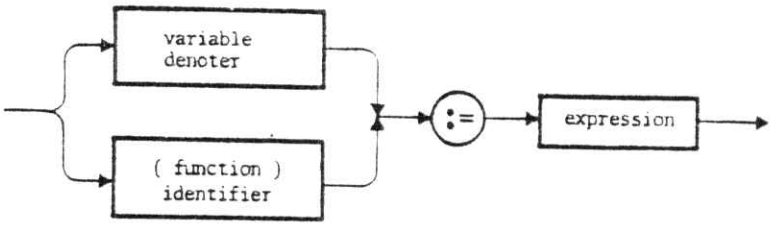
STATEMENT



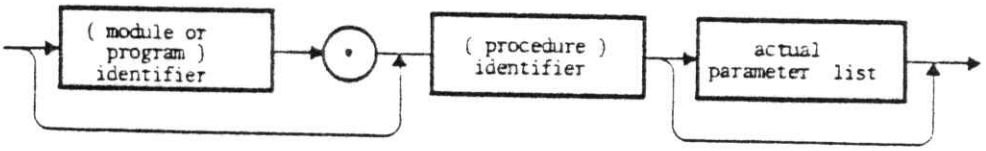
LABEL



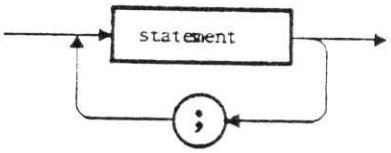
ASSIGNMENT STATEMENT



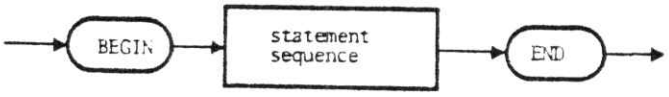
PROCEDURE CALL STATEMENT



STATEMENT SEQUENCE

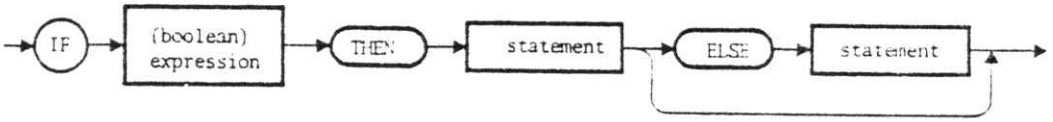


COMPOUND STATEMENT

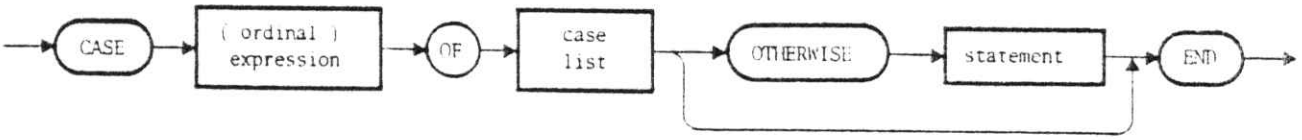


EMPTY STATEMENT

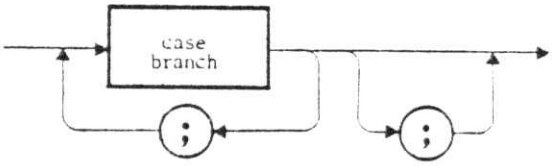
IF STATEMENT



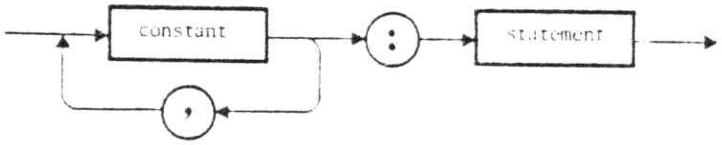
CASE STATEMENT



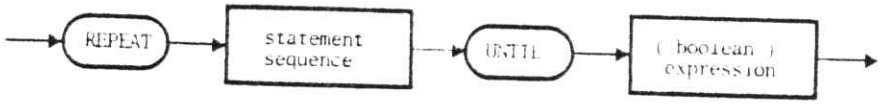
CASE LIST



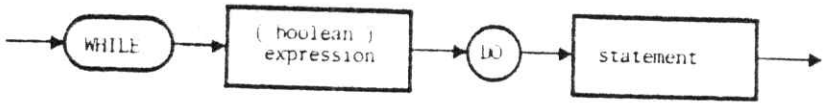
CASE BRANCH



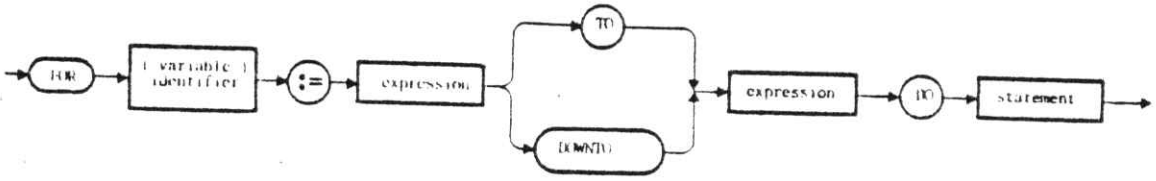
REPEAT STATEMENT



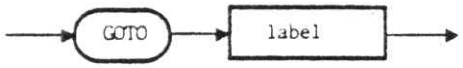
WHILE STATEMENT



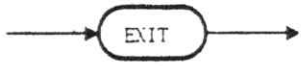
FOR STATEMENT



GOTO STATEMENT

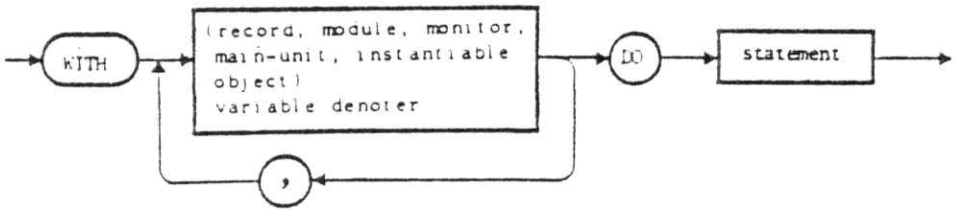


EXIT STATEMENT



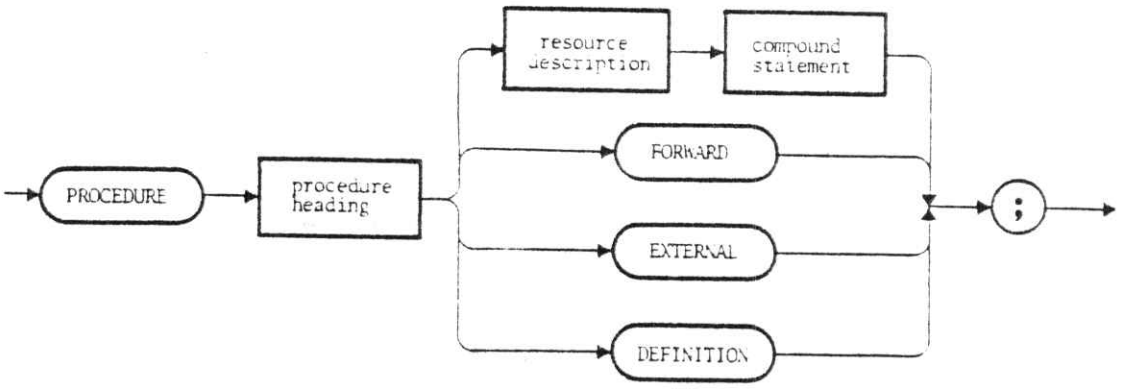
RETURN STATEMENT

WITH STATEMENT

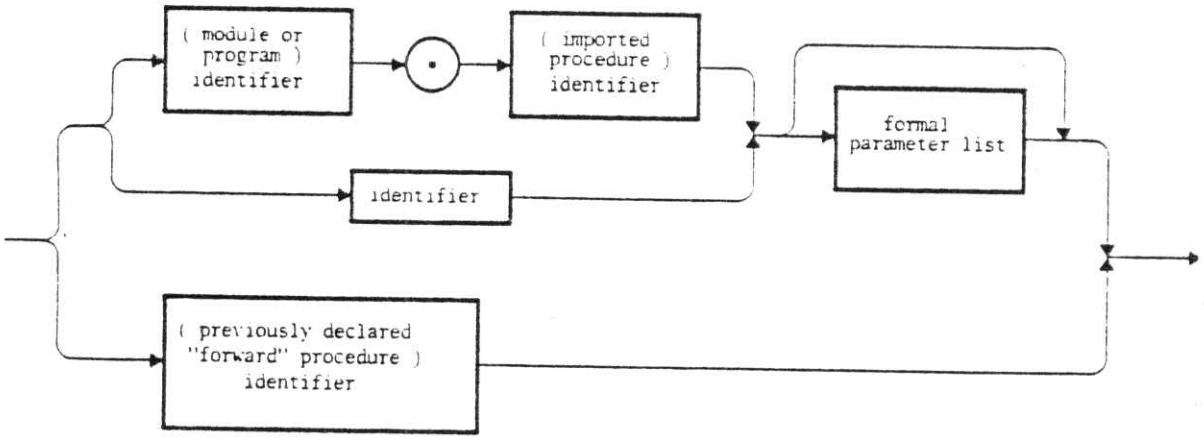


PROCEDURE AND FUNCTION

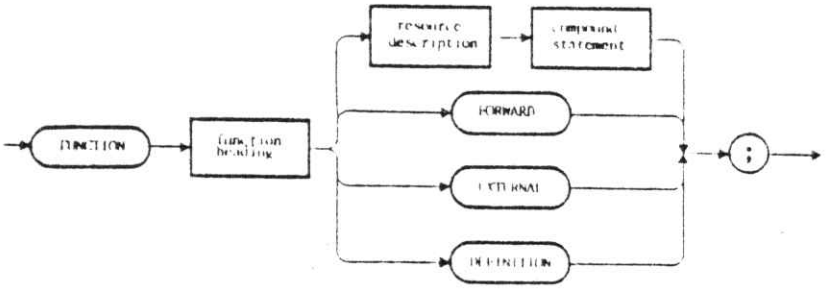
PROCEDURE DESCRIPTION



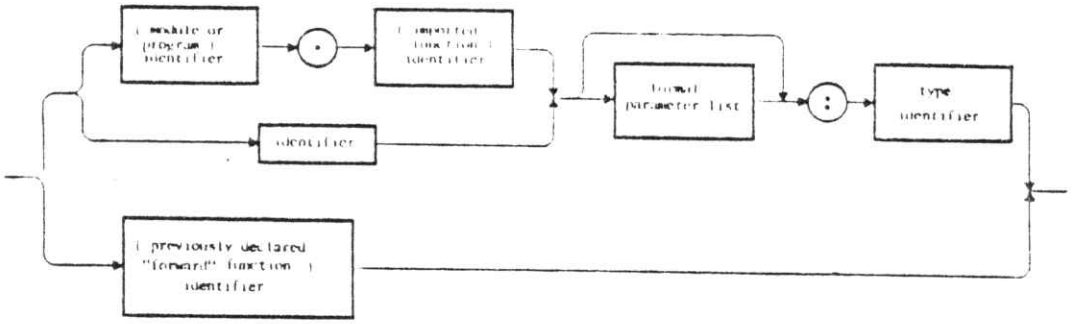
PROCEDURE HEADING



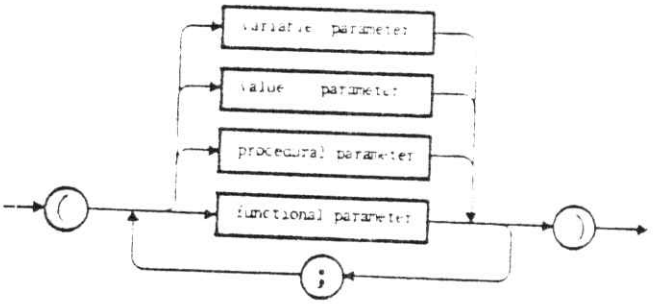
FUNCTION DESCRIPTION



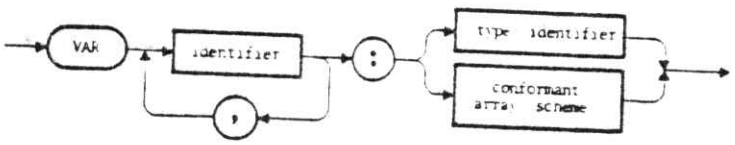
FUNCTION HEADING



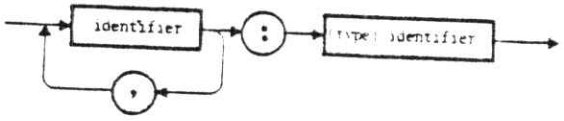
FORMAL PARAMETER LIST



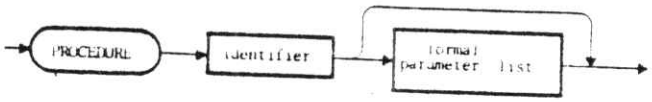
VARIABLE PARAMETER



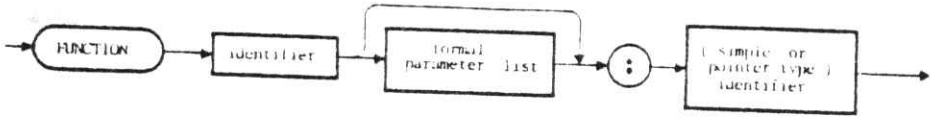
VALUE PARAMETER



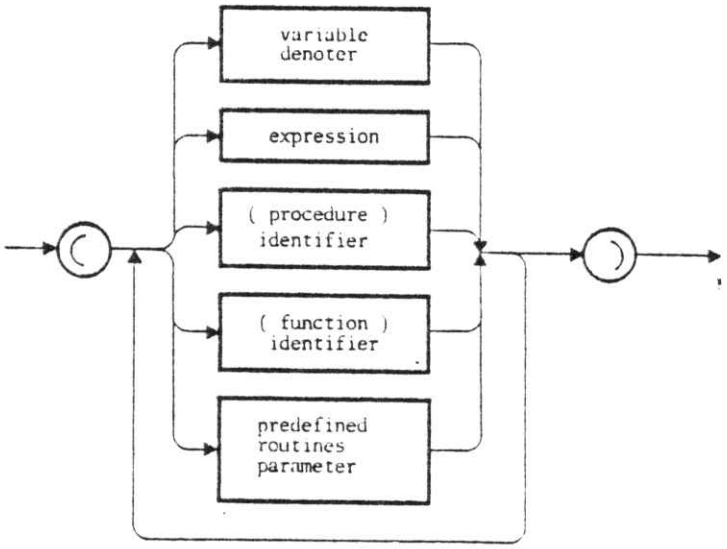
PROCEDURAL PARAMETER



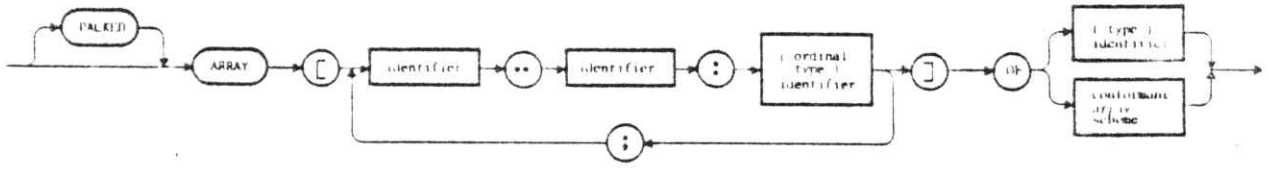
FUNCTIONAL PARAMETER



ACTUAL PARAMETER LIST



CONFORMANT ARRAY SCHEME



B. INDICE ANALITICO

ABS 13-31
ALBERI 16-7
ALLOC 22-14
ARCTAN 13-32
ARRAY SPARSI 16-1
CARATTERISTICHE FONDAMENTALI DEL LINGUAGGIO 11-2
CASE 12-9
CHIAMATA DI PROCEDURA 12-4
CHIAMATA ED ESECUZIONE DI PROCEDURE E FUNZIONI 13-6
CHR 13-36
CLOSE 13-20
COLLOCAZIONE DEL LINGUAGGIO 1-1
COME EVITARE IL SALTO
   Avanti 15-5
   Esterno ad una procedura o funzione 15-5
   Indietro 15-4
COMMENTI 2-10
COMPATIBILITA' 8-19
COMPATIBILITA' NELL' ASSEGNAZIONE 8-19
COMPILAZIONE DI LIBRERIA DI RISORSE 18-5
COMPILAZIONI SEPARATE 7-1
COS 13-31
COSTANTI 9-1
   Descrittori di costanti9-2
   Esadecimali 19-2
   Indicatori di costanti 9-1
   Ottali 19-1
COSTRUTTORI
   Array 8-8
   File 8-16
   Record 8-11
   Set 8-15
DEALLOC 22-15
DESCRIZIONE DI UNA PROCEDURA 13-1
DIRETTIVA AT 21-1
DISPOSE 13-29
DYNAMIC ARRAY 20-2
EFFETTO DI UNA CHIAMATA DI ROUTINE 6-1
ENUMERAZIONE 8-3
EOF 13-20
EOLN 13-29
ESECUZIONE DI PROGRAMMI E GESTIONE DELLA MEMORIA 6-1
ESEMPIO DI MACCHINA ASTRATTA 13-1
ESPORTAZIONE 17-4, 18-15
ESPRESSIONI 11-1
EXIT 12-17
EXP 13-31
FILES

File-testo 15-14
Normali 15-16
FOR 12-13
GESTIONE DELLA MEMORIA 6-1
GET 13-19,22-4
GOTO 12-15
HEAP 6-6
IDENTIFICATORI 2-7
IF 12-7
IMPORTAZIONE 17-4, 18-11
INPUT/OUTPUT, DETTAGLI 15-14
INSIEME DEI CARATTERI 2-3
INT 22-10
ISTANZE DI LIBRERIE DI RISORSE 18-1
ISTANZIATE
Costanti 19-4
Risorse 18-10
Routine 22-1
Variabili 21-2
ISTRUZIONI
Composta 12-5
Generalita' 12-1
Sequenza 12-4
Vuota 12-6
LIBRERIE DI RISORSE ISTANZIABILI 18-1
Come istanziare una libreria di risorse 18-1
Diagramma sintattico 18-7
LISTA DI ESPORTAZIONE 14-1
LISTA DI IMPORTAZIONE 14-3
LONGINT 22-10
LONGINTEGER 20-1
LROUND, LTRUNC 22-10
MAKEPRT 22-14
METODOLOGIA BOTTOM UP 16-10
MODULE 20-5
MODULI 14-1
Cos'e' 14-1
Diagramma sintattico 7-12
Particolarita' sui moduli 16-1
MONITOR
Cos'e' 17-8
Tipo 20-5
MOVEARRAY 22-13
MOVEFROM 22-13
MOVEINTO 22-12
NEW 13-28
NIDIFICAZIONE DELL' ISTRUZIONE IF 12-8
NOTAZIONE DEI DIAGRAMMI SINTATTICI 2-1
NOTAZIONE, VOCABOLARIO 2-1
NOTE STORICHE 1-1
NUMERI
Interi 2-5
Reali 2-5
ODD 13-37
OPERANDI 11-3
OPERATORI 11-4

Aritmetici 11-4
Booleani 11-5
Insiemi 11-5
Precedenze 11-6
Relazionali 11-6
ORD 11-35
PACK 13-33
PAGE 13-28
PARAMETRI
Funzionali 13-15
Input 15-11
Input/Output 15-12
Output 15-12
Passaggio 13-8
Procedurali 13-13
Schema di conformant array 13-16
Valore 13-12
Variabile 13-11
PASCAL+, COS'E' 1-1
PRED 13-37
PREDEFINITE
Funzioni aritmetiche 13-30
Funzioni ordinali 13-35
Procedure di conversione 22-9
Procedure e funzioni di conversione 13-32
Procedure e funzioni per la gestione di file 13-17
Procedure per l'allocazione dinamica 13-28
Risorse 4-5
Routine di comunicazione tra processi 22-11
Routine per la gestione dei file 22-2
Routine 22-10
Tipi semplici 8-5
PROBLEMA DEGLI PSEUDONOMI 15-13
PROCEDURE E FUNZIONI 13-1
PROCESSI
Cos'e' 17-1
Creazione e distruzione 17-2
PROGRAMMAZIONE
Considerazioni 15
Senza "goto" 15-4
Tramite macchine astratte 1-1
Concorrente 17-1
PROTEZIONE DI PROCESSI EFFETTIVI 17-3
PUNTO E VIRGOLA
nelle dichiarazioni 15-2
nelle intestazioni 15-3
nello statement 15-1
PUT 13-18, 22-5
READ 13-20, 13-22, 22-5
RECAST 22-9
REGOLE
Gerarchia 4-3
Ombra 4-3
Sequenziale 4-3
REPEAT 12-11, 15-7
RESET 13-19

RETURN 12-18
REWRITE 13-18
RICORSIONE 15-9
RIDEFINIZIONE ED USO DI RISORSE IMPORTATE 14-5
RIDEFINIZIONI 17-14
RIDEFINIZIONI ED USO DELLE RISORSE ISTANZIATE 18-10
RIFERIMENTI
   Buffer di file 10-4
   Campo di record 10-3
   Elemento di array 10-3
   Variabile importata 10-4
   Variabile intera 10-3
   Variabile puntata 10-3
RISORSA DI LIBRERIA
   Cos'e' 7-1
ROUND 13-32
ROUTINE
   Ricorsive 4-4
   Separate 7-6, 14-7
RUN TIME STACK 4-2
SEEK 22-3
SEEKEOF 22-3
SENSE 22-4
SEPARATORI 2-9
SEQUENZA DI SCRITTURA E DI COMPILAZIONE 3-5
SFRUTTAMENTO OTTIMALE DELLA MEMORIA AUSILIARIA 22-6
SIGNAL 22-11
SIMBOLI SPECIALI 2-8
SIN 13-31
SINCRONIZZAZIONE DI PROCESSI COOPERANTI 17-18
SIZE 22-12
SQR 13-31
SQRT 13-31
STRINGHE 2-8, 8-11
STRINGHE, SPECIALI INTERPRETAZIONI 19-3
STRUTTURA GERARCHICA DI UN PROGRAMMA 3-3
STRUTTURA LESSICALE 2-1, 2-3
   Liste 16-3
   LN 13-31
STRUTTURA DI UN' ESPRESSIONE 11-1
SUCC 13-36
TERMINOLOGIA RELATIVA ALLA NIDIFICAZIONE DELLE RISORSE 2-1
TIPI
   Compatibilita' 8-19
   Concetto 5-1
   Descrizione 8-2
   Indicatori 8-1
TIPI DEFINITI DALL'UTENTE 8-3
TIPI DI DATI INCAPSULATI 7-7,14-9
TIPI SEMPLICI 5-3, 8-3
TIPI STRUTTURATI 5-5, 8-8
TIPI SUBRANGE 8-4
TIPO
   Boolean 8-6
   Char 8-7
   Condition 17-21

Integer 8-6
Puntatore 8-17
Real 8-6
Text 8-17
TRUNC 13-32
UNITA' PRINCIPALE DI PROGRAMMA 7-9
UNPACK 13-34
UPDATE 22-2
VARIABILI
Indicatore 10-2
Descrizione 10-1
VOCABOLARIO 2-1
WAIT 22-11
WHILE 12-12, 15-7
WITH 12-19
WRITE 13-21, 13-25, 22-6
WRITELN 13-27