

Natural

Leitfaden zur Programmierung

Bestellnummer: NAT316D021ALL

Dieses Handbuch gilt für Natural ab Version 3.1.6 für Großrechner, Version 5.1.1 für Windows und Version 5.1.1 für UNIX und OpenVMS.

Hierin enthaltene Beschreibungen unterliegen Änderungen und Ergänzungen, die in nachfolgenden Update-Serien oder Neuausgaben bekanntgegeben werden.

Anmerkungen und Verbesserungsvorschläge der Leserinnen und Leser sind sehr willkommen. Bitte richten Sie Ihre Anmerkungen an:

Software AG
Dokumentation
Uhlandstraße 12
64297 Darmstadt

Telefax: 06151-92-1612

© Juni 2002, Software AG

Alle Rechte vorbehalten

Printed in the Federal Republic of Germany

Software AG und/oder Software AG Produkte sind entweder Warenzeichen oder eingetragene Warenzeichen der Software AG. Andere hier erwähnte Produkte und Unternehmensnamen können Warenzeichen ihrer jeweiligen Eigentümer sein.

INHALTSVERZEICHNIS

VORWORT	1
Zu diesem Handbuch	1
Kapitel-Übersicht	1
Beispielprogramme	2
Programmiermodi	2
Plattformspezifische Informationen	3
Deutsche Rechtschreibung	3
1. FELDER DEFINIEREN	5
DEFINE DATA-Statement	6
Struktur eines DEFINE DATA-Statements — Level-Nummern	8
Benutzervariablen	11
Namen von Benutzervariablen	12
Format und Länge von Benutzervariablen	13
Benutzerdefinierte Konstanten	15
Numerische Konstanten	15
Alphanumerische Konstanten	16
Datums- und Zeitkonstanten	18
Hexadezimale Konstanten	19
Logische Konstanten	20
Gleitkomma-Konstanten	20
Attribut-Konstanten	21
Konstanten mit Namen definieren	22
Ausgangswerte	23
Standard-Ausgangswerte	25
RESET-Statement	25
Felder redefinieren	26

Natural Leitfaden zur Programmierung

Array-Verarbeitung	29
Arrays definieren	30
Ausgangswerte für Arrays	31
Dreidimensionales Array	38
Arrays als Teil einer größeren Datenstruktur	39
Datenbank-Arrays	40
Arithmetische Ausdrücke in Index-Notationen	41
Arithmetische Funktionen bei Arrays	41
Datenblöcke	43
Datenblöcke definieren	44
Block-Hierarchien	46
2. DATENBANKZUGRIFFE	47
DDMs (Datendefinitionsmodule)	48
DDM anzeigen	49
Bestandteile eines DDMs	49
Datenbank-Arrays	52
Multiple Felder	53
Periodengruppen	54
Multiple Felder und Periodengruppen referenzieren	55
Multiple Felder innerhalb von Periodengruppen	56
Internen Zähler eines Datenbank-Arrays referenzieren	58
DEFINE DATA-Views	59
Statements für Datenbankzugriffe	61
READ-Statement	62
Syntax	63
Anzahl der zu lesenden Datensätze begrenzen	65
STARTING- und ENDING-Klausel bei READ-Statement	66
WHERE-Klausel beim READ-Statement	68
FIND-Statement	71
Syntax	71
Anzahl der zu verarbeitenden Datensätze begrenzen	72
WHERE-Klausel beim FIND-Statement	73
IF NO RECORDS FOUND-Bedingung	74

HISTOGRAM-Statement	76
Syntax	76
Anzahl der zu lesenden Werte begrenzen	77
STARTING- und ENDING-Klausel bei HISTOGRAM-STATEMENT	77
WHERE-Klausel beim HISTOGRAM-Statement	77
Datenbank-Verarbeitungsschleifen	79
Hierarchien von Verarbeitungsschleifen	81
Datenänderungen — Transaktionsverarbeitung	85
Logische Transaktionen	86
Datensatz-Kontrolle während einer Transaktion (“Hold”-Logik)	88
Transaktion abrechnen	90
Transaktion neu starten	90
Statements ACCEPT und REJECT	93
AT START/END OF DATA-Statements	97
AT START OF DATA-Statement	97
AT END OF DATA-Statement	97
3. AUSGABE VON DATEN	101
Layout einer Ausgabeseite — Übersicht	102
Statements DISPLAY und WRITE	105
DISPLAY-Statement	105
WRITE-Statement	107
Spaltenabstand — der SF-Parameter und die Notation nX	110
Tabulator-Notation nT	112
Zeilenvorschub — die Schrägstrich-Notation (/)	113
Index-Notation (n:n) für multiple Felder und Periodengruppen	115

Natural Leitfaden zur Programmierung

Seitenüberschriften und Seitenvorschübe	118
Standard-Seitenüberschrift	118
Seitenüberschrift unterdrücken — die NOTITLE-Option	119
Eigene Seitenüberschrift definieren — das WRITE TITLE-Statement	120
Logische Seite und physische Seite	122
Seitengröße — der PS-Parameter	123
Seitenvorschub — der EJ-Parameter	124
Seitenvorschub — die Statements EJECT und NEWPAGE	124
Seiten-Fußzeile — das WRITE TRAILER-Statement	127
AT TOP OF PAGE-Statement	128
AT END OF PAGE-Statement	128
Spaltenüberschriften	130
Standard-Spaltenüberschriften	131
Standard-Spaltenüberschriften unterdrücken — die NOHDR-Option	132
Eigene Spaltenüberschriften definieren	133
NOTITLE und NOHDR kombinieren	134
Spaltenüberschriften zentrieren — der HC-Parameter	135
Breite von Spaltenüberschriften — der HW-Parameter	135
Füllzeichen für Überschriften — die Parameter FC und GC	136
Unterstreichungszeichen für Überschriften — der UC-Parameter	138
Spaltenüberschriften unterdrücken — die Notation '/'	139
Parameter zur Beeinflussung der Ausgabe von Feldern	141
Vorangestellte Zeichen — der LC-Parameter	142
Einfügungszeichen — der IC-Parameter	142
Nachgestellte Zeichen — der TC-Parameter	142
Ausgabelänge — der AL- und NL-Parameter	143
Vorzeichen-Stelle — der SG-Parameter	143
Ausgabe identischer Werte unterdrücken — der IS-Parameter	146
Nullwerte anzeigen — der ZP-Parameter	148
Leerzeilenunterdrückung — der ES-Parameter	149
Editiermasken — der EM-Parameter	153
Editiermasken für numerische Felder	154
Editiermasken für alphanumerische Felder	154
Länge der Felder	154
Editiermasken für Datums- und Zeitfelder	155
Beispiele für Editiermasken	155

Vertikale Ausgaben	159
Kombination von DISPLAY und WRITE	159
Tabulator-Notation T*Feld	161
Positionierungsnotation x/y	162
DISPLAY VERT-Statement	164
Tabulator-Notation P*Feld	169
4. OBJEKTTYPEN	173
Welche Typen von Programmierobjekten gibt es?	174
Data Areas	175
Local Data Area	175
Global Data Area	177
Parameter Data Area	179
Programme, Subprogramme und Subroutinen	181
Modulare Anwendungsstruktur	181
Mehrere Stufen (Levels) aufgerufener Objekte	182
Programm	184
Subroutine	187
Subprogramm	191
Verarbeitungsablauf beim Aufruf eines Unterprogramms	193
Maps	194
Helproutinen	195
Helproutinen aufrufen	196
Helproutinen spezifizieren	197
Programmierhinweise für Helproutinen	197
Parameter an Helproutinen übergeben	198
Hilfe als eingeblendetes Fenster	200
Mehrfache Verwendung von Sourcecode — Copycode	202
Natural-Objekte dokumentieren — Text	203
Ereignisgesteuerte Anwendungen erstellen — Dialog	204
Verteilte objekt-basierte Anwendungen erstellen — Class	204
Nicht-Natural-Dateien benutzen — Ressource	204
Geteilte Ressourcen	205
Private Ressourcen	206

5. WEITERE PROGRAMMIERASPEKTE	207
Ende eines Programms — Das END-Statement	207
Ende einer Anwendung — Das STOP-Statement	207
Bedingte Verarbeitung — Das IF-Statement	208
Geschachtelte IF-Statements	210
Schleifenverarbeitung	212
Schleifendurchläufe bei Datenbankzugriffen begrenzen	213
Durchläufe bei Nicht-Datenbankzugriffsschleifen begrenzen — das REPEAT-Statement ..	215
Verarbeitungsschleife verlassen — das ESCAPE-Statement	217
Schleifen innerhalb von Schleifen	217
Statements innerhalb eines Programms referenzieren	219
Gruppenwechsel	223
AT BREAK-Statement	223
Automatische Gruppenwechsel-Verarbeitung	231
BEFORE BREAK PROCESSING-Statement	235
Programmabhängige Gruppenwechsel-Verarbeitung — das PERFORM BREAK PROCESSING-Statement	237
Datenberechnungen	241
Formate der Felder	241
COMPUTE-Statement	241
Statements MOVE und COMPUTE	243
Statements ADD, SUBTRACT, MULTIPLY und DIVIDE	243
COMPRESS-Statement	245
Mathematische Funktionen	249
Systemvariablen und Systemfunktionen	250
Systemvariablen	250
Systemfunktionen	252
Stack	255
Verarbeitung des Stack	256
Daten im Stack ablegen	257
Stack-Inhalt löschen	258

Datumsinformationen verarbeiten	259
Editiermasken für Datumsfelder and Datumssystemvariablen	259
Standard-Editiermaske für Datum — der DTFORM-Parameter	260
Datumsformat für alphanumerische Darstellung — der DF-Parameter	260
Datumsformat für Ausgabe — der DFOUT-Parameter	264
Datumsformat für Stack — der DFSTACK-Parameter	265
“Year Sliding Window” — der YSLW-Parameter	267
Kombinationen von DFSTACK und YSLW	270
Datumsformat für Standard-Seitenüberschriften — der DFTITLE-Parameter	274
6. REPORTING MODE UND STRUCTURED MODE	275
Allgemeine Informationen	275
Programmiermodus festlegen	276
7. PORTIERBARE VON NATURAL GENERIERTE PROGRAMME	283
Kompatibilität	283
Betrachtungen zum Endian-Modus	284
ENDIAN-Parameter	285
Von Natural generierte Programme übertragen	286
INDEX	289

VORWORT

Zu diesem Handbuch

Dieses Handbuch gilt für alle Plattformen, auf denen Natural verwendet werden kann.

Es enthält grundlegende Informationen zu verschiedenen Aspekten der Programmierung mit Natural. Sie sollten zuerst mit diesen Informationen vertraut sein, bevor Sie anfangen, Natural-Anwendungen zu schreiben.

Kapitel-Übersicht

1. Felder definieren (Seite 5)

Dieses Kapitel beschreibt, wie Sie die Felder definieren, die Sie in einem Programm verwenden möchten.

2. Datenbankzugriffe (Seite 47)

Dieses Kapitel beschreibt, wie Sie mit Natural auf Daten in einer Datenbank zugreifen können.

3. Ausgabe von Daten (Seite 101)

Dieses Kapitel beschreibt, wie Sie das Aussehen eines mit Natural erzeugten Ausgabe-Reports, d.h. die Art, in der die Daten angezeigt werden, beeinflussen können.

4. Objekttypen (Seite 173)

Innerhalb einer Anwendung können Sie verschiedene Typen von Programmierobjekten benutzen, um eine effiziente Anwendungsstruktur zu erhalten. Dieses Kapitel beschreibt die verschiedenen Typen von Natural-Programmierobjekten: Datenbereiche (Data Areas), Programme, Subprogramme, Subroutinen, Helprountinen, Maps usw.

5. Weitere Programmieraspekte (Seite 207)

Dieses Kapitel beschreibt verschiedene andere Aspekte der Programmierung mit Natural.

6. Reporting Mode und Structured Mode (Seite 275)

Dieses Kapitel beschreibt die Unterschiede zwischen den beiden Programmiermodi.

7. Portierbare mit Natural generierte Programme (Seite 275)

Ab Natural 5 sind generierte Programme über die Plattformen UNIX, OpenVMS und Windows hinweg portierbar.

Beispielprogramme

Der *Natural Leitfaden zur Programmierung* enthält zahlreiche Beispiele für Natural-Programme sowie Verweise auf weitere im Handbuch nicht abgebildete Beispielprogramme.

Den Sourcecode all dieser Programme finden Sie in der Natural-Library “SYSEXPG”. (Die Programme sind alle im *Structured Mode* geschrieben.)

Weitere Beispielprogramme für die Verwendung von Natural-Statements finden Sie in der Natural-Library “SYSEXRM”.

Welche Zugriffsmöglichkeiten Sie auf diese Libraries haben, erfahren Sie von Ihrem Natural-Administrator.

Die Beispielprogramme greifen auf Daten der Dateien “EMPLOYEES” (Personaldaten) und “VEHICLES” (Fahrzeugdaten) zu, die von der Software AG speziell zu Demonstrationszwecken erstellt wurden.

Programmiermodi

Natural bietet zwei Formen der Programmierung: *Reporting Mode* und *Structured Mode*. Grundsätzlich empfiehlt es sich, ausschließlich im *Structured Mode* zu programmieren, um eine klar strukturierte Anwendung zu gewährleisten. Daher beziehen sich alle Erklärungen und Beispiele in diesem Handbuch auf den *Structured Mode*.

Irgendwelche Besonderheiten des *Reporting Mode* werden nicht berücksichtigt. (Unterschiede zwischen den beiden Modi sind im Kapitel **Reporting Mode und Structured Mode** beschrieben.)

Plattformspezifische Informationen

Wo erforderlich, werden plattformspezifische Informationen in der aktuellen Dokumentation durch die folgenden Begriffe identifiziert:

Großrechner	Bezieht sich auf die Betriebssysteme OS/390, VSE/ESA, VM/CMS und BS2000/OSD, sowie auf alle von Natural unter diesen Betriebssystemen unterstützten TP-Monitore.
OpenVMS	Bezieht sich auf das Betriebssystem OpenVMS.
UNIX	Bezieht sich auf alle von Natural unterstützten UNIX-Systeme.
Windows	<p>Bezieht sich auf die folgenden Betriebssysteme:</p> <p>In einer Natural-Entwicklungsumgebung:</p> <ul style="list-style-type: none"> • Microsoft Windows NT • Microsoft Windows 2000 <p>In einer Natural-Laufzeitumgebung:</p> <ul style="list-style-type: none"> • Microsoft Windows 98 • Microsoft Windows NT • Microsoft Windows 2000

Deutsche Rechtschreibung

Dieses Handbuch ist in der alten deutschen Rechtschreibweise abgefaßt.

FELDER DEFINIEREN

Dieses Kapitel beschreibt, wie Sie die Felder definieren, die Sie in einem Programm verwenden möchten. Diese Felder können entweder Datenbankfelder oder benutzerdefinierte Felder sein. Die Informationen in diesem Kapitel beziehen sich auf alle Felder im allgemeinen und auf Benutzervariablen im besonderen. Die Besonderheiten von Datenbankfeldern sind im Kapitel **Datenbankzugriffe** (Seite 47) beschrieben.

Dieses Kapitel beschreibt folgende Punkte:

- das Statement DEFINE DATA
- Struktur eines DEFINE DATA-Statements — Level-Nummern
- Benutzervariablen
- Benutzerkonstanten
- Ausgangswerte für Variablen sowie das Statement RESET
- Felder redefinieren
- Arrays
- Datenblöcke.

Bitte beachten Sie, daß dieses Kapitel sich auf die Hauptaspekte des DEFINE DATA-Statements beschränkt. Weitere Optionen sind im *Natural Statements-Handbuch* beschrieben.

DEFINE DATA-Statement

Das erste Statement in einem Natural-Programm muß immer ein DEFINE DATA-Statement sein. In diesem Statement definieren Sie alle Felder — Datenbankfelder wie Benutzervariablen — die in dem Programm verwendet werden sollen.

Alle zu verwendenden Felder *müssen* im DEFINE DATA-Statement definiert werden.

Es gibt zwei Möglichkeiten, die Felder zu definieren:

- Die Felder können innerhalb des DEFINE DATA-Statements selbst definiert werden.
- Die Felder können außerhalb des Programms in einer Local Data Area oder Global Data Area definiert werden, wobei das DEFINE DATA-Statement diese Data Area referenziert.

Felder, die von mehreren Programmen/Unterprogrammen benutzt werden, sollten in einer programm-externen Data Area definiert werden.

Im Hinblick auf eine klare Anwendungsstruktur empfiehlt es sich in der Regel, Felder in programm-externen Data Areas zu definieren.

Data Areas werden mit dem Data-Area-Editor erstellt und gepflegt, der in Ihrem *Natural-Benutzerhandbuch* bzw. *User's Guide* beschrieben ist.

Im ersten Beispiel sind die Felder im DEFINE DATA-Statement des Programms definiert. Im zweiten Beispiel sind die gleichen Felder in einer Local Data Area definiert, und das DEFINE DATA-Statement enthält lediglich eine Referenz auf diese Data Area.

Beispiel 1 — Felder im DEFINE DATA-Statement definiert:

```
DEFINE DATA LOCAL
1 VIEWEMP VIEW OF EMPLOYEES
  2 NAME
  2 FIRST-NAME
  2 PERSONNEL-ID
1 #VARI-A (A20)
1 #VARI-B (N3.2)
1 #VARI-C (I4)
END-DEFINE
...
```

Beispiel 2 — Felder in einer separaten Data Area definiert:

Programm:

```
DEFINE DATA LOCAL
  USING LDA39
END-DEFINE
...
```

Local Data Area "LDA39":

I	T	L	Name	F	Leng	Index/Init/EM/Name/Comment
V	1		VIEWEMP			EMPLOYEES
	2		NAME	A	20	
	2		FIRST-NAME	A	20	
	2		PERSONNEL-ID	A	8	
	1		#VARI-A	A	20	
	1		#VARI-B	N	3.2	
	1		#VARI-C	I	4	

Struktur eines DEFINE DATA-Statements — Level-Nummern

Level-Nummern werden innerhalb eines DEFINE DATA-Statements verwendet, um die Struktur und Gruppierung der Definitionen zu zeigen. Dies ist relevant bei

- View-Definitionen
- Feldgruppen
- Redefinitionen.

Level-Nummern sind 1- oder 2-stellige Zahlen von 01 bis 99 (die vorangestellte “0” kann weggelassen werden).

Im allgemeinen sind Variablen-Definitionen auf Level 1.

Die Level-Angaben in View-Definitionen, Redefinitionen und Gruppen müssen lückenlos sein: es dürfen keine Level-Nummern ausgelassen werden.

Level-Nummern in View-Definitionen

Wenn Sie einen View definieren, geben Sie den View-Namen auf Level 1 an und die Felder, aus denen der View besteht, auf Level 2. (Näheres zu View-Definitionen finden Sie im Kapitel **Datenbankzugriffe**, Seite 47.)

Beispiel für Level-Nummern in einer View-Definition:

```
DEFINE DATA LOCAL
1 VIEWEMP VIEW OF EMPLOYEES
  2 NAME
  2 FIRST-NAME
  2 BIRTH
  ...
END-DEFINE
```

Level-Nummern in Feldgruppen

Mit der Definition von Gruppen ist es möglich, eine Reihe aufeinanderfolgender Felder auf einfache Weise zu referenzieren. Wenn Sie mehrere Felder unter einem gemeinsamen Gruppennamen definieren, können Sie diese Felder später im Programm referenzieren, indem Sie statt der Namen der einzelnen Felder lediglich den Gruppennamen angeben.

Der Gruppenname muß auf Level 1 definiert werden und die in der Gruppe enthaltenen Felder jeweils einen Level darunter.

Für Gruppennamen gelten die gleichen Namenskonventionen wie für Benutzervariablen.

Beispiel für Level-Nummern in Gruppe:

```
DEFINE DATA LOCAL
1 #FIELD A (N2.2)
1 #FIELD B (I4)
1 #GROUP A
  2 #FIELD C (A20)
  2 #FIELD D (A10)
  2 #FIELD E (N3.2)
1 #FIELD F (A2)
...
END-DEFINE
```

In diesem Beispiel sind die Felder #FIELD C, #FIELD D und #FIELD E unter dem gemeinsamen Gruppennamen #GROUP A definiert. Die anderen drei Felder sind nicht Teil der Gruppe. Bitte beachten Sie, daß #GROUP A nur als Gruppenname dient und selbst kein Feld ist (und daher auch keine Format-/Längendefinition hat).

Level-Nummern in Redefinitionen

Wenn Sie ein Feld redefinieren, muß die REDEFINE-Option auf dem gleichen Level sein wie die Definition des Feldes, das redefiniert wird; und die Felder, die sich aus der Redefinition ergeben, müssen einen Level darunter sein. (Näheres zu Redefinitionen finden Sie unter **Felder redefinieren**, Seite 26.)

Beispiel für Level-Nummern in Redefinition:

```
DEFINE DATA LOCAL
1 VIEWEMP VIEW OF STAFFDDM
  2 BIRTH
  2 REDEFINE BIRTH
    3 #YEAR-OF-BIRTH (N4)
    3 #MONTH-OF-BIRTH (N2)
    3 #DAY-OF-BIRTH (N2)
1 #FIELDA (A20)
1 REDEFINE #FIELDA
  2 #SUBFIELD1 (N5)
  2 #SUBFIELD2 (A10)
  2 #SUBFIELD3 (N5)
  ...
END-DEFINE
```

In diesem Beispiel wird das Datenbankfeld BIRTH als drei Benutzervariablen redefiniert, und die Benutzervariable #FIELDA wird als drei andere Benutzervariablen redefiniert.

Benutzervariablen

Von Ihnen selbst in einem Programm definierte Variablen werden als *Benutzervariablen* bezeichnet. Sie können Sie dazu verwenden, in einem Programm Werte oder Zwischenergebnisse zu speichern, die Sie später weiterverarbeiten oder ausgeben möchten.

Sie definieren eine Benutzervariable, indem Sie ihren Namen sowie ihr Format und ihre Länge im DEFINE DATA-Statement angeben.

Beispiel:

In diesem Beispiel ist eine Benutzervariable mit alphanumerischem Format und einer Länge von 10 Stellen unter dem Namen #FIELD1 definiert.

```
DEFINE DATA LOCAL  
1 #FIELD1 (A10)  
...  
END-DEFINE
```

Die folgenden Themen werden nachfolgend erörtert:

- Namen von Benutzervariablen
- Format und Länge von Benutzervariablen

Namen von Benutzervariablen

Der Name einer Benutzervariablen darf 1 bis 32 Zeichen lang sein.

Anmerkung:

Sie können Variablennamen verwenden, die über 32 Stellen lang sind (zum Beispiel um in komplexen Anwendungen die Lesbarkeit der Programme durch längere, "sprechendere" Variablennamen zu erhöhen); allerdings sind nur die ersten 32 Stellen signifikant und müssen eindeutig sein, die restlichen Zeichen werden von Natural ignoriert.

Der Name einer Benutzervariablen darf kein reserviertes Natural-Wort sein.

Innerhalb eines Natural-Programms sollten Sie für eine Benutzervariable und ein Datenbankfeld nicht den gleichen Namen verwenden.

Der Name einer Benutzervariablen darf folgende Zeichen enthalten:

Zeichen	Bedeutung
A bis Z	Buchstaben
0 bis 9	Ziffern
-	Bindestrich
@	"At"-Zeichen
_	Unterstrich
/	Schrägstrich
\$	Dollar-Zeichen
§	Paragraphenzeichen
&	Kaufmännisches Und-Zeichen
#	Rautenzeichen
+	Plus-Zeichen (nur als erstes Zeichen erlaubt)

Das erste Zeichen eines Variablennamens muß eines der folgenden Zeichen sein:

- ein Großbuchstabe
- #
- +
- &

Anmerkung:

In diesem Kapitel beginnen die Namen aller Benutzervariablen mit einem Rautenzeichen (#); dadurch werden Namenskonflikte mit Datenbankfeldern oder reservierten Natural-Wörtern von vorneherein ausgeschlossen.

Falls das erste Zeichen ein “#”, “+”, oder “&” ist, muß der Name aus mindestens einem weiteren Zeichen bestehen.

“+” darf nur bei applikationsunabhängigen Variablen (AIVs) und Variablen in einer Global Data Area als erstes Zeichen eines Namens verwendet werden. AIV-Namen *müssen* mit “+” anfangen.

“&” als erstes Zeichen eines Namens wird bei der dynamischen Source-Generierung verwendet (siehe RUN-Statement im *Natural Statements-Handbuch*) sowie als dynamisch ersetzbares Platzhalterzeichen in Processing Rules (siehe Map-Editor-Beschreibung in Ihrem *Natural-Benutzerhandbuch* bzw. *User’s Guide*).

Format und Länge von Benutzervariablen

Format und Länge einer Benutzervariablen werden in Klammern hinter dem Variablennamen angegeben.

Eine Benutzervariable kann eines der folgenden Formate haben:

A	Alphanumerisch
B	Binär
C	Dynamische Attribut-Zuweisung (C = attribute control)
D	Datum
F	Gleitkomma (F = floating point)
I	Integer (Ganzzahl)
L	Logisch
N	Numerisch, ungepackt
P	Numerisch, gepackt
T	Zeit (T = time)

Angaben über mögliche Längen von Benutzervariablen finden Sie im *Natural-Referenzhandbuch*.

Beispiele für Benutzervariablen:

```
DEFINE DATA LOCAL
#A1 (A10)      /* Alphanumerisch, 10 Stellen lang.
#A2 (B4)       /* Binär, 4 Stellen lang.
#A3 (P4)       /* Gepackt numerisch, 4 Stellen und 1 Vorzeichen-Stelle.
#A4 (N7.2)     /* Ungepackt numerisch,
               /* 7 Stellen vor, 2 nach dem Komma.
#A5 (N7.)      /* Ungültige Definition!!!
#A6 (P7.2)     /* Gepackt numerisch, 1 Vorzeichen-Stelle,
               /* 7 Stellen vor und 2 Stellen nach dem Komma.
#INT1 (I1)     /* Ganzzahl (Integer), 1 Byte.
#INT2 (I2)     /* Ganzzahl (Integer), 2 Bytes.
#INT3 (I3)     /* Ungültige Definition!!!
#INT4 (I4)     /* Ganzzahl (Integer), 4 Bytes.
#INT5 (I5)     /* Ungültige Definition!!!
#FLT4 (F4)     /* Gleitkomma (Floating Point), 4 Bytes.
#FLT8 (F8)     /* Gleitkomma (Floating Point), 8 Bytes.
#FLT2 (F2)     /* Ungültige Definition!!!
#DATE (D)      /* Datum (internes Format/Länge P6).
#TIME (T)      /* Zeit (internes Format/Länge P12).
#SWITCH (L)    /* Logisch, 1 Byte (TRUE oder FALSE).
...
END-DEFINE
```

Anmerkung:

Wird eine mit Format "P" definierte Benutzervariable mit einem DISPLAY-, WRITE- oder INPUT-Statement ausgegeben, setzt Natural für die Ausgabe das Format intern auf "N" um.

Benutzerdefinierte Konstanten

Konstanten können überall in Natural-Programmen verwendet werden. Dieser Abschnitt beschreibt, welche Arten von Konstanten es gibt und wie sie verwendet werden.

- Numerische Konstanten
- Alphanumerische Konstanten
- Datums- und Zeit-Konstanten
- Hexadezimale Konstanten
- Logische Konstanten
- Gleitkomma-Konstanten
- Attribut-Konstanten
- Konstanten mit Namen definieren.

Numerische Konstanten

Eine numerische Konstante kann aus 1 bis 29 Ziffern bestehen.

Eine numerische Konstante, die mit einem arithmetischen, COMPUTE- oder MOVE-Statement verwendet wird, kann ein Komma (Dezimalpunkt) und ein Vorzeichen enthalten.

Beispiele:

```
MOVE 3 TO #XYZ
```

```
COMPUTE #PRICE = 23.34
```

```
COMPUTE #XYZ = -103
```

```
COMPUTE #A = #B * 6074
```

Alphanumerische Konstanten

Eine alphanumerische Konstante kann 1 bis 253 alphanumerische Zeichen lang sein.

Eine alphanumerische Konstante muß entweder in Apostrophen (') oder in Anführungszeichen (") stehen.

Beispiele:

```
MOVE 'ABC' TO #XYZ
```

```
MOVE '% INCREASE' TO #TITLE
```

```
DISPLAY "LAST-NAME" NAME
```

Ein Apostroph, das Teil einer in Apostrophen stehenden alphanumerischen Konstanten ist, muß entweder durch doppelte Apostrophe oder durch ein einzelnes Anführungszeichen dargestellt werden.

Ein Apostroph, das Teil einer in Anführungszeichen stehenden alphanumerischen Konstanten ist, wird als einzelnes Apostroph dargestellt.

Beispiel:

Um folgende Ausgabe zu erhalten:

```
HE SAID, 'HELLO'
```

können Sie jede der folgenden Notationen verwenden:

```
WRITE 'HE SAID, ''HELLO'''
```

```
WRITE 'HE SAID, "HELLO"'
```

```
WRITE "HE SAID, ""HELLO"""
```

```
WRITE "HE SAID, 'HELLO'"
```

Eine alphanumerische Konstante, die dazu benutzt wird, einer Benutzervariablen einen Wert zuzuweisen, darf nicht über eine einzige Sourcecode-Zeile hinausgehen.

Mittels eines Bindestriches können mehrere alphanumerische Konstanten zu einem einzigen Wert verkettet werden.

Beispiele:

```
MOVE 'XXXXXX' –  
      'YYYYYY' TO #FIELD
```

```
MOVE "ABC" – 'DEF' TO #FIELD
```

Auf diese Weise können alphanumerische Konstanten auch mit hexadezimalen Konstanten verkettet werden.

Datums- und Zeitkonstanten

Eine Datumskonstante kann in Verbindung mit einer Variablen, die das Format “D” hat, verwendet werden. Datumskonstanten können folgende Form haben:

D'yyyy-mm-dd'	International
D'dd.mm.yyyy'	Deutsch
D'dd/mm/yyyy'	Europäisch
D'mm/dd/yyyy'	USA

wobei *dd* für den Tag, *mm* für den Monat und *yyyy* für die Jahreszahl steht.

Beispiel:

```
DEFINE DATA LOCAL
1 #DATE (D)
END-DEFINE
...
MOVE D'1997-08-11' TO #DATE
...
```

Das Standardformat für Datumsanzeigen wird vom Natural-Administrator mit dem Profilparameter DTFORM festgesetzt.

Eine Zeitkonstante kann in Verbindung mit einer Variablen, die das Format “T” hat, verwendet werden:

T'hh:ii:ss'

wobei *hh* für Stunden, *ii* für Minuten und *ss* für Sekunden steht.

Beispiel:

```
DEFINE DATA LOCAL
1 #TIME (T)
END-DEFINE
...
MOVE T'11:33:00' TO #TIME
```

Hexadezimale Konstanten

Hexadezimale Konstanten werden zur Eingabe von Werten benutzt, die nicht über auf der Tastatur verfügbare Zeichen eingegeben werden können.

Eine hexadezimale Konstante ist durch den Präfix "H" gekennzeichnet. Die Konstante selbst darf aus den Hexadezimalzeichen 0 – 9 und A – F bestehen und muß in Apostrophen stehen. Je zwei Hexadezimalzeichen stehen für ein Datenbyte.

Die hexadezimale Darstellung eines Zeichens ist unterschiedlich, je nachdem ob Ihr Computer ASCII- oder EBCDIC-Zeichensatz verwendet. Wenn Sie hexadezimale Konstanten auf einen anderen Computer übertragen, müssen Sie daher gegebenenfalls eine Zeichenumsetzung vornehmen.

ASCII-Beispiele:

H'313233' (entspricht der alphanumerischen Konstanten '123')

H'414243' (entspricht der alphanumerischen Konstanten 'ABC')

EBCDIC-Beispiele:

H'F1F2F3' (entspricht der alphanumerischen Konstanten '123')

H'C1C2C3' (entspricht der alphanumerischen Konstanten 'ABC')

Hexadezimale Konstanten können durch einen Bindestrich miteinander verkettet werden.

ASCII-Beispiel:

H'414243' – H'444546' (entspricht 'ABCDEF')

EBCDIC-Beispiel:

H'C1C2C3' – H'C4C5C6' (entspricht 'ABCDEF')

Logische Konstanten

Einem Feld mit Format “L” kann durch die logischen Konstanten “TRUE” (wahr) und “FALSE” (falsch) ein logischer Wert zugewiesen werden.

Beispiel:

```
DEFINE DATA LOCAL
1 #FLAG (L)
END-DEFINE
...
MOVE TRUE TO #FLAG
...
IF #FLAG ...
    statement ...
    MOVE FALSE TO #FLAG
END-IF
...
```

Gleitkomma-Konstanten

Gleitkomma-Konstanten können mit Variablen, die das Format “F” haben, benutzt werden.

Beispiel:

```
DEFINE DATA LOCAL
1 #FLT1 (F4)
END-DEFINE
...
COMPUTE #FLT1 = -5.34E+2
...
```

Attribut-Konstanten

Attribut-Konstanten können mit Variablen (Kontrollvariablen), die das Format “C” haben, verwendet werden. Attribut-Konstanten müssen in Klammern stehen.

Die folgenden Attribute können verwendet werden:

AD=D	Standard (default)	CD=BL	blau
AD=B	blinkend	CD=GR	grün
AD=I	intensiviert	CD=NE	neutral
AD=N	nicht angezeigt	CD=PI	pink
AD=V	invers (reverse video)	CD=RE	rot (red)
AD=U	unterstrichen	CD=TU	türkis
AD=C	kursiv (cursive/italic)	CD=YE	gelb (yellow)
AD=Y	dynamisches Attribut		
AD=P	geschützt (protected)		

Beispiel:

```

DEFINE DATA LOCAL
1 #ATTR (C)
1 #FIELD (A10)
END-DEFINE
...
MOVE (AD=I CD=BL) TO #ATTR
...
INPUT #FIELD (CV=#ATTR)
...

```

Konstanten mit Namen definieren

Wenn Sie denselben konstanten Wert mehrmals in einem Programm verwenden müssen, können Sie den Wartungsaufwand dadurch reduzieren, daß Sie eine Konstante mit Namen definieren:

Sie definieren ein Feld im DEFINE DATA-Statement, weisen ihm einen konstanten Wert zu, und verwenden im Programm den Feldnamen statt des konstanten Wertes. Falls Sie den Wert ändern müssen, brauchen Sie ihn nur einmal im DEFINE DATA-Statement ändern und nicht an jeder Stelle im Programm, an der er verwendet wird.

Sie geben den konstanten Wert in spitzen Klammern mit dem Schlüsselwort "CONSTANT" hinter der Felddefinition im DEFINE DATA-Statement an. Falls der Wert alphanumerisch ist, muß er in Apostrophen stehen.

Beispiel:

```
DEFINE DATA LOCAL
1 #FIELD A (N3) CONSTANT <100>
1 #FIELD B (A5) CONSTANT <'ABCDE'>
END-DEFINE
...
```

Während der Programmausführung kann der Wert einer solchen namentlich definierten Konstanten nicht geändert werden.

Ausgangswerte

Sie können einer Benutzervariablen einen Ausgangswert zuweisen. Sie geben den Ausgangswert in spitzen Klammern mit dem Schlüsselwort "INIT" hinter der Felddefinition im DEFINE DATA-Statement an. Falls der Wert alphanumerisch ist, muß er in Apostrophen stehen.

Beispiel:

```
DEFINE DATA LOCAL
1 #FIELD A (N3) INIT <100>
1 #FIELD B (A20) INIT <'ABC'>
END-DEFINE
...
```

Als Ausgangswert für ein Feld kann auch der Wert einer Natural-Systemvariablen genommen werden.

Beispiel für Systemvariable als Ausgangswert:

```
DEFINE DATA LOCAL
1 #MYDATE (D) INIT < *DATX >
END-DEFINE
...
```

Als Ausgangswert können Sie auch eine Variable vollständig oder teilweise mit einem bestimmten Zeichen oder einer Zeichenkette füllen (nur bei alphanumerischen Variablen möglich).

Mit der Option **FULL LENGTH** *<Zeichen>* füllen Sie das gesamte Feld mit dem/den angegebenen *Zeichen*.

Mit der Option **LENGTH** *n* *<Zeichen>* füllen Sie die ersten *n* Stellen des Feldes mit dem/den angegebenen *Zeichen*.

Beispiel für FULL LENGTH:

Hier wird das gesamte Feld mit Sternen gefüllt.

```
DEFINE DATA LOCAL
1 #FIELD (A25) INIT FULL LENGTH <'*'>
END-DEFINE
...
```

Beispiel für LENGTH *n*:

Hier werden die ersten 4 Stellen des Feldes mit Ausrufezeichen gefüllt.

```
DEFINE DATA LOCAL
1 #FIELD (A25) INIT LENGTH 4 <'!'>
END-DEFINE
...
```

Standard-Ausgangswerte

Wenn Sie für ein Feld keinen Ausgangswert angeben, wird das Feld je nach seinem Format mit einem Standard-Ausgangswert (Nullwert) initialisiert:

Format	Standard-Ausgangswert
B, F, I, N, P	0
A	Leerzeichen
L	F(ALSE)
D	D' '
T	T'00:00:00'
C	(AD=D)

RESET-Statement

Das RESET-Statement dient dazu, den Wert eines Feldes auf einen Nullwert oder einen bestimmten Ausgangswert zu setzen.

- RESET (ohne INITIAL) setzt den Wert jedes angegebenen Feldes auf einen Nullwert.
- RESET INITIAL setzt den Wert jedes angegebenen Feldes auf den für das Feld im DEFINE DATA-Statement definierten Ausgangswert.

Beispiel:

```

DEFINE DATA LOCAL
1 #FIELD A (N3) INIT <100>
1 #FIELD B (A20) INIT <'ABC'>
1 #FIELD C (I4) INIT <5>
END-DEFINE
...
...
RESET #FIELD A /* setzt Feldwert auf Null zurück
...
RESET INITIAL #FIELD A #FIELD B #FIELD C /* setzt Feldwerte auf
... /* Ausgangswerte zurück

```

Felder redefinieren

Redefinition dient dazu, das Format eines Feldes zu ändern oder ein einzelnes Feld in mehrere Teile aufzuteilen. Mit der REDEFINE-Option des DEFINE DATA-Statements kann ein einzelnes Feld — entweder eine Benutzervariable oder ein Datenbankfeld — als ein oder mehrere neue Feld/er redefiniert werden. Eine Gruppe kann ebenfalls redefiniert werden.

Wichtig:

Dynamische Variablen sind nicht zulässig.

Die REDEFINE-Option redefiniert die Byte-Positionen eines Feldes von links nach rechts, unabhängig vom Format. Die Byte-Positionen des ursprünglichen Feldes und des/der neudefinierten Feldes/Felder müssen einander entsprechen.

Eine Redefinition muß unmittelbar nach der Definition des ursprünglichen Feldes angegeben werden.

Im folgenden Beispiel wird das Datenbankfeld BIRTH als drei neue Benutzervariablen redefiniert:

```
DEFINE DATA LOCAL
01 EMPLOY-VIEW VIEW OF STAFFDDM
    02 NAME
    02 BIRTH
    02 REDEFINE BIRTH
        03 #BIRTH-YEAR (N4)
        03 #BIRTH-MONTH (N2)
        03 #BIRTH-DAY (N2)
END-DEFINE
...
```

Im folgenden Beispiel wird die Gruppe #VAR2, die aus zwei Benutzervariablen mit Format N bzw. P besteht, als eine neue Variable vom Format A redefiniert:

```
DEFINE DATA LOCAL
01 #VAR1 (A15)
01 #VAR2
    02 #VAR2A (N4.1)
    02 #VAR2B (P6.2)
01 REDEFINE #VAR2
    02 #VAR2RD (A10)
END-DEFINE
...
```

Mit der Notation **FILLER** *nX* können Sie in dem Feld, das redefiniert wird, *n* Füllbytes — d.h. Segmente, die nicht benutzt werden sollen — definieren. (Nachgestellte Füllbytes müssen nicht unbedingt angegeben werden.)

Im folgenden Beispiel wird die Benutzervariable #FIELD als drei neue Benutzervariablen, jede mit Format/Länge A2, redefiniert. Die FILLER-Notationen bedeuten, daß das 3. und 4. sowie das 7. bis 10. Byte des ursprünglichen Feldes nicht benutzt werden sollen.

```
DEFINE DATA LOCAL
1 #FIELD (A12)
1 REDEFINE #FIELD
    2 #RFIELD1 (A2)
    2 FILLER 2X
    2 #RFIELD2 (A2)
    2 FILLER 4X
    2 #RFIELD3 (A2)
END-DEFINE
...
```

Das folgende Programm veranschaulicht die Verwendung einer Redefinition:

```

** Example Program 'DDATAX01'
DEFINE DATA LOCAL
01 VIEWEMP VIEW OF EMPLOYEES
    02 NAME
    02 FIRST-NAME
    02 SALARY (1:1)
01 #PAY (N9)
01 REDEFINE #PAY
    02 FILLER 3X
    02 #USD (N3)
    02 #000 (N3)
END-DEFINE
READ (3) VIEWEMP BY NAME STARTING FROM 'JONES'
    MOVE SALARY (1) TO #PAY
    DISPLAY NAME FIRST-NAME #PAY #USD #000
END-READ
END

```

Beachten Sie, wie das Feld #PAY und die aus seiner Redefinition resultierenden Felder angezeigt werden:

Page		1	97-08-08 17:48:59		
NAME	FIRST-NAME	#PAY	#USD	#000	
JONES	VIRGINIA	46000	46	0	
JONES	MARSHA	50000	50	0	
JONES	ROBERT	31000	31	0	

Array-Verarbeitung

Natural unterstützt die Verarbeitung von sogenannten *Arrays*. Arrays sind mehrdimensionale Tabellen, d.h. zwei oder mehr logisch verwandte Elemente, die unter einem gemeinsamen Namen definiert werden.

Arrays können aus einzelnen Datenelementen mit mehreren Dimensionen bestehen oder aus hierarchischen Datenstrukturen, die sich wiederholende Strukturen oder individuelle Elemente aufweisen. Ein Natural-Array kann ein-, zwei- oder dreidimensional sein. Es kann eine unabhängige Variable, Teil einer größeren Datenstruktur oder Teil einer Datenbanksicht sein.

Die folgenden Themen werden nachfolgend erörtert:

- Arrays definieren
- Ausgangswerte für Arrays
- Ausgangswerte für eindimensionale Arrays zuweisen
- Ausgangswerte für zweidimensionale Arrays zuweisen
- Dreidimensionales Array
- Arrays als Teil einer größeren Datenstruktur
- Datenbank-Arrays
- Arithmetische Ausdrücke in Index-Notationen
- Arithmetische Funktionen bei Arrays.

Arrays definieren

Um eine Array-Variable zu definieren, geben Sie hinter Format und Länge einen Schrägstrich und danach eine sogenannte *Index-Notation*, d.h. die Anzahl der Ausprägungen des Arrays, an.

Wichtig:

Dynamische Variablen sind nicht zulässig.

Das folgende Array hat zum Beispiel drei Ausprägungen, wobei jede Ausprägung Format/Länge A10 hat:

```
DEFINE DATA LOCAL  
1 #ARRAY (A10/1:3)  
END-DEFINE  
...
```

Ein zweidimensionales Array definieren Sie, indem Sie für beide Dimensionen eine Index-Notation angeben:

```
DEFINE DATA LOCAL  
1 #ARRAY (A10/1:3,1:4)  
END-DEFINE  
...
```

Ein zweidimensionales Array kann man sich als Tabelle vorstellen. Das im obigen Beispiel definierte Array wäre demnach eine Tabelle, die aus 3 “Zeilen” und 4 “Spalten” besteht:

Ausgangswerte für Arrays

Um einer oder mehreren Ausprägungen eines Arrays einen Ausgangswert zuzuweisen, verwenden Sie, ähnlich wie für "einfache" Variablen (vgl. Seite 23), eine INIT-Angabe.

Ausgangswerte für eindimensionale Arrays zuweisen

Die folgenden Beispiele veranschaulichen, wie einem eindimensionalen Array Ausgangswerte zugewiesen werden:

Um einer einzelnen Ausprägung einen Ausgangswert zuzuweisen, geben Sie an:

```
1 #ARRAY (A1/1:3) INIT (2) <'A'>
```

"A" wird der zweiten Ausprägung zugewiesen.

Um allen Ausprägungen den gleichen Ausgangswert zuzuweisen, geben Sie an:

```
1 #ARRAY (A1/1:3) INIT ALL <'A'>
```

"A" wird jeder Ausprägung zugewiesen. Stattdessen könnten Sie auch angeben:

```
1 #ARRAY (A1/1:3) INIT (*) <'A'>
```

Um einem Bereich von mehreren Ausprägungen den gleichen Ausgangswert zuzuweisen, geben Sie an:

```
1 #ARRAY (A1/1:3) INIT (2:3) <'A'>
```

"A" wird der zweiten bis dritten Ausprägung zugewiesen.

Um jeder Ausprägung einen anderen Ausgangswert zuzuweisen, geben Sie an:

```
1 #ARRAY (A1/1:3) INIT <'A','B','C'>
```

"A" wird der ersten Ausprägung zugewiesen, "B" der zweiten und "C" der dritten.

Um verschiedenen (aber nicht allen) Ausprägungen verschiedene Ausgangswerte zuzuweisen, geben Sie an:

```
1 #ARRAY (A1/1:3) INIT (1) <'A'> (3) <'C'>
```

“A” wird der ersten Ausprägung zugewiesen und “C” der dritten; der zweiten Ausprägung wird kein Wert zugewiesen.

Stattdessen könnten Sie auch angeben:

```
1 #ARRAY (A1/1:3) INIT <'A',, 'C'>
```

Wenn weniger Ausgangswerte angegeben werden als Ausprägungen vorhanden sind, bleiben die letzten Ausprägungen leer:

```
1 #ARRAY (A1/1:3) INIT <'A', 'B'>
```

“A” wird der ersten Ausprägung zugewiesen und “B” der zweiten; der dritten Ausprägung wird kein Wert zugewiesen.

Ausgangswerte für zweidimensionale Arrays zuweisen

Die im folgenden aufgeführten Beispiele veranschaulichen, wie einem zweidimensionalen Array Ausgangswerte zugewiesen werden.

Für die Beispiele gehen wir von einem zweidimensionalen Array aus, das drei Ausprägungen in der ersten Dimension (“Zeilen”) und vier Ausprägungen in der zweiten Dimension (“Spalten”) hat:

```
1 #ARRAY (A1/1:3,1:4)
```

↓ Erste Dimension (1:3), Zweite Dimension (1:4) →

(1,1)	(1,2)	(1,3)	(1,4)
(2,1)	(2,2)	(2,3)	(2,4)
(3,1)	(3,2)	(3,3)	(3,4)

Die erste Gruppe von Beispielen veranschaulicht, wie den Ausprägungen eines zweidimensionalen Arrays der *gleiche* Ausgangswert zugewiesen wird; die zweite Gruppe von Beispielen veranschaulicht, wie *unterschiedliche* Ausgangswerte zugewiesen werden.

Beachten Sie bei den Beispielen insbesondere die Verwendung der Notationen “*” und “V”. Beide Notationen beziehen sich auf *alle* Ausprägungen der betreffenden Dimension: Mit “*” werden alle Ausprägungen der betreffenden Dimension mit dem *gleichen* Wert initialisiert, mit “V” werden alle Ausprägungen der betreffenden Dimension mit *unterschiedlichen* Werten initialisiert.

Die folgenden Themen werden nachfolgend erörtert:

- Gleichen Wert zuweisen
- Unterschiedliche Werte zuweisen.

Gleichen Wert zuweisen

Um einer Ausprägung einen Ausgangswert zuzuweisen, geben Sie an:

```
1 #ARRAY (A1/1:3) INIT (2) <'A'>
```

		A	

Um einer Ausprägung der zweiten Dimension — in allen Ausprägungen der ersten Dimension — den gleichen Ausgangswert zuzuweisen, geben Sie an:

```
1 #ARRAY (A1/1:3,1:4) INIT (*,3) <'A'>
```

		A	
		A	
		A	

Um einem Bereich von Ausprägungen der ersten Dimension — in allen Ausprägungen der zweiten Dimension — den gleichen Ausgangswert zuzuweisen, geben Sie an:

```
1 #ARRAY (A1/1:3,1:4) INIT (2:3,*) <'A'>
```

A	A	A	A
A	A	A	A

Um einem Bereich von Ausprägungen in beiden Dimensionen den gleichen Ausgangswert zuzuweisen, geben Sie an:

```
1 #ARRAY (A1/1:3,1:4) INIT (2:3,1:2) <'A'>
```

A	A		
A	A		

Um allen Ausprägungen (in beiden Dimensionen) den gleichen Ausgangswert zuzuweisen, geben Sie an:

```
1 #ARRAY (A1/1:3,1:4) INIT ALL <'A'>
```

A	A	A	A
A	A	A	A
A	A	A	A

Stattdessen könnten Sie auch angeben:

```
1 #ARRAY (A1/1:3,1:4) INIT (*,*) <'A'>
```

Unterschiedliche Werte zuweisen

```
1 #ARRAY (A1/1:3,1:4) INIT (V,2) <'A','B','C'>
```

	A		
	B		
	C		

```
1 #ARRAY (A1/1:3,1:4) INIT (V,2:3) <'A','B','C'>
```

	A	A	
	B	B	
	C	C	

```
1 #ARRAY (A1/1:3,1:4) INIT (V,*) <'A','B','C'>
```

A	A	A	A
B	B	B	B
C	C	C	C

```
1 #ARRAY (A1/1:3,1:4) INIT (V,*) <'A',,'C'>
```

A	A	A	A
C	C	C	C

```
1 #ARRAY (A1/1:3,1:4) INIT (V,*) <'A','B'>
```

A	A	A	A
B	B	B	B

```
1 #ARRAY (A1/1:3,1:4) INIT (V,1) <'A','B','C'>
    (V,3) <'D','E','F'>
```

A		D	
B		E	
C		F	

```
1 #ARRAY (A1/1:3,1:4) INIT (3,V) <'A','B','C','D'>
```

A	B	C	D

```
1 #ARRAY (A1/1:3,1:4) INIT (*,V) <'A','B','C','D'>
```

A	B	C	D
A	B	C	D
A	B	C	D

```
1 #ARRAY (A1/1:3,1:4) INIT (2,1) <'A'> (*,2) <'B'>
    (3,3) <'C'> (3,4) <'D'>
```

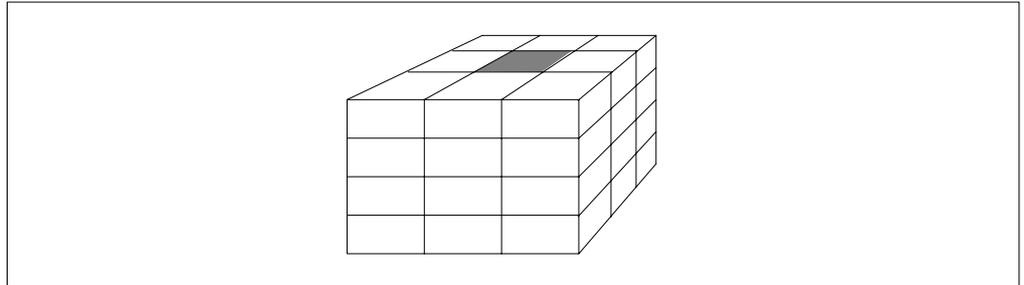
	B		
A	B		
	B	C	D

```
1 #ARRAY (A1/1:3,1:4) INIT (2,1) <'A'> (V,2) <'B','C','D'>
    (3,3) <'E'> (3,4) <'F'>
```

	B		
A	C		
	D	E	F

Dreidimensionales Array

Ein dreidimensionales Array könnte man sich folgendermaßen vorstellen:



Das oben dargestellte Array müßte wie folgt definiert werden (wobei gleichzeitig dem dunkel markierten Feld #FIELD2, das die Position Zeile 1, Spalte 2, Ebene 2 hat, ein Ausgangswert zugewiesen wird):

```

DEFINE DATA LOCAL
1 #ARRAY2
  2 #ROW (1:4)
    3 #COLUMN (1:3)
      4 #PLANE (1:3)
        5 #FIELD2 (P3) INIT (1,2,2) <100>
END-DEFINE
...

```

Wenn man das gleiche Array im Data-Area-Editor als Local Data Area definiert, sieht dies so aus:

I	T	L	Name	F	Leng	Index/Init/EM/Name/Comment
			1 #ARRAY2			
			2 #ROW			(1:4)
			3 #COLUMN			(1:3)
			4 #PLANE			(1:3)
I			5 #FIELD2	P	3	

Arrays als Teil einer größeren Datenstruktur

Die Mehrdimensionalität von Arrays ermöglicht es, Datenstrukturen analog zu COBOL- oder PL1-Strukturen zu definieren.

Beispiel:

```
DEFINE DATA LOCAL
1 #AREA
  2 #FIELD1 (A10)
  2 #GROUP1 (1:10)
    3 #FIELD2 (P2)
    3 #FIELD3 (N1/1:4)
END-DEFINE
...
```

Im obigen Beispiel hat der Datenbereich #AREA insgesamt eine Größe von:

$10 + (10 * (2 + (1 * 4)))$ Bytes = 70 Bytes.

#FIELD1 ist alphanumerisch und 10 Bytes lang. #GROUP1 ist ein Unterbereich von #AREA, kommt zehnmal vor und besteht aus zwei Feldern: #FIELD2 und #FIELD3. #FIELD2 ist gepackt numerisch und 2 Bytes lang; #FIELD3 ist das zweite Feld von #GROUP1, ist numerisch, 1 Byte lang und kommt viermal vor.

Wollen Sie eine bestimmte Ausprägung von #FIELD3 referenzieren, sind hierzu zwei Angaben erforderlich: erstens die der betreffenden Ausprägung von #GROUP1 und zweitens die der betreffenden Ausprägung von #FIELD3. Falls #FIELD3 beispielsweise an anderer Stelle im Programm in einem ADD-Statement referenziert würde, sähe dies folgendermaßen aus:

```
ADD 2 TO #FIELD3 (3,2)
```

Datenbank-Arrays

Adabas unterstützt Array-Strukturen innerhalb einer Datenbank in Form von *multiplen Feldern* und *Periodengruppen*. Diese sind im Kapitel **Datenbankzugriffe** (Seite 47) beschrieben.

Das folgende Beispiel zeigt einen DEFINE DATA-View, der ein multiples Feld enthält, zunächst programmintern und dann in einer programmexternen Local Data Area definiert:

```

DEFINE DATA LOCAL
  1 EMPLOYEES-VIEW VIEW OF EMPLOYEES
    2 NAME
    2 ADDRESS-LINE (1:10) /* ← MULTIPLES FELD
END-DEFINE
...

```

I	T	L	Name	F	Leng	Index/Init/EM/Name/Comment
V	1		EMPLOYEES-VIEW			EMPLOYEES
	2		NAME	A	20	
M	2		ADDRESS-LINE	A	20	(1:10) /* MU-FIELD

Arithmetische Ausdrücke in Index-Notationen

Zur Bestimmung eines Indexbereiches eines Arrays können auch einfache arithmetische Ausdrücke verwendet werden.

Beispiele:

MA (I:I+5) 6 Werte des Feldes "MA" werden referenziert, beginnend mit Wert I und endend mit Wert I + 5.

MA (I+2:J-3) Die Werte des Feldes "MA" von I + 2 bis J - 3 werden referenziert.

In derartigen Index-Angaben dürfen keine anderen Rechenzeichen als "+" und "-" verwendet werden.

Arithmetische Funktionen bei Arrays

Arithmetische Funktionen lassen sich innerhalb von Arrays auf Tabellenebene, auf Zeilen-/Spaltenebene und auf Feldebene einsetzen. Allerdings sind mit Array-Variablen nur einfache arithmetische Funktionen erlaubt, die höchstens ein oder zwei Operanden enthalten sowie möglicherweise eine dritte Variable als Ergebnisfeld. Werden Indexbereiche definiert, so sind nur die Operatoren "+" und "-" zulässig.

Beispiele für Array-Arithmetik:

Die folgenden Beispiele gehen von den folgenden Felddefinitionen aus:

```
DEFINE DATA LOCAL
01 #A (N5/1:10,1:10)
01 #B (N5/1:10,1:10)
01 #C (N5)
END-DEFINE
...
```

1. ADD #A(*,*) TO #B(*,*)

Die Werte des Arrays #B werden um die Werte des Arrays #A erhöht.

2. ADD 4 TO #A(*,2)

Die Werte der 2. Spalte des Arrays #A werden um 4 erhöht.

3. ADD 2 TO #A(2,*)

Die Werte der 2. Zeile des Arrays #A werden um 2 erhöht.

4. ADD #A(2,*) TO #B(4,*)

Die Werte der 2. Zeile des Arrays #A werden den Werten der 4. Zeile des Arrays #B hinzuaddiert.

5. ADD #A(2,*) TO #B(*,2)

Diese Operation ist nicht erlaubt und würde von Natural als Syntaxfehler zurückgewiesen. Es ist nicht gestattet, bei arithmetischen Funktionen Zeilen mit Spalten zu vermischen.

6. ADD #A(2,*) TO #C

Alle Werte der 2. Zeile des Arrays #A werden zu dem Skalarwert #C hinzuaddiert.

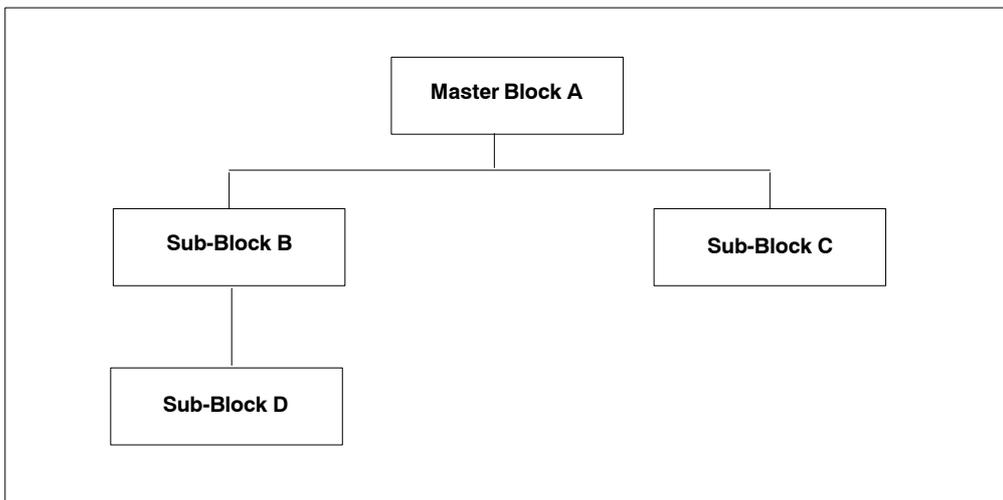
7. ADD #A(2,5:7) TO #C

Die Werte der 5. bis 7. Spalte der 2. Zeile des Arrays #A werden dem Skalarwert #C hinzuaddiert.

Datenblöcke

Um Speicherplatz zu sparen, können Sie eine Global Data Area mit Datenblöcken definieren: Datenblöcke können sich bei der Programmausführung überlagern und so Speicherplatz sparen.

Die folgende Grafik zeigt eine Struktur, in der die Blöcke B und C die gleiche Speicheradresse erhalten würden; daher wäre es nicht möglich, beide Blöcke gleichzeitig zu verwenden. So würde in obigem Beispiel eine Veränderung von Block B bewirken, daß der Inhalt von Block C zerstört wird.



Die folgenden Themen werden nachfolgend erörtert:

- Datenblöcke definieren
- Block-Hierarchien.

Datenblöcke definieren

Datenblöcke werden im Data-Area-Editor definiert. Sie legen die Hierarchie der Datenblöcke fest, indem Sie angeben, welcher Block welchem Block untergeordnet ist. Dies erreichen Sie, indem Sie den Namen des übergeordneten Blocks im Kommentarfeld der Blockdefinition angeben.

Im Beispiel unten sind SUB-BLOCKB und SUB-BLOCKC dem MASTER-BLOCKA untergeordnet: SUB-BLOCKD wiederum ist SUB-BLOCKB untergeordnet.

Die maximale Anzahl von Ebenen in der Blockhierarchie ist 8 (einschließlich des Master-Blocks).

Beispiel:

Global Data Area G-BLOCK:

I	T	L	Name	F	Leng	Index/Init/EM/Name/Comment
B			MASTER-BLOCKA			
	1		MB-DATA01	A	10	
B			SUB-BLOCKB			MASTER-BLOCKA
	1		SBB-DATA01	A	20	
B			SUB-BLOCKC			MASTER-BLOCKA
	1		SBC-DATA01	A	40	
B			SUB-BLOCKD			SUB-BLOCKB
	1		SBD-DATA01	A	40	

Welche Datenblöcke von welchem Programm verwendet werden, wird in den DEFINE DATA-Statements der betreffenden Programme angegeben, wie die folgenden Beispiele zeigen:

Programm 1:

```
DEFINE DATA GLOBAL
    USING G-BLOCK
    WITH MASTER-BLOCKA
END-DEFINE
```

Programm 2:

```
DEFINE DATA GLOBAL
    USING G-BLOCK
    WITH MASTER-BLOCKA.SUB-BLOCKB
END-DEFINE
```

Programm 3:

```
DEFINE DATA GLOBAL
    USING G-BLOCK
    WITH MASTER-BLOCKA.SUB-BLOCKC
END-DEFINE
```

Programm 4:

```
DEFINE DATA GLOBAL
    USING G-BLOCK
    WITH MASTER-BLOCKA.SUB-BLOCKB.SUB-BLOCKD
END-DEFINE
```

Bei der hier gezeigten Struktur kann Programm 1 sich die in MASTER-BLOCKA definierten Daten mit Programm 2, Programm 3 oder Programm 4 teilen. Es ist für die Programme 2 und 3 jedoch nicht möglich, sich die Daten aus SUB-BLOCKB und SUB-BLOCKC zu teilen, da beide Blöcke auf derselben Ebene definiert sind und daher den gleichen Speicherbereich belegen.

Block-Hierarchien

Bei der Verwendung von Datenblöcken ist folgender Fall zu beachten. Unten sind drei Programme aufgeführt, die eine Datenblock-Hierarchie benutzen:

Programm 1:

```
DEFINE DATA GLOBAL
    USING G-BLOCK
    WITH MASTER-BLOCKA.SUB-BLOCKB
END-DEFINE
*
MOVE 1234 TO SBB-DATA01
FETCH 'PROGRAM2'
END
```

Programm 2:

```
DEFINE DATA GLOBAL
    USING G-BLOCK
    WITH MASTER-BLOCKA
END-DEFINE
*
FETCH 'PROGRAM3'
END
```

Programm 3:

```
DEFINE DATA GLOBAL
    USING G-BLOCK
    WITH MASTER-BLOCKA.SUB-BLOCKB
END-DEFINE
*
WRITE SBB-DATA01
END
```

Programm 1 benutzt die Global Data Area G-BLOCK mit MASTER-BLOCKA und SUB-BLOCKB. Dieses Programm modifiziert ein Feld von SUB-BLOCKB. Programm 2 wird mit FETCH aufgerufen. Programm 2, das lediglich MASTER-BLOCKA in seiner Datendefinition enthält, löscht den Inhalt von SUB-BLOCKB. Der Grund hierfür liegt darin, daß ein Programm auf Stufe 1 (z.B. ein Programm, das mit FETCH aufgerufen wird) alle Datenblöcke, die auf einer niedrigeren Ebene in der Hierarchie liegen als die in seiner eigenen Datendefinition, zurücksetzt (d.h. deren Inhalt löscht). Nun wird Programm 3 mittels FETCH aufgerufen. Dieses Programm soll das in Programm 1 modifizierte Feld anzeigen; das WRITE-Statement liefert aber einen leeren Bildschirm. Weitere Informationen über Programm-Stufen finden Sie in dem Kapitel **Objekttypen** (Seite 173).

DATENBANKZUGRIFFE

Dieses Kapitel beschreibt verschiedene Aspekte des Zugriffs auf Daten in einer Datenbank mit Natural:

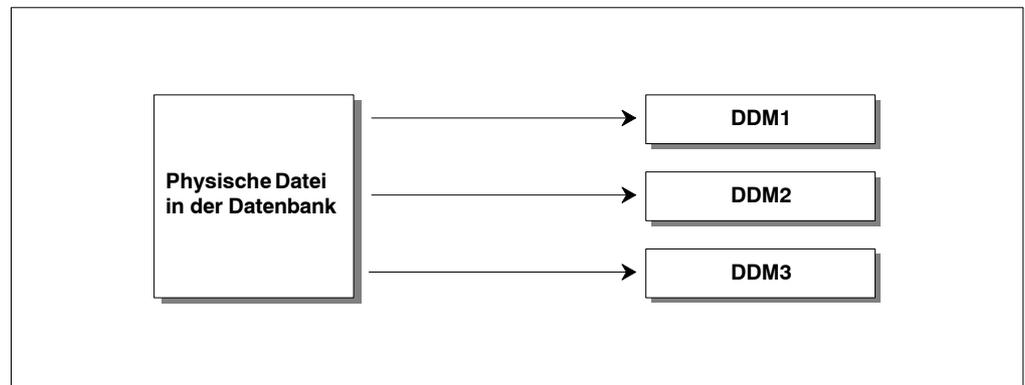
- DDMs (Datendefinitionsmodule)
- Datenbank-Arrays (multiple Felder und Periodengruppen)
- DEFINE DATA-Views
- Statements für Datenbankzugriffe
- READ-Statement
- FIND-Statement
- HISTOGRAM-Statement
- Datenbank-Verarbeitungsschleifen
- Datenbank-Änderungen und Transaktionslogik (die Statements STORE, DELETE, UPDATE, END TRANSACTION und BACKOUT TRANSACTION)
- die Statements ACCEPT und REJECT
- die Statements AT START OF DATA und AT END OF DATA.

DDMs (Datendefinitionsmodule)

Damit Natural auf eine Datenbank-Datei zugreifen kann, ist eine logische Definition der physischen Datei erforderlich. Eine solche logische Dateidefinition wird *DDM* (Datendefinitionsmodul) genannt.

Das DDM enthält Informationen über die einzelnen Felder der Datei — Informationen, die bei der Verwendung dieser Felder in einem Natural-Programm relevant sind. Ein DDM stellt eine logische Sicht (View) auf eine physische Datenbank-Datei dar.

Für jede physische Datei einer Datenbank können ein oder mehrere DDMs definiert werden.



DDMs werden vom Natural-Administrator mit Predict definiert (oder, falls Predict nicht verfügbar ist, mit der entsprechenden Natural-Funktion, wie im *Natural Benutzerhandbuch* beschrieben).

Ein DDM enthält die datenbank-internen Feldnamen der Datenbankfelder sowie ihre "externen" Feld-Langnamen (d.h. die in einem Natural-Programm verwendeten Feldnamen). Außerdem sind im DDM Format und Länge der Felder definiert, sowie weitere Angaben, die verwendet werden, wenn ein Feld in einem DISPLAY- oder WRITE-Statement benutzt wird (Spaltenüberschriften, Editiermasken usw.).

Die folgenden Themen werden nachfolgend erörtert:

- DDM anzeigen
- Bestandteile eines DDMs.

DDM anzeigen

Falls Sie den Namen des von Ihnen benötigten DDMs nicht wissen, können Sie sich mit dem Systemkommando **LIST DDM** eine Liste aller verfügbaren DDMs anzeigen lassen. Von der Liste können Sie dann ein DDM zur Anzeige auswählen.

Um ein DDM, dessen Namen Sie kennen, anzuzeigen, verwenden Sie das Systemkommando **LIST DDM** *ddm-name*.

Zum Beispiel:

LIST DDM EMPLOYEES

Sie erhalten dann eine Liste aller in dem DDM definierten Felder mit verschiedenen Angaben zu jedem Feld:

Bestandteile eines DDMs

Für jedes Feld enthält ein DDM folgende Angaben:

Spalte	Erklärung
T	Der <i>Typ</i> des Feldes: <i>kein Eintrag</i> — Elementarfeld. Dieser Feldtyp kann nur einen Wert pro Datensatz enthalten. M — Multiples Feld. Dieser Feldtyp kann mehr als einen Wert pro Datensatz enthalten. P — Periodengruppe. Dies ist eine Gruppe von Feldern, die mehr als eine Ausprägung pro Datensatz haben können. G — Gruppe. Dies ist eine Anzahl von Feldern, die unter einem gemeinsamen Gruppennamen definiert sind. Dadurch können mehrere Felder gleichzeitig durch Angabe des Gruppennamen statt aller einzelnen Feldnamen referenziert werden. * — Kommentarzeile.
L	Die <i>Level</i> -Nummer des Feldes. Levels werden bei der Strukturierung und Gruppierung von Felddefinitionen verwendet. Dies ist relevant bei View-Definitionen (siehe Seite 59), Redefinitionen (siehe Seite 10) und Feldgruppen (siehe Seite 9).
DB	Der zweistellige datenbank- <i>interne Feldname</i> .

Spalte	Erklärung
Name	<p>Der 3- bis 32-stellige <i>externe Feldname</i>. Dies ist der Name, der in einem Natural-Programm zur Referenzierung des Feldes verwendet wird.</p> <p>HD= bezeichnet eine Standard-Spaltenüberschrift, die erscheint, wenn Feldwerte über ein DISPLAY-Statement ausgegeben werden; falls keine Spaltenüberschrift angegeben ist, wird der Feld-Longname als Spaltenüberschrift verwendet.</p> <p>EM= bezeichnet eine Standard-Editiermaske, die verwendet wird, wenn Feldwerte über ein DISPLAY-Statement ausgegeben werden.</p>
F	Das <i>Format</i> des Feldes (A=alphanumerisch, N=numerisch ungepackt, P=gepackt numerisch, usw.).
Leng	Die <i>Länge</i> des Feldes. Bei numerischen Feldern ist die Länge in der Form " <i>nn.m</i> " angegeben, wobei " <i>nn</i> " für die Stellen vor dem Komma (Dezimalpunkt) und " <i>m</i> " für die Stellen nach dem Komma steht.
S	<p>(= Suppression) Die für das Feld gültige Art der Leerwertunterdrückung (Komprimierung):</p> <p>N — (<i>null-value suppression</i>) bedeutet, daß Nullwerte des Feldes nicht gelesen werden, wenn das Feld als Grundlage für eine Suchabfrage (z.B. in der WITH-Klausel eines FIND-Statements), in einem HISTOGRAM-Statement oder in einem READ LOGICAL-Statement benutzt wird.</p> <p>F — (<i>fixed storage</i>) bedeutet, daß das Feld mit einer fixen Länge gespeichert wird (d.h. es wird nicht komprimiert).</p> <p>Kein Eintrag bedeutet <i>normale Komprimierung</i>, d.h. nachfolgende Leerzeichen in einem alphanumerischen Feld und vorangestellte Nullen in einem numerischen Feld werden unterdrückt.</p>
D	<p>Der <i>Deskriptor</i>-Typ des Feldes; zum Beispiel:</p> <p>D — normaler Deskriptor, N — kein Deskriptor, P — phonetischer Deskriptor, U — Subdeskriptor, S — Superdeskriptor.</p> <p>Keine Angabe in dieser Spalte bedeutet, daß das Feld kein Deskriptor ist.</p> <p>Ein Deskriptor kann als Grundlage für einen Datenbankzugriff verwendet werden. Ein mit "D" oder "S" markiertes Feld kann in der BY-Klausel eines READ-Statements verwendet werden. Sobald ein Datensatz mit einem READ-Statement von der Datenbank gelesen worden ist, kann ein DISPLAY-Statement alle in der "D"-Spalte mit "D" oder gar nicht markierten Felder referenzieren.</p>
Remarks	Diese Spalte kann <i>Kommentare/Anmerkungen</i> zu dem Feld enthalten.

Oberhalb der Liste von Feldern wird folgendes angezeigt: die Nummer der Datei, von der das DDM erstellt wurde (DDM FNR), die Nummer der Datenbank, in der diese Datei gespeichert ist (DDM DBID), sowie gegebenenfalls das "Default Sequence"-Feld, d.h. der Name des Feldes, das das logisch sequentielle Lesen der Datei steuert, falls im READ LOGICAL-Statement eines Programms kein entsprechendes Feld angegeben wird.

Datenbank-Arrays

Adabas unterstützt Array-Strukturen innerhalb der Datenbank in Form von *multiplen Feldern* und *Periodengruppen*.

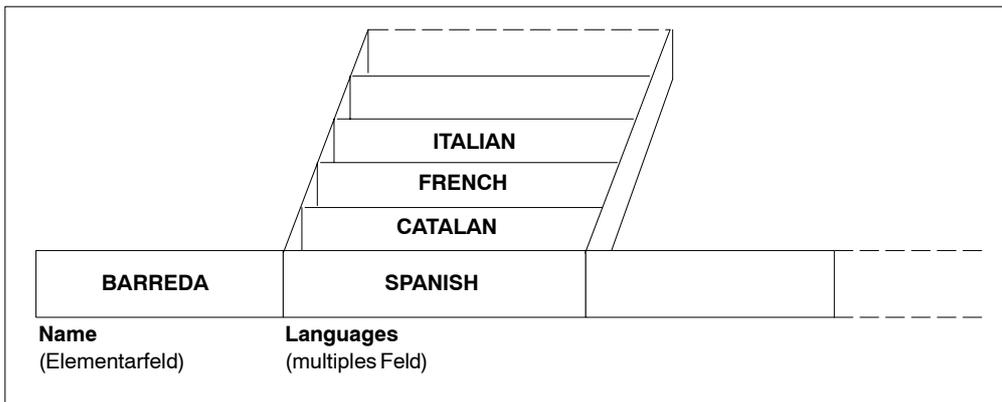
Die folgenden Themen werden nachfolgend erörtert:

- Multiple Felder
- Periodengruppen
- Multiple Felder und Periodengruppen referenzieren
- Multiple Felder innerhalb von Periodengruppen
- Multiple Felder innerhalb von Periodengruppen referenzieren
- Internen Zähler eines Datenbank-Arrays referenzieren.

Multiple Felder

Ein multiples Feld ist ein Feld, das innerhalb eines Datensatzes mehr als einen Wert (bis zu 191) haben kann.

Beispiel:



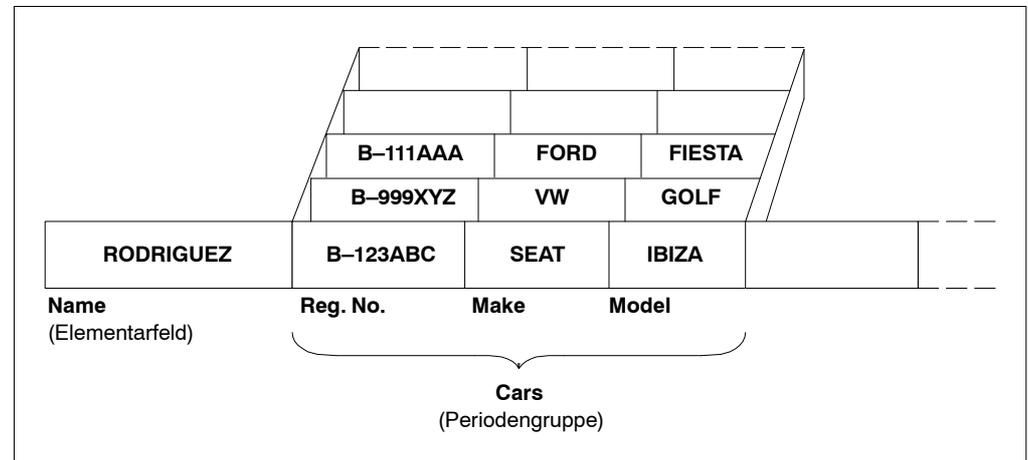
Angenommen, obige Abbildung zeigt einen Datensatz aus einer Personendatei: das erste Feld (Name) ist ein Elementarfeld, das nur einen Wert enthalten kann, nämlich den Namen der Person; das zweite Feld (Languages) enthält die Sprachen, die die Person spricht, und ist ein multiples Feld, da eine Person mehrere Sprachen sprechen kann.

Periodengruppen

Eine Periodengruppe ist eine Gruppe von Feldern (wobei es sich um Elementarfelder und/oder multiple Felder handeln kann), die innerhalb eines Datensatzes mehr als eine Ausprägung (bis zu 191) haben kann.

Bei multiplen Feldern werden die verschiedenen Werte eines Feldes auch als *Ausprägungen* bezeichnet, d.h. mit der Anzahl der Ausprägungen ist die Anzahl der Werte, die das Feld enthält, gemeint, und eine bestimmte Ausprägung bezeichnet einen bestimmten Wert. Analog dazu ist bei einer Periodengruppe mit Ausprägung eine Gruppe von Werten gemeint.

Beispiel:



Angenommen, obige Abbildung zeigt einen Datensatz aus einer Fahrzeugdatei: das erste Feld (Name) ist ein Elementarfeld, das den Namen einer Person enthält; Cars ist eine Periodengruppe, die die Fahrzeuge dieser Person enthält. Die Periodengruppe besteht aus drei Feldern, die für jedes Fahrzeug das KFZ-Kennzeichen (Reg. No.), die Marke (Make) und das Modell (Model) enthalten. Jede Ausprägung von Cars enthält jeweils die Werte für ein Fahrzeug.

Multiple Felder und Periodengruppen referenzieren

Um eine oder mehrere Ausprägungen eines multiplen Feldes oder einer Periodengruppe zu referenzieren, geben Sie hinter dem Feldnamen eine *Index-Notation* an.

Beispiele:

Die folgenden Beispiele verwenden das multiple Feld LANGUAGES und die Periodengruppe CARS aus den Abbildungen auf den vorigen Seiten.

Die verschiedenen Werte des multiplen Feldes LANGUAGES können wie folgt referenziert werden:

LANGUAGES (1)	Referenziert den ersten Wert ("SPANISH").
LANGUAGES (X)	Der Inhalt der Variablen X bestimmt den zu referenzierenden Wert.
LANGUAGES (1:3)	Referenziert die ersten drei Werte ("SPANISH", "CATALAN" und "FRENCH").
LANGUAGES (6:10)	Referenziert den sechsten bis zehnten Wert.
LANGUAGES (X:Y)	Die Inhalte der Variablen X und Y bestimmen die zu referenzierenden Werte.

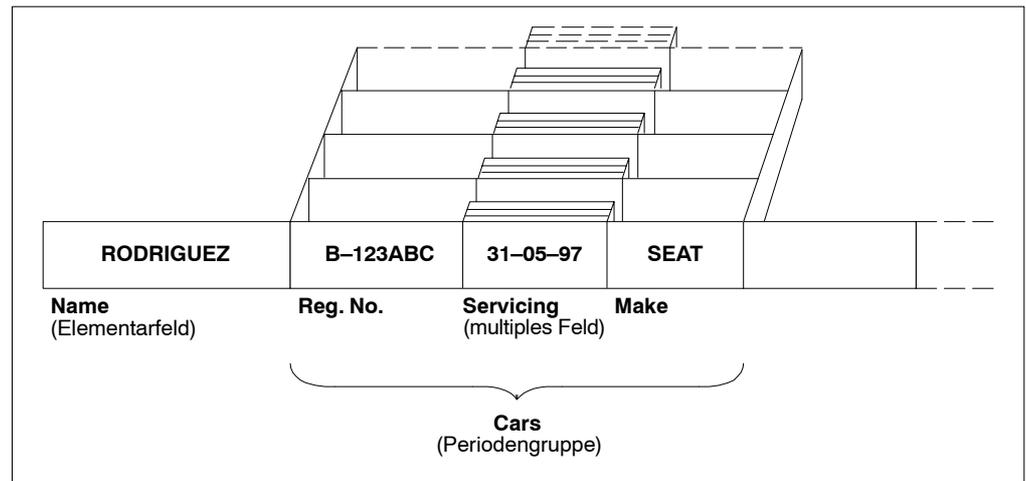
Die verschiedenen Ausprägungen der Periodengruppe CARS können in der gleichen Weise referenziert werden:

CARS (1)	Referenziert die erste Ausprägung ("B-123ABC/SEAT/IBIZA").
CARS (X)	Der Inhalt der Variablen X bestimmt die zu referenzierende Ausprägung.
CARS (1:2)	Referenziert die ersten beiden Ausprägungen ("B-123ABC/SEAT/IBIZA" und "B-999XYZ/VW/GOLF").
CARS (4:7)	Referenziert die vierte bis siebte Ausprägung.
CARS (X:Y)	Die Inhalte der Variablen X und Y bestimmen die zu referenzierenden Ausprägungen.

Multiple Felder innerhalb von Periodengruppen

Ein Adabas-Array kann bis zu zwei Dimensionen haben: ein multiples Feld innerhalb einer Periodengruppe.

Beispiel:



Angenommen, obige Abbildung zeigt einen Datensatz aus einer Fahrzeugdatei: das erste Feld (Name) ist ein Elementarfeld, das den Namen einer Person enthält; "Cars" ist eine Periodengruppe, die die Fahrzeuge dieser Person enthält. Die Periodengruppe besteht aus drei Feldern, die für jedes Fahrzeug das KFZ-Kennzeichen (Reg. No.), die Inspektionstermine (Servicing) und die Marke (Make) enthalten. Innerhalb der Periodengruppe "Cars" ist "Servicing" ein multiples Feld, das die verschiedenen Inspektionstermine jedes Autos enthält.

Multiple Felder innerhalb von Periodengruppen referenzieren

Um eine oder mehrere Ausprägungen eines multiplen Feldes innerhalb einer Periodengruppe zu referenzieren, geben Sie eine "zweidimensionale" Index-Notation hinter dem Feldnamen an.

Beispiele:

Die folgenden Beispiele verwenden das multiple Feld **SERVICING** und die Periodengruppe **CARS** aus der obigen Abbildung. Die verschiedenen Werte des multiplen Feldes können wie folgt referenziert werden:

SERVICING (1,1)	Referenziert den ersten Wert von SERVICING in der ersten Ausprägung von CARS ("31-05-97")
SERVICING (1:5,1)	Referenziert jeweils den ersten Wert von SERVICING in den ersten fünf Ausprägungen von CARS .
SERVICING (1:5,1:10)	Referenziert jeweils die ersten zehn Werte von SERVICING in den ersten fünf Ausprägungen von CARS .

Internen Zähler eines Datenbank-Arrays referenzieren

Es ist manchmal erforderlich, ein multiples Feld oder eine Periodengruppe zu referenzieren, ohne die Anzahl der Werte bzw. Ausprägungen eines Datensatzes zu kennen. Adabas zählt intern die Anzahl der Werte eines multiplen Feldes und die Anzahl der Ausprägungen einer Periodengruppe. Dieser interne Zähler kann mit einem READ-Statement abgelesen werden, indem man unmittelbar vor dem Feldnamen "C*" angibt:

Die Anzahl wird jeweils in Format/Länge N3 zurückgegeben. Siehe *Natural-Referenzhandbuch* für weitere Informationen.

Beispiele:

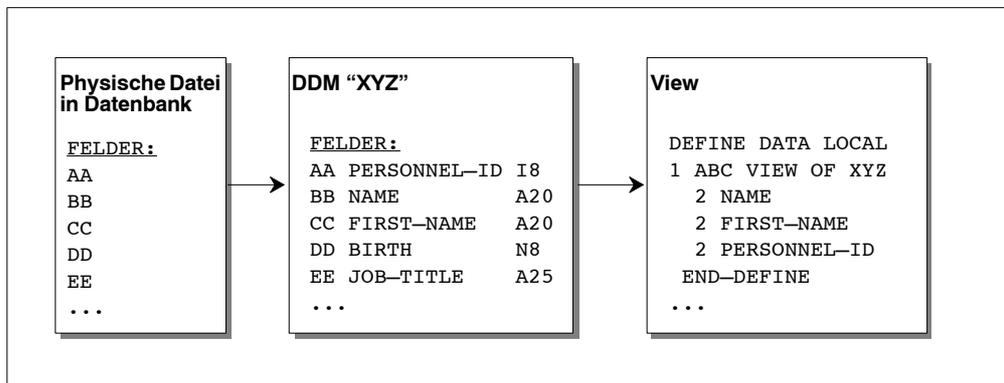
C*LANGUAGES	Liefert die Anzahl der Werte des multiplen Feldes LANGUAGES.
C*CARS	Liefert die Anzahl der Ausprägungen der Periodengruppe CARS.
C*SERVICING(1)	Liefert die Anzahl der Werte des multiplen Feldes SERVICING in der ersten Ausprägung einer Periodengruppe (ausgehend von der Annahme, daß SERVICING ein multiples Feld innerhalb einer Periodengruppe ist).

DEFINE DATA-Views

Um Datenbankfelder in einem Natural-Programm verwenden zu können, müssen Sie sie in einem sogenannten *View* angeben.

In dem View geben Sie den Namen des DDMs an, aus dem die Felder stammen, sowie die Namen der Datenbankfelder selbst (d.h. ihre Langnamen, nicht ihre datenbank-internen Kurznamen).

Sie definieren einen solchen Datenbank-View entweder im DEFINE DATA-Statement des Programms selbst oder in einer Local oder Global Data Area außerhalb des Programms, wobei das DEFINE DATA-Statement dann diese Data Area referenziert (wie im Kapitel **Felder definieren** beschrieben, vgl. Seite 5).



Auf Level 1 geben Sie den View-Namen wie folgt an:

1 *view-name* **VIEW OF** *ddm-name*

wobei *view-name* der von Ihnen gewählte Name für den View ist und *ddm-name* der Name des DDMs, aus dem die im View angegebenen Felder stammen. Darunter, auf Level 2, geben Sie die Namen der Datenbankfelder aus dem DDM an.

In der obigen Abbildung hat der View den Namen "ABC" und er umfaßt die Felder NAME, FIRST-NAME und PERSONNEL-ID aus dem DDM "XYZ".

Format und Länge eines Datenbankfeldes brauchen im View nicht angegeben zu werden, da sie bereits im zugrundeliegenden DDM definiert sind.

Ein View kann ein komplettes DDM umfassen oder einen Ausschnitt daraus. Die Reihenfolge der Felder im View braucht nicht mit der Reihenfolge der Felder im zugrundeliegenden DDM übereinzustimmen.

Wie später in diesem Kapitel noch gezeigt wird, wird der View-Name in Datenbankzugriffs-Statements verwendet, um zu bestimmen, auf welche Datenbank zugegriffen werden soll.

Statements für Datenbankzugriffe

Um Daten von einer Datenbank zu lesen, stehen folgende Statements zur Verfügung:

- **READ** — Mit diesem Statement können Sie eine Reihe von Datensätzen in einer bestimmten Reihenfolge von der Datenbank lesen.
- **FIND** — Mit diesem Statement können Sie von einer Datenbank diejenigen Datensätze lesen, die ein bestimmtes Suchkriterium erfüllen.
- **HISTOGRAM** — Mit diesem Statement können Sie nur die Werte eines einzelnen Datenbankfeldes lesen oder herausfinden, wieviele Datensätze ein bestimmtes Suchkriterium erfüllen.

READ-Statement

Das READ-Statement dient dazu, Datensätze von einer Datenbank zu lesen. Die Datensätze können von der Datenbank gelesen werden:

- in der Reihenfolge, in der sie physisch auf der Datenbank gespeichert sind (READ IN PHYSICAL SEQUENCE) oder
- in der Reihenfolge der Adabas-internen Satznummern (READ BY ISN) oder
- in logischer Reihenfolge der Werte eines Deskriptorfeldes (READ IN LOGICAL SEQUENCE).

In diesem Handbuch wird lediglich READ IN LOGICAL SEQUENCE behandelt, da dies die am häufigsten verwendete Form des READ-Statements ist; Informationen zu den anderen beiden Möglichkeiten finden Sie unter der Beschreibung des READ-Statements im *Natural Statements-Handbuch*.

Die folgenden Themen werden nachfolgend behandelt:

- Syntax
- Anzahl der zu lesenden Datensätze begrenzen
- STARTING- und ENDING-Klauseln
- WHERE-Klausel.

Syntax

Die Grundform des READ-Statements ist:

READ *view* **IN LOGICAL SEQUENCE BY** *descriptor*

oder kürzer:

READ *view* **LOGICAL BY** *descriptor*

view ist der Name eines im DEFINE DATA-Statement definierten Views (wie weiter oben in diesem Kapitel beschrieben).

descriptor ist der Name eines in diesem View definierten Datenbankfeldes. Die Werte dieses Feldes bestimmen die Reihenfolge, in der die Datensätze von der Datenbank gelesen werden.

Wenn Sie einen Deskriptor angeben, erübrigt sich die Angabe des Wortes "LOGICAL":

READ *view* **BY** *descriptor*

Wenn Sie keinen Deskriptor angeben, werden die Datensätze in der Reihenfolge der Werte des im DDM als Standard-Deskriptor (unter "Default Sequence") definierten Feldes gelesen. Wenn Sie keinen Deskriptor angeben, müssen Sie allerdings das Schlüsselwort "LOGICAL" angeben:

READ *view* **LOGICAL**

Beispiel:

```

** Example Program 'READX01'
DEFINE DATA LOCAL
1 MYVIEW VIEW OF EMPLOYEES
  2 PERSONNEL-ID
  2 NAME
  2 JOB-TITLE
END-DEFINE
READ (6) MYVIEW BY NAME
  DISPLAY NAME PERSONNEL-ID JOB-TITLE
END-READ
END

```

Das READ-Statement im obigen Beispiel liest Datensätze von der Mitarbeiter-Datei EMPLOYEES in alphabetischer Reihenfolge der (im Feld NAME enthaltenen) Nachnamen.

Das obige Programm erzeugt folgende Ausgabe, wobei die Informationen zu jedem Mitarbeiter in alphabetischer Reihenfolge der Nachnamen angezeigt werden:

Page	1	97-08-19 13:16:04	
NAME	PERSONNEL ID	CURRENT POSITION	
ABELLAN	60008339	MAQUINISTA	
ACHIESON	30000231	DATA BASE ADMINISTRATOR	
ADAM	50005800	CHEF DE SERVICE	
ADKINSON	20008800	PROGRAMMER	
ADKINSON	20009800	DBA	
ADKINSON	20011000	SALES PERSON	

Falls Sie die Mitarbeiterdaten in der Reihenfolge der (im Feld BIRTH enthaltenen) Geburtsdaten lesen und ausgeben möchten, wäre dazu folgendes READ-Statement geeignet:

```

READ MYVIEW BY BIRTH

```

Sie können nur ein Feld angeben, das im zugrundeliegenden DDM als “Deskriptor” definiert ist (es kann auch ein Subdeskriptor, Superdeskriptor oder Hyperdeskriptor sein).

Anzahl der zu lesenden Datensätze begrenzen

Wie im Beispielprogramm auf der vorigen Seite gezeigt, können Sie die Anzahl der Datensätze, die gelesen werden sollen, begrenzen, indem Sie hinter dem Schlüsselwort READ in Klammern eine Zahl angeben:

```
READ (6) MYVIEW BY NAME
```

In diesem Beispiel würde das READ-Statement maximal 6 Datensätze lesen.

Ohne diese Limit-Notation würde das obige READ-Statement *sämtliche* Datensätze von der EMPLOYEES-Datei in der Reihenfolge der Nachnamen von A bis Z lesen.

STARTING- und ENDING-Klausel bei READ-Statement

Mit dem READ-Statement können Sie das Suchkriterium für die zu lesenden Datensätze durch einen bestimmten *Wert* eines Deskriptorfeldes weiter einschränken. Mit der Option EQUAL TO/STARTING FROM in einer BY- bzw. WITH-Klausel können Sie festlegen, *ab welchem Wert* die Datensätze gelesen werden sollen. Mit der Option THRU/ENDING AT können Sie darüber hinaus bestimmen, *bis zu welchem Wert* gelesen werden soll.

Wünschen Sie beispielsweise eine Liste aller Mitarbeiter in der Reihenfolge der Tätigkeitsbezeichnungen (JOB-TITLE) von "TRAINEE" bis "Z", würden Sie eines der folgenden Statements verwenden:

```
READ MYVIEW WITH JOB-TITLE = 'TRAINEE'  
  
READ MYVIEW WITH JOB-TITLE STARTING FROM 'TRAINEE'  
  
READ MYVIEW BY JOB-TITLE = 'TRAINEE'  
  
READ MYVIEW BY JOB-TITLE STARTING FROM 'TRAINEE'
```

Bitte beachten Sie, daß der Wert hinter dem Gleichheitszeichen (=) bzw. der STARTING FROM-Option in Apostrophen stehen muß. Bei einem numerischen Wert ist diese *Text-Notation* nicht erforderlich.

Es ist nicht möglich, die Optionen BY und WITH gleichzeitig zu verwenden; es ist jeweils nur eine von beiden gestattet.

Durch Angabe einer THRU- bzw. ENDING AT-Klausel können Sie darüber hinaus festlegen, bis zu welchem Punkt Datensätze gelesen werden sollen.

Um nur Datensätze mit der Tätigkeitsbezeichnung "TRAINEE" zu lesen, müßten Sie folgendes angeben:

```
READ MYVIEW BY JOB-TITLE STARTING FROM 'TRAINEE' THRU 'TRAINEE'
```

```
READ MYVIEW WITH JOB-TITLE EQUAL TO 'TRAINEE'  
ENDING AT 'TRAINEE'
```

Um alle Datensätze mit Tätigkeitsbezeichnungen, die mit "A" oder "B" anfangen, zu lesen, müßten Sie folgendes angeben:

```
READ MYVIEW BY JOB-TITLE = 'A' THRU 'C'
```

```
READ MYVIEW WITH JOB-TITLE STARTING FROM 'A' ENDING AT 'C'
```

Die Werte werden gelesen bis einschließlich des Wertes, der nach THRU/ENDING AT spezifiziert wird. In den beiden obigen Beispielen werden alle Datensätze mit Tätigkeitsbezeichnungen, die mit "A" oder "B" anfangen, gelesen; gäbe es eine Tätigkeitsbezeichnung "C", würde diese auch gelesen werden, aber nicht der nächsthöhere Wert "CA".

WHERE-Klausel beim READ-Statement

Mit einer WHERE-Klausel können Sie ein zusätzliches Suchkriterium angeben.

Zum Beispiel, wenn Sie nur die Datensätze derjenigen Mitarbeiter mit Tätigkeitsbezeichnung "TRAINEE", die in US-Dollar bezahlt werden, lesen wollen, würden Sie folgendes angeben:

```
READ MYVIEW WITH JOB-TITLE = 'TRAINEE'
                WHERE CURR-CODE = 'USD'
```

Die WHERE-Klausel kann auch zusammen mit einer BY-Klausel verwendet werden, zum Beispiel:

```
READ MYVIEW BY NAME
                WHERE SALARY = 20000
```

Die WHERE-Klausel unterscheidet sich in zwei Punkten von einer BY- bzw. WITH-Klausel:

- Das in der WHERE-Klausel angegebene Feld muß kein Deskriptor sein.
- In der WHERE-Klausel wird eine logische Bedingung angegeben. Folgende logische Operatoren können dabei verwendet werden:

EQUAL TO	EQ	=
NOT EQUAL TO	NE	≠
LESS THAN	LT	<
LESS THAN OR EQUAL TO	LE	<=
GREATER THAN	GT	>
GREATER THAN OR EQUAL TO	GE	>=

Das folgende Programm veranschaulicht die Verwendung der Klauseln STARTING FROM, ENDING AT und WHERE:

```

** Example Program 'READX02'
DEFINE DATA LOCAL
1 MYEMP VIEW OF EMPLOYEES
  2 NAME
  2 JOB-TITLE
  2 INCOME (1:2)
  3 CURR-CODE
  3 SALARY
  3 BONUS (1:1)
END-DEFINE
*
READ (3) MYVIEW WITH JOB-TITLE = 'TRAINEE' THRU 'TRAINEE'
      WHERE CURR-CODE (*) = 'USD'
      DISPLAY NOTITLE NAME / JOB-TITLE 5X INCOME (1:2)
      SKIP 1
END-READ
END

```

Es erzeugt folgende Ausgabe:

NAME CURRENT POSITION	INCOME		
	CURRENCY CODE	ANNUAL SALARY	BONUS
SENKO	USD	23000	0
TRAINEE	USD	21800	0
BANGART	USD	25000	0
TRAINEE	USD	23000	0
LINCOLN	USD	24000	0
TRAINEE	USD	22000	0

Weiteres Beispiel für READ-Statement:

Siehe Programm READX03 in Library SYSEXPB.

FIND-Statement

Das FIND-Statement dient dazu, Datensätze von einer Datenbank zu lesen, die ein bestimmtes Suchkriterium erfüllen.

Die folgenden Themen werden nachfolgend behandelt:

- Syntax
- Anzahl der zu verarbeitenden Datensätze begrenzen
- WHERE-Klausel
- IF NO RECORDS FOUND-Bedingung.

Syntax

Die Grundform des FIND-Statements ist:

FIND RECORDS IN *view* **WITH** *field = value*

oder kürzer:

FIND *view* **WITH** *field = value*

view ist der Name eines im DEFINE DATA-Statement definierten Views (wie weiter oben in diesem Kapitel beschrieben).

field ist der Name eines in diesem View definierten Datenbankfeldes. Sie können nur ein *field* angeben, das im zugrundeliegenden DDM als “Deskriptor” definiert ist (es kann auch ein Subdeskriptor, Superdeskriptor, Hyperdeskriptor oder phonetischer Deskriptor sein).

Anzahl der zu verarbeitenden Datensätze begrenzen

Ähnlich wie beim READ-Statement können Sie die Anzahl der Datensätze, die verarbeitet werden sollen, begrenzen, indem Sie hinter dem Schlüsselwort FIND in Klammern eine Zahl angeben:

```
FIND (6) RECORDS IN MYVIEW WITH NAME = 'CLEGG'
```

In diesem Beispiel würde das FIND-Statement maximal 6 Datensätze verarbeiten.

Ohne diese Limit-Notation würden alle Datensätze, die das Suchkriterium erfüllen, verarbeitet werden.

Anmerkung:

*Wenn das FIND-Statement eine WHERE-Klausel enthält (siehe unten), werden Datensätze, die die WHERE-Klausel nicht erfüllen, bei der Ermittlung des Limits **nicht** berücksichtigt.*

WHERE-Klausel beim FIND-Statement

Mit der WHERE-Klausel des FIND-Statements können Sie ein zusätzliches Selektionskriterium angeben, das ausgewertet wird, *nachdem* ein (über die WITH-Klausel ausgewählter) Datensatz gelesen wurde und *bevor* der ausgewählte Datensatz weiterverarbeitet wird.

Beispiel für WHERE-Klausel:

```

** Example Program 'FINDX01'
DEFINE DATA LOCAL
1 MYVIEW VIEW OF EMPLOYEES
  2 PERSONNEL-ID
  2 NAME
  2 JOB-TITLE
  2 CITY
END-DEFINE
*
FIND MYVIEW WITH CITY = 'PARIS'
      WHERE JOB-TITLE = 'INGENIEUR COMMERCIAL'
      DISPLAY NOTITLE CITY JOB-TITLE PERSONNEL-ID NAME
END-FIND
END

```

Wie Sie sehen, werden in diesem Beispiel nur die Datensätze, die die Kriterien der WITH-Klausel *und* der WHERE-Klausel erfüllen, im DISPLAY-Statement verarbeitet.

CITY	CURRENT POSITION	PERSONNEL ID	NAME
PARIS	INGENIEUR COMMERCIAL	50007300	CAHN
PARIS	INGENIEUR COMMERCIAL	50006500	MAZUY
PARIS	INGENIEUR COMMERCIAL	50004400	VALLY
PARIS	INGENIEUR COMMERCIAL	50002800	BRETON
PARIS	INGENIEUR COMMERCIAL	50001000	GIGLEUX

IF NO RECORDS FOUND-Bedingung

Falls keine Datensätze gefunden werden, die die in der WITH- und WHERE-Klausel angegebenen Suchkriterien erfüllen, werden die innerhalb der FIND-Verarbeitungsschleife angegebenen Statements nicht ausgeführt (für das Beispiel auf der vorigen Seite hieße dies, daß das DISPLAY-Statement nicht ausgeführt würde und folglich keine Mitarbeiterdaten angezeigt würden).

Das FIND-Statement bietet jedoch auch eine IF NO RECORDS FOUND-Klausel, in der Sie eine Verarbeitung angeben können, die ausgeführt werden soll für den Fall, daß kein Datensatz die Suchkriterien erfüllt.

Beispiel für IF NO RECORDS FOUND-Klausel:

```
** Example Program 'FINDX02'
DEFINE DATA LOCAL
1 MYVIEW VIEW OF EMPLOYEES
  2 NAME
  2 FIRST-NAME
END-DEFINE
*
FIND MYVIEW WITH NAME = 'BLACKMORE'
  IF NO RECORDS FOUND
    WRITE 'NO PERSON FOUND.'
  END-NOREC
  DISPLAY NAME FIRST-NAME
END-FIND
END
```

Das obige Programm wählt alle Datensätze aus, in denen das Feld NAME den Wert "BLACKMORE" enthält. Von jedem ausgewählten Datensatz werden der Name (NAME) und der Vorname (FIRST-NAME) angezeigt. Falls in der Datei kein Datensatz mit NAME = 'BLACKMORE' gefunden wird, wird das in der IF NO RECORDS FOUND-Klausel angegebene WRITE-Statement ausgeführt:

Page	1	97-08-19	11:44:00
	NAME	FIRST-NAME	
NO PERSON FOUND.			

Weitere Beispiele für FIND-Statement:

Siehe Programme FINDX07, FINDX08, FINDX09, FINDX10 und FINDX11 in Library SYSEXPB.

HISTOGRAM-Statement

Das HISTOGRAM-Statement dient dazu, entweder die Werte eines einzelnen Datenbankfeldes zu lesen oder herauszufinden, wieviele Datensätze ein bestimmtes Suchkriterium erfüllen.

Das HISTOGRAM-Statement kann auf keine anderen Datenbankfelder zugreifen als auf das im HISTOGRAM-Statement angegebene Feld.

Die folgenden Themen werden nachfolgend behandelt:

- Syntax
- Anzahl der zu lesenden Werte begrenzen
- STARTING- und ENDING-Klausel
- WHERE-Klausel.

Syntax

Die Grundform des HISTOGRAM-Statements ist:

HISTOGRAM VALUE IN *view* **FOR** *field*

oder kürzer:

HISTOGRAM *view field*

view ist der Name eines im DEFINE DATA-Statement definierten Views (wie weiter oben in diesem Kapitel beschrieben). *field* ist der Name des in diesem View definierten Datenbankfeldes.

Anzahl der zu lesenden Werte begrenzen

Ähnlich wie beim READ-Statement können Sie die Anzahl der Werte, die gelesen werden sollen, begrenzen, indem Sie hinter dem Schlüsselwort HISTOGRAM in Klammern eine Zahl angeben:

```
HISTOGRAM (6) MYVIEW FOR NAME
```

In diesem Beispiel würden nur die ersten 6 Werte des Feldes NAME gelesen.

Ohne diese Limit-Notation würden alle Werte gelesen.

STARTING- und ENDING-Klausel bei HISTOGRAM-STATEMENT

Wie das READ-Statement bietet auch das HISTOGRAM-Statement eine STARTING FROM-Klausel und eine ENDING AT- (bzw. THRU-)Klausel, mit denen Sie den Bereich der zu lesenden Werte durch Angabe eines Startwertes und eines Endwertes eingrenzen können.

Beispiele:

```
HISTOGRAM MYVIEW FOR NAME STARTING FROM 'BOUCHARD'
```

```
HISTOGRAM MYVIEW FOR NAME STARTING FROM 'BOUCHARD' ENDING AT 'LANIER'
```

```
HISTOGRAM MYVIEW FOR NAME FROM 'BLOOM' THRU 'ROESER'
```

WHERE-Klausel beim HISTOGRAM-Statement

Das HISTOGRAM-Statement bietet außerdem eine WHERE-Klausel, in der Sie ein zusätzliches Selektionskriterium angeben können, das ausgewertet wird, *nachdem* ein Wert gelesen wurde und *bevor* der Wert weiterverarbeitet wird. Das in der WHERE-Klausel angegebene Feld muß dasselbe sein wie das in der Hauptklausel des HISTOGRAM-Statements angegebene.

Beispiel für HISTOGRAM-Statement:

```

** Example Program 'HISTOX01'
DEFINE DATA LOCAL
1 MYVIEW VIEW OF EMPLOYEES
  2 CITY
END-DEFINE
*
LIMIT 8
HISTOGRAM MYVIEW CITY STARTING FROM 'M'
  DISPLAY NOTITLE CITY 'NUMBER OF/PERSONS' *NUMBER *COUNTER
END-HISTOGRAM
END

```

CITY	NUMBER OF PERSONS	CNT
MADISON	3	1
MADRID	41	2
MAILLY LE CAMP	1	3
MAMERS	1	4
MANSFIELD	4	5
MARSEILLE	2	6
MATLOCK	1	7
MELBOURNE	2	8

In dem obigen Programm werden mit dem HISTOGRAM-Statement außerdem die Systemvariablen *NUMBER und *COUNTER ausgewertet und mit dem DISPLAY-Statement ausgegeben. *NUMBER enthält die Anzahl der Datensätze, in denen der zuletzt gelesene Wert vorkommt; *COUNTER enthält die Gesamtanzahl der bisher gelesenen Werte.

Datenbank-Verarbeitungsschleifen

Natural initiiert automatisch die Schleifen, die zur Verarbeitung von Daten erforderlich sind, die mit einem FIND-, READ- oder HISTOGRAM-Statement von einer Datenbank ausgewählt wurden.

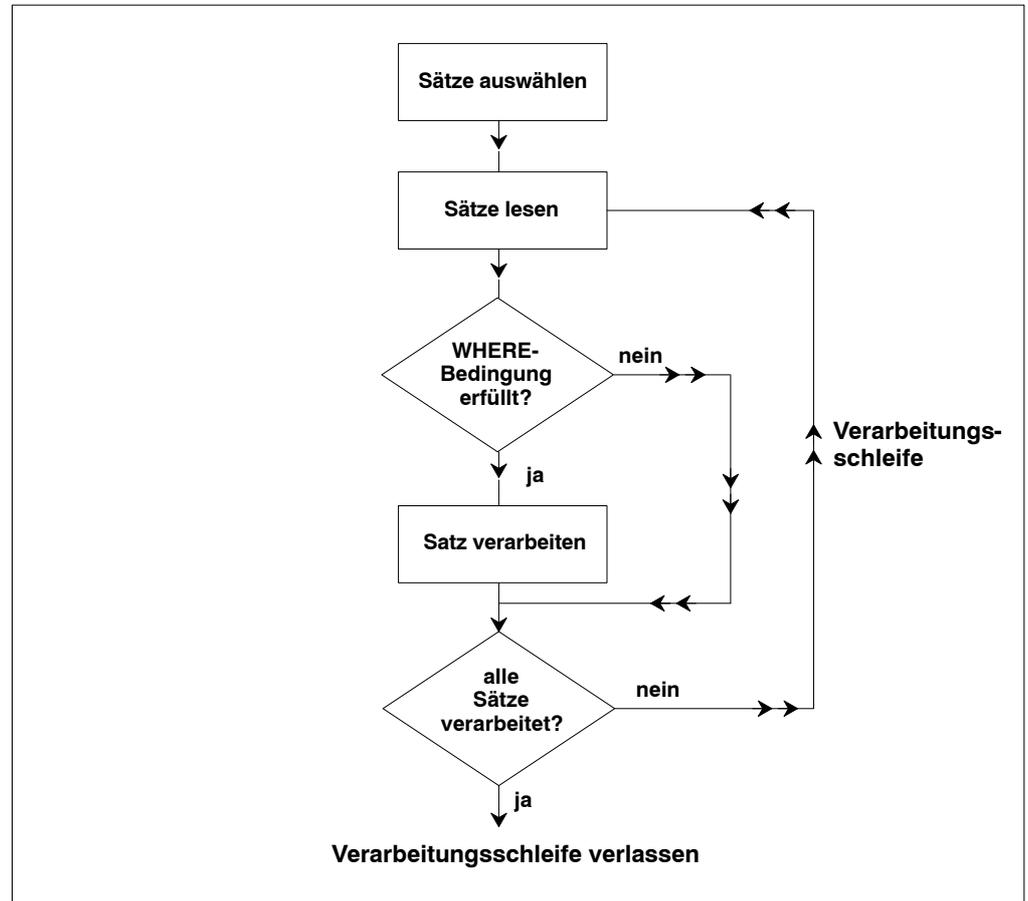
Beispiel:

```
** Example Program 'FINDX03'  
DEFINE DATA LOCAL  
1 MYVIEW VIEW OF EMPLOYEES  
  2 NAME  
  2 FIRST-NAME  
  2 CITY  
END-DEFINE  
*  
FIND MYVIEW WITH NAME = 'ADKINSON'  
  DISPLAY NAME FIRST-NAME CITY  
END-FIND  
END
```

Die obige FIND-Schleife wählt von der Datei EMPLOYEES alle Datensätze aus, in denen das Feld NAME den Wert "ADKINSON" enthält, und verarbeitet die ausgewählten Datensätze. Im Beispiel besteht die Verarbeitung in der Anzeige bestimmter Feldwerte aus jedem der ausgewählten Datensätze.

Wenn obiges FIND-Statement zusätzlich zu der WITH-Klausel noch eine WHERE-Klausel enthielte, würden nur diejenigen der ausgewählten Datensätze verarbeitet, die die WITH- und die WHERE-Bedingung erfüllen.

Das folgende Diagramm zeigt den logischen Ablauf einer Datenbank-Verarbeitungsschleife:



Hierarchien von Verarbeitungsschleifen

Die Verwendung mehrerer FIND- bzw. READ-Statements führt zu einer Hierarchie ineinander geschachtelter Schleifen, wie das folgende Beispiel zeigt:

Beispiel für Verarbeitungsschleifen-Hierarchie:

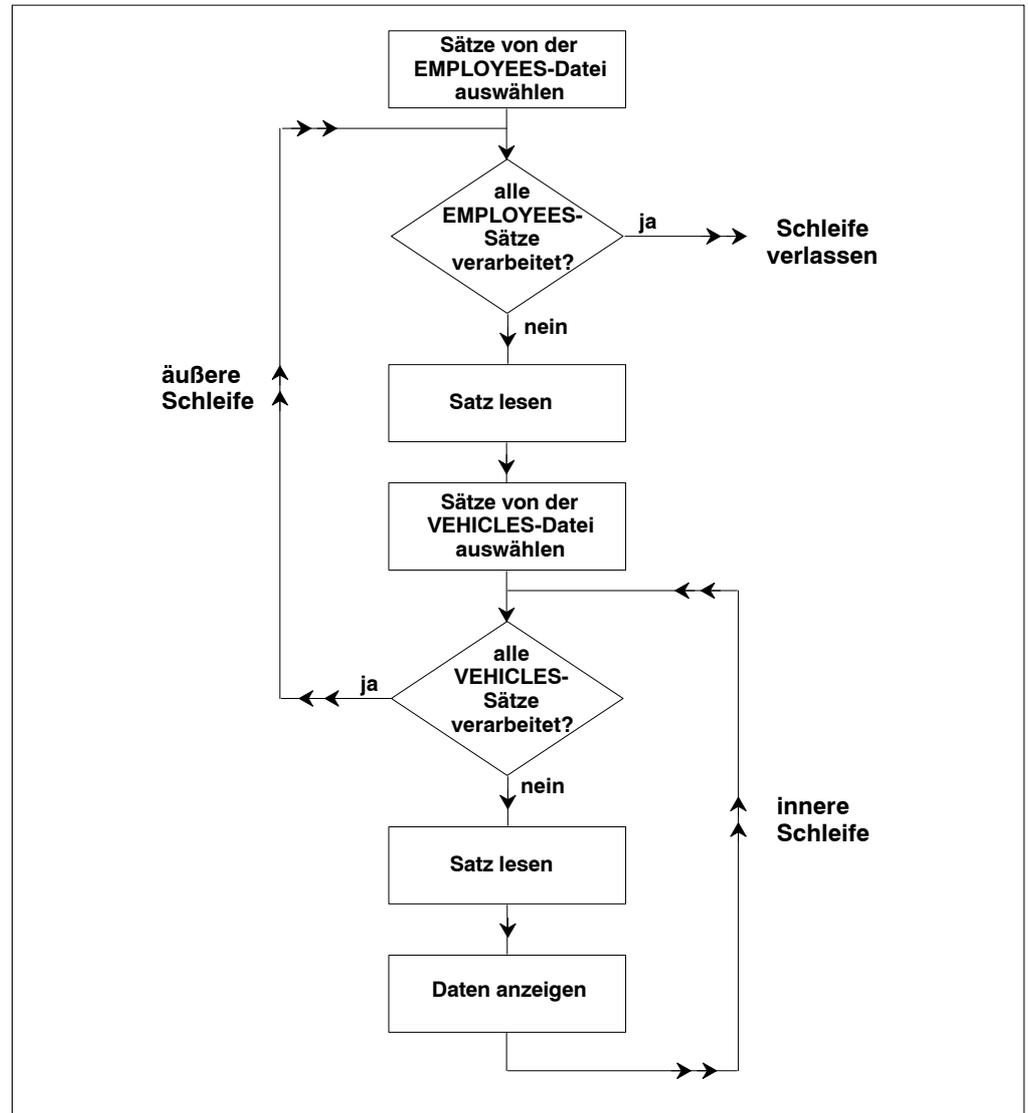
```
** Example Program 'FINDX04'  
DEFINE DATA LOCAL  
1 PERSONVIEW VIEW OF EMPLOYEES  
  2 PERSONNEL-ID  
  2 NAME  
1 AUTOVIEW VIEW OF VEHICLES  
  2 PERSONNEL-ID  
  2 MAKE  
  2 MODEL  
END-DEFINE  
*  
EMP. FIND PERSONVIEW WITH NAME = 'ADKINSON'  
VEH.  FIND AUTOVIEW WITH PERSONNEL-ID = PERSONNEL-ID (EMP.)  
      DISPLAY NAME MAKE MODEL  
      END-FIND  
      END-FIND  
END
```

Im obigen Programm werden zunächst alle Datensätze mit Namen "ADKINSON" von der Datei EMPLOYEES ausgewählt. Dann wird jeder dieser Datensätze wie folgt verarbeitet:

1. Mit dem zweiten FIND-Statement werden für alle von der Datei EMPLOYEES gelesenen Personen die dazugehörigen Fahrzeuge (VEHICLES) gesucht, und zwar unter Verwendung der Personalnummern (PERSONNEL-ID) aus den mit dem ersten FIND-Statement von der EMPLOYEES-Datei ausgewählten Datensätzen.
2. Dann werden folgende Werte angezeigt (DISPLAY): der NAME jeder gefundenen Person (diese Informationen werden von der EMPLOYEES-Datei gelesen) und Marke und Modell (MAKE und MODEL) des dazugehörigen Fahrzeugs (diese Informationen kommen von der VEHICLES-Datei).

Das zweite FIND-Statement initiiert innerhalb der äußeren FIND-Schleife des ersten FIND-Statements eine innere Schleife, wie das folgende Diagramm veranschaulicht.

Das Diagramm zeigt den logischen Ablauf der Hierarchie von Verarbeitungsschleifen des Beispielprogramms auf der vorigen Seite:



Es ist auch möglich, eine Verarbeitungsschleifen-Hierarchie aufzubauen, in der zwei ineinander verschachtelte Schleifen auf dieselbe Datei zugreifen, wie das folgende Beispiel zeigt.

Beispiel für geschachtelte FIND-Schleifen, die auf dieselbe Datei zugreifen:

```

** Example Program 'FINDX05'
DEFINE DATA LOCAL
1 PERSONVIEW VIEW OF EMPLOYEES
  2 NAME
  2 FIRST-NAME
  2 CITY
1 #NAME (A40)
END-DEFINE
*
WRITE TITLE LEFT JUSTIFIED
  'PEOPLE IN SAME CITY AS:' #NAME / 'CITY:' CITY SKIP 1
FIND PERSONVIEW WITH NAME = 'JONES'
      WHERE FIRST-NAME = 'LAUREL'
  COMPRESS NAME FIRST-NAME INTO #NAME
  FIND PERSONVIEW WITH CITY = CITY
      DISPLAY NAME FIRST-NAME CITY
  END-FIND
END-FIND
END

```

Im obigen Programm werden zunächst in der Datei EMPLOYEES alle Personen mit Namen "JONES" und Vornamen "LAUREL" gesucht. Dann werden zu jeder gefundenen Person alle Personen, die in derselben Stadt wohnen, in der EMPLOYEES-Datei gesucht, und es wird eine Liste dieser Personen erzeugt. Alle mit dem DISPLAY-Statement ausgegebenen Feldwerte werden mit dem zweiten FIND-Statement gelesen.

```

PEOPLE IN SAME CITY AS: JONES LAUREL
CITY: BALTIMORE

```

NAME	FIRST-NAME	CITY
JENSEN	MARTHA	BALTIMORE
LAWLER	EDDIE	BALTIMORE
FORREST	CLARA	BALTIMORE
ALEXANDER	GIL	BALTIMORE
NEEDHAM	SUNNY	BALTIMORE
ZINN	CARLOS	BALTIMORE
JONES	LAUREL	BALTIMORE

Weitere Beispiele für geschachtelte READ- und FIND-Statements:

Siehe Programme READX04 und LIMITX01 in Library SYSEXP.

Datenänderungen — Transaktionsverarbeitung

Die folgenden Themen werden nachfolgend behandelt:

- Logische Transaktionen
- Datensatz-Kontrolle während einer Transaktion (“Hold“-Logik)
- Transaktion abbrechen
- Transaktion neu starten.

Logische Transaktionen

Natural führt Veränderungen auf der Datenbank auf der Grundlage von *Transaktionen* aus, d.h. alle Veränderungszugriffe werden in logische Transaktionseinheiten gegliedert. Eine Transaktion ist die kleinste (von Ihnen definierte) Verarbeitungseinheit, die vollständig ausgeführt werden muß, um die logische Konsistenz der gespeicherten Daten zu gewährleisten.

Eine logische Transaktion kann aus einem oder mehreren datenverändernden Statements (DELETE, STORE, UPDATE) bestehen und auf eine oder mehrere Dateien zugreifen. Eine logische Transaktion kann sich auch über mehrere Natural-Programme erstrecken.

Eine logische Transaktion beginnt, sobald ein Datensatz in den "Hold"-Status gestellt wird. Dies erfolgt durch Natural automatisch, wenn ein Satz zwecks Änderung gelesen wird, wenn also z.B. in einer FIND-Schleife ein UPDATE- oder DELETE-Statement steht.

Das Ende einer logischen Transaktion wird im Programm durch ein END TRANSACTION-Statement bestimmt. Dieses Statement gewährleistet, daß alle durch die Transaktion bewirkten Änderungen erfolgreich durchgeführt werden, und gibt anschließend alle während der Transaktion gehaltenen Datensätze wieder frei.

Beispiel:

```
DEFINE DATA LOCAL
1 MYVIEW VIEW OF EMPLOYEES
  2 NAME
END-DEFINE
FIND MYVIEW WITH NAME = 'SMITH'
  DELETE
  END TRANSACTION
END-FIND
END
```

Jeder gefundene Satz würde hier in den "Hold"-Status gestellt, gelöscht und anschließend — wenn das END TRANSACTION-Statement ausgeführt wird — aus dem "Hold"-Status wieder freigegeben.

Anmerkung:

Mit dem Profilparameter OPRB kann der Natural-Administrator festlegen, ob Natural am Ende jedes Programms ein END TRANSACTION-Statement generieren soll. Einzelheiten hierzu sagt Ihnen Ihr Natural-Administrator.

Beispiel für STORE-Statement:

Sieh Programm STOREX01 in Library SYSEXP.

Datensatz-Kontrolle während einer Transaktion (“Hold”-Logik)

Wird Natural zusammen mit Adabas eingesetzt, so wird jeder Datensatz, der verändert werden soll, solange ins “Hold” gestellt, bis die Transaktion entweder durch ein END TRANSACTION- oder BACKOUT TRANSACTION-Statement beendet oder aufgrund einer Zeitüberschreitung abgebrochen wird.

Solange ein Datensatz für einen Benutzer im “Hold” steht, haben andere Benutzer keine Möglichkeit, diesen Datensatz zu ändern. Ein Benutzer, der dies tun will, gelangt in den Wartestatus (Wait) und erhält die Kontrolle über den gewünschten Satz erst, wenn der erste Benutzer seine Transaktion beendet/abgebrochen hat.

Um zu vermeiden, daß ein Benutzer im Wartestatus bleibt, ist es möglich den Session-Parameter WH (Wait Hold) entsprechend zu setzen (siehe *Natural-Referenzhandbuch*).

Beim Programmieren sollten Sie folgendes bezüglich der Hold-Logik bedenken:

- Die Zeit, für die ein Datensatz höchstens ins “Hold” gestellt werden kann, wird von Adabas durch das “Transaction Time Limit” (Transaktionszeitbegrenzung; Adabas TT-Parameter) begrenzt. Wird diese Zeit überschritten, erhält man eine entsprechende Fehlermeldung, und Veränderungen, die nach dem letzten END TRANSACTION-Statement erfolgen, werden rückgängig gemacht.
- Die Anzahl der ISNs im “Hold” und mögliche Transaktionszeitüberschreitungen ergeben sich aus der Größe einer Transaktion, d.h. aus der Platzierung des END TRANSACTION-Statements. In diesem Zusammenhang sollten Sie die Nutzung von Restart-Möglichkeiten in Betracht ziehen. Falls die Mehrzahl der zu verarbeitenden Datensätze nicht verändert werden soll, empfiehlt es sich beispielsweise, ein GET-Statement zu verwenden, um das “Halten” von Sätzen zu steuern. Damit spart man viele END TRANSACTION-Statements und verringert gleichzeitig die Zahl der in den “Hold”-Status gestellten ISNs. Bei Verarbeitung umfangreicher Dateien sollte bedacht werden, daß für ein GET-Statement ein zusätzlicher Adabas-Aufruf erforderlich ist. Ein Beispiel für die Verwendung eines GET-Statements sehen Sie im folgenden.

Beispiel für GET-Statement:

```
DEFINE DATA LOCAL
1 EMPLOY-VIEW VIEW OF EMPLOYEES
  2 NAME
  2 SALARY (1)
END-DEFINE
RD. READ EMPLOY-VIEW BY NAME
  IF SALARY (1) > 30000
GE.  GET EMPLOY-VIEW *ISN (RD.)
      COMPUTE SALARY (1) = SALARY (1) * 1.15
      UPDATE (GE.)
      END TRANSACTION
  END-IF
END-READ
END
```

Auf Großrechnern wird das “Halten” von Datensätzen auch vom Profilparameter RI gesteuert, der vom Natural-Administrator gesetzt wird.

Transaktion abbrechen

Innerhalb einer aktiven logischen Transaktion, d.h. bevor das END TRANSACTION-Statement ausgeführt wird, können Sie durch Verwendung eines BACKOUT TRANSACTION-Statements den Abbruch der Transaktion bewirken. Dadurch werden alle vorgenommenen Änderungen (einschließlich hinzugefügter und gelöschter Datensätze) rückgängig gemacht und die von der Transaktion gehaltenen Datensätze freigegeben.

Transaktion neu starten

Mit dem END TRANSACTION-Statement können Sie auch transaktionsbezogene Informationen speichern. Falls die Verarbeitung der Transaktion nicht ordnungsgemäß beendet werden kann, können Sie beim Neustarten (Restart) der Transaktion diese Informationen mit einem GET TRANSACTION DATA-Statement lesen, um festzustellen, an welchem Punkt die Verarbeitung fortgesetzt werden muß.

Beispiel für Verwendung von Transaktionsdaten beim Neustarten einer Transaktion:

Im folgenden Beispielprogramm werden Daten der Dateien EMPLOYEES und VEHICLES verändert. Wenn das Programm nach einem Abbruch neu gestartet wird, werden Sie durch eine Restart-Prozedur darüber informiert, welcher Datensatz der Datei EMPLOYEES vor dem Abbruch zuletzt verarbeitet wurde, und können die Verarbeitung dann an dieser Stelle wiederaufnehmen. Es bestünde zusätzlich die Möglichkeit, Angaben über den zuletzt bearbeiteten Satz der VEHICLES-Datei in die Restart-Transaktionsmeldung einzufügen.

```

** Example Program 'GETTRX01'
DEFINE DATA LOCAL
01 PERSON VIEW OF EMPLOYEES
  02 PERSONNEL-ID      (A8)
  02 NAME              (A20)
  02 FIRST-NAME       (A20)
  02 MIDDLE-I         (A1)
  02 CITY             (A20)
01 AUTO VIEW OF VEHICLES
  02 PERSONNEL-ID      (A8)
  02 MAKE              (A20)
  02 MODEL            (A20)
01 ET-DATA
  02 #APPL-ID          (A8) INIT <' '>
  02 #USER-ID          (A8)
  02 #PROGRAM          (A8)
  02 #DATE             (A10)
  02 #TIME             (A8)
  02 #PERSONNEL-NUMBER (A8)
END-DEFINE
*
GET TRANSACTION DATA #APPL-ID #USER-ID #PROGRAM
                    #DATE      #TIME      #PERSONNEL-NUMBER
*
IF #APPL-ID NOT = 'NORMAL'      /* IF LAST EXECUTION ENDED ABNORMALLY
  AND #APPL-ID NOT = ' '
  INPUT (AD=OIL)
  // 20T '*** LAST SUCCESSFUL TRANSACTION ***' (I)
  / 20T '*****'
  /// 25T 'APPLICATION:' #APPL-ID
  / 32T 'USER:' #USER-ID
  / 29T 'PROGRAM:' #PROGRAM
  / 24T 'COMPLETED ON:' #DATE 'AT' #TIME
  / 20T 'PERSONNEL NUMBER:' #PERSONNEL-NUMBER
END-IF
* program continued on next page

```

```

* continued from previous page
REPEAT
  INPUT (AD=MIL) // 20T 'ENTER PERSONNEL NUMBER:' #PERSONNEL-NUMBER
  IF #PERSONNEL-NUMBER = 99999999
    ESCAPE BOTTOM
  END-IF
  FIND1. FIND PERSON WITH PERSONNEL-ID = #PERSONNEL-NUMBER
  IF NO RECORDS FOUND
    REINPUT 'SPECIFIED NUMBER DOES NOT EXIST; ENTER ANOTHER ONE.'
  END-NOREC
  FIND2. FIND AUTO WITH PERSONNEL-ID = #PERSONNEL-NUMBER
  IF NO RECORDS FOUND
    WRITE 'PERSON DOES NOT OWN ANY CARS'
  END-NOREC
  IF *COUNTER (FIND1.) = 1 /* FIRST PASS THROUGH THE LOOP
    INPUT (AD=M)
      / 20T 'EMPLOYEES/AUTOMOBILE DETAILS' (I)
      / 20T '-----'
    /// 20T 'NUMBER:' PERSONNEL-ID (AD=O)
      / 22T 'NAME:' NAME ' ' FIRST-NAME ' ' MIDDLE-I
      / 22T 'CITY:' CITY
      / 22T 'MAKE:' MAKE
      / 21T 'MODEL:' MODEL
    UPDATE (FIND1.) /* UPDATE THE EMPLOYEES FILE
  ELSE /* SUBSEQUENT PASSES THROUGH THE LOOP
    INPUT NO ERASE (AD=M) ////////// 20T MAKE / 20T MODEL
  END-IF
  UPDATE (FIND2.) /* UPDATE THE VEHICLES FILE
  MOVE *APPLIC-ID TO #APPL-ID
  MOVE *INIT-USER TO #USER-ID
  MOVE *PROGRAM TO #PROGRAM
  MOVE *DAT4E TO #DATE
  MOVE *TIME TO #TIME
  END TRANSACTION #APPL-ID #USER-ID #PROGRAM
                    #DATE #TIME #PERSONNEL-NUMBER
  END-FIND /* FOR VEHICLES (FIND2.)
  END-FIND /* FOR EMPLOYEES (FIND1.)
  END-REPEAT /* FOR REPEAT
  STOP /* Simulate abnormal transaction end
  END TRANSACTION 'NORMAL '
  END

```

Statements ACCEPT und REJECT

Die Statements ACCEPT und REJECT können Sie zur Auswahl von Datensätzen anhand von Ihnen definierter logischer Auswahlkriterien verwenden.

Sie können ACCEPT und REJECT zusammen mit den Datenbankzugriffs-Statements READ, FIND und HISTOGRAM verwenden.

Beispiel für ACCEPT-Statement:

```

** Example Program 'ACCEPX01'
DEFINE DATA LOCAL
1 MYVIEW VIEW OF EMPLOYEES
  2 NAME
  2 JOB-TITLE
  2 CURR-CODE (1:1)
  2 SALARY (1:1)
END-DEFINE
READ (20) MYVIEW BY NAME WHERE CURR-CODE (1) = 'USD'
  ACCEPT IF SALARY (1) >= 40000
  DISPLAY NAME JOB-TITLE SALARY (1)
END-READ
END

```

NAME	CURRENT POSITION	ANNUAL SALARY
ADKINSON	DBA	46700
ADKINSON	MANAGER	47000
ADKINSON	MANAGER	47000
AFANASSIEV	DBA	42800
ALEXANDER	DIRECTOR	48000
ANDERSON	MANAGER	50000
ATHERTON	ANALYST	43000
ATHERTON	MANAGER	40000

Mit einem ACCEPT- oder REJECT-Statement können Sie zusätzlich zu der WHERE- und WITH-Bedingung eines READ-Statements weitere logische Auswahlkriterien angeben. Das ACCEPT- bzw. REJECT-Auswahlkriterium wird erst ausgewertet, *nachdem* die über das READ-Statement ausgewählten Datensätze gelesen worden sind.

Die folgenden logischen Operatoren können in einem ACCEPT- bzw. REJECT-Statement verwendet werden (weitere Einzelheiten hierzu finden Sie im *Natural-Referenzhandbuch*):

EQUAL TO	EQ	=
NOT EQUAL TO	NE	\neq
LESS THAN	LT	<
LESS THAN OR EQUAL TO	LE	\leq
GREATER THAN	GT	>
GREATER THAN OR EQUAL TO	GE	\geq

Außerdem können Sie die Boole'schen Operatoren AND, OR und NOT zur Verknüpfung logischer Bedingungen einsetzen; mit Klammern können Sie die Bedingungen in logische Einheiten unterteilen.

Das folgende Programm zeigt die Verwendung des Boole'schen Operators AND in einem ACCEPT-Statement:

Beispiel für ACCEPT-Statement mit Operator AND:

```
** Example Program 'ACCEPX02'  
DEFINE DATA LOCAL  
1 MYVIEW VIEW OF EMPLOYEES  
  2 NAME  
  2 JOB-TITLE  
  2 CURR-CODE (1:1)  
  2 SALARY    (1:1)  
END-DEFINE  
READ (20) MYVIEW BY NAME WHERE CURR-CODE (1) = 'USD'  
  ACCEPT IF SALARY (1) >= 40000  
          AND SALARY (1) <= 45000  
  DISPLAY NAME JOB-TITLE SALARY (1)  
END-READ  
END
```

Das folgende Programm zeigt die Verwendung des Boole'schen Operators OR in einem REJECT-Statement. Das Programm erzeugt die gleiche Ausgabe wie das vorherige, da gleichzeitig die logischen Operatoren umgekehrt wurden:

Beispiel für REJECT-Statement mit Operator OR:

```

** Example Program 'ACCEPX03'
DEFINE DATA LOCAL
1 MYVIEW VIEW OF EMPLOYEES
  2 NAME
  2 JOB-TITLE
  2 CURR-CODE (1:1)
  2 SALARY (1:1)
END-DEFINE
READ (20) MYVIEW BY NAME WHERE CURR-CODE (1) = 'USD'
  REJECT IF SALARY (1) < 40000
    OR SALARY (1) > 45000
  DISPLAY NAME JOB-TITLE SALARY (1)
END-READ
END

```

NAME	CURRENT POSITION	ANNUAL SALARY
AFANASSIEV	DBA	42800
ATHERTON	ANALYST	43000
ATHERTON	MANAGER	40000

Weitere Beispiele für ACCEPT- und REJECT-Statements:

Siehe Programme ACCEPX04, ACCEPX05 und ACCEPX06 in Library SYSEXP.

AT START/END OF DATA-Statements

AT START OF DATA-Statement

Mit dem Statement AT START OF DATA können Sie eine beliebige Verarbeitung angeben, die ausgeführt werden soll, nachdem der erste Datensatz einer Datenbank-Verarbeitungsschleife gelesen worden ist.

Das AT START OF DATA-Statement muß innerhalb der betreffenden Verarbeitungsschleife stehen.

Erzeugt das AT START OF DATA-Statement eine Ausgabe, so wird diese *vor der ersten Feldwert-Ausgabe* ausgegeben. Standardmäßig erfolgt die Ausgabe linksbündig auf der Seite.

AT END OF DATA-Statement

Mit dem Statement AT END OF DATA können Sie eine beliebige Verarbeitung angeben, die ausgeführt werden soll, nachdem alle Datensätze in einer Datenbank-Verarbeitungsschleife verarbeitet worden sind.

Das AT END OF DATA-Statement muß innerhalb der betreffenden Verarbeitungsschleife stehen.

Erzeugt das AT END OF DATA-Statement eine Ausgabe, so wird diese *nach der letzten Feldwert-Ausgabe* ausgegeben. Standardmäßig erfolgt die Ausgabe linksbündig auf der Seite.

Das Beispielprogramm auf der folgenden Seite veranschaulicht die Verwendung der Statements AT START OF DATA und AT END OF DATA. Das AT START OF DATA-Statement enthält die Systemvariable *TIME zur Anzeige der Uhrzeit. Das AT END OF DATA-Statement enthält die Systemfunktion OLD, um den Namen der zuletzt ausgewählten Person anzuzeigen.

Beispiel für AT START OF DATA- und AT END OF DATA-Statements:

```

** Example Program 'ATSTAX01'
DEFINE DATA LOCAL
1 MYVIEW VIEW OF EMPLOYEES
  2 CITY
  2 NAME
  2 JOB-TITLE
  2 INCOME (1:1)
    3 CURR-CODE
    3 SALARY
    3 BONUS (1:1)
END-DEFINE
WRITE TITLE 'XYZ EMPLOYEE ANNUAL SALARY AND BONUS REPORT' /
READ (3) MYVIEW BY CITY STARTING FROM 'E'
  DISPLAY GIVE SYSTEM FUNCTIONS
    NAME (AL=15) JOB-TITLE (AL=15) INCOME (1)
  AT START OF DATA
    WRITE 'RUN TIME:' *TIME /
  END-START
  AT END OF DATA
    WRITE / 'LAST PERSON SELECTED:' OLD (NAME) /
  END-ENDDATA
END-READ
AT END OF PAGE
  WRITE / 'AVERAGE SALARY:' AVER (SALARY(1))
END-ENDPAGE
END

```

Das Programm erzeugt folgende Ausgabe:

XYZ EMPLOYEE ANNUAL SALARY AND BONUS REPORT				
NAME	CURRENT POSITION	INCOME		
		CURRENCY CODE	ANNUAL SALARY	BONUS
RUN TIME: 11:18:58.2				
DUYVERMAN	PROGRAMMER	USD	34000	0
PRATT	SALES PERSON	USD	38000	9000
MARKUSH	TRAINEE	USD	22000	0
LAST PERSON SELECTED: MARKUSH				
AVERAGE SALARY:		31333		

Weitere Beispiele für AT START OF DATA- und AT END OF DATA-Statements:

Siehe Programme ATENDX01, ATSTAX02 und WRITEX09 in Library SYSEXP.

AUSGABE VON DATEN

Dieses Kapitel beschreibt verschiedene Möglichkeiten, wie Sie die Form eines mit Natural erzeugten Ausgabe-Reports, d.h. die Art und Weise, in der die Daten angezeigt werden, beeinflussen können.

Unter anderem werden folgende Punkte behandelt:

- Layout einer Ausgabeseite — Übersicht
- Statements `DISPLAY` und `WRITE`
- Index-Notation (n:n) für multiple Felder und Periodengruppen
- Seitenüberschriften und Seitenvorschübe
- Spaltenüberschriften
- Parameter zur Beeinflussung der Ausgabe von Feldern
- Editiermasken — der `EM`-Parameter
- Vertikale Ausgaben.

Layout einer Ausgabeseite — Übersicht

Das folgende Programm veranschaulicht die allgemeine Form einer Ausgabeseite:

```
** Example Program 'OUTPUX01'
DEFINE DATA LOCAL
  1 EMP-VIEW VIEW OF EMPLOYEES
    2 NAME
    2 FIRST-NAME
    2 BIRTH
END-DEFINE
*
WRITE TITLE '***** Page Title *****'
WRITE TRAILER '***** Page Trailer *****'
AT TOP OF PAGE
  WRITE '===== Top of Page ====='
END-TOPPAGE
AT END OF PAGE
  WRITE '===== End of Page ====='
END-ENDPAGE
*
READ (10) EMP-VIEW BY NAME
  DISPLAY NAME FIRST-NAME BIRTH (EM=YYYY-MM-DD)
  AT START OF DATA
    WRITE '>>>>> Start of Data >>>>>'
  END-START
  AT END OF DATA
    WRITE '<<<<< End of Data <<<<<'
  END-ENDDATA
END-READ
END
```

```
***** Page Title *****
===== Top of Page =====
      NAME              FIRST-NAME          DATE
                        OF
                        BIRTH
-----
>>>>> Start of Data >>>>>
ABELLAN                KEPA                1961-04-08
ACHIESON               ROBERT             1963-12-24
ADAM                   SIMONE             1952-01-30
ADKINSON               JEFF               1951-06-15
ADKINSON               PHYLLIS            1956-09-17
ADKINSON               HAZEL              1954-03-19
ADKINSON               DAVID              1946-10-12
ADKINSON               CHARLIE            1950-03-02
ADKINSON               MARTHA             1970-01-01
ADKINSON               TIMMIE             1970-03-03
<<<<< End of Data <<<<<
***** Page Trailer *****
===== End of Page =====
```

Folgende Statements haben Auswirkungen auf das Aussehen einer Ausgabe:

Statement	Funktion
WRITE TITLE	Mit diesem Statement können Sie eine Seiten-Kopfzeile angeben, d.h. Text, der am Anfang einer Seite ausgegeben werden soll.
WRITE TRAILER	Mit diesem Statement können Sie eine Seiten-Fußzeile angeben, d.h. Text, der am Ende einer Seite ausgegeben werden soll.
AT TOP OF PAGE	Mit diesem Statement können Sie eine Verarbeitung angeben, die immer dann ausgeführt werden soll, wenn eine neue Ausgabeseite erzeugt wird. Erzeugt diese Verarbeitung eine Ausgabe, dann wird diese unter der Seiten-Kopfzeile ausgegeben.
AT END OF PAGE	Mit diesem Statement können Sie eine Verarbeitung angeben, die immer dann ausgeführt werden soll, wenn eine Seitenende-Bedingung vorliegt. Erzeugt diese Verarbeitung eine Ausgabe, dann wird diese unter der (mit dem WRITE TRAILER-Statement erzeugten) Seiten-Fußzeile ausgegeben.
AT START OF DATA	Mit diesem Statement können Sie eine Verarbeitung angeben, die ausgeführt werden soll, nachdem in einer Datenbank-Verarbeitungsschleife der erste Datensatz gelesen worden ist. Erzeugt diese Verarbeitung eine Ausgabe, dann wird diese vor dem ersten Feldwert ausgegeben.
AT END OF DATA	Mit diesem Statement können Sie eine Verarbeitung angeben, die ausgeführt werden soll, nachdem in einer Datenbank-Verarbeitungsschleife alle Datensätze verarbeitet worden sind. Erzeugt diese Verarbeitung eine Ausgabe, dann wird diese unmittelbar nach dem letzten Feldwert ausgegeben.
DISPLAY/WRITE	Mit diesen Statements steuern Sie die Art, in der gelesene Feldwerte ausgegeben werden.

Die Statements AT START OF DATA und AT END OF DATA sind im Kapitel **Datenbankzugriffe** (Seite 97) beschrieben. Die anderen oben aufgeführten Statements sind in den folgenden Abschnitten des vorliegenden Kapitels beschrieben.

Statements DISPLAY und WRITE

Mit den Statements DISPLAY und WRITE geben Sie Daten aus und steuern die Art und Weise, in der die Informationen ausgegeben werden.

DISPLAY-Statement

Das DISPLAY-Statement erzeugt eine Ausgabe in Spaltenform; d.h. die Werte eines Feldes werden jeweils in einer Spalte untereinander ausgegeben. Wenn mehrere Felder ausgegeben werden, d.h. wenn mehrere Spalten erzeugt werden, werden diese Spalten nebeneinander ausgegeben.

Die Reihenfolge, in der die Felder ausgegeben werden, bestimmen Sie durch die Reihenfolge, in der Sie die Feldnamen im DISPLAY-Statement angeben.

Das DISPLAY-Statement im folgenden Programm zeigt für jede Person zuerst die Personalnummer (PERSONNEL-ID) an, dann den Nachnamen (NAME) und zuletzt die Tätigkeitsbezeichnung (JOB-TITLE):

```

** Example Program 'DISPLX01'
DEFINE DATA LOCAL
1 VIEWEMP VIEW OF EMPLOYEES
  2 PERSONNEL-ID
  2 NAME
  2 BIRTH
  2 JOB-TITLE
END-DEFINE
READ (3) VIEWEMP BY BIRTH
  DISPLAY PERSONNEL-ID NAME JOB-TITLE
END-READ
END

```

PERSONNEL ID	NAME	CURRENT POSITION
30020013	GARRET	TYPIST
30016112	TAILOR	WAREHOUSEMAN
20017600	PIETSCH	SECRETARY

Page 1 99-01-22 11:31:01

Um die Reihenfolge der Spalten in der Ausgabe zu ändern, ändern Sie einfach die Reihenfolge der Feldnamen im DISPLAY-Statement. Falls Sie beispielsweise zuerst die Nachnamen, dann die Tätigkeitsbezeichnungen und zuletzt die Personalnummern ausgeben möchten, müßte das entsprechende DISPLAY-Statement folgendermaßen aussehen:

```

** Example Program 'DISPLX02'
DEFINE DATA LOCAL
1 VIEWEMP VIEW OF EMPLOYEES
  2 PERSONNEL-ID
  2 NAME
  2 BIRTH
  2 JOB-TITLE
END-DEFINE
READ (3) VIEWEMP BY BIRTH
  DISPLAY NAME JOB-TITLE PERSONNEL-ID
END-READ
END

```

Page	1	99-01-22	11:32:06
NAME	CURRENT POSITION	PERSONNEL ID	
GARRET	TYPIST	30020013	
TAILOR	WAREHOUSEMAN	30016112	
PIETSCH	SECRETARY	20017600	

Über jeder Spalte wird eine Spaltenüberschrift ausgegeben. Verschiedene Möglichkeiten, die Ausgabe dieser Überschriften zu beeinflussen, sind weiter unten in diesem Kapitel beschrieben.

WRITE-Statement

Das WRITE-Statement wird zur Erzeugung unformatierter (d.h. nicht in Spalten unterteilter) Ausgaben benutzt. Im Gegensatz zum DISPLAY-Statement gilt für das WRITE-Statement folgendes:

- Es führt, wenn nötig, einen automatischen Zeilenvorschub aus; d.h. ein Feld oder Textelement, das nicht mehr in eine Zeile paßt, wird automatisch in der nächsten Zeile ausgegeben.
- Es erzeugt keine Spaltenüberschriften.
- Bei mehreren Werten pro Feld werden diese nicht untereinander sondern nebeneinander ausgegeben.

Die beiden Beispielprogramme auf der folgenden Seite veranschaulichen die grundsätzlichen Unterschiede zwischen DISPLAY- und WRITE-Statement.

Sie können beide Statements auch miteinander kombinieren; diese Möglichkeit ist weiter unten in diesem Kapitel beschrieben.

Beispiel für DISPLAY-Statement:

```
** Example Program 'DISPLX03'  
DEFINE DATA LOCAL  
1 VIEWEMP VIEW OF EMPLOYEES  
  2 NAME  
  2 FIRST-NAME  
  2 SALARY (1:3)  
END-DEFINE  
READ (2) VIEWEMP BY NAME STARTING FROM 'JONES'  
  DISPLAY NAME FIRST-NAME SALARY (1:3)  
END-READ  
END
```

Page	1	97-08-14 11:44:00	
	NAME	FIRST-NAME	ANNUAL SALARY
	JONES	VIRGINIA	46000 42300 39300
	JONES	MARSHA	50000 46000 42700

Beispiel für WRITE-Statement:

```
** Example Program 'WRITEX01'  
DEFINE DATA LOCAL  
1 VIEWEMP VIEW OF EMPLOYEES  
  2 NAME  
  2 FIRST-NAME  
  2 SALARY (1:3)  
END-DEFINE  
READ (2) VIEWEMP BY NAME STARTING FROM 'JONES'  
  WRITE NAME FIRST-NAME SALARY (1:3)  
END-READ  
END
```

Page	1			97-08-14	11:45:00
JONES		VIRGINIA	46000	42300	39300
JONES		MARSHA	50000	46000	42700

Spaltenabstand — der SF-Parameter und die Notation nX

Standardmäßig sind die mit einem DISPLAY-Statement ausgegebenen Spalten jeweils durch *eine* Leerstelle voneinander getrennt.

Mit dem Session-Parameter SF (Spacing Factor = Spaltenabstand) können Sie angeben, wieviele Leerstellen zwischen den Spalten einer DISPLAY-Ausgabe eingefügt werden sollen. Sie können die Anzahl der Leerstellen auf einen Wert von 1 bis 30 setzen.

Der Parameter kann in einem FORMAT-Statement angegeben werden und gilt dann für den ganzen Report. Oder er kann in einem DISPLAY-Statement angegeben werden, und zwar auf Statement-Ebene, aber nicht auf Feld-Ebene.

Mit der Notation nX können Sie die Anzahl der Leerstellen (n) zwischen zwei bestimmten Spalten angeben.

Eine nX -Notation hat Vorrang vor einer SF-Parameterangabe.

```

** Example Program 'DISPLX04'
DEFINE DATA LOCAL
1 VIEWEMP VIEW OF EMPLOYEES
  2 PERSONNEL-ID
  2 NAME
  2 BIRTH
  2 JOB-TITLE
END-DEFINE
FORMAT SF=3
READ (3) VIEWEMP BY BIRTH
  DISPLAY PERSONNEL-ID NAME 5X JOB-TITLE
END-READ
END

```

Das obige Beispielprogramm erzeugt folgende Ausgabe, wobei die ersten beiden Spalten aufgrund des SF-Parameters im FORMAT-Statement durch 3 Leerstellen voneinander getrennt sind, während die zweite und dritte Spalte aufgrund der Notation "5X" im DISPLAY-Statement durch 5 Leerstellen voneinander getrennt sind:

PERSONNEL ID	NAME	CURRENT POSITION
30020013	GARRET	TYPIST
30016112	TAILOR	WAREHOUSEMAN
20017600	PIETSCH	SECRETARY

Die Notation *nX* kann auch in einem WRITE-Statement verwendet werden, um Leerstellen zwischen Ausgabe-Elementen einzufügen:

```
WRITE PERSONNEL-ID 5X NAME 3X JOB-TITLE
```

Mit dem obigen Statement werden zwischen den Feldern PERSONNEL-ID und NAME 5 Leerstellen und zwischen NAME und JOB-TITLE 3 Leerstellen eingefügt.

Tabulator-Notation *nT*

Mit der Tabulator-Notation *nT*, die im DISPLAY- und im WRITE-Statement verwendet werden kann, können Sie die Spalte *n* bestimmen, ab der ein Ausgabe-Element ausgegeben werden soll.

```

** Example Program 'DISPLX05'
DEFINE DATA LOCAL
1 VIEWEMP VIEW OF EMPLOYEES
  2 NAME
  2 FIRST-NAME
END-DEFINE
READ (3) VIEWEMP BY NAME STARTING FROM 'JONES'
  DISPLAY 5T NAME 30T FIRST-NAME
END-READ
END

```

Das obige Programm erzeugt folgende Ausgabe, wobei das Feld NAME ab Spalte 5 (vom linken Seitenrand aus gezählt) und das Feld FIRST-NAME ab Spalte 30 ausgegeben wird:

Page	1	97-08-21	11:46:01
	NAME		FIRST-NAME
	-----		-----
	JONES		VIRGINIA
	JONES		MARSHA
	JONES		ROBERT

Zeilenvorschub — die Schrägstrich-Notation (/)

Mit einem Schrägstrich “/” in einem DISPLAY- oder WRITE-Statement bewirken Sie einen Zeilenvorschub.

- Bei einem DISPLAY-Statement bewirkt ein Schrägstrich einen Zeilenvorschub *zwischen Feldern* und *innerhalb von Text*.
- Bei einem WRITE-Statement bewirkt ein Schrägstrich nur *zwischen Feldern* einen Zeilenvorschub; innerhalb von Text wird er wie ein gewöhnliches Textzeichen behandelt.

Zwischen Feldern muß dem Schrägstrich je ein Leerzeichen vor- und nachgestellt werden.

Für mehrfachen Zeilenvorschub geben Sie mehrere Schrägstriche an.

Beispiel für Zeilenvorschub in DISPLAY-Statement

```
** Example Program 'DISPLX06'
DEFINE DATA LOCAL
1 VIEWEMP VIEW OF EMPLOYEES
  2 NAME
  2 FIRST-NAME
  2 DEPARTMENT
END-DEFINE
READ (3) VIEWEMP BY NAME STARTING FROM 'JONES'
  DISPLAY NAME / FIRST-NAME 'DEPART-/MENT' DEPARTMENT
END-READ
END
```

Das obige DISPLAY-Statement erzeugt einen Zeilenvorschub nach jedem Wert des Feldes NAME sowie innerhalb des Textes “DEPART-MENT”:

NAME FIRST-NAME	DEPART- MENT
JONES VIRGINIA	SALE
JONES MARSHA	MGMT
JONES ROBERT	TECH

Beispiel für Zeilenvorschub in WRITE-Statement:

```

** Example Program 'WRITEX02'
DEFINE DATA LOCAL
1 VIEWEMP VIEW OF EMPLOYEES
  2 NAME
  2 FIRST-NAME
  2 DEPARTMENT
END-DEFINE
READ (3) VIEWEMP BY NAME STARTING FROM 'JONES'
  WRITE NAME / FIRST-NAME 'DEPART-/MENT' DEPARTMENT //
END-READ
END

```

Das obige WRITE-Statement erzeugt einen Zeilenvorschub nach jedem Wert des Feldes NAME und einen doppelten Zeilenvorschub nach jedem Wert des Feldes DEPARTMENT, aber keinen innerhalb des Textes "DEPART-/MENT":

Page	1	97-08-14	11:45:12
JONES			
VIRGINIA	DEPART-/MENT	SALE	
JONES			
MARSHA	DEPART-/MENT	MGMT	
JONES			
ROBERT	DEPART-/MENT	TECH	

Weitere Beispiele für DISPLAY- und WRITE-Statements:

Siehe Programme DISPLX13, WRITEX08, DISPLX14, WRITEX09 und DISPLX21 in Library SYSEXP.

Index-Notation ($n:n$) für multiple Felder und Periodengruppen

Mit einer Index-Notation ($n:n$) können Sie angeben, wieviele Werte eines multiplen Feldes bzw. wieviele Ausprägungen einer Periodengruppe ausgegeben werden sollen.

Beispiel: Das Feld INCOME im DDM EMPLOYEES ist eine Periodengruppe und enthält das jährliche Einkommen eines Mitarbeiters für jedes Jahr der Betriebszugehörigkeit. Die Daten werden in chronologischer Reihenfolge gespeichert, wobei das Einkommen des jeweils letzten Jahres in der Ausprägung "1" zu finden ist.

Wollen Sie das Jahreseinkommen eines Mitarbeiters in den letzten drei Jahren angezeigt bekommen — d.h. Ausprägungen "1" bis "3" —, fügen Sie im WRITE- bzw. DISPLAY-Statement hinter dem betreffenden Feldnamen die Notation "(1:3)" ein (wie im folgenden Beispielprogramm gezeigt).

Beispiel für Index-Notation in DISPLAY-Statement:

```

** Example Program 'DISPLX07'
DEFINE DATA LOCAL
1 VIEWEMP VIEW OF EMPLOYEES
  2 PERSONNEL-ID
  2 NAME
  2 BIRTH
  2 INCOME (1:3)
    3 CURR-CODE
    3 SALARY
    3 BONUS (1:1)
END-DEFINE
READ (3) VIEWEMP BY BIRTH
  DISPLAY PERSONNEL-ID NAME INCOME (1:3)
  SKIP 1
END-READ
END

```

Wenn mehrere Werte eines multiplen Feldes über ein DISPLAY-Statement ausgegeben werden, werden diese, wie Sie sehen, untereinander ausgegeben:

Page		1	99-01-22 11:36:58		
PERSONNEL ID	NAME	INCOME			
		CURRENCY CODE	ANNUAL SALARY	BONUS	
30020013	GARRET	UKL	4200	0	
		UKL	4150	0	
			0	0	
30016112	TAILOR	UKL	7450	0	
		UKL	7350	0	
		UKL	6700	0	
20017600	PIETSCH	USD	22000	0	
		USD	20200	0	
		USD	18700	0	

Da bei Verwendung eines WRITE-Statements die Werte nebeneinander (statt untereinander) ausgegeben werden, kann dies eventuell einen — möglicherweise unerwünschten — automatischen Zeilenvorschub auslösen.

Wenn Sie statt einer ganzen Periodengruppe nur ein Feld (z.B. SALARY) aus einer Periodengruppe benutzen und, wie im folgenden Beispiel zwischen NAME und JOB-TITLE, zusätzlich einen Zeilenvorschub “/” einfügen, wird der Report übersichtlicher:

Beispiel für Index-Notation in WRITE-Statement:

```

** Example Program 'WRITEX03'
DEFINE DATA LOCAL
1 VIEWEMP VIEW OF EMPLOYEES
  2 PERSONNEL-ID
  2 NAME
  2 BIRTH
  2 JOB-TITLE
  2 SALARY (1:3)
END-DEFINE
READ (3) VIEWEMP BY BIRTH
  WRITE PERSONNEL-ID NAME / JOB-TITLE SALARY (1:3)
  SKIP 1
END-READ
END

```

Page	1				99-01-22	11:37:18
30020013	GARRET					
TYPIST		4200	4150	0		
30016112	TAILOR					
WAREHOUSEMAN		7450	7350	6700		
20017600	PIETSCH					
SECRETARY		22000	20200	18700		

Seitenüberschriften und Seitenvorschübe

Dieser Abschnitt beschreibt verschiedene Möglichkeiten, wie Sie den Seitenumbruch in einem Report sowie die Ausgabe von Seitenüberschriften am Anfang jeder Seite des Reportes beeinflussen können.

- Standard-Seitenüberschrift
- Seitenüberschrift unterdrücken — die NOTITLE-Option
- Eigene Seitenüberschrift definieren — das WRITE TITLE-Statement
- Logische Seite und physische Seite
- Seitengröße — der PS-Parameter
- Seitenvorschub — der EJ-Parameter
- Seitenvorschub — die Statements EJECT und NEWPAGE
- Seiten-Fußzeile — das WRITE TRAILER-Statement
- AT TOP OF PAGE-Statement
- AT END OF PAGE-Statement

Standard-Seitenüberschrift

Natural generiert für jede über ein DISPLAY oder WRITE-Statement erzeugte Ausgabeseite automatisch eine Standard-Kopfzeile. Diese Kopfzeile enthält die Seitennummer sowie Datum und Uhrzeit.

```
WRITE 'HELLO'  
END
```

Das obige Programm erzeugt folgende Ausgabe mit einer Standard-Kopfzeile:

Page	1	97-08-14	18:27:35
HELLO			

Seitenüberschrift unterdrücken — die NOTITLE-Option

Falls Sie Ihren Report ohne Kopfzeile ausgeben möchten, geben Sie im DISPLAY- bzw. WRITE-Statement das Schlüsselwort “NOTITLE” an.

```
WRITE NOTITLE 'HELLO'  
END
```

Das obige Programm erzeugt folgende Ausgabe ohne Kopfzeile:

```
HELLO
```

Eigene Seitenüberschrift definieren — das WRITE TITLE-Statement

Wenn Sie statt der Natural-Standard-Kopfzeile eine eigene Kopfzeile ausgeben möchten, verwenden Sie dazu das Statement WRITE TITLE. Mit diesem Statement geben Sie (in Apostrophen) den Text Ihrer Kopfzeile an.

```
WRITE TITLE 'THIS IS MY PAGE TITLE'  
WRITE 'HELLO'  
END
```

HELLO

THIS IS MY PAGE TITLE

Mit der SKIP-Option des WRITE TITLE-Statements können Sie bestimmen, wieviele Leerzeilen unter der Kopfzeile ausgegeben werden sollen. Nach dem Schlüsselwort SKIP geben Sie die Anzahl der gewünschten Leerzeilen an:

```
WRITE TITLE 'THIS IS MY PAGE TITLE' SKIP 2  
WRITE 'HELLO'  
END
```

HELLO

THIS IS MY PAGE TITLE

SKIP kann nicht nur in einem WRITE TITLE-Statement, sondern auch als eigenständiges Statement verwendet werden.

Standardmäßig wird die Kopfzeile zentriert und ohne Unterstreichung ausgegeben. Das WRITE TITLE-Statement bietet Ihnen aber auch die Möglichkeit, eine Seitenüberschrift linksbündig (LEFT JUSTIFIED) und/oder unterstrichen (UNDERLINED) auszugeben.

```
WRITE TITLE LEFT JUSTIFIED UNDERLINED 'THIS IS MY PAGE TITLE' SKIP 2
WRITE 'HELLO'
END
```

THIS IS MY PAGE TITLE
HELLO

Standardmäßig werden Überschriften mit einem Bindestrich (–) unterstrichen. Mit dem Parameter UC können Sie allerdings auch ein anderes Zeichen angeben, das als Unterstreichungszeichen verwendet werden soll (wie weiter unten in diesem Kapitel beschrieben).

Das WRITE TITLE-Statement wird jedesmal ausgeführt, wenn eine neue Reportseite initiiert wird.

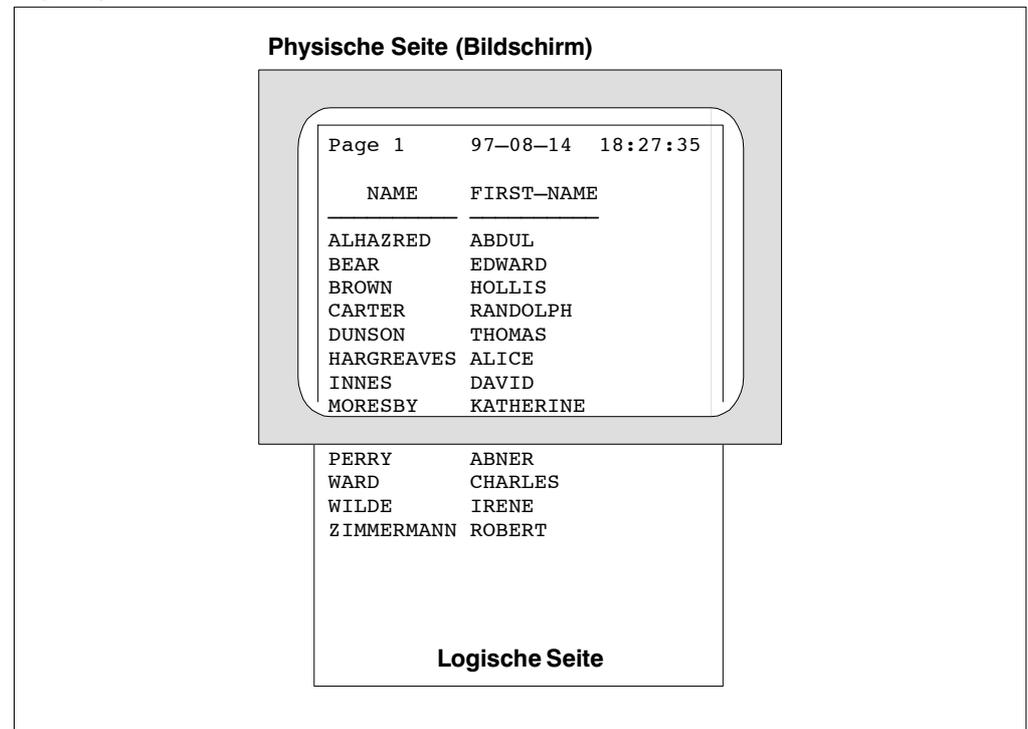
Logische Seite und physische Seite

Eine *logische Seite* ist die von einem Natural-Programm erzeugte Ausgabe.

Eine *physische Seite* ist Ihr Bildschirm, auf dem die Ausgabe angezeigt wird; oder es ist das Stück Papier, auf dem die Ausgabe ausgedruckt wird.

Die Länge der logischen Seite ergibt sich aus der Anzahl der vom Natural-Programm ausgegebenen Zeilen.

Falls mehr Zeilen ausgegeben werden als auf einen Bildschirm passen, ist die logische Seite länger als die physische Seite, und die restlichen Zeilen werden auf dem nächsten Schirm angezeigt.



Falls Informationen, die Sie unten auf dem Schirm anzeigen möchten (z.B. mit einem WRITE TRAILER- oder AT END OF PAGE-Statement erzeugte Ausgaben), erst auf dem nächsten Schirm ausgegeben werden, verkleinern Sie die logische Seitenlänge entsprechend (mit dem Session-Parameter PS, wie unten beschrieben).

Seitengröße — der PS-Parameter

Mit dem Parameter PS (Page Size) bestimmen Sie die maximale Anzahl der Zeilen einer (logischen) Ausgabeseite.

Wenn die Anzahl der mit dem PS-Parameter angegebenen Zeilen erreicht ist, dann erfolgt ein Seitenvorschub (es sei denn, der Seitenvorschub wird über ein NEWPAGE- oder ein EJECT-Statement gesteuert; siehe unten).

Der PS-Parameter kann entweder auf Session-Ebene mit dem Systemkommando GLOBALS gesetzt werden oder innerhalb eines Programms mit den folgenden Statements:

- auf Report-Ebene:

`FORMAT PS=nn`

- auf Statement-Ebene:

`DISPLAY (PS=nn)`

`WRITE (PS=nn)`

`WRITE TITLE (PS=nn)`

`WRITE TRAILER (PS=nn)`

`INPUT (PS=nn)`

Seitenvorschub — der EJ-Parameter

Mit dem Session-Parameter EJ bestimmen Sie, ob Seitenvorschübe ausgeführt werden sollen oder nicht. Standardmäßig gilt EJ=ON, d.h. Seitenvorschübe werden wie angegeben ausgeführt. Wenn Sie EJ=OFF angeben, werden Seitenvorschub-Informationen ignoriert. Dies kann bei Testläufen, bei denen Seitenumbrüche keine Rolle spielen, sinnvoll sein, um Papier zu sparen.

Der Seitenvorschub-Parameter EJ kann auf Session-Ebene mit dem Systemkommando GLOBALS gesetzt werden:

GLOBALS EJ=OFF

Seitenvorschub — die Statements EJECT und NEWPAGE

Das EJECT-Statement hat Priorität vor dem EJ-Parameter. Das EJECT-Statement bewirkt einen Seitenvorschub, *ohne* daß auf der neuen Seite eine Kopfzeile oder Standard-Seitenüberschrift generiert wird. An Seitenanfang und Seitenende gebundene Verarbeitungen wie WRITE TITLE, AT TOP OF PAGE, *PAGE-NUMBER, WRITE TRAILER oder AT END OF PAGE werden *nicht* ausgeführt.

Das NEWPAGE-Statement hingegen bewirkt einen Seitenvorschub *mit* Ausführung der für Seitenanfang und Seitenende festgelegten Verarbeitungen. Eine Fußzeile wird ausgegeben, falls spezifiziert; eine standardmäßige oder benutzerdefinierte Kopfzeile wird auf der neuen Seite ausgegeben (es sei denn, das betreffende DISPLAY- bzw. WRITE-Statement enthält die Option NOTITLE).

Wird kein NEWPAGE-Statement verwendet, so ergibt sich der Seitenvorschub aus der mit dem Parameter PS definierten Seitenlänge (siehe oben).

EJECT/NEWPAGE WHEN LESS THAN n LINES LEFT

Das NEWPAGE- wie das EJECT-Statement erlauben es, eine WHEN LESS THAN n LINES LEFT-Klausel anzugeben. Mit dieser Klausel geben Sie eine Zeilenanzahl n an; das NEWPAGE- bzw. EJECT-Statement wird dann nur ausgeführt, wenn zum Zeitpunkt der Verarbeitung des Statements weniger als n Zeilen auf der aktuellen Seite zur Verfügung stehen.

Beispiel:

```
...  
FORMAT PS=55  
...  
NEWPAGE WHEN LESS THAN 7 LINES LEFT  
...
```

In diesem Beispiel ist die Seitenlänge mit 55 Zeilen angegeben.

Sind zu dem Zeitpunkt, zu dem das NEWPAGE-Statement verarbeitet wird, auf der aktuellen Seite nur noch 6 oder weniger Zeilen übrig, wird das NEWPAGE-Statement ausgeführt. Sind 7 oder mehr übrig, wird es nicht ausgeführt, und der Seitenvorschub erfolgt in Abhängigkeit vom PS-Parameter, also nach 55 Zeilen.

NEWPAGE WITH TITLE

Das NEWPAGE-Statement bietet darüber hinaus eine WITH TITLE-Option. Ohne diese Option wird entweder die Standard-Kopfzeile ausgegeben oder ein WRITE TITLE-Statement bzw. eine NOTITLE-Option ausgeführt. Mit der WITH TITLE-Option können Sie für einen mit NEWPAGE ausgelösten Seitenvorschub eine eigene Kopfzeile ausgeben, die dann Priorität vor allen anderen Seitenüberschrift-Anweisungen hat. Die Syntax der WITH TITLE-Klausel entspricht der des WRITE TITLE-Statements.

Beispiel:

```
NEWPAGE WITH TITLE LEFT JUSTIFIED 'PEOPLE LIVING IN BOSTON:'
```

Das folgende Beispielprogramm zeigt die Verwendung des PS-Parameters und des NEWPAGE-Statements. Außerdem wird hier die Natural-Systemvariable *PAGE-NUMBER verwendet, die jeweils die aktuelle Seitenzahl enthält.

```

** Example Program 'NEWPAX01'
DEFINE DATA LOCAL
1 VIEWEMP OF EMPLOYEES
  2 NAME
  2 CITY
  2 DEPT
END-DEFINE
FORMAT PS=20
READ (5) VIEWEMP BY CITY STARTING FROM 'M'
  DISPLAY NAME 'DEPT' DEPT 'LOCATION' CITY
  AT BREAK OF CITY
    NEWPAGE WITH TITLE LEFT JUSTIFIED
    'EMPLOYEES BY CITY - PAGE:' *PAGE-NUMBER
  END-BREAK
END-READ
END

```

Beachten Sie, wann der Seitenvorschub erfolgt, sowie die Kopfzeile der neuen Seite:

Page	1	97-08-19	18:27:35
	NAME	DEPT	LOCATION
	-----	-----	-----
	FICKEN	TECH10	MADISON
	KELLOGG	TECH10	MADISON
	ALEXANDER	SALE20	MADISON

EMPLOYEES BY CITY - PAGE:	2		
	NAME	DEPT	LOCATION
	-----	-----	-----
	DE JUAN	SALE03	MADRID
	DE LA MADRID	PROD01	MADRID

Seiten-Fußzeile — das WRITE TRAILER-Statement

Mit dem Statement WRITE TRAILER können Sie einen Text (in Apostrophen) angeben, der als Fußzeile am Ende jeder Seite ausgegeben werden soll.

```
WRITE TRAILER 'THIS IS THE END OF THE PAGE'
```

Das Statement wird ausgeführt vor einem SKIP- oder NEWPAGE-Statement oder am Ende einer logischen Seite.

Die Prüfung, ob das Ende einer logischen Seite erreicht ist, erfolgt erst, *nachdem* ein WRITE- oder DISPLAY-Statement vollständig ausgeführt ist. Daher kann es vorkommen, daß der Umfang einer logischen Seite (d.h. die Anzahl der mit einem DISPLAY- bzw. WRITE-Statement ausgegebenen Zeilen) eine physische Seite überschreitet, bevor das WRITE TRAILER-Statement ausgeführt wird.

Um sicherzustellen, daß die Fußzeilen jeweils am Ende einer physischen Seite erscheinen, sollten Sie die logische Seitenlänge (Session-Parameter PS) so festlegen, daß sie entsprechend kleiner als die physische Seitenlänge ist.

Standardmäßig wird die Seitenunterschrift zentriert auf der Seite ausgegeben. Das WRITE TRAILER-Statement bietet Ihnen aber auch die Möglichkeit, eine Fußzeile linksbündig (LEFT JUSTIFIED) und/oder unterstrichen (UNDERLINED) auszugeben:

```
WRITE TRAILER LEFT JUSTIFIED UNDERLINED 'THIS IS THE END OF THE PAGE'
```

AT TOP OF PAGE-Statement

Mit dem Statement AT TOP OF PAGE können Sie eine beliebige Verarbeitung angeben, die jedesmal ausgeführt werden soll, wenn eine neue Reportseite beginnt.

Erzeugt das AT TOP OF PAGE-Statement eine Ausgabe, so wird diese unterhalb der Seiten-Kopfzeile (mit einer Leerzeile dazwischen) ausgegeben. Standardmäßig erfolgt die Ausgabe linksbündig auf der Seite.

AT END OF PAGE-Statement

Mit dem Statement AT END OF PAGE können Sie eine beliebige Verarbeitung angeben, die jedesmal ausgeführt werden soll, wenn das Ende einer Reportseite erreicht wird.

Erzeugt das AT END OF PAGE-Statement eine Ausgabe, so wird diese unterhalb der (mit dem WRITE TRAILER-Statement angegebenen) Seiten-Fußzeile ausgegeben. Standardmäßig erfolgt die Ausgabe linksbündig auf der Seite.

Dieselben Überlegungen bezüglich logischer und physischer Seitenlängen, die für das WRITE TRAILER-Statement gelten (vgl. oben), treffen auch auf das AT END OF PAGE-Statement zu.

Weitere Beispiele für die Statements WRITE TITLE, WRITE TRAILER, AT TOP OF PAGE, AT END OF PAGE und SKIP:

Siehe Programme WTITLX01, DISPLX21, ATENPX01, ATTOPX01, SKIPX01 und SKIPX02 in Library SYSEXPB.

Weiteres Beispiel für NOTITLE-Option:

Siehe Programm DISPLX20 in Library SYSEXPB.

Weiteres Beispiel für NEWPAGE- und EJECT-Statements:

Siehe Programm NEWPAX02 in Library SYSEXPB.

Spaltenüberschriften

Dieser Abschnitt beschreibt verschiedene Möglichkeiten, wie Sie die Anzeige der von einem DISPLAY-Statement erzeugten Spaltenüberschriften beeinflussen können.

- Standard-Spaltenüberschriften
- Standard-Spaltenüberschriften unterdrücken — die NOHDR-Option
- Eigene Spaltenüberschriften definieren
- NOTITLE und NOHDR kombinieren
- Spaltenüberschriften zentrieren — der HC-Parameter
- Breite von Spaltenüberschriften — der HW-Parameter
- Füllzeichen für Überschriften — die Parameter FC und GC
- Unterstreichungszeichen für Überschriften — der UC-Parameter
- Spaltenüberschriften unterdrücken — die Notation `'/'`

Standard-Spaltenüberschriften

Standardmäßig wird jedes mit einem DISPLAY-Statement ausgegebene Datenbankfeld mit einer (im DDM für das Feld definierten) Standard-Spaltenüberschrift ausgegeben.

```
** Example Program 'DISPLX01'  
DEFINE DATA LOCAL  
1 VIEWEMP VIEW OF EMPLOYEES  
  2 PERSONNEL-ID  
  2 NAME  
  2 BIRTH  
  2 JOB-TITLE  
END-DEFINE  
READ (3) VIEWEMP BY BIRTH  
  DISPLAY PERSONNEL-ID NAME JOB-TITLE  
END-READ  
END
```

Das obige Beispielprogramm verwendet Standard-Spaltenüberschriften und erzeugt folgende Ausgabe:

Page	1		99-01-22	11:31:01
PERSONNEL ID	NAME		CURRENT POSITION	
30020013	GARRET		TYPIST	
30016112	TAILOR		WAREHOUSEMAN	
20017600	PIETSCH		SECRETARY	

Standard-Spaltenüberschriften unterdrücken — die NOHDR-Option

Wünschen Sie in Ihrem Report keine Spaltenüberschriften, geben Sie im DISPLAY-Statement das Schlüsselwort “NOHDR” an.

```
DISPLAY NOHDR PERSONNEL-ID NAME JOB-TITLE
```

Eigene Spaltenüberschriften definieren

Wenn Sie statt der Standard-Spaltenüberschriften eigene Spaltenüberschriften ausgeben möchten, geben Sie unmittelbar vor dem jeweiligen Feld *'Text'* (in Apostrophen) an, wobei *Text* die für das Feld zu verwendende Spaltenüberschrift ist.

```
** Example Program 'DISPLX08'  
DEFINE DATA LOCAL  
1 VIEWEMP VIEW OF EMPLOYEES  
  2 PERSONNEL-ID  
  2 NAME  
  2 BIRTH  
  2 JOB-TITLE  
END-DEFINE  
READ (3) VIEWEMP BY BIRTH  
  DISPLAY PERSONNEL-ID  
          'EMPLOYEE' NAME  
          'POSITION' JOB-TITLE  
END-READ  
END
```

Das obige Programm enthält für das Feld NAME die Spaltenüberschrift "EMPLOYEE" und für das Feld JOB-TITLE die Spaltenüberschrift "POSITION"; für das Feld PERSONNEL-ID wird die Standard-Spaltenüberschrift verwendet. Das Programm erzeugt folgende Ausgabe:

Page	1		99-01-22	11:39:53
PERSONNEL ID	EMPLOYEE		POSITION	
30020013	GARRET		TYPIST	
30016112	TAILOR		WAREHOUSEMAN	
20017600	PIETSCH		SECRETARY	

NOTITLE und NOHDR kombinieren

Zur Ausgabe eines Reports ohne Kopfzeilen und Spaltenüberschriften geben Sie die Optionen NOTITLE und NOHDR gleichzeitig an, und zwar in der folgenden Reihenfolge:

```
DISPLAY NOTITLE NOHDR PERSONNEL-ID NAME JOB-TITLE
```

Spaltenüberschriften zentrieren — der HC-Parameter

Standardmäßig werden Spaltenüberschriften zentriert über den Spalten ausgegeben. Mit dem Parameter HC (Header Centering) können Sie die Ausrichtung der Spaltenüberschriften beeinflussen:

- Wenn Sie **HC=L** angeben, werden die Spaltenüberschriften linksbündig ausgerichtet.
- Wenn Sie **HC=R** angeben, werden die Spaltenüberschriften rechtsbündig ausgerichtet.
- Wenn Sie **HC=C** angeben, werden die Spaltenüberschriften zentriert.

Sie können den HC-Parameter in einem FORMAT-Statement angeben; er gilt dann für den gesamten Report. Sie können ihn auch in einem DISPLAY-Statement angeben, und zwar sowohl auf Statement- wie auf Feldebene.

```
DISPLAY (HC=L) PERSONNEL-ID NAME JOB-TITLE
```

Breite von Spaltenüberschriften — der HW-Parameter

Mit dem Parameter HW (Header Width) bestimmen Sie die Breite einer von einem DISPLAY-Statement erzeugten Spalte.

- Standardmäßig gilt HW=ON, d.h. die Breite einer DISPLAY-Spalte wird entweder durch die Länge des Feldes oder durch die Länge der Spaltenüberschrift bestimmt, je nachdem was länger ist.
- Wenn Sie HW=OFF angeben, wird die Breite einer DISPLAY-Spalte allein durch die Länge des Feldes bestimmt. Bitte beachten Sie, daß HW=OFF nur bei DISPLAY-Statements, die *keine* Spaltenüberschriften erzeugen, wirkt; d.h. bei einem ersten DISPLAY-Statement mit NOHDR-Option, oder bei einem nachfolgenden DISPLAY-Statement (vgl. *Natural-Referenzhandbuch*).

Sie können den HW-Parameter in einem FORMAT-Statement verwenden; er gilt dann für den gesamten Report. Sie können ihn auch in einem DISPLAY-Statement angeben, und zwar sowohl auf Statement- wie auf Feldebene.

Füllzeichen für Überschriften — die Parameter FC und GC

Mit dem Parameter FC (Filler Character) bestimmen Sie das *Füllzeichen*, das auf beiden Seiten der von einem DISPLAY-Statement erzeugten *Überschrift* über die gesamte Breite der Spalte erscheint. Voraussetzung ist, daß die Spaltenbreite durch die Feldlänge und nicht durch die Überschrift bestimmt wird (vgl. HW-Parameter oben), sonst hat der FC-Parameter keine Wirkung.

Wenn eine Feldgruppe oder eine Periodengruppe mit einem DISPLAY-Statement ausgegeben wird, wird eine *Gruppenüberschrift* über den Überschriften der einzelnen Felder der Gruppe ausgegeben. Mit dem Parameter GC (Group Filler Character) bestimmen Sie das *Füllzeichen*, das auf beiden Seiten der Gruppenüberschrift erscheinen soll.

Während der FC-Parameter für Überschriften einzelner Felder gilt, bezieht sich der GC-Parameter auf Überschriften für Feldgruppen.

Sie können die Parameter FC und GC in einem FORMAT-Statement verwenden; sie gelten dann für den gesamten Report. Sie können sie auch in einem DISPLAY-Statement angeben, und zwar sowohl auf Statement- wie auf Feldebene.

```
** Example Program 'FORMAX01'  
DEFINE DATA LOCAL  
1 VIEWEMP VIEW OF EMPLOYEES  
  2 NAME  
  2 INCOME (1:1)  
    3 CURR-CODE  
    3 SALARY  
  3 BONUS (1:1)  
END-DEFINE  
FORMAT FC=* GC=$  
READ (3) VIEWEMP BY NAME  
  DISPLAY NAME (FC==) INCOME (1)  
END-READ  
END
```

Das obige Programm erzeugt folgende Ausgabe:

=====NAME=====	\$\$\$\$\$\$\$\$\$\$\$INCOME\$\$\$\$\$\$\$\$\$\$\$		
	CURRENCY	**ANNUAL**	**BONUS**
	CODE	SALARY	
ABELLAN	PTA	1450000	0
ACHIESON	UKL	10500	0
ADAM	FRA	159980	23000

Unterstreichungszeichen für Überschriften — der UC-Parameter

Standardmäßig werden Kopfzeilen und Überschriften mit einem Bindestrich (–) unterstrichen.

Mit dem Parameter UC (Underlining Character) können Sie ein anderes Zeichen bestimmen, das als Unterstreichungszeichen verwendet werden soll.

Sie können den UC-Parameter in einem FORMAT-Statement verwenden; er gilt dann für den gesamten Report. Sie können ihn auch in einem DISPLAY-Statement angeben, und zwar sowohl auf Statement- wie auf Feldebene.

```

** Example Program 'FORMAX02'
DEFINE DATA LOCAL
1 VIEWEMP VIEW OF EMPLOYEES
  2 PERSONNEL-ID
  2 NAME
  2 BIRTH
  2 JOB-TITLE
END-DEFINE
FORMAT UC==
WRITE TITLE LEFT JUSTIFIED UNDERLINED 'EMPLOYEES REPORT' SKIP 1
READ (3) VIEWEMP BY BIRTH
  DISPLAY PERSONNEL-ID (UC=*) NAME JOB-TITLE
END-READ
END

```

Im obigen Programm ist der UC-Parameter auf Programmebene und auf Feldebene gesetzt: das im FORMAT-Statement angegebene Unterstreichungszeichen (=) gilt für den ganzen Report — außer für das Feld PERSONNEL-ID, für das ein anderes Unterstreichungszeichen (*) angegeben ist. Das Programm erzeugt folgende Ausgabe:

EMPLOYEES REPORT		
=====		
PERSONNEL ID	NAME	CURRENT POSITION
*****	=====	=====
30020013	GARRET	TYPIST
30016112	TAILOR	WAREHOUSEMAN
20017600	PIETSCH	SECRETARY

Spaltenüberschriften unterdrücken — die Notation `'/'`

Mit der Notation Apostroph-Schrägstrich-Apostroph (`'/'`) können Sie die Ausgabe von Standard-Spaltenüberschriften für einzelne Felder in einem `DISPLAY`-Statement unterdrücken. Im Gegensatz zur `NOHDR`-Option, mit der die Ausgabe von Standard-Spaltenüberschriften für sämtliche Spalten unterdrückt werden kann, kann dies mit der `'/'`-Notation für eine einzelne Spalte erreicht werden.

Die Notation wird in einem `DISPLAY`-Statement jeweils unmittelbar vor dem Namen des Feldes angegeben, dessen Spaltenüberschrift unterdrückt werden soll.

Vergleichen Sie hierzu folgende Beispiele:

```
DISPLAY NAME PERSONNEL-ID JOB-TITLE
```

In diesem Fall werden die Standardüberschriften aller drei Spalten ausgegeben:

Page	1	97-04-19 17:37:27	
NAME	PERSONNEL ID	CURRENT POSITION	
ABELLAN	60008339	MAQUINISTA	
ACHIESON	30000231	DATA BASE ADMINISTRATOR	
ADAM	50005800	CHEF DE SERVICE	
ADKINSON	20008800	PROGRAMMER	
ADKINSON	20009800	DBA	
ADKINSON	20011000	SALES PERSON	

```
DISPLAY '/' NAME PERSONNEL-ID JOB-TITLE
```

In diesem Fall wird mit der Notation '/' die Spaltenüberschrift für das Feld NAME unterdrückt:

Page	1	97-04-19 17:38:45	
	PERSONNEL ID	CURRENT POSITION	
ABELLAN	60008339	MAQUINISTA	
ACHIESON	30000231	DATA BASE ADMINISTRATOR	
ADAM	50005800	CHEF DE SERVICE	
ADKINSON	20008800	PROGRAMMER	
ADKINSON	20009800	DBA	
ADKINSON	20011000	SALES PERSON	

Weitere Beispiele für Spaltenüberschriften:

Siehe Programme DISPLX15 und DISPLX16 in Library SYSEXP.

Parameter zur Beeinflussung der Ausgabe von Feldern

Natural bietet eine Reihe von Parametern, mit denen Sie die Art, in der Felder ausgegeben werden, beeinflussen können:

- Mit den Parametern LC, IC und TC können Sie Zeichen angeben, die vor bzw. nach einem Feld bzw. vor einem Feldwert angezeigt werden sollen.
- Mit den Parametern AL und NL können Sie die Ausgabelänge von Feldern vergrößern oder verkleinern.
- Mit dem Parameter SG können Sie bestimmen, ob negative Werte mit oder ohne Minuszeichen angezeigt werden sollen.
- Mit dem Parameter IS können Sie die Anzeige von Feldwerten, die mit dem jeweils vorigen Feldwert identisch sind, unterdrücken.
- Mit dem Parameter ZP können Sie bestimmen, ob Feldwerte, die “0” sind, angezeigt werden sollen oder nicht.
- Mit dem Parameter ES können Sie die Anzeige von Leerzeilen, die von einem DISPLAY- oder WRITE-Statement generiert werden, unterdrücken.

Die folgenden Themen werden nachfolgend erörtert:

- Vorangestellte Zeichen — der LC-Parameter
- Einfügezeichen — der IC-Parameter
- Nachgestellte Zeichen — der TC-Parameter
- Ausgabelänge — der AL- und NL-Parameter
- Vorzeichen-Stelle — der SG-Parameter
- Ausgabe identischer Werte unterdrücken — der IS-Parameter
- Nullwerte anzeigen — der ZP-Parameter
- Leerzeilenunterdrückung — der ES-Parameter

Vorangestellte Zeichen — der LC-Parameter

Mit dem Parameter LC (Leading Characters) geben Sie an, welche Zeichen *unmittelbar vor einem Feld* ausgegeben werden, das von einem DISPLAY-Statement ausgegeben wird. Die Breite der Ausgabespalte wird entsprechend vergrößert. Sie können 1 bis 10 Zeichen angeben.

Standardmäßig sind die Werte in alphanumerischen Feldern linksbündig ausgerichtet und die Werte in numerischen Feldern rechtsbündig. (Mit dem AD-Parameter können Sie diese Ausrichtung ändern; vgl. *Natural-Referenzhandbuch*). Bei einem alphanumerischen Feld erscheint ein vorangestelltes Zeichen daher unmittelbar vor dem Feldwert; bei einem numerischen Feld kann es dagegen vorkommen, daß zwischen dem LC-Zeichen und dem Feldwert Leerstellen bleiben.

Der LC-Parameter kann mit den Statements FORMAT und DISPLAY verwendet werden, und zwar sowohl auf Statement- wie auf Feldebene.

Einfügungszeichen — der IC-Parameter

Mit dem Parameter IC (Insertion Characters) geben Sie an, welche Zeichen *unmittelbar vor einem Feldwert* eingefügt werden, der von einem DISPLAY-Statement ausgegeben wird. Sie können 1 bis 10 Zeichen angeben.

Bei einem numerischen Feld werden die Einfügungszeichen unmittelbar vor die erste signifikante Stelle, die ausgegeben wird, gestellt, und zwar ohne Leerstellen zwischen dem Einfügungszeichen und dem Feldwert. Bei alphanumerischen Feldern hat der IC-Parameter die gleiche Wirkung wie der LC-Parameter.

Die Parameter LC und IC dürfen nicht beide gleichzeitig für ein Feld angegeben werden.

Der IC-Parameter kann mit den Statements FORMAT und DISPLAY verwendet werden, und zwar sowohl auf Statement- wie auf Feldebene.

Nachgestellte Zeichen — der TC-Parameter

Mit dem Parameter TC (Trailing Characters) geben Sie an, welche Zeichen *unmittelbar hinter einem Feld* ausgegeben werden, das von einem DISPLAY-Statement ausgegeben wird. Die Breite der Ausgabespalte wird entsprechend vergrößert. Sie können 1 bis 10 Zeichen angeben.

Der TC-Parameter kann mit den Statements FORMAT und DISPLAY verwendet werden, und zwar sowohl auf Statement- wie auf Feldebene.

Ausgabelänge — der AL- und NL-Parameter

Mit dem AL-Parameter bestimmen Sie die *Ausgabelänge* eines alphanumerischen Feldes; mit dem NL-Parameter bestimmen Sie die *Ausgabelänge* eines numerischen Feldes. Diese Länge ist die Länge, in der das Feld ausgegeben wird, und kann kürzer oder länger sein als die tatsächliche Länge des Feldes (die für ein Datenbankfeld im DDM bzw. für eine Benutzervariable im DEFINE DATA-Statement definiert ist).

Beide Parameter können mit den Statements FORMAT, DISPLAY, WRITE und INPUT verwendet werden, und zwar auf Statement- und Feldebene.

Anmerkung:

Wenn eine Editiermaske angegeben ist, gilt diese vor einer AL- bzw. NL-Angabe. Editiermasken sind auf Seite 153 beschrieben.

Vorzeichen-Stelle — der SG-Parameter

Mit dem Parameter SG (Sign Position) bestimmen Sie, ob numerische Felder eine zusätzliche Stelle zur Anzeige des Vorzeichens erhalten sollen.

- Standardmäßig gilt SG=ON, d.h. numerische Felder erhalten eine Vorzeichen-Stelle.
- Wenn Sie SG=OFF angeben, werden negative Werte in numerischen Feldern ohne Minuszeichen (-) ausgegeben.

Der SG-Parameter kann mit den Statements FORMAT, DISPLAY, WRITE und INPUT verwendet werden, und zwar auf Statement- und Feldebene.

Anmerkung:

Wenn eine Editiermaske angegeben ist, gilt diese vor einer SG-Angabe. Editiermasken sind auf Seite 153 beschrieben.

Beispielprogramm ohne Parameter:

```

** Example Program 'FORMAX03'
DEFINE DATA LOCAL
1 VIEWEMP VIEW OF EMPLOYEES
  2 NAME
  2 FIRST-NAME
  2 SALARY (1:1)
  2 BONUS (1:1,1:1)
END-DEFINE
READ (5) VIEWEMP BY NAME STARTING FROM 'JONES'
  DISPLAY NAME FIRST-NAME
    SALARY (1:1) BONUS (1:1,1:1)
END-READ
END

```

Das obige Programm enthält keine Parameterangaben und erzeugt folgende Ausgabe:

Page	1	97-08-15 17:25:19	
NAME	FIRST-NAME	ANNUAL SALARY	BONUS
JONES	VIRGINIA	46000	9000
JONES	MARSHA	50000	0
JONES	ROBERT	31000	0
JONES	LILLY	24000	0
JONES	EDWARD	37600	0

Beispielprogramm mit Parametern AL, NL, LC, IC und TC:

```

** Example Program 'FORMAX04'
DEFINE DATA LOCAL
1 VIEWEMP VIEW OF EMPLOYEES
  2 NAME
  2 FIRST-NAME
  2 SALARY (1:1)
  2 BONUS (1:1,1:1)
END-DEFINE
FORMAT AL=10 NL=6
READ (5) VIEWEMP BY NAME STARTING FROM 'JONES'
  DISPLAY NAME (LC=*) FIRST-NAME (TC=*)
    SALARY (1:1)(IC=$) BONUS (1:1,1:1)(LC=>)
END-READ
END

```

Das obige Programm erzeugt folgende Ausgabe. Vergleichen Sie sie mit der Ausgabe des vorigen Programms, um zu sehen, wie sich die einzelnen Parameter auswirken:

NAME	FIRST-NAME	ANNUAL SALARY	BONUS
*JONES	VIRGINIA	* \$46000 >	9000
*JONES	MARSHA	* \$50000 >	0
*JONES	ROBERT	* \$31000 >	0
*JONES	LILLY	* \$24000 >	0
*JONES	EDWARD	* \$37600 >	0

Wie Sie im obigen Beispiel sehen, schließt eine mit einem AL- oder NL-Parameter angegebene Ausgabelänge die mit einem LC-, IC- oder TC-Parameter angegebenen Zeichen nicht mit ein: die Breite der NAME-Spalte ist z.B. 11 Stellen — 10 für den Feldwert (AL=10) plus 1 vorangestelltes Zeichen.

Die Spalten SALARY und BONUS sind jeweils 8 Stellen breit — 6 Stellen für den Feldwert (NL=6), plus 1 vorangestelltes bzw. eingefügtes Zeichen, plus 1 Vorzeichen-Stelle (da SG=ON gilt).

Ausgabe identischer Werte unterdrücken — der IS-Parameter

Mit dem Parameter IS (Identical Suppress) können Sie die mehrmalige Ausgabe identischer Werte in aufeinanderfolgenden Zeilen, die von einem WRITE- oder DISPLAY-Statement erzeugt werden, unterdrücken.

- Standardmäßig gilt IS=OFF. Dies bedeutet, daß identische Werte angezeigt werden.
- Ist IS=ON gesetzt, wird ein Wert, der identisch mit dem vorherigen Wert des Feldes ist, nicht angezeigt.

Sie können den IS-Parameter in einem FORMAT-Statement angeben; er gilt dann für den gesamten Report. Sie können ihn auch in einem DISPLAY- oder WRITE-Statement angeben, und zwar sowohl auf Statement- wie auf Feldebene.

Die Wirkung des Parameters IS=ON kann für einen Datensatz mit dem Statement SUSPEND IDENTICAL SUPPRESS ausgesetzt werden. Näheres zu diesem Statement finden Sie im *Natural Statements-Handbuch*.

Vergleichen Sie die Ausgaben der beiden folgenden Beispielprogramme miteinander, um die Wirkung des IS-Parameters zu sehen. Im zweiten Programm wird die Anzeige identischer Werte im Feld NAME unterdrückt.

Beispielprogramm ohne IS-Parameter:

```
** Example Program 'FORMAX05'
DEFINE DATA LOCAL
1 VIEWEMP VIEW OF EMPLOYEES
  2 NAME
  2 FIRST-NAME
END-DEFINE
READ (3) VIEWEMP BY NAME STARTING FROM 'JONES'
  DISPLAY NAME FIRST-NAME
END-READ
END
```

Page	1	97-08-18 17:25:19
	NAME	FIRST-NAME
	JONES	VIRGINIA
	JONES	MARSHA
	JONES	ROBERT

Beispielprogramm mit IS-Parameter:

```
** Example Program 'FORMAX06'  
DEFINE DATA LOCAL  
1 VIEWEMP VIEW OF EMPLOYEES  
  2 NAME  
  2 FIRST-NAME  
END-DEFINE  
FORMAT IS=ON  
READ (3) VIEWEMP BY NAME STARTING FROM 'JONES'  
  DISPLAY NAME FIRST-NAME  
END-READ  
END
```

Page	1	97-08-18	17:26:02
	NAME	FIRST-NAME	
	-----	-----	
JONES		VIRGINIA MARSHA ROBERT	

Nullwerte anzeigen — der ZP-Parameter

Mit dem Parameter `ZP` (Zero Printing) bestimmen Sie wie ein Feldwert, der Null ist, ausgegeben wird.

- Standardmäßig gilt `ZP=ON`, d.h. für einen Feldwert, der Null ist, wird eine “0” (bei numerischen Feldern) bzw. der ganze Feldwert (bei Zeitfeldern) ausgegeben.
- Wenn Sie `ZP=OFF` angeben, wird ein Feldwert, der Null ist, gar nicht ausgegeben.

Sie können den `ZP`-Parameter in einem `FORMAT`-Statement angeben; er gilt dann für den gesamten Report. Sie können ihn auch in einem `DISPLAY`- oder `WRITE`-Statement angeben, und zwar sowohl auf Statement- wie auf Feldebene.

Leerzeilenunterdrückung — der ES-Parameter

Mit dem Parameter ES (Empty Line Suppression) können Sie die Ausgabe von mit einem DISPLAY- oder WRITE-Statement erzeugten Leerzeilen unterdrücken.

- Standardmäßig gilt ES=OFF. Dies bedeutet, daß alle Zeilen, die Leerwerte enthalten, angezeigt werden.
- Wenn Sie ES=ON angeben, wird eine mit einem DISPLAY- oder WRITE-Statement erzeugte Zeile, die nur Leerwerte enthält, unterdrückt. Die Verwendung des ES-Parameters empfiehlt sich, wenn bei der Ausgabe von multiplen Feldern oder Periodengruppen die Ausgabe vieler Leerzeilen zu erwarten ist.

Sie können den ES-Parameter in einem FORMAT-Statement angeben; er gilt dann für den gesamten Report. Sie können ihn auch in einem DISPLAY- oder WRITE-Statement angeben, und zwar auf Statement-Ebene.

Anmerkung:

Um die Leerwertunterdrückung auch für numerische Werte zu erhalten, muß für die betreffenden Felder neben ES=ON auch der Parameter ZP=OFF gesetzt werden, was bewirkt, daß Nullwerte in Leerwerte umgesetzt und dann ebenfalls nicht ausgegeben werden.

Vergleichen Sie die Ausgaben der beiden folgenden Beispielprogramme miteinander, um die Wirkung der Parameter ZP und ES zu sehen.

Beispielprogramm ohne Parameter ZP und ES:

```

** Example Program 'FORMAX07'
DEFINE DATA LOCAL
1 VIEWEMP VIEW OF EMPLOYEES
  2 NAME
  2 FIRST-NAME
  2 BONUS (1:2,1:1)
END-DEFINE
READ (4) VIEWEMP BY NAME STARTING FROM 'JONES'
  DISPLAY NAME FIRST-NAME BONUS (1:2,1:1)
END-READ
END

```

NAME	FIRST-NAME	BONUS
JONES	VIRGINIA	9000
JONES	MARSHA	6750
JONES	ROBERT	0
JONES	LILLY	0
		0

Beispielprogramm mit den Parametern ZP und ES:

```
** Example Program 'FORMAX08'  
DEFINE DATA LOCAL  
1 VIEWEMP VIEW OF EMPLOYEES  
  2 NAME  
  2 FIRST-NAME  
  2 BONUS (1:2,1:1)  
END-DEFINE  
FORMAT ES=ON  
READ (4) VIEWEMP BY NAME STARTING FROM 'JONES'  
  DISPLAY NAME FIRST-NAME BONUS (1:2,1:1)(ZP=OFF)  
END-READ  
END
```

Page	1		97-08-18 17:27:12
	NAME	FIRST-NAME	BONUS
	JONES	VIRGINIA	9000 6750
	JONES	MARSHA	
	JONES	ROBERT	
	JONES	LILLY	

**Weitere Beispiele für die Parameter LC, IC, TC, AL, NL, IS, ZP und ES
und das SUSPEND IDENTICAL SUPPRESS-Statement:**

Siehe Programme DISPLX17, DISPLX18, DISPLX19, SUSPEX01, SUSPEX02 und
COMPRX03 in Library SYSEXP.

Editiermasken — der EM-Parameter

Der EM-Parameter wird dazu verwendet, für ein numerisches oder alphanumerisches Feld eine sogenannte *Editiermaske* anzugeben, d.h. das Format, in dem die Feldwerte ausgegeben werden sollen, Zeichen für Zeichen festzulegen.

Beispiel:

```
DISPLAY NAME (EM=X^X^X^X^X^X^X^X^X^X)
```

In diesem Beispiel steht jedes “X” für ein Zeichen eines ausgegebenen alphanumerischen Feldwertes und jedes “^” für eine Leerstelle. Der Name “JOHNSON” würde in diesem Fall wie folgt ausgegeben:

```
J O H N S O N
```

Der EM-Parameter kann auf Programm-Ebene (in einem FORMAT-Statement), auf Statement-Ebene (in einem DISPLAY-, WRITE-, INPUT-, MOVE EDITED- oder PRINT-Statement) oder auf Feldebene (in einem DISPLAY-, WRITE- oder INPUT-Statement) angegeben werden.

Eine mit dem EM-Parameter definierte Editiermaske hat Vorrang vor einer im DDM für das betreffende Feld definierten Standard-Editiermaske. Falls EM=OFF gesetzt worden ist, wird überhaupt keine Editiermaske verwendet. Eine auf Feldebene definierte Editiermaske hat Vorrang vor einer auf Statement-Ebene definierten Editiermaske, welche wiederum Vorrang vor einer auf Programm-Ebene definierten Editiermaske hat.

Die folgenden Themen werden nachfolgend erörtert:

- Editiermasken für numerische Felder
- Editiermasken für alphanumerische Felder
- Länge der Felder
- Editiermasken für Datums- und Zeitfelder
- Beispiele für Editiermasken

Editiermasken für numerische Felder

Bei Editiermasken für numerische Felder (Formate N, I, P, F) geben Sie für jede auszugebende Ziffer eine “9” an, und ein “Z” für jede Ziffer, die nur ausgegeben werden soll, wenn sie nicht 0 ist. Ein Dezimalkomma wird durch einen Punkt “.” angegeben. Stellen nach dem Komma dürfen nicht mit “Z” angegeben werden. Weitere Zeichen dürfen vor- oder nachgestellt oder eingefügt werden, z.B. Vorzeichen.

Editiermasken für alphanumerische Felder

Editiermasken für alphanumerische Felder müssen für jedes auszugebende alphanumerische Zeichen ein “X” enthalten. Auch hier dürfen weitere Zeichen (bis auf einige Ausnahmen) vor-, nachgestellt oder hinzugefügt werden (in Apostrophen oder ohne).

Leerstellen in numerischen wie alphanumerischen Feldern werden mit “^” gekennzeichnet.

Länge der Felder

Wenn Sie für ein Feld eine Editiermaske definieren, beachten Sie bitte die Länge des Feldes. Ist die Editiermaske länger als das Feld, hat dies unvorhersehbare Auswirkungen. Ist die Editiermaske kürzer als das Feld, kann es sein, daß ein Feldwert nur unvollständig ausgegeben wird.

Beispiele:

Nehmen wir an, ein alphanumerisches Feld ist 12 Stellen lang und der ausgegebene Feldwert ist “JOHNSON”, dann würden folgende Editiermasken in folgenden Ausgaben resultieren:

EM=X.X.X.X.X Ausgabe: J.O.H.N.S

EM=***XXXXXX*** Ausgabe: ***JOHNSO**

Editiermasken für Datums- und Zeitfelder

Editiermasken für Datumsfelder können die Zeichen “D” für Tag, “M” für Monat und “Y” für Jahr in verschiedenen Kombinationen enthalten. Editiermasken für Zeitfelder können die Zeichen “H” für Stunde, “I” für Minute, “S” für Sekunde und “T” für Zehntelsekunde in verschiedenen Kombinationen enthalten.

Im Zusammenhang mit Editiermasken für Datums- und Zeitfelder siehe auch die Datums- und Uhrzeit-Systemvariablen im *Natural Referenzhandbuch* (Seite 79).

Beispiele für Editiermasken

Im folgenden sehen Sie ein paar Beispiele für Editiermasken zusammen mit möglichen Ausgaben, die sie erzeugen. Zusätzlich ist die jeweilige Kurzschreibweise angegeben. Sie können Kurz- oder Langschreibweise wahlweise verwenden.

Editiermaske	Kurzschreibweise	Ausgabe A	Ausgabe B
EM=999.99	EM=9(3).9(2)	367.32	005.40
EM=ZZZZZ9	EM=Z(5)9(1)	0	579
EM=X^XXXXXX	EM=X(1)^X(5)	B LUE	A 19379
EM=XXX...XX	EM=X(3)...X(2)	BLU...E	AAB...01
EM=MM.DD.YY	*	01.05.87	12.22.86
EM=HH.II.SS.T	**	08.54.12.7	14.32.54.3

* Verwenden Sie eine Datums-Systemvariable.

** Verwenden Sie eine Uhrzeit-Systemvariable.

Weitere Informationen zu Editiermasken finden Sie unter Session-Parameter EM im *Natural Referenzhandbuch*.

Beispielprogramm ohne EM-Parameter:

```

** Example Program 'EDITMX01'
DEFINE DATA LOCAL
1 VIEWEMP VIEW OF EMPLOYEES
  2 NAME
  2 JOB-TITLE
  2 SALARY (1:3)
  2 CITY
END-DEFINE
READ (3) VIEWEMP BY NAME STARTING FROM 'JONES'
  DISPLAY 'N A M E' NAME /
          'OCCUPATION' JOB-TITLE
          'SALARY' SALARY (1:3)
          'LOCATION' CITY

  SKIP 1
END-READ
END

```

Das obige Programm erzeugt die folgende Ausgabe unter Verwendung von Standard-Editiermasken (soweit vorhanden):

Page		1	97-08-19 17:26:19	
N A M E		SALARY	LOCATION	
OCCUPATION				
JONES		46000	TULSA	
MANAGER		42300		
		39300		
JONES		50000	MOBILE	
DIRECTOR		46000		
		42700		
JONES		31000	MILWAUKEE	
PROGRAMMER		29400		
		27600		

Beispielprogramm mit EM-Parametern:

```

** Example Program 'EDITMX02'
DEFINE DATA LOCAL
1 VIEWEMP VIEW OF EMPLOYEES
  2 NAME
  2 FIRST-NAME
  2 JOB-TITLE
  2 SALARY (1:3)
END-DEFINE
READ (3) VIEWEMP BY NAME STARTING FROM 'JONES'
  DISPLAY 'N A M E' NAME (EM=X^X^X^X^X^X^X^X^X^X^X^X^X^X^X^X^X) /
    FIRST-NAME (EM=...X(10)...)
    'OCCUPATION' JOB-TITLE (EM=' ___ 'X(12))
    'SALARY' SALARY (1:3) (EM=' USD 'ZZZ,999)

  SKIP 1
END-READ
END

```

Das obige Programm erzeugt folgende Ausgabe. Vergleichen Sie sie mit der des vorigen Programms, um zu sehen, wie die EM-Angaben sich auf die Anzeige der Felder auswirken:

Page		1	97-08-19 17:26:29	
N A M E		OCCUPATION	SALARY	
FIRST-NAME				
J O N E S		___ MANAGER	USD	46,000
..VIRGINIA	...		USD	42,300
			USD	39,300
J O N E S		___ DIRECTOR	USD	50,000
..MARSHA	...		USD	46,000
			USD	42,700
J O N E S		___ PROGRAMMER	USD	31,000
..ROBERT	...		USD	29,400
			USD	27,600

Weitere Beispiele für Editiermasken:

Siehe Programme EDITMX03, EDITMX04 und EDITMX05 in Library SYSEXPB.

Vertikale Ausgaben

Natural bietet Ihnen zwei Möglichkeiten, die verschiedenen Daten eines Datensatzes bei der Ausgabe *untereinander* anzuordnen:

- mit einer Kombination von DISPLAY- und WRITE-Statement,
- mit einer VERT-Klausel in einem DISPLAY-Statement.

Kombination von DISPLAY und WRITE

Wie weiter oben in diesem Kapitel beschrieben, erzeugt das DISPLAY-Statement normalerweise Ausgaben in Spaltenform mit Standardüberschriften, während das WRITE-Statement die Daten nebeneinander ohne Überschriften anordnet.

Sie können die Merkmale dieser beiden Statements miteinander verbinden, um eine vertikale Ausgabe von Feldwerten zu erzeugen.

Das DISPLAY-Statement ordnet die Werte eines Feldes untereinander an, und zwar Datensatz für Datensatz; die verschiedenen Felder eines Datensatzes werden nebeneinander ausgegeben.

Durch ein dem DISPLAY-Statement nachgestelltes WRITE-Statement haben Sie die Möglichkeit, in einem WRITE-Statement angegebene Text und/oder Feldwerte zwischen den einzelnen mit dem DISPLAY-Statement ausgegebenen Datensätzen einzufügen.

Das folgende Programm zeigt diese Kombination von DISPLAY und WRITE:

```
** Example Program 'WRITEX04'  
DEFINE DATA LOCAL  
1 VIEWEMP VIEW OF EMPLOYEES  
  2 NAME  
  2 JOB-TITLE  
  2 CITY  
  2 DEPT  
END-DEFINE  
READ (3) VIEWEMP BY CITY STARTING FROM 'SAN FRANCISCO'  
  DISPLAY NAME JOB-TITLE  
  WRITE 20T 'DEPT:' DEPT  
  SKIP 1  
END-READ  
END
```

Es erzeugt folgende Ausgabe:

Page	1	97-08-19 17:52:19
NAME	CURRENT POSITION	
KOLENCE	MANAGER DEPT: TECH05	
GOSDEN	ANALYST DEPT: TECH10	
WALLACE	SALES PERSON DEPT: SALE20	

Tabulator-Notation *T*Feld*

Im vorigen Beispiel ergibt sich die Position des Feldes DEPT aus der Tabulator-Notation *nT* (in diesem Fall "20T", d.h. die Ausgabe beginnt in der 20. Bildschirmspalte).

Durch die Notation *T*Feld* können Sie die WRITE-Ausgabe nach der Position eines im vorangegangenen DISPLAY-Statement ausgegebenen *Feldes* ausrichten.

Im folgenden Beispiel wird die Position der vom WRITE-Statement erzeugten Ausgabe mittels der Notation "T*JOB-TITLE" nach der Position des Feldes JOB-TITLE ausgerichtet:

```

** Example Program 'WRITEX05'
DEFINE DATA LOCAL
1 VIEWEMP VIEW OF EMPLOYEES
  2 NAME
  2 JOB-TITLE
  2 DEPT
  2 CITY
END-DEFINE
READ (3) VIEWEMP BY CITY STARTING FROM 'SAN FRANCISCO'
  DISPLAY NAME JOB-TITLE
  WRITE T*JOB-TITLE 'DEPT:' DEPT
  SKIP 1
END-READ
END

```

NAME	CURRENT POSITION
KOLENCE	MANAGER DEPT: TECH05
GOSDEN	ANALYST DEPT: TECH10
WALLACE	SALES PERSON DEPT: SALE20

Positionierungsnotation x/y

Wenn bei der Kombination von DISPLAY und WRITE die vom WRITE-Statement erzeugte Ausgabe über mehrere Zeilen und/oder Spalten gehen soll, empfiehlt sich die Verwendung der Notation x/y (Zahl-Schrägstrich-Zahl), mit der Sie angeben können, in welcher Zeile/Spalte etwas ausgegeben werden soll. Die Zahl vor dem Schrägstrich gibt die Zeile an, die Zahl hinter dem Schrägstrich die Spalte.

Das folgende Programm veranschaulicht die Verwendung dieser Notation:

```
** Example Program 'WRITEX06'  
DEFINE DATA LOCAL  
1 VIEWEMP VIEW OF EMPLOYEES  
  2 NAME  
  2 FIRST-NAME  
  2 MIDDLE-I  
  2 ADDRESS-LINE (1:1)  
  2 CITY  
  2 ZIP  
END-DEFINE  
READ (3) VIEWEMP BY CITY STARTING FROM 'NEW YORK'  
  DISPLAY 'NAME AND ADDRESS' NAME  
  WRITE  1/5  FIRST-NAME          1/30 MIDDLE-I  
         2/5  ADDRESS-LINE (1:1)  
         3/5  CITY                3/30 ZIP /  
END-READ  
END
```

```
Page      1                               97-08-19 17:55:47  
  
NAME AND ADDRESS  
-----  
  
RUBIN  
  SYLVIA                L  
  2003 SARAZEN PLACE  
  NEW YORK              10036  
  
WALLACE  
  MARY                  P  
  12248 LAUREL GLADE C  
  NEW YORK              10036  
  
KELLOGG  
  HENRIETTA            S  
  1001 JEFF RYAN DR.  
  NEWARK                19711
```

DISPLAY VERT-Statement

Standardmäßig gibt Natural die Felder nebeneinander aus. Mit der VERT-Klausel können Sie erreichen, daß bei einem DISPLAY-Statement die Werte der verschiedenen Felder eines Datensatzes nicht nebeneinander, sondern untereinander (in vertikaler Anordnung) ausgegeben werden. Mit einer HORIZ-Klausel können Sie dies im selben DISPLAY-Statement wieder rückgängig machen und zur horizontalen Ausgabe zurückkehren.

Die Ausgabe von Spaltenüberschriften beim DISPLAY VERT wird über eine AS-Klausel gesteuert:

- Ohne AS-Klausel werden keine Spaltenüberschriften ausgegeben.
- AS CAPTIONED bewirkt die Ausgabe der Standard-Spaltenüberschriften.
- AS *Text* bewirkt, daß *Text* als Spaltenüberschrift ausgegeben wird. Beachten Sie hierbei, daß ein Schrägstrich (/) innerhalb von *Text*-Elementen eines DISPLAY-Statements einen Zeilenvorschub bewirkt.
- AS *Text* CAPTIONED bewirkt, daß *Text* als Spaltenüberschrift ausgegeben wird und außerdem die Standard-Spaltenüberschrift in jeder Ausgabezeile dem jeweiligen Feldwert direkt vorangestellt wird.

Die folgenden Beispielprogramme veranschaulichen die Verwendung des DISPLAY VERT-Statements.

DISPLAY VERT ohne AS-Klausel

Das folgende Programm verwendet keine AS-Klausel, d.h. es werden keine Spaltenüberschriften ausgegeben.

```
** Example Program 'DISPLX09'  
DEFINE DATA LOCAL  
1 VIEWEMP VIEW OF EMPLOYEES  
  2 NAME  
  2 FIRST-NAME  
  2 CITY  
END-DEFINE  
READ (3) VIEWEMP BY CITY STARTING FROM 'NEW YORK'  
  DISPLAY VERT NAME FIRST-NAME / CITY  
  SKIP 2  
END-READ  
END
```

Beachten Sie, daß alle Feldwerte vertikal, d.h. untereinander, ausgegeben werden:

Page	1	97-08-19	17:55:47
RUBIN SYLVIA NEW YORK WALLACE MARY NEW YORK KELLOGG HENRIETTA NEWARK			

DISPLAY VERT AS CAPTIONED und HORIZ

Das folgende Programm enthält eine VERT- und eine HORIZ-Klausel, die bewirken, daß einige Ausgaben vertikal und andere horizontal angeordnet sind, sowie eine AS CAPTIONED-Klausel zur Ausgabe der Standard-Spaltenüberschriften.

```

** Example Program 'DISPLX10'
DEFINE DATA LOCAL
1 VIEWEMP VIEW OF EMPLOYEES
  2 NAME
  2 FIRST-NAME
  2 CITY
  2 JOB-TITLE
  2 SALARY (1:1)
END-DEFINE
READ (3) VIEWEMP BY CITY STARTING FROM 'NEW YORK'
  DISPLAY VERT AS CAPTIONED NAME FIRST-NAME
    HORIZ JOB-TITLE SALARY (1:1)
  SKIP 1
END-READ
END

```

Page 1		97-08-19 17:55:47
NAME FIRST-NAME	CURRENT POSITION	ANNUAL SALARY
RUBIN SYLVIA	SECRETARY	17000
WALLACE MARY	ANALYST	38000
KELLOGG HENRIETTA	DIRECTOR	52000

DISPLAY VERT AS *text*

Das folgende Programm enthält eine *AS Text*-Klausel, die bewirkt, daß der angegebene *Text* als Spaltenüberschrift ausgegeben wird.

```

** Example Program 'DISPLX11'
DEFINE DATA LOCAL
1 VIEWEMP VIEW OF EMPLOYEES
  2 NAME
  2 FIRST-NAME
  2 CITY
  2 JOB-TITLE
  2 SALARY (1:1)
END-DEFINE
READ (3) VIEWEMP BY CITY STARTING FROM 'NEW YORK'
  DISPLAY VERT AS 'EMPLOYEES' NAME FIRST-NAME
    HORIZ JOB-TITLE SALARY (1:1)
  SKIP 1
END-READ
END

```

EMPLOYEES	CURRENT POSITION	ANNUAL SALARY
RUBIN SYLVIA	SECRETARY	17000
WALLACE MARY	ANALYST	38000
KELLOGG HENRIETTA	DIRECTOR	52000

DISPLAY VERT AS *text* CAPTIONED

Das folgende Programm enthält eine AS *Text* CAPTIONED-Klausel.

```

** Example Program 'DISPLX12'
DEFINE DATA LOCAL
1 VIEWEMP VIEW OF EMPLOYEES
  2 NAME
  2 FIRST-NAME
  2 CITY
  2 JOB-TITLE
  2 SALARY (1:1)
END-DEFINE
READ (3) VIEWEMP BY CITY STARTING FROM 'NEW YORK'
  DISPLAY VERT AS 'EMPLOYEES' CAPTIONED NAME FIRST-NAME
    HORIZ JOB-TITLE SALARY (1:1)
  SKIP 1
END-READ
END

```

Diese Klausel bewirkt, daß die Standard-Spaltenüberschriften (NAME und FIRST-NAME) vor den Feldwerten ausgegeben werden:

Page	1	97-04-19	17:55:47
	EMPLOYEES	CURRENT POSITION	ANNUAL SALARY
NAME RUBIN		SECRETARY	17000
FIRST-NAME SYLVIA			
NAME WALLACE		ANALYST	38000
FIRST-NAME MARY			
NAME KELLOGG		DIRECTOR	52000
FIRST-NAME HENRIETTA			

Tabulator-Notation **P****Feld*

Bei einer Kombination von DISPLAY VERT-Statement mit nachfolgendem WRITE-Statement können Sie mit der Notation **P****Feld* die Feld-Ausgabe des WRITE-Statements nach der Zeilen- und Spalten-Position eines im DISPLAY VERT-Statements angegebenen *Feldes* ausrichten.

Im folgenden Programm werden die Felder SALARY und BONUS in der gleichen Spalte ausgegeben, SALARY in jeder ersten Zeile, BONUS in jeder zweiten Zeile.

Der Text “***SALARY PLUS BONUS***” ist nach SALARY ausgerichtet, d.h. der Text wird in der gleichen Spalte wie SALARY und in der ersten Zeile ausgegeben. Der Text “(IN US DOLLARS)” hingegen ist nach BONUS ausgerichtet; entsprechend wird dieser Text in der gleichen Spalte wie BONUS und in der zweiten Zeile ausgegeben.

```

** Example Program 'WRITEX07'
DEFINE DATA LOCAL
1 VIEWEMP VIEW OF EMPLOYEES
  2 CITY
  2 NAME
  2 JOB-TITLE
  2 SALARY (1:1)
  2 BONUS (1:1,1:1)
END-DEFINE
READ (3) VIEWEMP BY CITY STARTING FROM 'LOS ANGELES'
  DISPLAY NAME JOB-TITLE VERT AS 'INCOME' SALARY (1) BONUS (1,1)
  WRITE P*SALARY '***SALARY PLUS BONUS***'
      P*BONUS '(IN US DOLLARS)'
  SKIP 1
END-READ
END

```

Page	1	97-08-19	18:14:11
NAME	CURRENT POSITION	INCOME	
POORE JR	SECRETARY	25000	0
		SALARY PLUS BONUS (IN US DOLLARS)	
PREPARATA	MANAGER	46000	9000
		SALARY PLUS BONUS (IN US DOLLARS)	
MARKUSH	TRAINEE	22000	0
		SALARY PLUS BONUS (IN US DOLLARS)	

Weiteres Beispiel für DISPLAY VERT- mit WRITE-Statement:

Siehe Programm WRITEX10 in Library SYSEXPB.

OBJEKTTYPEN

Dieses Kapitel umfasst die folgenden Themen:

- Welche Typen von Programmierobjekten gibt es?
- Data Areas
- Programme, Subprogramme und Subroutinen
- Maps
- Helproutinen
- Mehrfache Verwendung von Sourcecode — Copycode
- Natural-Objekte dokumentieren — Text
- Ereignisgesteuerte Anwendungen erstellen — Dialog
- Verteilte objekt-basierte Anwendungen erstellen — Class
- Nicht-Natural-Dateien benutzen — Ressource.

Welche Typen von Programmierobjekten gibt es?

Innerhalb einer Natural-Anwendung können verschiedene Typen von Programmierobjekten verwendet werden, um eine Anwendung möglichst effizient zu strukturieren.

Es gibt folgende Typen von Natural-Programmierobjekten:

- Local Data Area
- Global Data Area
- Parameter Data Area
- Program
- Subprogram
- Subroutine
- Helproutine
- Map
- Copycode
- Text
- Dialog
- Class.

All diese Objekte erstellen und pflegen Sie mit den Natural-Editoren:

- Local Data Areas, Global Data Areas und Parameter Data Areas werden mit dem *Data-Area-Editor* erstellt und gepflegt.
- Maps werden mit dem *Map-Editor* erstellt und gepflegt.
- Dialoge werden mit dem *Dialog-Editor* erstellt und gepflegt.
- Classes (Klassen) werden mit dem *Class Builder* erstellt und gepflegt.
- Alle anderen oben aufgeführten Objekttypen werden mit dem *Programm-Editor* erstellt und gepflegt.

Die Editoren sind in Ihrem *Natural Benutzerhandbuch* bzw. *User's Guide* beschrieben.

Data Areas

Wie im Kapitel **Felder definieren** (Seite 5) erläutert, müssen alle Felder, die in einem Programm verwendet werden sollen, in einem DEFINE DATA-Statement definiert werden.

Die Felder können entweder innerhalb des DEFINE DATA-Statements selbst definiert werden; oder sie können außerhalb des Programms in einem separaten Datenbereich (Data Area) definiert werden, der dann vom DEFINE DATA-Statement referenziert wird.

Mit Natural können Sie drei Arten von Data Areas anlegen:

Local Data Area	In einer Local Data Area definieren Sie die Datenelemente, die nur von einem einzigen Natural-Modul in einer Anwendung benutzt werden sollen.
Global Data Area	In einer Global Data Area definieren Sie die Datenelemente, die von mehreren Natural-Programmen, -Unterprogrammen, usw. in einer Anwendung benutzt werden sollen.
Parameter Data Area	In einer Parameter Data Area definieren Sie die Felder, die als Parameter an ein Subprogramm, eine externe Subroutine bzw. eine Helpoutine übergeben werden.

Local Data Area

Als “local” definierte Variablen können nur von einem einzigen Natural-Modul benutzt werden. Sie haben zwei Möglichkeiten, lokale Daten zu definieren:

- Sie können die Daten innerhalb des Programms definieren.
- Sie können die Daten in einer Local Data Area außerhalb des Programms definieren.

Im ersten Beispiel sind die Felder innerhalb des DEFINE DATA-Statements des Programms definiert. Im zweiten Beispiel sind dieselben Felder in einer Local Data Area definiert, und das DEFINE DATA-Statement enthält lediglich eine Referenz auf diese Data Area.

Beispiel 1 — Im DEFINE DATA-Statement definierte Felder:

```
DEFINE DATA LOCAL
1 VIEWEMP VIEW OF EMPLOYEES
  2 NAME
  2 FIRST-NAME
  2 PERSONNEL-ID
1 #VARI-A (A20)
1 #VARI-B (N3.2)
1 #VARI-C (I4)
END-DEFINE
...
```

Beispiel 2 — In einer separaten Data Area definierte Felder:

Programm:

```
DEFINE DATA LOCAL
  USING LDA39
END-DEFINE
...
```

Local Data Area "LDA39":

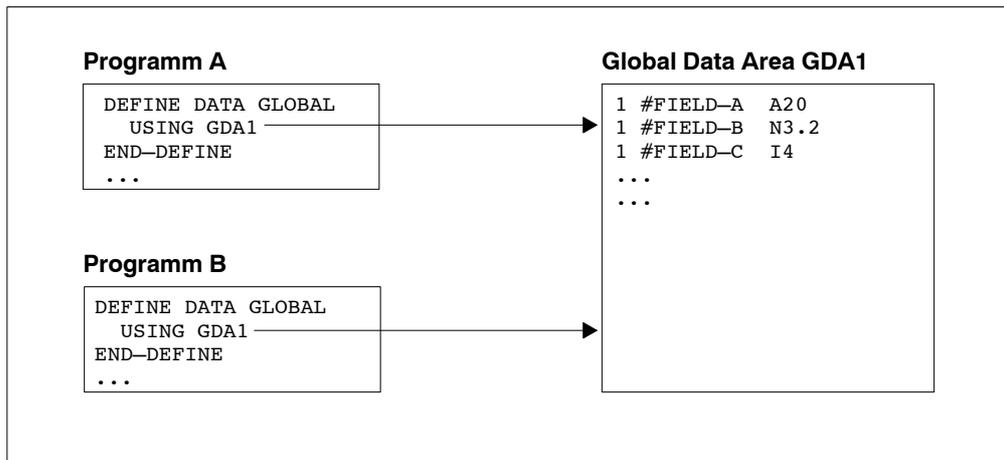
I	T	L	Name	F	Leng	Index/Init/EM/Name/Comment
V	1		VIEWEMP			EMPLOYEES
	2		NAME	A	20	
	2		FIRST-NAME	A	20	
	2		PERSONNEL-ID	A	8	
	1		#VARI-A	A	20	
	1		#VARI-B	N	3.2	
	1		#VARI-C	I	4	

Um eine klar strukturierte Anwendung zu erhalten, ist es in der Regel besser, Felder in Data Areas außerhalb der Programme zu definieren.

Global Data Area

In einer Global Data Area definieren Sie die Datenelemente, die von mehr als einem Programm, Unterprogramm, usw. in einer Anwendung benutzt werden sollen.

In einer Global Data Area definierte Variablen können von mehreren Objekten in einer Anwendung referenziert werden.



Die Global Data Area und die Objekte, von denen sie referenziert wird, müssen sich in derselben Library (bzw. einer Steplib) befinden.

Global Data Areas werden mit dem Data-Area-Editor erstellt. Die Referenzierung der Global Data Area erfolgt im Programm/Unterprogramm über das DEFINE DATA-Statement. Eine einzige Global Data Area kann von beliebig vielen Programmen, externen Subroutinen und Helproutinen benutzt werden.

Jedes Programm/Unterprogramm kann nur eine einzige Global Data Area referenzieren, d.h. ein DEFINE DATA-Statement darf nicht mehr als eine GLOBAL-Klausel enthalten.

Anmerkung:

Beim Aufbau einer Anwendung, in der mehrere Objekte eine Global Data Area benutzen, beachten Sie bitte, daß jede Veränderung der Global Data Area sich auf alle diese Programme/Routinen auswirkt, die diese Data Area referenzieren. Daher müssen nach der Veränderung einer Global Data Area alle Objekte, die sie benutzen, neu katalogisiert (STOW) werden.

Wann werden Global Data Areas initialisiert?

Eine Global Data Area wird bei der ersten Benutzung initialisiert. Die Global Data Area bleibt in der aktuellen Natural-Session so lange aktiv (d.h. der Inhalt der Variablen in der Global Data Area bleibt erhalten), bis:

- die nächste Anmeldung (LOGON) erfolgt oder
- eine andere Global Data Area auf der gleichen Programm-Stufe (Levels werden später in diesem Kapitel beschrieben) benutzt wird oder
- ein RELEASE VARIABLES-Statement ausgeführt wird. In diesem Fall werden die Variablen der Global Data Area zurückgesetzt, sobald entweder die Ausführung des Level-1-Programms abgeschlossen ist oder das Programm über ein FETCH- oder RUN-Statement ein anderes Programm aufruft.

Anmerkung:

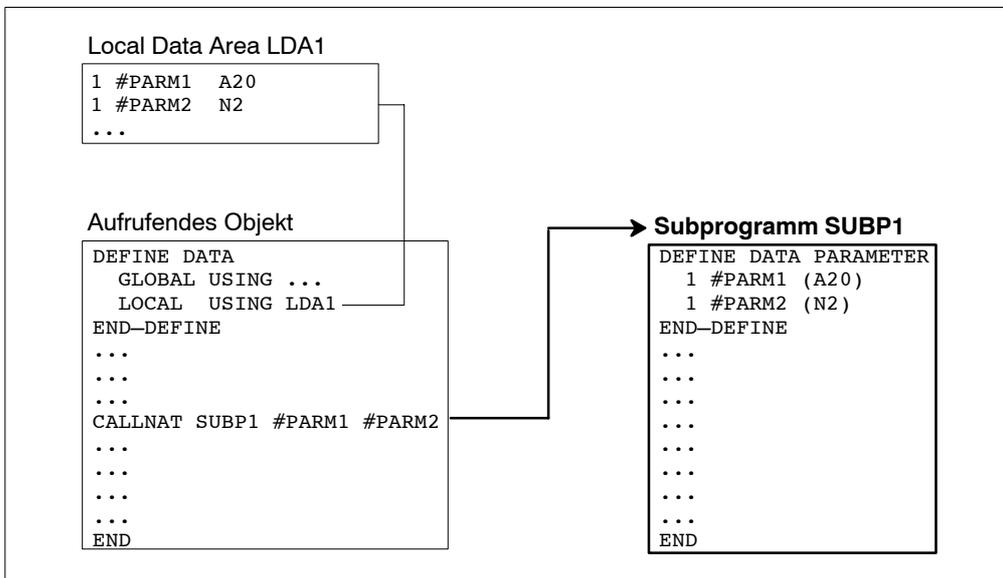
Wenn eine GDA mit Namen "COMMON" in einer Library vorhanden ist, wird das Programm mit Namen ACOMMON automatisch aufgerufen, wenn Sie sich mit LOGON für diese Library anmelden.

Parameter Data Area

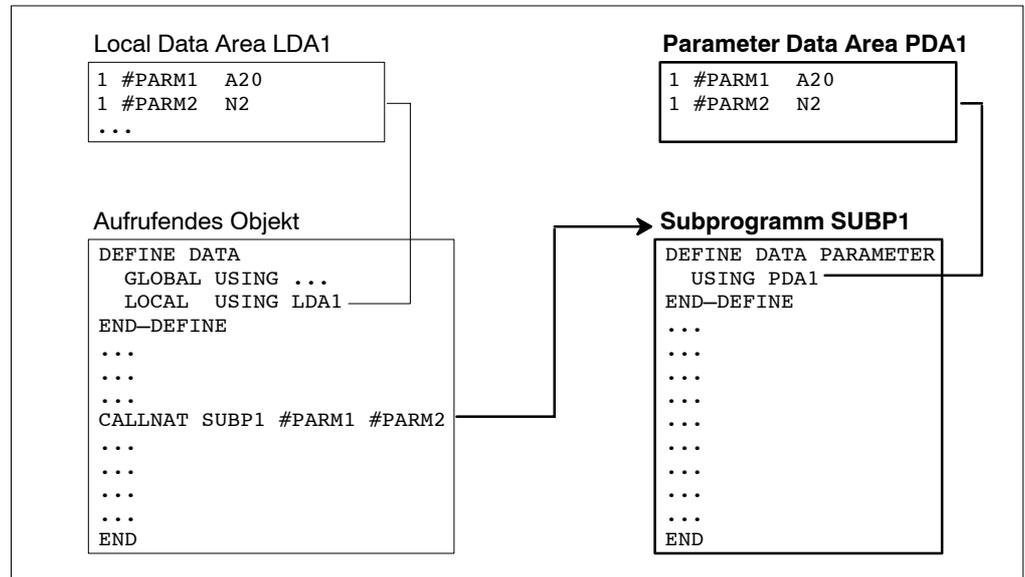
Parameter Data Areas werden von Subprogrammen und externen Subroutinen verwendet.

Ein Subprogramm wird mit einem CALLNAT-Statement aufgerufen. Mit dem CALLNAT-Statement können Parameter von dem aufrufenden Objekt an das Subprogramm übergeben werden. Diese Parameter müssen im Subprogramm in einem DEFINE DATA PARAMETER-Statement definiert werden: sie können entweder in der PARAMETER-Klausel des DEFINE DATA-Statements selbst definiert werden oder in einer separaten Parameter Data Area, die von dem DEFINE DATA PARAMETER-Statement referenziert wird.

Innerhalb des DEFINE DATA PARAMETER-Statements definierte Parameter:



In einer Parameter Data Area definierte Parameter:



In der gleichen Weise müssen Parameter, die mit einem PERFORM-Statement an eine externe Subroutine übergeben werden, in der externen Subroutine in einem DEFINE DATA PARAMETER-Statement definiert werden.

Im aufrufenden Objekt müssen die an das Subprogramm bzw. die Subroutine übergebenen Parametervariablen nicht in einer Parameter Data Area definiert werden; in der obigen Abbildung sind sie in einer vom aufrufenden Objekt benutzten Local Data Area definiert (man hätte sie aber auch in einer Global Data Area definieren können).

Reihenfolge, Format und Länge der im CALLNAT- bzw. PERFORM-Statement des aufrufenden Objekts angegebenen Parameter muß genau mit Reihenfolge, Format und Länge der Felder, die im DEFINE DATA PARAMETER-Statement des aufgerufenen Subprogramms bzw. der aufgerufenen Subroutine definiert sind, übereinstimmen.

Die Namen der Variablen im aufrufenden Objekt und dem aufgerufenen Subprogramm bzw. der aufgerufenen Subroutine brauchen nicht dieselben zu sein (da die Übergabe der Parameter nach Speicheradressen erfolgt und nicht nach Namen).

Programme, Subprogramme und Subroutinen

Die folgenden Themen werden nachfolgend behandelt:

- Modulare Anwendungsstruktur
- Mehrere Stufen (Levels) aufgerufener Objekte
- Programm
- Subroutine
- Subprogramm
- Verarbeitungsablauf beim Aufruf eines Unterprogramms.

Modulare Anwendungsstruktur

Eine typische Natural-Anwendung besteht nicht aus einem einzigen großen Programm, sondern ist in mehrere Module aufgeteilt. Jedes dieser Module stellt eine funktionale Einheit von überschaubarer Größe dar, und jedes Modul ist mit den anderen Modulen der Anwendung auf eine klar definierte Weise verbunden. Dadurch ergibt sich eine klar strukturierte Anwendung, was die Entwicklung und anschließende Wartung erheblich erleichtert und beschleunigt.

Ein Hauptprogramm, das ausgeführt wird, kann andere Programme, Subprogramme, Subroutinen, Helproutinen und Maps aufrufen. Diese Objekte können ihrerseits wiederum andere Objekte aufrufen (eine Subroutine kann beispielsweise eine andere Subroutine aufrufen). Dadurch kann die modulare Struktur einer Anwendung äußerst komplex und vielschichtig werden.

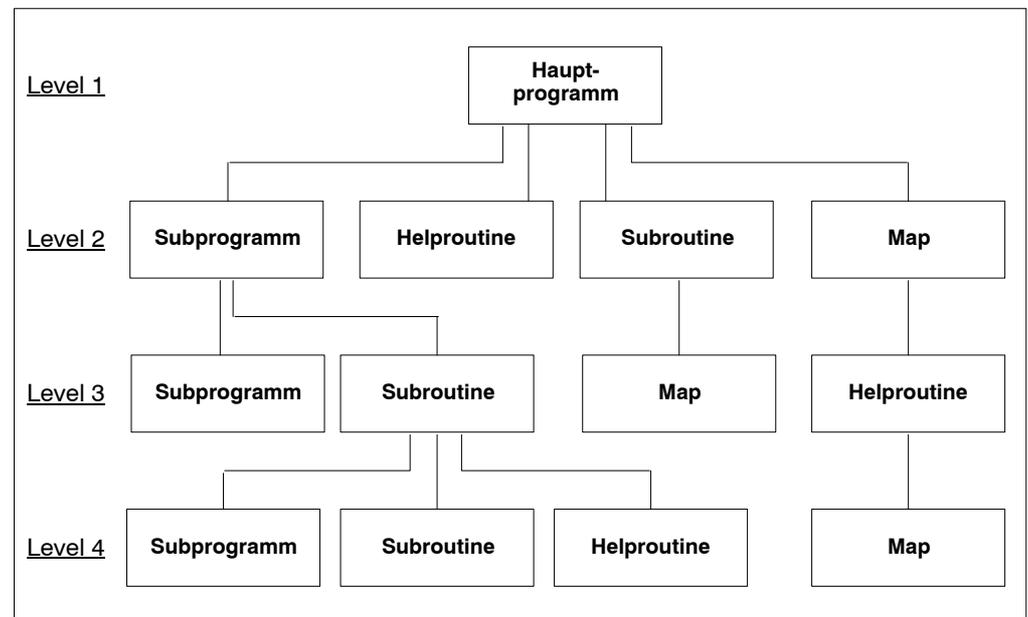
Mehrere Stufen (Levels) aufgerufener Objekte

Ein aufgerufenes Objekt ist jeweils eine Stufe unter dem Objekt, von dem es aufgerufen wurde; d.h. mit jedem Aufruf eines untergeordneten Objekts erhöht sich die Stufennummer um 1.

Ein Programm, das selbständig ausgeführt wird, wird auf Stufe 1 eingeordnet; Subprogramme, Subroutinen, Maps oder Helproutinen, die direkt von diesem Hauptprogramm aufgerufen werden, sind auf Stufe 2; ruft eine dieser Subroutinen ihrerseits eine andere Subroutine auf, so ist letztere auf Stufe 3.

Wird von einem Objekt über ein FETCH-Statement ein anderes Programm aufgerufen, so wird dies als Hauptprogramm eingestuft und auf Stufe 1 eingeordnet. Ein Programm, das mit FETCH RETURN aufgerufen wird, wird dagegen als Unterprogramm eingestuft und ist eine Stufe unter dem Objekt, von dem es aufgerufen wurde.

Die folgende Abbildung ist ein Beispiel für mehrere Stufen (Levels) aufgerufener Objekte und zeigt, wie diese Stufen gezählt werden:



Um die Level-Nummer des Objekts, das gerade ausgeführt wird, zu erfahren, können Sie die Systemvariable *LEVEL verwenden (die im *Natural Referenzhandbuch* beschrieben ist).

Der vorliegende Abschnitt behandelt folgende Natural-Objekttypen, die als Unterprogramme aufgerufen werden können:

- Programm
- Subroutine
- Subprogramm

Helprountines und Maps werden zwar auch von anderen Objekten aufgerufen, sind aber keine Unterprogramme im eigentlichen Sinne und werden daher in späteren Abschnitten dieses Kapitels behandelt.

Grundsätzlich unterscheiden sich Programme, Subprogramme und Subroutinen dadurch voneinander, wie Daten zwischen ihnen übergeben werden können und welche Data Areas sie gemeinsam benutzen können. Die Entscheidung, welchen Objekttyp Sie verwenden, ergibt sich daher im wesentlichen aus der Datenstruktur Ihrer Anwendung.

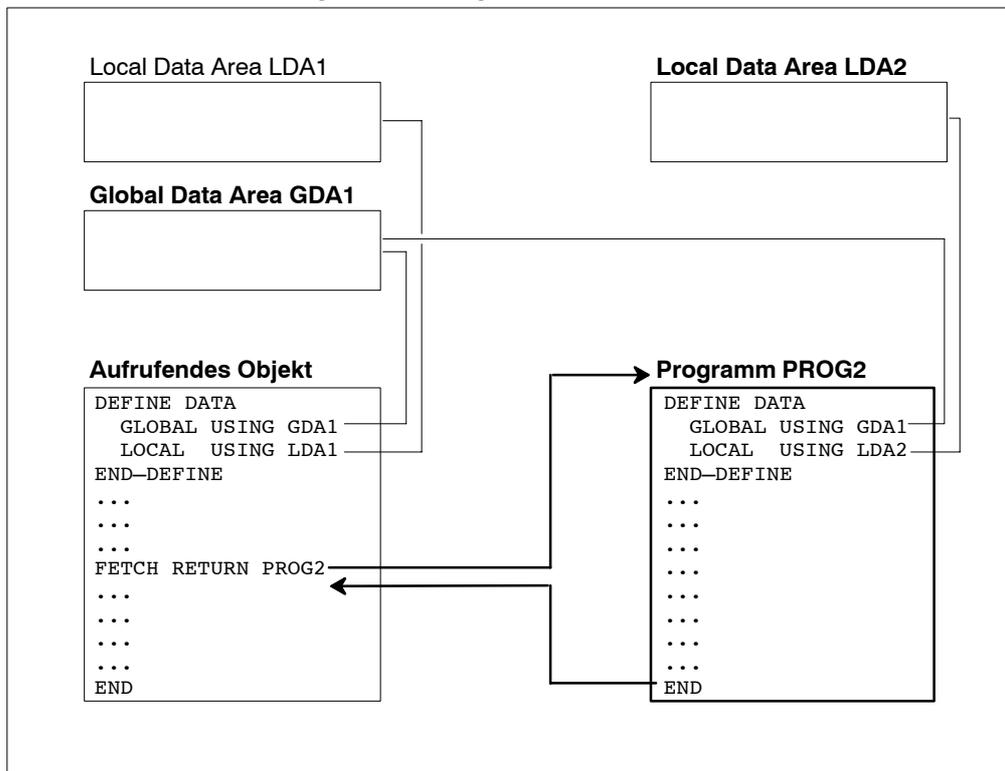
Programm

Ein Programm kann selbständig ausgeführt — und getestet — werden. Um ein Source-Programm zu kompilieren und anschließend auszuführen, verwenden Sie das Systemkommando RUN. Um ein Programm auszuführen, das bereits in kompilierter Form existiert, verwenden Sie das Systemkommando EXECUTE.

Ein Programm kann von einem anderen Objekt mit einem FETCH- oder FETCH RETURN-Statement aufgerufen werden. Das aufrufende Objekt kann ein Programm, ein Subprogramm, eine Subroutine oder eine Helproutine sein.

- Wenn ein Programm mit FETCH RETURN aufgerufen wird, wird die Ausführung des aufrufenden Objekts unterbrochen — nicht beendet —, und das aufgerufene Programm wird als *Unterprogramm* aktiviert. Wenn die Ausführung des aufgerufenen Programms beendet ist, wird das aufrufende Objekt reaktiviert und seine Ausführung mit dem nächsten Statement nach dem FETCH RETURN-Statement fortgesetzt.
- Wenn ein Programm mit FETCH aufgerufen wird, wird die Ausführung des aufrufenden Objekts beendet, und das aufgerufene Programm wird als *Hauptprogramm* aktiviert. Das aufrufende Objekt wird nach beendeter Ausführung des aufgerufenen Programms nicht reaktiviert.

Mit FETCH RETURN aufgerufenes Programm:

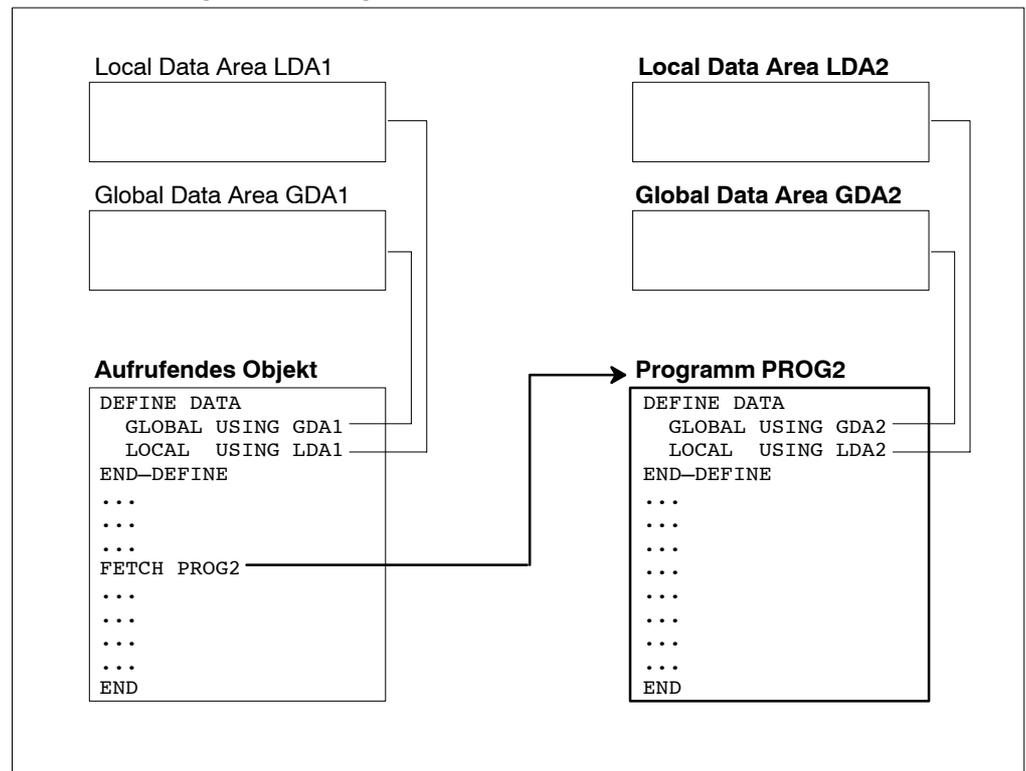


Ein mit `FETCH RETURN` aufgerufenes Programm kann auf die vom aufrufenden Objekt benutzte Global Data Area zugreifen.

Darüber hinaus kann jedes Programm seine eigene Local Data Area haben, in der die nur in diesem Programm verwendeten Felder definiert sind.

Ein mit `FETCH RETURN` aufgerufenes Programm kann jedoch keine eigene Global Data Area haben.

Mit FETCH aufgerufenes Programm:



Ein mit FETCH als Hauptprogramm aufgerufenes Programm verwendet in der Regel seine eigene Global Data Area (wie in der obigen Abbildung gezeigt). Es könnte allerdings auch dieselbe Global Data Area verwenden wie das aufrufende Objekt.

Anmerkung:

Ein Source-Programm kann auch mit einem RUN-Statement aufgerufen werden; siehe RUN-Statement im Natural Statements-Handbuch.

Subroutine

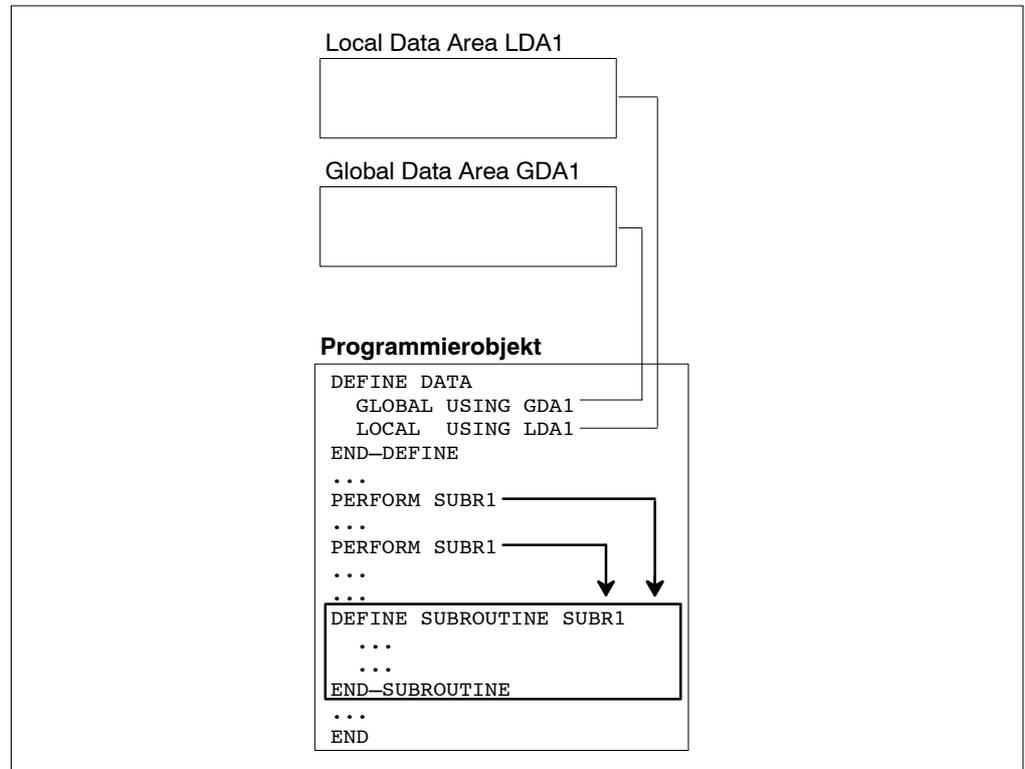
Die Statements, aus denen eine Subroutine besteht, müssen innerhalb eines DEFINE SUBROUTINE ... END-SUBROUTINE-Statement-Blocks definiert werden.

Eine Subroutine wird mit einem PERFORM-Statement aufgerufen.

Eine Subroutine kann eine *interne Subroutine* oder eine *externe Subroutine* sein:

- Eine *interne Subroutine* wird innerhalb des Objekts, das das sie aufrufende PERFORM-Statement enthält, definiert.
- Eine *externe Subroutine* wird als separates Objekt — vom Typ Subroutine — außerhalb des Objektes, das sie aufruft, definiert.

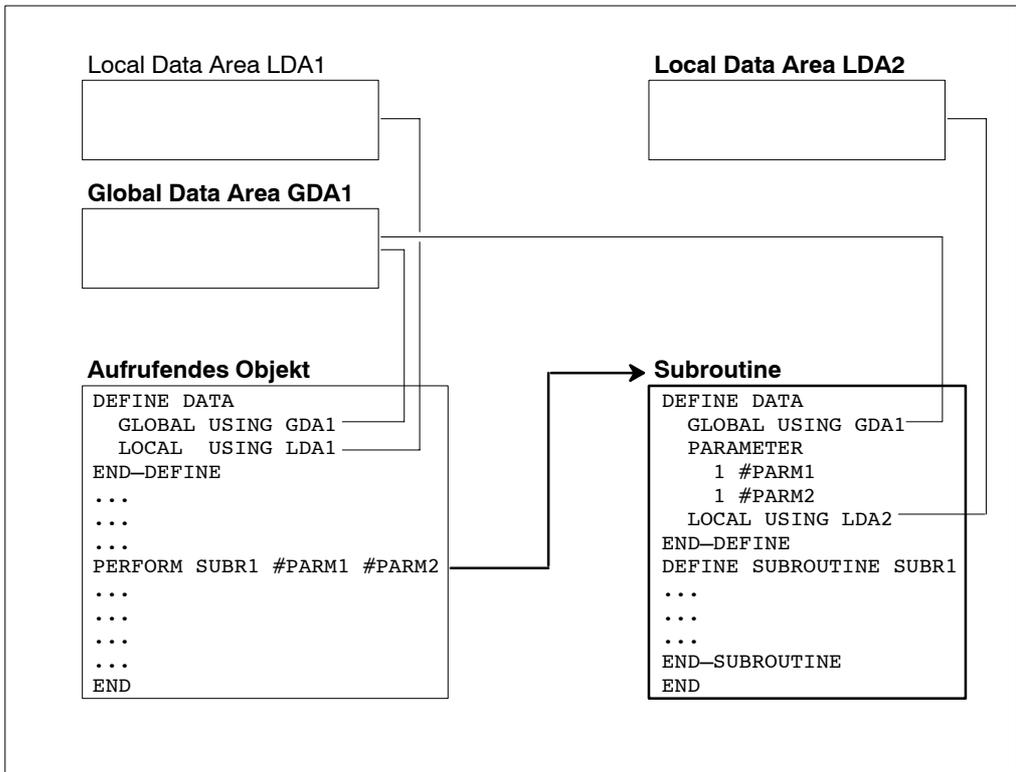
Falls Sie einen Code-Block haben, der innerhalb eines Objekts mehrmals ausgeführt werden soll, ist es sinnvoll, eine interne Subroutine zu verwenden. Sie müssen diesen Block dann nur einmal innerhalb eines DEFINE SUBROUTINE-Statement-Blocks kodieren und rufen ihn dann mit mehreren PERFORM-Statements auf.

Interne Subroutine:

Eine interne Subroutine kann in einem Programmierobjekt vom Typ Programm, Subprogramm, Subroutine oder Helproutine enthalten sein.

Wenn eine interne Subroutine so groß ist, daß sie die Lesbarkeit des Objekts, in dem sie enthalten ist, beeinträchtigt, mag es ratsam sein, sie in eine externe Subroutine zu tun, um die Lesbarkeit der Anwendung zu verbessern.

Externe Subroutine:



Eine externe Subroutine — also ein Objekt vom Typ Subroutine — kann nicht selbständig ausgeführt werden. Sie muß von einem anderen Objekt aufgerufen werden. Das aufrufende Objekt kann ein Programm, ein Subprogramm, eine Subroutine oder eine Helproutine sein.

Welche Daten einer internen Subroutine zur Verfügung stehen

Eine interne Subroutine kann auf die Local Data Area und die Global Data Area des Objektes, in dem sie enthalten ist, zugreifen.

Welche Daten einer externen Subroutine zur Verfügung stehen

Eine externe Subroutine kann auf die Global Data Area des aufrufenden Objekts zugreifen.

Darüber hinaus können mit dem PERFORM-Statement Parameter von dem aufrufenden Objekt an die externe Subroutine übergeben werden. Diese Parameter müssen entweder im DEFINE DATA PARAMETER-Statement der Subroutine oder in einer von der Subroutine verwendeten Parameter Data Area definiert werden.

Außerdem kann eine externe Subroutine eine eigene Local Data Area haben, in der die Felder definiert sind, die nur innerhalb der Subroutine verwendet werden.

Eine externe Subroutine kann jedoch keine eigene Global Data Area haben.

Subprogramm

Ein Subprogramm enthält in der Regel eine allgemein verfügbare Standardfunktion, die von verschiedenen Objekten in einer Anwendung benutzt wird.

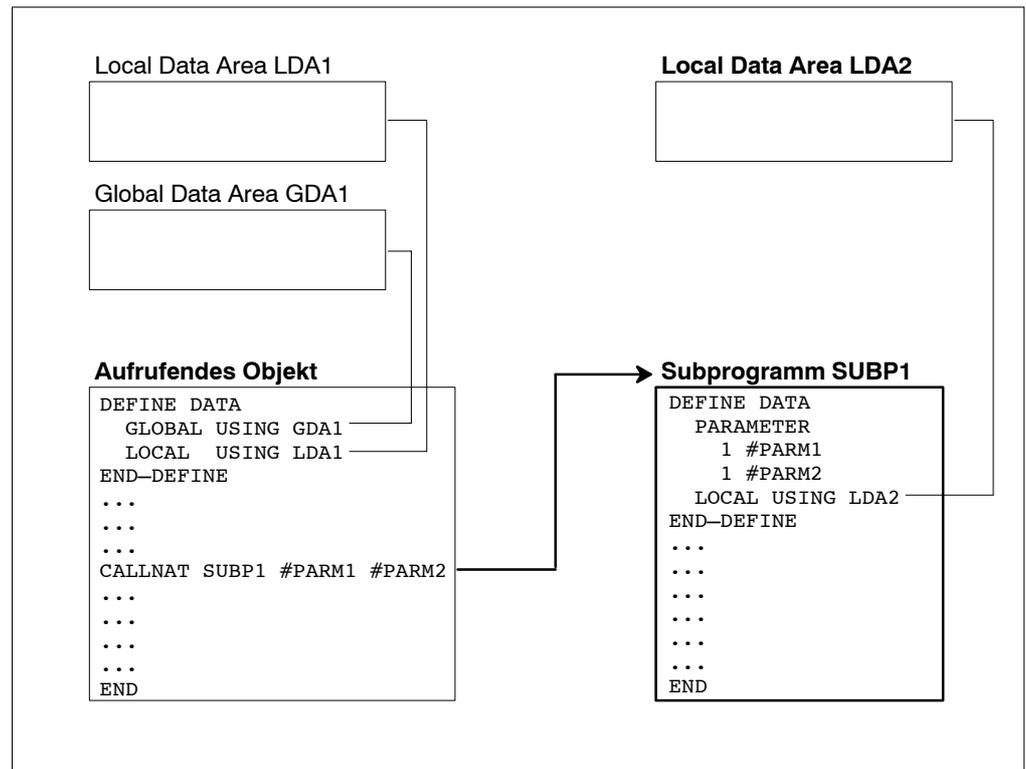
Ein Subprogramm kann nicht selbständig ausgeführt werden. Es muß von einem anderen Objekt aufgerufen werden. Das aufrufende Objekt kann ein Programm, ein Subprogramm, eine Subroutine oder eine Helpoutine sein.

Ein Subprogramm wird mit einem CALLNAT-Statement aufgerufen.

Wenn das CALLNAT-Statement ausgeführt wird, wird die Ausführung des aufrufenden Objekts unterbrochen und das Subprogramm ausgeführt. Nach der Ausführung des Subprogramms wird die Ausführung des aufrufenden Objekts mit dem nächsten Statement nach dem CALLNAT-Statement fortgesetzt.

Welche Daten einem Subprogramm zur Verfügung stehen

Mit dem CALLNAT-Statement können Parameter von dem aufrufenden Objekt an das Subprogramm übergeben werden. Diese Parameter sind die einzigen Daten, die dem Subprogramm vom aufrufenden Objekt zur Verfügung stehen. Sie müssen entweder im DEFINE DATA PARAMETER-Statement des Subprogramms oder in einer vom Subprogramm verwendeten Parameter Data Area definiert werden.



Außerdem kann ein Subprogramm eine eigene Local Data Area haben, in der die Felder definiert sind, die innerhalb des Subprogramms verwendet werden.

Wenn ein Subprogramm seinerseits eine Subroutine oder Helproutine aufruft, kann es eine eigene Global Data Area haben und diese mit der Subroutine/ Helproutine teilen.

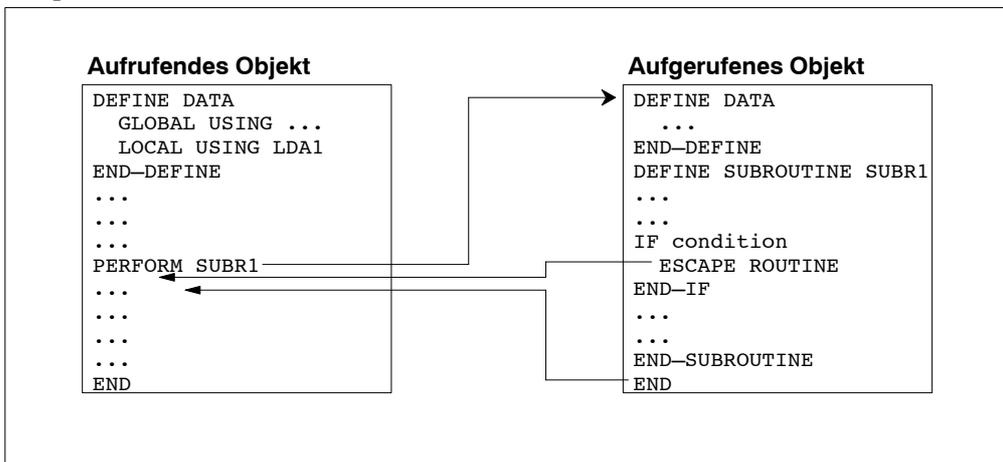
Verarbeitungsablauf beim Aufruf eines Unterprogramms

Wenn ein CALLNAT-, PERFORM- oder FETCH RETURN-Statement, das ein Unterprogramm — ein Subprogram, eine externe Subroutine bzw. ein Programm — aufruft, ausgeführt wird, wird die Ausführung des aufrufenden Objekts unterbrochen, und die Ausführung des Unterprogramms beginnt.

Die Ausführung des Unterprogramms wird fortgesetzt, bis entweder sein END-Statement erreicht ist oder die Verarbeitung des Unterprogramms durch die Ausführung eines ESCAPE ROUTINE-Statements gestoppt wird.

In beiden Fällen wird die Verarbeitung des aufrufenden Objekts mit dem nächsten Statement nach dem CALLNAT-, PERFORM- bzw. FETCH RETURN-Statement, mit dem das Unterprogramm aufgerufen wurde, fortgesetzt.

Beispiel:



Maps

Maps (Bildschirmmasken) sind derjenige Teil einer Anwendung, den die Benutzer auf ihren Bildschirmen sehen.

Der Dialog mit dem Benutzer erfolgt über Eingabe-Maps. Eine Eingabe-Map (*input map*) wird mit einem INPUT USING MAP-Statement aufgerufen.

Wenn eine Anwendung einen Ausgabe-Report erzeugt, kann dieser Report mittels einer Ausgabe-Map (*output map*) auf dem Bildschirm angezeigt werden. Eine Ausgabe-Map wird mit einem WRITE USING MAP-Statement aufgerufen.

Maps werden mit dem Map-Editor erstellt, der in Ihrem *Natural Benutzerhandbuch* bzw. *User's Guide* beschrieben ist.

Die Verarbeitung einer Map kann mit einem ESCAPE ROUTINE-Statement in einer Processing Rule (Verarbeitungsregel) gestoppt werden.

Helpmaps werden mit dem Map-Editor erstellt. Im Prinzip sind sie wie andere Maps, aber wenn sie als Hilfe zugewiesen werden, werden zusätzliche Prüfungen vorgenommen, um ihre Verwendbarkeit für Hilfe-Zwecke zu gewährleisten.

Helproutinen

Helproutinen haben besondere Eigenschaften, die die Verarbeitung von Hilfe-Aufrufen erleichtern.

Helproutinen werden mit dem Programm-Editor erstellt. Sie ermöglichen den Aufbau komplexer interaktiver Hilfe-Systeme.

Die folgenden Themen werden nachfolgend erörtert:

- Helproutinen aufrufen
- Helproutinen spezifizieren
- Programmierhinweise für Helproutinen
- Parameter an Helproutinen übergeben
- Hilfe als eingeblendetes Fenster.

Helproutinen aufrufen

Sie fordern Hilfe an, indem Sie das Hilfe-Zeichen (Standardzeichen ist “?”) in einem Feld eingeben oder die Hilfe-Taste (normalerweise PF1) drücken. Dadurch rufen Sie eine Helproutine auf.

Anmerkungen:

- ① *Das Hilfe-Zeichen ist nur einmal einzugeben.*
- ② *Das Hilfe-Zeichen muß das einzige geänderte Zeichen in der Eingabe-Zeichenkette sein.*
- ③ *Das Hilfe-Zeichen muß das erste Zeichen in der Eingabe-Zeichenkette sein.*
- ④ *Wird eine Helproutine für ein numerisches Feld angegeben, läßt Natural nichtsdestoweniger die Eingabe eines Fragezeichens zum Aufrufen der Helproutine zu. Ungeachtet dessen überprüft Natural, ob es sich bei Eingaben in das Feld um gültige numerische Eingaben handelt.*

Die Hilfe-Taste kann — falls sie nicht bereits festgelegt ist — mit einem SET KEY-Statement bestimmt werden:

```
SET KEY PF1=HELP
```

Sie können eine Helproutine nur dann aufrufen, wenn sie in dem Program oder der Map, von dem/der sie aufgerufen wird, spezifiziert ist.

Helproutinen spezifizieren

Eine Helproutine kann folgendermaßen angegeben werden:

- in einem Programm: auf Statement-Ebene und auf Feldebene
- in einer Map: auf Map-Ebene und auf Feldebene.

Wenn Sie Hilfe für ein Feld anfordern, dem keine Helproutine zugeordnet ist, oder wenn Sie Hilfe anfordern, ohne daß ein bestimmtes Feld referenziert wird, dann wird die auf der jeweiligen Statement- oder Map-Ebene angegebene Helproutine aufgerufen.

Eine Helproutine kann auch über ein REINPUT USING HELP-Statement aufgerufen werden (entweder im Programm selbst oder in einer Processing Rule). Falls das REINPUT USING HELP-Statement eine MARK-Option enthält, wird die Helproutine für das MARKierte Feld aufgerufen. Ist keine feldspezifische Helproutine angegeben, wird die Helproutine für die betreffende Map aufgerufen.

Ein REINPUT-Statement in einer Helproutine kann sich nur auf INPUT-Statements innerhalb derselben Helproutine beziehen.

Es gibt zwei Möglichkeiten, eine Helproutine anzugeben, entweder mit dem Session-Parameter HE in einem INPUT-Statement:

```
INPUT (HE='HELP2112')
```

oder im “Extended Field Editing”-Bereich des Map-Editors (wie in Ihrem *Natural Benutzerhandbuch* bzw. *User's Guide* beschrieben).

Der Name einer Helproutine kann entweder eine alphanumerische Konstante sein oder eine alphanumerische Variable, die den Namen enthält. Falls es sich um eine Konstante handelt, muß der Name der Helproutine in Apostrophen angegeben werden.

Programmierhinweise für Helproutinen

Die Verarbeitung einer Helproutine kann mit einem ESCAPE ROUTINE-Statement gestoppt werden.

Bitte beachten Sie, daß ein END OF TRANSACTION- bzw. BACKOUT TRANSACTION-Statement in einer Helproutine die Transaktionslogik des Hauptprogramms beeinflußt.

Parameter an Helprouinen übergeben

Eine Helproutine kann auf die gerade aktive Global Data Area zugreifen (aber keine eigene Global Data Area haben). Außerdem kann sie eine eigene Local Data Area haben.

Darüber hinaus können Daten von der/an die Helproutine über Parameter übergeben werden. Eine Helproutine kann bis zu 20 explizite Parameter und einen impliziten Parameter haben. Die expliziten Parameter werden mit dem Operanden "HE" hinter dem Namen der Helproutine ausgegeben:

```
HE='MYHELP', '001'
```

Der implizite Parameter ist das Feld, für das die Helproutine aufgerufen wurde:

```
INPUT #A (A5) (HE='YOURHELP', '001')
```

wobei "001" der explizite Parameter ist und "#A" der implizite Parameter, also das Feld.

Dies wird im DEFINE DATA PARAMETER-Statement der Helproutine wie folgt definiert:

```
DEFINE DATA PARAMETER
1 #PARM1 (A3)          /* expliziter Parameter
1 #PARM2 (A5)          /* impliziter Parameter
END-DEFINE
```

Bitte beachten Sie, daß im obigen Beispiel der implizite Parameter #PARM2 auch weggelassen werden kann. Der implizite Parameter dient dazu, auf das Feld zuzugreifen, für das Hilfe angefordert wurde, sowie dazu, Daten von der Helproutine an das Feld zu übergeben. Es wäre beispielsweise denkbar, als Helproutine ein Rechenprogramm zu haben und das Rechenergebnis an das Feld zu übergeben.

Anmerkung:

Wenn Hilfe angefordert wird, so wird die Helproutine aufgerufen, bevor irgendwelche Daten vom Bildschirm an die Datenbereiche des Programms weitergegeben werden. Das bedeutet, daß Helprouinen nicht auf Daten zugreifen können, die während derselben Bildschirmtransaktion eingegeben wurden.

Nach Beenden der Helproutine werden auf dem Bildschirm die Feldwerte aktualisiert, die durch die Helproutine verändert wurden — mit Ausnahme der Felder, deren Inhalte bereits vorher vom Benutzer verändert worden waren, aber inklusive des Feldes, für das Hilfe angefordert wurde.

(Ausnahme: wenn das Feld, für das Hilfe angefordert wurde, durch dynamische Attribute (Parameter DY) in mehrere Teile unterteilt wird und der Teil, in den das Fragezeichen eingegeben wurde sich *nach* einem vom Benutzer veränderten Teil befindet, wird eine durch die Helproutine bewirkte Veränderung des Feldinhalts *nicht* wirksam.)

Anmerkung:

Kontrollvariablen werden nach Beenden der Helproutine nicht noch einmal ausgewertet, selbst wenn sie innerhalb der Helproutine verändert wurden.

Gleichheitszeichen-Option

Es ist möglich, das Gleichheitszeichen (=) als expliziten Parameter anzugeben:

```
INPUT PERSONNEL-NUMBER (HE='HELPROUT',=)
```

Dieser Parameter wird dann als internes Feld (A65) verarbeitet, welches den Namen des Feldes bzw. der Map enthält. Die entsprechende Helproutine würde beispielsweise folgendermaßen beginnen:

```
DEFINE DATA PARAMETER
1 FNAME (A65)           /* enthält 'PERSONNEL-NUMBER'
1 FVALUE (N8)          /* Feldwert (wahlweise)
END-DEFINE
```

Diese Möglichkeit kann dazu eingesetzt werden, auf eine übergreifende Helproutine zuzugreifen, die den Feldnamen liest und feldspezifische Hilfe liefert, indem sie auf die Online-Dokumentation der Anwendung oder auf das PREDICT-Data-Dictionary zugreift.

Array-Felder

Ist das Feld, für das Hilfe aufgerufen wird, Teil eines Arrays, dann werden seine Indexangaben als implizite Parameter (1–3 je nach Rang ungeachtet der expliziten Parameter) angegeben. Diese Parameter haben das Format I2.

```
INPUT A(*,*) (HE='HELPROUT',=)
```

Die entsprechende Helproutine würde wie folgt beginnen:

```
DEFINE DATA PARAMETER
1 FNAME (A65)           /* enthält 'A'
1 FVALUE (N8)          /* Wert des ausgewählten Elements
1 FINDEX1 (I2)         /* Index der 1. Dimension
1 FINDEX2 (I2)         /* Index der 2. Dimension
END-DEFINE
...
```

Hilfe als eingeblendetes Fenster

Sie können die Größe eines Hilfe-Schirms so festlegen, daß sie kleiner ist als die Größe Ihres Bildschirms. In diesem Fall wird der Hilfe-Schirm als eingerahmtes Fenster auf dem Bildschirm eingeblendet:

```

*****
                                     PERSONNEL INFORMATION

PLEASE ENTER NAME: ? _____
PLEASE ENTER CITY:  _____
+-----+
TYPE IN . TO STOP  !
                   ! Type in the name of an   !
                   ! employee in the first    !
                   ! field and press ENTER.   !
                   ! You will then receive   !
                   ! a list of all employees  !
                   ! of that name.           !
                   !                           !
                   ! For a list of employees  !
                   ! of a certain name who   !
                   ! live in a certain city,  !
                   ! type in a name in the   !
                   ! first field and a city   !
                   ! in the second field     !
                   ! and press ENTER.        !
***** ! *****
+-----+

```

Innerhalb einer Helprououtine haben Sie folgende Möglichkeiten, die Größe eines Fensters zu bestimmen:

- in einem FORMAT-Statement (z.B. FORMAT PS=15 LS=30)
- über ein INPUT USING MAP-Statement; in diesem Fall gilt die für die verwendete Map (in den “Map Settings”) festgelegte Größe
- durch ein DEFINE WINDOW-Statement; mit diesem Statement können Sie ein Fenster entweder explizit definieren oder dies Natural überlassen (Natural wird dann die Größe des Fensters je nach Inhalt festlegen).

Die Position des eingeblendeten Fensters wird automatisch in Abhängigkeit von der Position des Feldes, für das Hilfe angefordert wurde, bestimmt. Natural plaziert das Fenster automatisch möglichst nahe an das Feld, ohne es zu überdecken. Mit dem DEFINE WINDOW-Statement können Sie die Position des Fensters auch selbst bestimmen.

Weitere Informationen über Bildschirmfenster finden Sie im *Natural Statements-Handbuch* unter DEFINE WINDOW sowie im *Natural Referenzhandbuch* unter Terminalkommando %W.

Mehrfache Verwendung von Sourcecode — Copycode

Copycode ist ein Stück Sourcecode, das mit einem INCLUDE-Statement in ein anderes Objekt eingefügt werden kann.

Wenn Sie einen Statement-Block haben, der in identischer Form in mehreren Objekten erscheinen soll, können Sie Copycode verwenden, anstatt den Statement-Block mehrmals zu kodieren. Dadurch reduziert sich der Kodieraufwand, und gleichzeitig ist sichergestellt, daß die Blöcke tatsächlich identisch sind.

Der Copycode wird bei der Kompilierung eingefügt; d.h. die Sourcecode-Zeilen des Copycode werden nicht physisch in den Sourcecode des Objekts, das das INCLUDE-Statement enthält, eingefügt, sondern sie werden bei der Kompilierung berücksichtigt und sind so Bestandteil des resultierenden Objektmoduls.

Wenn Sie also den Sourcecode eines Copycode verändern, müssen Sie auch alle Objekte, in denen dieser Copycode verwendet wird, neu mit STOW kompilieren.

Copycode kann nicht selbständig ausgeführt werden. Er kann nicht mit dem Systemkommando STOW in Objektform sondern nur in Sourceform mit dem Systemkommando SAVE gespeichert werden.

Weitere Informationen zu Copycode finden Sie bei der Beschreibung des INCLUDE-Statements im *Natural Statements-Handbuch*.

Natural-Objekte dokumentieren — Text

Der Natural-Objekttyp “Text” wird dafür verwendet, Texte anstatt Programme zu schreiben. Sie können einen beliebigen Text schreiben (eine Syntax-Prüfung gibt es nicht). Sie können diesen Objekttyp dafür benutzen, eine Dokumentation für Natural-Objekte zu erstellen, die wesentlich umfangreicher sein kann, als es z.B. innerhalb des Sourcecodes eines Objektes möglich wäre.

“Text” kann auch hilfreich sein, wenn Ihnen PREDICT nicht zur Dokumentation von Objekten zur Verfügung steht.

Den Text schreiben Sie im Natural-Programm-Editor. Der einzige Unterschied zur Programmerstellung liegt darin, daß der Text, den Sie schreiben, unverändert bleibt, d.h. es wird weder eine Umsetzung von Klein- in Großbuchstaben noch eine Leerzeilenunterdrückung vorgenommen (vorausgesetzt, in Ihrem Editorprofil ist Empty Line Suppression auf “N” und Editing in Lower Case auf “Y” gesetzt).

Eine ausführliche Beschreibung des Editorprofils finden Sie im *Natural Benutzerhandbuch* bzw. *User's Guide for Windows*.

“Text” kann nur in Sourceform (mit dem Systemkommando SAVE) gespeichert werden, aber nicht in Objektform (mit dem Systemkommando STOW). Er kann nicht mit RUN ausgeführt, sondern lediglich im Editor angezeigt werden.

Ereignisgesteuerte Anwendungen erstellen — Dialog

Dialoge werden im Zusammenhang mit sogenannter “ereignisgesteuerter Programmierung” (event-driven programming) bei der Erstellung von Natural-Anwendungen für graphische Benutzeroberflächen (GUIs) verwendet.

Informationen zu Dialogen und ereignisgesteuerter Programmierung finden Sie im *Natural User’s Guide for Windows*.

Verteilte objekt-basierte Anwendungen erstellen — Class

Klassen (Classes) werden im Zusammenhang mit NaturalX bei der Erstellung sogenannter “verteilter objekt-basierter Anwendungen” für den Einsatz in Client/Server-Umgebungen verwendet.

Informationen zu Klassen finden Sie in der *NaturalX-Dokumentation*.

Nicht-Natural-Dateien benutzen — Ressource

Ressourcen stehen nur in Verbindung mit Natural unter Windows 98 und Windows NT/2000 zur Verfügung.

Natural unterscheidet zwischen zwei Arten von Ressourcen:

- **Geteilte Ressourcen**
Eine geteilte Ressource ist eine Nicht-Natural-Datei, die in einer Natural-Anwendung verwendet und im Natural-Library-System verwaltet wird.
- **Private Ressourcen**
Eine private Ressource ist eine Datei, die nur einem Natural-Objekt zugewiesen ist und als Teil dieses Objektes betrachtet wird. Ein Objekt kann höchstens eine private Ressourcen-Datei haben. Zur Zeit haben nur Natural-Dialoge private Ressourcen.

Sowohl geteilte als auch private Ressourcen, die zu einer Natural-Library gehören, werden in einem Subdirectory (Unterverzeichnis) mit Namen `..\RES` in dem Directory (Verzeichnis) gepflegt, das die Natural-Library im Dateisystem darstellt.

Geteilte Ressourcen

Eine geteilte Ressource ist eine Nicht-Natural-Datei, die in einer Natural-Anwendung verwendet und im Natural-Library-System gepflegt wird. Eine als geteilte Ressource zu verwendende Nicht-Natural-Datei muß im Subdirectory (Unterverzeichnis) mit Namen `..\RES` einer Natural-Library enthalten sein.

Beispiel — Eine geteilte Ressource verwenden

Die Bitmap `MYPICTURE.BMP` soll in einer Bitmap Control in einem Dialog `MYDLG` angezeigt werden, der in der Library `MYLIB` enthalten ist. Zuerst wird die Bitmap durch Verschieben in das Directory (Verzeichnis) `..\MYLIB\RES` in die Natural-Library `MYLIB` gestellt. Das folgende Programmcode-Stück vom Dialog `MYDLG` zeigt, wie sie dann der Bitmap Control zugewiesen wird:

```
DEFINE DATA LOCAL
  01 #BM-1 HANDLE OF BITMAP
  ...
END-DEFINE
* (Erstellen der ausgelassenen Bitmap Control.)
...
#BM-1.BITMAP-FILE-NAME := "MYPICTURE.BMP" ...
```

Die Vorteile der Verwendung der Bitmap als geteilte Ressource sind folgende:

- Der Dateiname kann im Natural-Dialog ohne einen Pfadnamen angegeben werden.
- Die Datei kann in einer Natural-Library zusammen mit dem sie verwendenden Natural-Objekt gehalten werden.

Anmerkung:

In früheren Natural-Versionen wurden Nicht-Natural-Dateien gewöhnlich in einem Directory (Verzeichnis) gehalten, das mit der Umgebungsvariable `NATGUI_BMP` definiert wurde. Vorhandene, diese Methode verwendende Anwendungen arbeiten genauso wie vorher, weil Natural immer nach einer geteilten Ressourcen-Datei in diesem Verzeichnis sucht, wenn es in der aktuellen Library nicht gefunden wurde.

Private Ressourcen

Private Ressourcen werden intern von Natural verwendet, um binäre Daten zu speichern, die Bestandteil der Natural-Objekte sind. Diese Dateien werden von der Dateinamens-Extension NR* erkannt, wobei * ein Zeichen ist, das vom Natural-Objektyp abhängig ist. Natural verwaltet automatisch private Ressourcen-Dateien und ihren Inhalt.

Ein Natural-Objekt kann maximal eine private Ressourcen-Datei haben. Zur Zeit haben nur Natural-Dialoge eine private Ressourcen-Datei. Diese Datei wird verwendet, um die Konfiguration der ActiveX Controls zu speichern, die in einem Dialog definiert werden und mit ihren eigenen Property Pages konfiguriert sind. Wie eine ActiveX Control konfiguriert wird, entnehmen Sie den ActiveX Control Property Pages.

Beispiel — Private Ressourcen

Der Name der privaten Ressourcen-Datei des Dialogs MYDLG ist MYDLG.NR3. Natural erstellt, ändert und löscht diese Datei automatisch je nach Erfordernis, wenn der Dialog erstellt, geändert, gelöscht wird, usw. Die private Ressourcen-Datei wird verwendet, um binäre Daten zu speichern, die mit dem Dialog MYDLG verbunden sind.

WEITERE PROGRAMMIERASPEKTE

In diesem Kapitel werden folgende Themen behandelt:

- das END-Statement zum Beenden eines Programms
- das STOP-Statement zum Beenden einer Anwendung
- das IF-Statement für bedingte Verarbeitung
- Schleifenverarbeitung
- Gruppenwechsel-Verarbeitung
- Datenberechnungen
- Systemvariablen und Systemfunktionen
- der Stack
- Verarbeitung von Datumsinformationen.

Ende eines Programms — Das END-Statement

Das END-Statement dient dazu, das Ende eines Programms, eines Subprogramms, einer externen Subroutine bzw. einer Helproutine zu kennzeichnen.

Jedes dieser Objekte muß als letztes Statement ein END-Statement enthalten.

Jedes Objekt darf nur ein END-Statement enthalten.

Ende einer Anwendung — Das STOP-Statement

Das STOP-Statement dient dazu, die Ausführung einer Natural-Anwendung abubrechen. Ganz gleich, wo ein STOP-Statement in einer Anwendung ausgeführt wird, beendet es sofort die Ausführung der gesamten Anwendung.

Bedingte Verarbeitung — Das IF-Statement

Mit dem IF-Statement können Sie eine logische Bedingung definieren und Statements angeben, die in Abhängigkeit von dieser logischen Bedingung verarbeitet werden sollen.

Das IF-Statement hat drei Bestandteile: IF, THEN und ELSE.

- Mit der IF-Klausel geben Sie eine logische Bedingung an, die erfüllt werden soll.
- Mit der THEN-Klausel geben Sie die Statements an, die ausgeführt werden sollen, wenn diese Bedingung erfüllt wird.
- Mit der (wahlweise verwendbaren) ELSE-Klausel haben Sie zusätzlich die Möglichkeit, Statements anzugeben, die ausgeführt werden sollen, wenn die Bedingung *nicht* erfüllt wird.

Ein IF-Statement hat also folgende allgemeine Form:

```
IF Bedingung
  THEN führe Statement(s) aus
  ELSE führe andere(s) Statement(s) aus
END-IF
```

Falls Sie wünschen, daß eine bestimmte Verarbeitung nur ausgeführt werden soll, wenn eine IF-Bedingung *nicht* erfüllt wird, können Sie die Klausel THEN IGNORE verwenden, d.h. die IF-Bedingung wird ignoriert, wenn sie erfüllt wird.

Weitere Informationen zu logischen Bedingungen finden Sie im Kapitel **Allgemeine Informationen** des *Natural-Referenzhandbuchs*.

Beispiel für IF-Statement:

```

** Example Program 'IFX01'
DEFINE DATA LOCAL
  1 MYVIEW VIEW OF EMPLOYEEES
    2 NAME
    2 BIRTH
    2 CITY
    2 SALARY (1:1)
END-DEFINE
*
LIMIT 7
READ MYVIEW BY CITY STARTING FROM 'C'
  IF SALARY (1) LT 40000 THEN
    WRITE NOTITLE '*****' NAME 30X 'SALARY LT 40000'
  ELSE
    DISPLAY NAME BIRTH (EM=YYYY-MM-DD) SALARY (1)
  END-IF
END-READ
END

```

Der IF-Statement-Block im obigen Programm bewirkt folgende bedingte Verarbeitung:

- Wenn (IF) das Gehalt weniger als 40000 beträgt, dann (THEN) soll das WRITE-Statement ausgeführt werden.
- Andernfalls (ELSE), d.h. wenn das Gehalt 40000 und mehr beträgt, soll das DISPLAY-Statement ausgeführt werden.

Das Programm erzeugt folgende Ausgabe:

NAME	DATE OF BIRTH	ANNUAL SALARY	
***** KEEN			SALARY LT 40000
***** FORRESTER			SALARY LT 40000
***** JONES			SALARY LT 40000
***** MELKANOFF			SALARY LT 40000
DAVENPORT	1948-12-25	42000	
GEORGES	1949-10-26	182800	
***** FULLERTON			SALARY LT 40000

Geschachtelte IF-Statements

Es ist möglich, mehrere IF-Statements ineinander zu verschachteln, zum Beispiel, indem Sie die Ausführung einer THEN-Klausel durch ein weiteres, in der THEN-Klausel angegebenes IF-Statement von einer zusätzlichen Bedingung abhängig machen.

Beispiel für geschachtelte IF-Statements:

```

** Example Program 'IFX02'
DEFINE DATA LOCAL
  1 MYVIEW VIEW OF EMPLOYEES
    2 NAME
    2 CITY
    2 SALARY (1:1)
    2 BIRTH
    2 PERSONNEL-ID
  1 MYVIEW2 VIEW OF VEHICLES
    2 PERSONNEL-ID
    2 MAKE
  1 #BIRTH (D)
END-DEFINE
*
MOVE EDITED '19450101' TO #BIRTH (EM=YYYYMMDD)
*
LIMIT 20
FND1. FIND MYVIEW WITH CITY = 'BOSTON'
      SORTED BY NAME
      IF SALARY (1) LESS THAN 20000
        THEN WRITE NOTITLE '*****' NAME 30X 'SALARY LT 20000'
      ELSE
        IF BIRTH GT #BIRTH
          FIND MYVIEW2 WITH PERSONNEL-ID = PERSONNEL-ID (FND1.)
            DISPLAY (IS=ON) NAME BIRTH (EM=YYYY-MM-DD)
              SALARY (1) MAKE (AL=8 IS=OFF)
        END-FIND
      END-IF
END-IF
SKIP 1
END-FIND
END

```

Das obige Programm mit geschachtelten IF-Statements erzeugt folgende Ausgabe:

NAME	DATE OF BIRTH	ANNUAL SALARY	MAKE
***** COHEN			SALARY LT 20000
CREMER	1972-12-14	20000	FORD
***** FLEMING			SALARY LT 20000
***** GREENACRE			SALARY LT 20000
PERREAULT	1950-05-12	30500	CHRYSLER
***** SHAW			SALARY LT 20000
STANWOOD	1946-09-08	31000	CHRYSLER FORD

Weiteres Beispiel für IF-Statement:

Siehe Programm IFX03 in Library SYSEXPG.

Schleifenverarbeitung

Eine Verarbeitungsschleife ist eine Gruppe von Statements, deren Ausführung so oft wiederholt wird, bis eine bestimmte Bedingung erfüllt ist, oder solange eine bestimmte Bedingung gegeben ist.

Verarbeitungsschleifen lassen sich in Datenbankschleifen und Nicht-Datenbankschleifen unterteilen:

- *Datenbankschleifen* werden von Natural automatisch erzeugt, um die Daten, die mit einem READ-, FIND- oder HISTOGRAM-Statement von einer Datenbank gelesen werden, zu verarbeiten. Diese Statements sind im Kapitel **Datenbankzugriffe** (Seite 47) beschrieben.
- *Nicht-Datenbankschleifen* (d.h. Schleifen ohne Datenbankzugriff) werden mit folgenden Statements erzeugt: REPEAT, FOR, CALL FILE, CALL LOOP, SORT und READ WORK FILE.

Es können mehrere Verarbeitungsschleifen gleichzeitig aktiv sein. In einer gerade aktiven, d.h. noch nicht abgeschlossenen Schleife können weitere Schleifen eingebettet werden.

Jede Verarbeitungsschleife muß durch ein entsprechendes END-...-Statement beendet werden (z.B. END-REPEAT, END-FOR, usw.).

Das SORT-Statement, mit dem das Sortierprogramm des Betriebssystems aufgerufen wird, beendet alle aktiven Schleifen und löst eine neue Schleife aus.

Die folgenden Themen werden nachfolgend behandelt:

- Schleifendurchläufe bei Datenbankzugriffen begrenzen
- Durchläufe bei Nicht-Datenbankzugriffsschleifen begrenzen — das REPEAT-Statement
- Verarbeitungsschleife verlassen — das ESCAPE-Statement
- Schleifen innerhalb von Schleifen
- Statements innerhalb eines Programms referenzieren.

Schleifendurchläufe bei Datenbankzugriffen begrenzen

Bei den Statements READ, FIND und HISTOGRAM haben Sie drei Möglichkeiten, die Anzahl, wie oft eine Verarbeitungsschleife durchlaufen werden soll, zu begrenzen:

- mit dem Session-Parameter LT
- mit einem LIMIT-Statement
- oder mit einer Limit-Notation im READ-/FIND-/HISTOGRAM-Statement selbst.

LT-Session-Parameter

Mit dem Systemkommando GLOBALS können Sie den Session-Parameter LT angeben, der die Anzahl der Datensätze, die in einer Datenbank-Verarbeitungsschleife gelesen werden sollen, begrenzt.

Beispiel:

```
GLOBALS LT=100
```

Dieses Limit gilt für alle READ-, FIND- und HISTOGRAM-Schleifen in der gesamten Session.

LIMIT-Statement

In einem Programm können Sie die Anzahl der Datensätze, die in einer Datenbank-Verarbeitungsschleife gelesen werden sollen, mit einem LIMIT-Statement begrenzen.

Beispiel:

```
LIMIT 100
```

Das LIMIT-Statement gilt für alle nachfolgenden READ-, FIND- und HISTOGRAM-Schleifen im Programm, es sein denn, es wird durch ein anderes LIMIT-Statement oder eine Limit-Notation außer Kraft gesetzt.

Limit-Notation

In einem READ-, FIND- oder HISTOGRAM-Statement selbst können Sie die Anzahl der Datensätze, die gelesen werden sollen, in Klammern unmittelbar hinter dem Statement-Namen angeben.

Beispiel:

```
READ (10) VIEWXYZ BY NAME
```

Diese Limit-Notation hat Vorrang vor allen anderen Limits, gilt aber nur für das betreffende Statement.

Wenn das mit dem LT-Parameter angegebene Limit kleiner ist als ein mit einem LIMIT-Statement oder einer Limit-Notation angegebenes, dann hat das LT-Limit Vorrang vor diesen anderen Limits.

Durchläufe bei Nicht-Datenbankzugriffsschleifen begrenzen — das REPEAT-Statement

Anfang und Ende von Verarbeitungsschleifen, die keinen Datenbankzugriff beinhalten, basieren auf einer logischen oder sonstwie die Schleife begrenzenden Bedingung.

Stellvertretend für Nicht-Datenbankschleifen-Statements wird hier das Statement REPEAT behandelt.

Mit dem REPEAT-Statement geben Sie ein oder mehrere Statements an, die wiederholt ausgeführt werden sollen. Außerdem können Sie eine logische Bedingung angeben, so daß die Statements nur ausgeführt werden, solange oder bis diese Bedingung erfüllt ist. Die Bedingung geben Sie in einer UNTIL-Klausel oder in einer WHILE-Klausel an:

- Bei einer UNTIL-Klausel wird die Schleife so oft ausgeführt, *bis* (UNTIL) die logische Bedingung erfüllt ist, d.h. die Schleife wird beendet, sobald der in der Bedingung angegebene Zustand erreicht ist.
- Bei einer WHILE-Klausel wird die Schleife ausgeführt, *während* (WHILE) der in der Bedingung angegebene Zustand besteht, d.h. die Schleife wird beendet, sobald die Bedingung nicht mehr erfüllt wird.

Wenn Sie keine logische Bedingung angeben, muß die REPEAT-Schleife mit einem ESCAPE-, STOP- oder TERMINATE-Statement verlassen werden:

- Ein ESCAPE-Statement (siehe nächsten Abschnitt) beendet die Verarbeitung der Schleife und setzt die Verarbeitung außerhalb der Schleife fort.
- Ein STOP-Statement bricht die Ausführung der gesamten Natural-Anwendung ab.
- Ein TERMINATE-Statement bricht die Ausführung der Natural-Anwendung ab und beendet die Natural-Session.

Beispiel für REPEAT-Statement:

```

** Example Program 'REPEAX01'
DEFINE DATA LOCAL
1 MYVIEW VIEW OF EMPLOYEES
  2 NAME
  2 SALARY (1:1)
1 #PAY1 (N8)
END-DEFINE
*
READ (5) MYVIEW BY NAME WHERE SALARY (1) = 30000 THRU 39999
  MOVE SALARY (1) TO #PAY1
  REPEAT WHILE #PAY1 LT 40000
    MULTIPLY #PAY1 BY 1.1
    DISPLAY NAME (IS=ON) SALARY (1)(IS=ON) #PAY1
  END-REPEAT
  SKIP 1
END-READ
END

```

Das obige Programm erzeugt folgende Ausgabe:

Page	1	97-08-19 18:42:53	
NAME	ANNUAL SALARY	#PAY1	
ADKINSON	34500	37950	41745
	33500	36850	40535
	36000	39600	43560
AFANASSIEV	37000	40700	
ALEXANDER	34500	37950	41745

Verarbeitungsschleife verlassen — das ESCAPE-Statement

Mit dem ESCAPE-Statement können Sie die Ausführung einer Verarbeitungsschleife abbrechen, und zwar aufgrund einer logischen Bedingung.

Das ESCAPE-Statement kann Teil eines IF-Statements sein oder an eines der Statements AT END OF DATA, AT END OF PAGE oder AT BREAK geknüpft sein; es kann aber auch als eigenständiges Statement in Ausführung der einer Verarbeitungsschleife zugrundeliegenden logischen Bedingungen stehen.

Mit dem ESCAPE-Statement haben Sie die Optionen TOP und BOTTOM, mit denen Sie festlegen, wo die Verarbeitung fortgesetzt werden soll, nachdem die Schleife mit ESCAPE verlassen wurde:

- Bei ESCAPE TOP wird die Verarbeitung am Anfang der Schleife, d.h. mit dem nächsten Schleifendurchlauf, fortgesetzt.
- Bei ESCAPE BOTTOM wird die Verarbeitung mit dem ersten Statement, das nach der Schleife kommt, fortgesetzt.

Sie können innerhalb einer Verarbeitungsschleife auch mehrere ESCAPE-Statements angeben.

Weitere Informationen und Beispiele zum ESCAPE-Statement finden Sie im *Natural Statements-Handbuch*.

Schleifen innerhalb von Schleifen

Mit Natural haben Sie die Möglichkeit, Schleifen innerhalb von Schleifen auszulösen und so eine ganze Hierarchie ineinander verschachtelter Schleifenkonstruktionen aufzubauen. Sind mehrere Datenbankzugriffsschleifen ineinander verschachtelt, so durchläuft jeder gelesene Datensatz, der die Auswahlkriterien erfüllt, nacheinander die einzelnen Schleifen, bevor der nächste Datensatz verarbeitet wird.

Mehrere Datenbankzugriffs- und Nicht-Datenbankzugriffsschleifen können ineinander verschachtelt werden. Verarbeitungsschleifen können auch Teil einer bedingten Verarbeitung sein.

Das folgende Programm zeigt eine Hierarchie zweier Verarbeitungsschleifen, wobei sich eine FIND-Schleife innerhalb einer anderen FIND-Schleife befindet.

Beispiel für geschachtelte FIND-Statements:

```

** Example Program 'FINDX06'
DEFINE DATA LOCAL
1 EMPLOY-VIEW VIEW OF EMPLOYEES
  2 CITY
  2 NAME
  2 PERSONNEL-ID
1 VEH-VIEW VIEW OF VEHICLES
  2 MAKE
  2 PERSONNEL-ID
END-DEFINE
*
FND1. FIND EMPLOY-VIEW WITH CITY = 'NEW YORK' OR = 'BEVERLEY HILLS'
      FIND (1) VEH-VIEW WITH PERSONNEL-ID = PERSONNEL-ID (FND1.)
          DISPLAY NOTITLE NAME CITY MAKE
      END-FIND
END-FIND
END

```

Das obige Programm liest Daten von mehreren Dateien. Die äußere FIND-Schleife wählt von der EMPLOYEES-Datei alle Personen aus, die in New York oder Beverley Hills wohnen. Für jeden in der äußeren Schleife ausgewählten Datensatz wird die innere FIND-Schleife durchlaufen, in der die Fahrzeugdaten der betreffenden Personen von der VEHICLES-Datei gelesen werden. Das Programm erzeugt folgende Ausgabe:

NAME	CITY	MAKE
RUBIN	NEW YORK	FORD
OLLE	BEVERLEY HILLS	GENERAL MOTORS
ADKINSON	BEVERLEY HILLS	FORD
WALLACE	NEW YORK	MAZDA
SPEISER	BEVERLEY HILLS	FORD

Statements innerhalb eines Programms referenzieren

Statement-Referenzierung dient dazu, in einem Programm auf ein vorhergehendes Statement zu verweisen (d.h. dieses Statement zu “referenzieren”), um eine Verarbeitung für einen bestimmten Bereich von Daten auszuführen, Natural Standard-Referenzierung (die bei jedem betroffenen Statement im *Natural Statements-Handbuch* beschrieben ist) aufzuheben, oder zu Programmdokumentationszwecken.

Sie können jedes Natural-Statement referenzieren, das eine Verarbeitungsschleife initiiert und/oder auf Datenelemente in einer Datenbank zugreift (z.B.: READ, FIND, HISTOGRAM, SORT, REPEAT, FOR).

Enthält ein Programm mehrere Verarbeitungsschleifen, so kann man ein bestimmtes Datenbankfeld eindeutig identifizieren, indem man das Statement referenziert, welches zuerst auf das entsprechende Feld in der Datenbank zugriff. (Welche Felder bei welchem Statement referenziert werden dürfen, ersehen Sie im *Natural Statements-Handbuch* in den Operandentabellen der einzelnen Statements aus der Spalte “Referenzierung erlaubt”.)

Außerdem kann eine Referenzierungsnotation in einigen Statements angegeben werden, z.B. bei: AT START OF DATA, AT END OF DATA, AT BREAK und ESCAPE BOTTOM. Normalerweise bezieht sich bei einem AT START OF DATA-, AT END OF DATA- oder AT BREAK-Statement die schleifenbeendende Gruppenwechsel-Bedingung auf die jeweils *äußerste* aktive READ-, FIND-, HISTOGRAM-, SORT- oder READ WORK FILE-Schleife. Mit einer Referenzierungsnotation können Sie die Bedingung auf eine beliebige andere aktive Schleife beziehen.

Wenn Sie bei einem ESCAPE BOTTOM-Statement ein Statement referenzieren, wird die Verarbeitung unmittelbar nach der durch das referenzierte Statement identifizierten Schleife fortgesetzt.

Zur Statement-Referenzierung können Sie entweder ein sogenanntes *Statement-Label* oder die *Sourcecode-Zeilenummer* verwenden.

Ein Statement-Label ist eine Zeichenkette, deren letztes Zeichen ein Punkt (.) sein muß. Der Punkt identifiziert die Zeichenkette als Label.

Ein Statement, das referenziert werden soll, wird mit einem Label markiert, indem das Label an den Anfang der Zeile gestellt wird, in der das Statement steht, Zum Beispiel:

```
0030 ...  
0040 READ1. READ VIEWXYZ BY NAME  
0050 ...
```

In dem Statement, das das markierte Statement referenziert, wird das Label in Klammern an der in der Statement-Syntax dafür vorgesehenen Stelle (siehe Syntaxdiagramme im *Natural Statements-Handbuch*) eingefügt. Zum Beispiel:

```
AT BREAK (READ1.) OF NAME
```

Wenn Sie Sourcecode-Zeilenummern zur Referenzierung verwenden, müssen Sie diese immer vierstellig (vorangestellte Nullen dürfen nicht weggelassen werden) und in Klammern angeben. Zum Beispiel:

```
AT BREAK (0040) OF NAME
```

Bezieht sich in einem DISPLAY- oder WRITE-Statement ein bestimmtes Feld auf ein vorhergegangenes Statement, so wird das Label bzw. die Zeilenummer in Klammern hinter dem jeweiligen Feldnamen angegeben. Zum Beispiel:

```
DISPLAY NAME (READ1.) JOB-TITLE (READ1.) MAKE MODEL
```

Sourcecode-Zeilenummern und Statement-Labels können wahlweise verwendet werden.

Das folgende Programm verwendet Sourcecode-Zeilennummern zur Referenzierung. In diesem Beispiel beziehen sich die Zeilennummern auf Statements, die aufgrund der Programmstruktur ohnehin — auch ohne explizite Referenzierung — referenziert worden wären.

Beispiel mit Zeilennummern:

```
0010 ** Example Program 'LABELX01'
0020 DEFINE DATA LOCAL
0030 1 MYVIEW1 VIEW OF EMPLOYEES
0040   2 NAME
0050   2 FIRST-NAME
0060   2 PERSONNEL-ID
0070 1 MYVIEW2 VIEW OF VEHICLES
0080   2 PERSONNEL-ID
0090   2 MAKE
0100 END-DEFINE
0110 *
0120 LIMIT 15
0130 READ MYVIEW1 BY NAME STARTING FROM 'JONES'
0140   FIND MYVIEW2 WITH PERSONNEL-ID = PERSONNEL-ID (0130)
0150     IF NO RECORDS FOUND
0160       MOVE '***NO CAR***' TO MAKE
0170     END-NOREC
0180     DISPLAY NOTITLE NAME (0130) (IS=ON) FIRST-NAME (0130) (IS=ON)
0190       MAKE (0140)
0200   END-FIND /* (0140)
0210 END-READ /* (0130)
0220 END
```

Das folgende Beispiel zeigt die Verwendung von Statement-Labels. Es ist mit dem vorigen Beispielprogramm identisch bis auf die Tatsache, daß zur Referenzierung der Statements Labels anstelle von Zeilennummern verwendet werden.

Beispiel mit Labels:

```

0010 ** Example Program 'LABELX02'
0020 DEFINE DATA LOCAL
0030 1 MYVIEW1 VIEW OF EMPLOYEES
0040   2 NAME
0050   2 FIRST-NAME
0060   2 PERSONNEL-ID
0070 1 MYVIEW2 VIEW OF VEHICLES
0080   2 PERSONNEL-ID
0090   2 MAKE
0100 END-DEFINE
0110 *
0120 LIMIT 15
0130 RD. READ MYVIEW1 BY NAME STARTING FROM 'JONES'
0140   FD. FIND MYVIEW2 WITH PERSONNEL-ID = PERSONNEL-ID (FD.)
0150     IF NO RECORDS FOUND
0160       MOVE '***NO CAR***' TO MAKE
0170     END-NOREC
0180     DISPLAY NOTITLE NAME (RD.) (IS=ON) FIRST-NAME (RD.) (IS=ON)
0190       MAKE (FD.)
0200   END-FIND /* (FD.)
0210 END-READ /* (RD.)
0220 END

```

Beide Programme erzeugen folgende Ausgabe:

NAME	FIRST-NAME	MAKE
JONES	VIRGINIA	***NO CAR***
	MARSHA	CHRYSLER
		CHRYSLER
	ROBERT	GENERAL MOTORS
	LILLY	***NO CAR***
	EDWARD	GENERAL MOTORS
	MARTHA	***NO CAR***
	LAUREL	GENERAL MOTORS
	KEVIN	DATSUN
	GREGORY	FORD
JOPER	MANFRED	***NO CAR***
JOUSSELIN	DANIEL	RENAULT
JUBE	GABRIEL	***NO CAR***
JUNG	ERNST	***NO CAR***
JUNKIN	JEREMY	***NO CAR***
KAISER	REINER	***NO CAR***

Gruppenwechsel

Ein Gruppenwechsel (“break”) findet statt, wenn der Wert eines Kontrollfeldes sich ändert.

Die Ausführung von Statements kann von einem solchen Gruppenwechsel abhängig gemacht werden. Ein Gruppenwechsel kann auch zur Auswertung von Natural-Systemfunktionen verwendet werden. Systemfunktionen sind weiter unten in diesem Kapitel beschrieben.

- AT BREAK-Statement
- Automatische Gruppenwechsel-Verarbeitung
- BEFORE BREAK PROCESSING-Statement
- Programmabhängige Gruppenwechsel-Verarbeitung — das PERFORM BREAK PROCESSING-Statement.

AT BREAK-Statement

Mit dem Statement AT BREAK können Sie eine Verarbeitung angeben, die immer dann ausgeführt werden soll, wenn ein Gruppenwechsel erfolgt, d.h. wenn der Wert eines Kontrollfeldes, das Sie im AT BREAK-Statement angeben, sich ändert. Als Kontrollfeld können Sie ein Datenbankfeld oder eine Benutzervariable verwenden.

Gruppenwechsel basierend auf einem Datenbankfeld

Das Feld, welches als Kontrollfeld in einem AT BREAK-Statement angegeben wird, ist üblicherweise ein Datenbankfeld.

Beispiel:

```
...  
AT BREAK OF DEPT  
  statements  
END-BREAK  
...
```

In diesem Beispiel ist das Datenbankfeld DEPT das Kontrollfeld; wechselt der Wert des Feldes, beispielsweise von “SALE01” auf “SALE02”, würde dies die Ausführung der im AT BREAK-Statement angegebenen *Statements* auslösen.

Es ist auch möglich, statt eines ganzen Feldes nur einen Teil eines Feldes als Kontrollfeld zu nehmen. Mit der Notation “/n/” können Sie festlegen, daß nur die ersten *n* Stellen eines Feldes auf einen Wertwechsel überprüft werden sollen.

Beispiel:

```
...  
AT BREAK OF DEPT /4/  
    statements  
END-BREAK  
...
```

In diesem Beispiel würden die angegebenen *Statements* nur ausgeführt, wenn sich der Wert der ersten 4 Stellen des Feldes DEPT ändern würde, beispielsweise von “SALE” auf “TECH”; ein Wechsel von “SALE01” auf “SALE02” hingegen würde ignoriert und der AT BREAK-Block nicht ausgeführt werden.

Beispiel für AT BREAK-Statement mit einem Datenbankfeld:

```
** Example Program 'ATBEX01'
DEFINE DATA LOCAL
1 MYVIEW VIEW OF EMPLOYEES
  2 NAME
  2 CITY
  2 COUNTRY
  2 JOB-TITLE
  2 SALARY (1:1)
END-DEFINE
*
READ (5) MYVIEW BY CITY WHERE COUNTRY = 'USA'
  DISPLAY CITY (AL=9) NAME 'POSITION' JOB-TITLE 'SALARY' SALARY (1)
  AT BREAK OF CITY
    WRITE / OLD(CITY) (EM=X^X^X^X^X^X^X^X^X^X^X)
      5X 'AVERAGE:' T*SALARY AVER(SALARY (1)) //
      COUNT(SALARY(1)) 'RECORDS FOUND' /
  END-BREAK
  AT END OF DATA
    WRITE 'TOTAL (ALL RECORDS):' T*SALARY(1) TOTAL(SALARY(1))
  END-ENDDATA
END-READ
END
```

Im obigen Programm wird das erste WRITE-Statement ausgeführt, wenn der Wert des Feldes CITY sich ändert. Im AT BREAK-Statement werden die Systemfunktionen OLD, AVER und COUNT ausgewertet (und in dem WRITE-Statement ausgegeben). In dem AT END OF DATA-Statement wird die Systemfunktion TOTAL ausgewertet. Das Programm erzeugt folgende Ausgabe:

CITY	NAME	POSITION	SALARY
AIKEN	SENKO	PROGRAMMER	31500
A I K E N	AVERAGE :		31500
1 RECORDS FOUND			
ALBUQUERQ	HAMMOND	SECRETARY	22000
ALBUQUERQ	ROLLING	MANAGER	34000
ALBUQUERQ	FREEMAN	MANAGER	34000
ALBUQUERQ	LINCOLN	ANALYST	41000
A L B U Q U E R Q U E	AVERAGE :		32750
4 RECORDS FOUND			
TOTAL (ALL RECORDS) :			162500

Gruppenwechsel basierend auf einer Benutzervariablen

Auch eine Benutzervariable kann als Kontrollfeld in einem AT BREAK-Statement verwendet werden.

Im folgenden Programm wird die Benutzervariable #LOCATION als Kontrollfeld verwendet.

Beispiel für AT BREAK-Statement mit einer Benutzervariablen:

```

** Example Program 'ATBREX02'
DEFINE DATA LOCAL
1 MYVIEW VIEW OF EMPLOYEES
  2 CITY
  2 COUNTRY
  2 JOB-TITLE
  2 SALARY (1:1)
1 #LOCATION (A20)
END-DEFINE
*
READ (5) MYVIEW BY CITY WHERE COUNTRY = 'USA'
  BEFORE BREAK PROCESSING
    COMPRESS CITY 'USA' INTO #LOCATION
  END-BEFORE
  DISPLAY #LOCATION 'POSITION' JOB-TITLE 'SALARY' SALARY (1)
  AT BREAK OF #LOCATION
    SKIP 1
  END-BREAK
END-READ
END

```

Das obige Programm erzeugt folgende Ausgabe:

#LOCATION	POSITION	SALARY
AIKEN USA	PROGRAMMER	31500
ALBUQUERQUE USA	SECRETARY	22000
ALBUQUERQUE USA	MANAGER	34000
ALBUQUERQUE USA	MANAGER	34000
ALBUQUERQUE USA	ANALYST	41000

Gruppenwechsel auf mehreren Ebenen

Mit der Notation “/n/” können Sie, wie bereits erläutert, den Teil eines Feldes zum Kontrollfeld eines Gruppenwechsels machen. Sie können auch mehrere AT BREAK-Statements miteinander kombinieren, wobei bei einem Gruppenwechsel ein ganzes Feld und bei einem anderen ein Teil dieses Feldes Kontrollfeld ist. In diesem Fall muß der übergeordnete Gruppenwechsel (ganzes Feld) vor dem untergeordneten (Teil des Feldes) angegeben werden, d.h. im ersten AT BREAK-Statement muß das ganze Feld, im zweiten das Teilfeld als Kontrollfeld angegeben werden.

Das folgende Beispielprogramm zeigt dies anhand des Feldes DEPT und den ersten 4 Stellen dieses Feldes (DEPT /4/).

Beispiel 1 für mehrere AT BREAK-Statements:

```

** Example Program 'ATBEX03'
DEFINE DATA LOCAL
1 MYVIEW VIEW OF EMPLOYEES
  2 NAME
  2 JOB-TITLE
  2 DEPT
  2 SALARY (1:1)
  2 CURR-CODE (1:1)
END-DEFINE
READ MYVIEW BY DEPT STARTING FROM 'SALE40' ENDING AT 'TECH10'
      WHERE SALARY(1) GT 47000 AND CURR-CODE(1) = 'USD'

  AT BREAK OF DEPT
    WRITE '*** LOWEST BREAK LEVEL ***' /
  END-BREAK
  AT BREAK OF DEPT /4/
    WRITE '*** HIGHEST BREAK LEVEL ***'
  END-BREAK
  DISPLAY DEPT NAME 'POSITION' JOB-TITLE
END-READ
END
    
```

Page	1	97-08-19	18:24:16
DEPARTMENT CODE	NAME	POSITION	
TECH05	HERZOG	MANAGER	
TECH05	LAWLER	MANAGER	
TECH05	MEYER	MANAGER	
*** LOWEST BREAK LEVEL ***			
TECH10	DEKKER	DBA	
*** LOWEST BREAK LEVEL ***			
*** HIGHEST BREAK LEVEL ***			

Im folgenden Programm wird jedesmal, wenn sich der Wert des Feldes DEPT ändert, eine Leerzeile ausgegeben; und jedesmal, wenn sich der Wert in den ersten 4 Stellen von DEPT ändert, wird über die Systemfunktion COUNT die Anzahl der verarbeiteten Datensätze ermittelt.

Beispiel 2 für mehrere AT BREAK-Statements:

```

** Example Program 'ATBEX04'
DEFINE DATA LOCAL
1 MYVIEW VIEW OF EMPLOYEES
  2 DEPT
  2 REDEFINE DEPT
    3 #GENDEP (A4)
  2 NAME
  2 SALARY (1)
END-DEFINE
WRITE TITLE '** PERSONS WITH SALARY > 30000, SORTED BY DEPARTMENT **' /
LIMIT 9
READ MYVIEW BY DEPT FROM 'A' WHERE SALARY(1) > 30000
  DISPLAY 'DEPT' DEPT NAME 'SALARY' SALARY(1)
  AT BREAK OF DEPT
    SKIP 1
  END-BREAK
  AT BREAK OF DEPT /4/
    WRITE COUNT(SALARY(1)) 'RECORDS FOUND IN:' OLD(#GENDEP) /
  END-BREAK
END-READ
END

```

** PERSONS WITH SALARY > 30000, SORTED BY DEPARTMENT **		
DEPT	NAME	SALARY
ADMA01	JENSEN	180000
ADMA01	PETERSEN	105000
ADMA01	MORTENSEN	320000
ADMA01	MADSEN	149000
ADMA01	BUHL	642000
ADMA02	HERMANSEN	391500
ADMA02	PLOUG	162900
ADMA02	HANSEN	234000
8 RECORDS FOUND IN: ADMA		
COMP01	HEURTEBISE	168800
1 RECORDS FOUND IN: COMP		

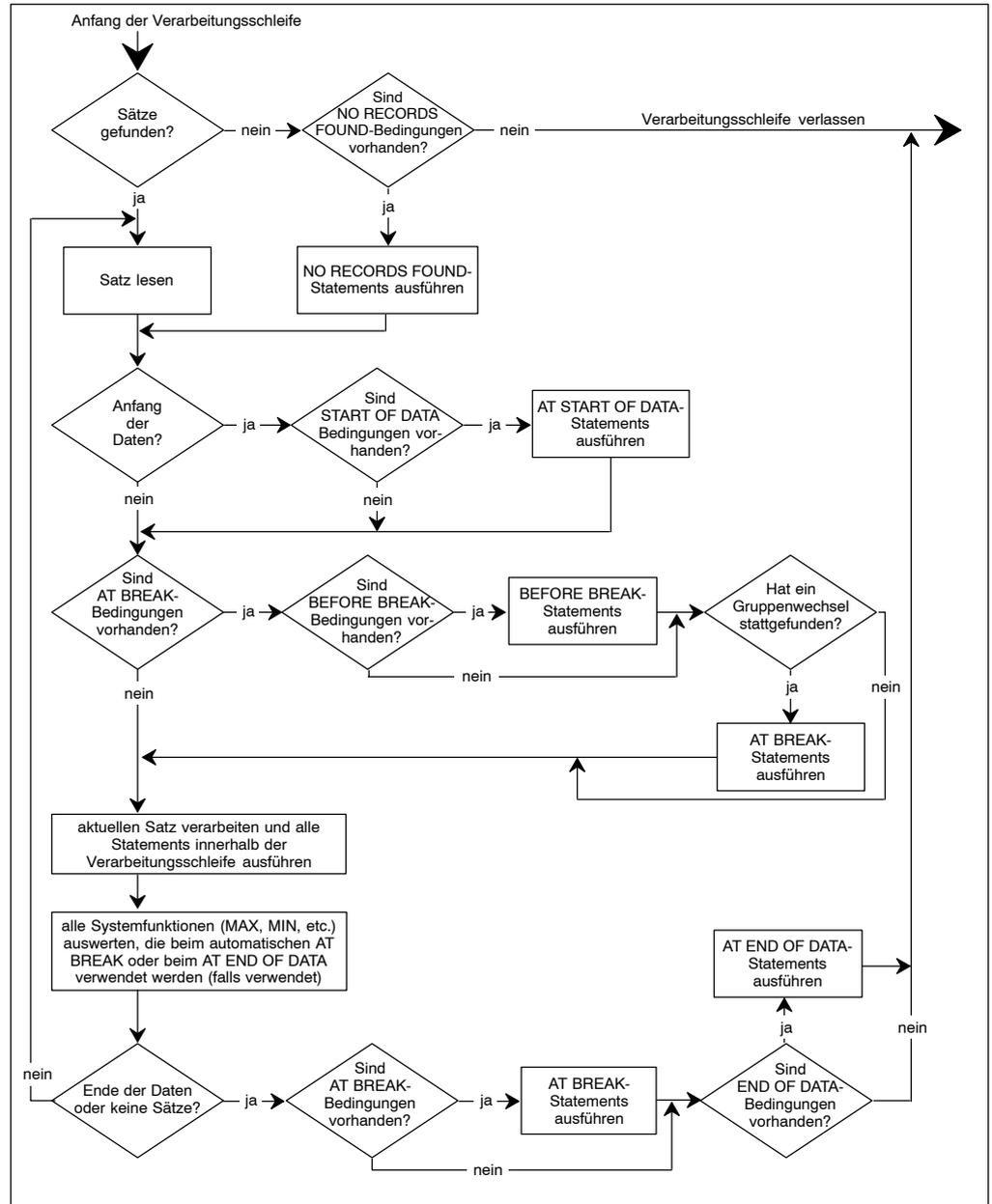
Automatische Gruppenwechsel-Verarbeitung

Automatische Gruppenwechsel-Verarbeitung ist aktiv, wenn innerhalb einer FIND-, READ-, HISTOGRAM-, SORT- oder READ WORK FILE-Schleife ein AT BREAK-Statement steht.

Hierbei wird der Wert des im AT BREAK-Statement angegebenen Kontrollfeldes nur bei den Datensätzen überprüft, die die WITH- und WHERE-Auswahlkriterien der Verarbeitungsschleife erfüllen.

Natural-Systemfunktionen (AVER, MAX, MIN, usw.) werden für jeden Datensatz ausgewertet, nachdem alle in der Verarbeitungsschleife enthaltenen Statements ausgeführt worden sind. Datensätze, die aufgrund des WHERE-Kriteriums nicht verarbeitet werden, werden bei der Auswertung der Systemfunktionen nicht berücksichtigt.

Die Abbildung auf der folgenden Seite veranschaulicht den Verarbeitungsablauf eines automatischen Gruppenwechsels.



CITY	NAME	SALARY	CURRENCY
SALT LAKE CITY	ANDERSON	50000	USD
SALT LAKE CITY	SAMUELSON	24000	USD
S A L T L A K E C I T Y	— MINIMUM:	24000	USD
	— AVERAGE:	37000	USD
	— MAXIMUM:	50000	USD
	— SUM:	74000	USD
		2 RECORDS FOUND	
SAN DIEGO	GEE	60000	USD
S A N D I E G O	— MINIMUM:	60000	USD
	— AVERAGE:	60000	USD
	— MAXIMUM:	60000	USD
	— SUM:	60000	USD
		1 RECORDS FOUND	
	TOTAL (ALL RECORDS):	134000	USD

BEFORE BREAK PROCESSING-Statement

Mit dem Statement `BEFORE BREAK PROCESSING` können Sie Statements angeben, die unmittelbar vor einem Gruppenwechsel ausgeführt werden sollen, d.h. bevor der Wert des Kontrollfeldes geprüft wird, bevor die Statements im `AT BREAK`-Block ausgeführt werden und bevor Systemfunktionen ausgewertet werden.

Beispiel für BEFORE BREAK PROCESSING-Statement:

```

** Example Program 'BEFORX01'
DEFINE DATA LOCAL
1 MYVIEW VIEW OF EMPLOYEES
  2 NAME
  2 FIRST-NAME
  2 SALARY (1:1)
  2 BONUS (1:1,1:1)
1 #INCOME (P11)
END-DEFINE
*
LIMIT 5
READ MYVIEW BY NAME FROM 'B'
  BEFORE BREAK PROCESSING
    COMPUTE #INCOME = SALARY(1) + BONUS(1,1)
  END-BEFORE
  DISPLAY NOTITLE NAME FIRST-NAME (AL=10)
    'ANNUAL/INCOME' #INCOME
    'SALARY' SALARY(1) (LC==) / '+ BONUS' BONUS(1,1) (IC=+)
  AT BREAK OF #INCOME
    WRITE T*#INCOME '-'(24)
  END-BREAK
END-READ
END

```

NAME	FIRST-NAME	ANNUAL INCOME	SALARY + BONUS
BACHMANN	HANS	297546 =	293546 +4000
BAECKER	JOHANNES	420244 =	413644 +6600
BAECKER	KARL	52650 =	48600 +4050
BAGAZJA	MARJAN	152700 =	129700 +23000
BAILLET	PATRICK	198500 =	188000 +10500

Programmabhängige Gruppenwechsel-Verarbeitung — das **PERFORM BREAK PROCESSING**-Statement

Bei automatischer Gruppenwechsel-Verarbeitung werden die im **AT BREAK**-Block angegebenen Statements jedesmal ausgeführt, wenn sich der Wert des angegebenen Kontrollfeldes ändert — unabhängig von der Position des **AT BREAK**-Statements in der Verarbeitungsschleife.

Mit einem **PERFORM BREAK PROCESSING**-Statement können Sie selbst festlegen, wo in einer Verarbeitungsschleife eine Gruppenwechsel-Verarbeitung ausgeführt werden soll. Das **PERFORM BREAK PROCESSING**-Statement wird dann ausgeführt, wenn es im Verarbeitungsablauf des Programms angetroffen wird.

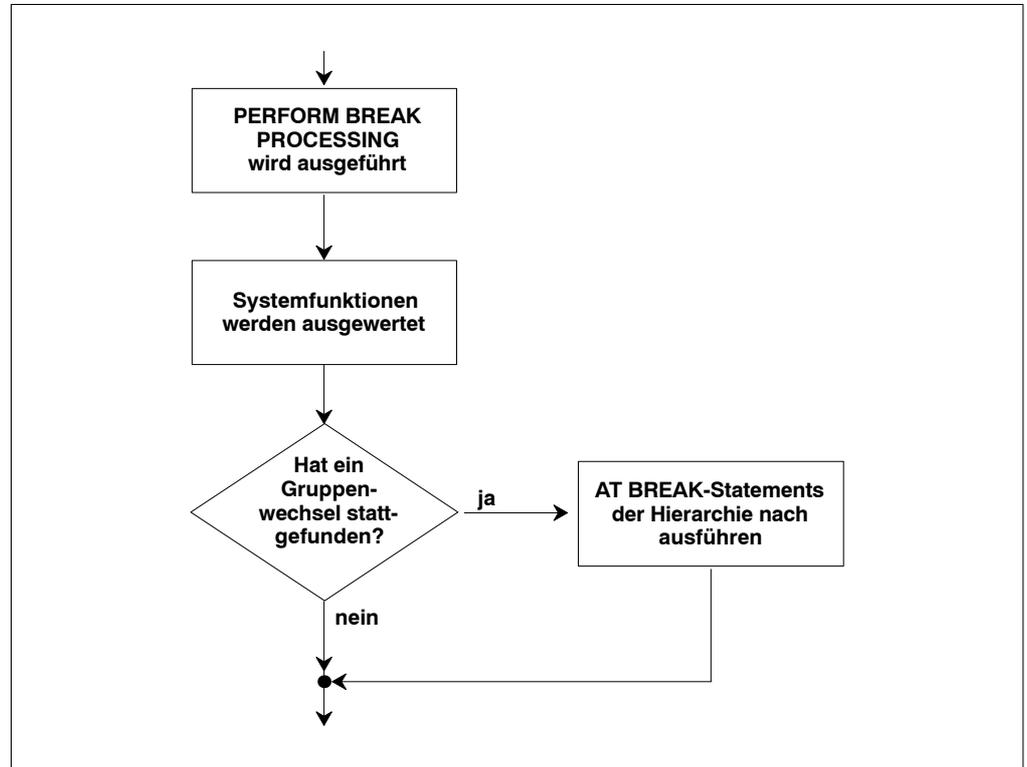
Unmittelbar nach dem **PERFORM BREAK PROCESSING**-Statement geben Sie einen oder mehrere **AT BREAK**-Statement-Blöcke an:

```
...  
PERFORM BREAK PROCESSING  
  AT BREAK OF field1  
    statements  
  END-BREAK  
  AT BREAK OF field2  
    statements  
  END-BREAK  
...
```

Wenn ein **PERFORM BREAK PROCESSING**-Statement ausgeführt wird, prüft Natural, ob ein Gruppenwechsel stattgefunden hat, d.h. ob der Wert des angegebenen Kontrollfeldes sich geändert hat; ist dies der Fall, dann werden die angegebenen Statements ausgeführt.

Bei **PERFORM BREAK PROCESSING** werden Systemfunktionen ausgewertet, *bevor* Natural prüft, ob ein Gruppenwechsel stattgefunden hat.

Folgende Abbildung zeigt den logischen Ablauf einer programmabhängigen Gruppenwechsel-Verarbeitung:



Beispiel für PERFORM BREAK PROCESSING-Statement:

```

** Example Program 'PERFBX01'
DEFINE DATA LOCAL
1 MYVIEW VIEW OF EMPLOYEES
  2 NAME
  2 DEPT
  2 SALARY (1:1)
1 #CNTL (N2)
END-DEFINE
*
LIMIT 7
READ MYVIEW BY DEPT
  AT BREAK OF DEPT          /* ← automatischer Gruppenwechsel
  SKIP 1
  WRITE 'SUMMARY FOR ALL SALARIES      '
    'SUM:'  SUM(SALARY(1))
    'TOTAL:' TOTAL(SALARY(1))
  ADD 1 TO #CNTL
END-BREAK
IF SALARY (1) GREATER THAN 100000 OR BREAK #CNTL
  PERFORM BREAK PROCESSING /* ← programmabhängiger Gruppenwechsel
  AT BREAK OF #CNTL
    WRITE 'SUMMARY FOR SALARY GREATER 100000'
      'SUM:'  SUM(SALARY(1))
      'TOTAL:' TOTAL(SALARY(1))
  END-BREAK
END-IF
IF SALARY (1) GREATER THAN 150000 OR BREAK #CNTL
  PERFORM BREAK PROCESSING /* ← programmabhängiger Gruppenwechsel
  AT BREAK OF #CNTL
    WRITE 'SUMMARY FOR SALARY GREATER 150000'
      'SUM:'  SUM(SALARY(1))
      'TOTAL:' TOTAL(SALARY(1))
  END-BREAK
END-IF
  DISPLAY NAME DEPT SALARY(1)
END-READ
END

```

Page	1					97-08-18	17:11:11
NAME	DEPARTMENT CODE	ANNUAL SALARY					
JENSEN	ADMA01	180000					
PETERSEN	ADMA01	105000					
MORTENSEN	ADMA01	320000					
MADSEN	ADMA01	149000					
BUHL	ADMA01	642000					
SUMMARY FOR ALL SALARIES			SUM:	1396000	TOTAL:	1396000	
SUMMARY FOR SALARY GREATER 100000			SUM:	1396000	TOTAL:	1396000	
SUMMARY FOR SALARY GREATER 150000			SUM:	1142000	TOTAL:	1142000	
HERMANSEN	ADMA02	391500					
PLOUG	ADMA02	162900					
SUMMARY FOR ALL SALARIES			SUM:	554400	TOTAL:	1950400	
SUMMARY FOR SALARY GREATER 100000			SUM:	554400	TOTAL:	1950400	
SUMMARY FOR SALARY GREATER 150000			SUM:	554400	TOTAL:	1696400	

Weiteres Beispiel für AT BREAK-Statement:

Siehe Programm ATBREX06 in Library SYSEXP.

Datenberechnungen

Dieser Abschnitt beschreibt die arithmetischen Statements COMPUTE, ADD, SUBTRACT, MULTIPLY und DIVIDE, sowie die Statements MOVE und COMPRESS, mit denen Werte von einem Feld in ein anderes übertragen werden.

- Formate der Felder
- COMPUTE-Statement
- Statements MOVE und COMPUTE
- Statements ADD, SUBTRACT, MULTIPLY und DIVIDE
- COMPRESS-Statement
- Mathematische Funktionen.

Formate der Felder

Um die Verarbeitung zu optimieren, sollten Benutzervariablen, die in arithmetischen Statements verwendet werden, mit Format P (gepackt numerisch) definiert werden.

COMPUTE-Statement

Mit dem COMPUTE-Statement können Sie Rechenoperationen ausführen. Die folgenden Operatoren stehen Ihnen hierbei zur Verfügung:

Potenzierung	**
Multiplikation	*
Division	/
Addition	+
Subtraktion	-

Logische Gruppen können mittels Klammerung gebildet werden.

Beispiel 1:

```
COMPUTE LEAVE-DUE = LEAVE-DUE * 1.1
```

Der Wert des Feldes LEAVE-DUE wird mit 1,1 multipliziert und das Ergebnis in LEAVE-DUE gestellt.

Beispiel 2:

```
COMPUTE #A = SQRT (#B)
```

Die Quadratwurzel des Feldwertes von #B wird errechnet und dem Feld #A zugewiesen. SQRT (für square root = Quadratwurzel) ist eine von mehreren mathematischen Funktionen, die mit den Statements COMPUTE, ADD, SUBTRACT, MULTIPLY und DIVIDE verwendet werden können. Einen Überblick über mathematische Funktionen finden Sie weiter unten in diesem Kapitel.

Beispiel 3:

```
COMPUTE #INCOME = BONUS (1,1) + SALARY (1)
```

Der erste Bonus des laufenden Jahres und das derzeitige Gehalt werden addiert, und das Ergebnis wird in das Feld #INCOME gestellt.

Statements MOVE und COMPUTE

Mit den Statements MOVE und COMPUTE stellen Sie den Wert eines Operanden in ein oder mehrere Felder. Der Operand kann eine Text- oder Zahlenkonstante, ein Datenbankfeld, eine Benutzervariable, eine Systemvariable und in bestimmten Fällen auch eine Systemfunktion sein.

Die Statements MOVE und COMPUTE unterscheiden sich in ihrer Syntax dahingehend voneinander, daß beim MOVE-Statement der zu verschiebende Wert links angegeben wird, und beim COMPUTE-Statement der zuzuweisende Wert rechts angegeben wird, wie folgende Beispiele zeigen:

Beispiele:

```
MOVE NAME TO #LAST-NAME
```

```
COMPUTE #LAST-NAME = NAME
```

Statements ADD, SUBTRACT, MULTIPLY und DIVIDE

Mit den folgenden Statements ADD, SUBTRACT, MULTIPLY und DIVIDE können Sie Rechenoperationen ausführen.

Beispiele:

```
ADD +5 -2 -1 GIVING #A
```

```
SUBTRACT 6 FROM 11 GIVING #B
```

```
MULTIPLY 3 BY 4 GIVING #C
```

```
DIVIDE 3 INTO #D GIVING #E
```

Alle vier Statements haben eine ROUNDED-Option, mit der Sie gerundete Werte erhalten können.

Weitere Informationen zu diesen Statements finden Sie im *Natural Statements-Handbuch*.

Das folgende Programm veranschaulicht die Verwendung von Benutzervariablen in arithmetischen Statements. Es berechnet Alter und Einkommen von drei Mitarbeitern und gibt die Ergebnisse aus.

Beispiel für MOVE-, SUBTRACT- und COMPUTE-Statements:

```

** Example Program 'COMPUX01'
DEFINE DATA LOCAL
  1 MYVIEW VIEW OF EMPLOYEES
    2 NAME
    2 BIRTH
    2 JOB-TITLE
    2 SALARY          (1:1)
    2 BONUS           (1:1,1:1)
  1 #DATE             (N8)
  1 REDEFINE #DATE
    2 #YEAR           (N4)
    2 #MONTH          (N2)
    2 #DAY            (N2)
  1 #BIRTH-YEAR      (A4)
  1 REDEFINE #BIRTH-YEAR
    2 #BIRTH-YEAR-N  (N4)
  1 #AGE             (N3)
  1 #INCOME          (P9)
END-DEFINE
*
MOVE *DATN TO #DATE
*
READ (3) MYVIEW BY NAME STARTING FROM 'JONES'
  MOVE EDITED BIRTH (EM=YYYY) TO #BIRTH-YEAR
  SUBTRACT #BIRTH-YEAR-N FROM #YEAR GIVING #AGE
  COMPUTE #INCOME = BONUS (1:1,1:1) + SALARY (1:1)
  DISPLAY NAME 'POSITION' JOB-TITLE #AGE #INCOME
END-READ
END

```

NAME	POSITION	#AGE	#INCOME
JONES	MANAGER	58	55000
JONES	DIRECTOR	53	50000
JONES	PROGRAMMER	43	31000

COMPRESS-Statement

Mit dem COMPRESS-Statement fassen Sie den Inhalt zweier oder mehrerer Operanden in einem einzigen alphanumerischen Feld zusammen.

Führende Nullen in einem numerischen Feld bzw. nachfolgende Leerstellen in einem alphanumerischen Feld werden unterdrückt, bevor der Feldwert in das Zielfeld übertragen wird.

Standardmäßig werden die übertragenen Werte im Zielfeld jeweils durch ein Leerzeichen voneinander getrennt. Andere Trennmöglichkeiten sind im *Natural Statements-Handbuch* beschrieben.

Beispiel:

```
COMPRESS 'NAME:' FIRST-NAME #LAST-NAME INTO #FULLNAME
```

In diesem Beispiel werden eine Textkonstante ('NAME:'), ein Datenbankfeld (FIRST-NAME) und eine Benutzervariable (#LAST-NAME) mittels eines COMPRESS-Statements in einer Benutzervariablen (#FULLNAME) zusammengefaßt.

Weitere Informationen zum COMPRESS-Statement siehe *Natural Statements-Handbuch*.

Beispiel für die Statements COMPRESS und MOVE:

```

** Example Program 'COMPRX01'
DEFINE DATA LOCAL
1 MYVIEW OF EMPLOYEES
  2 NAME
  2 FIRST-NAME
  2 MIDDLE-I
1 #LAST-NAME (A15)
1 #FULL-NAME (A30)
END-DEFINE
*
READ (3) MYVIEW BY NAME STARTING FROM 'JONES'
  MOVE NAME TO #LAST-NAME
  COMPRESS 'NAME:' FIRST-NAME MIDDLE-I #LAST-NAME INTO #FULL-NAME
  DISPLAY #FULL-NAME (UC==) FIRST-NAME 'I' MIDDLE-I (AL=1) NAME
END-READ
END

```

Obiges Programm veranschaulicht die Verwendung der Statements MOVE und COMPRESS. Beachten Sie vor allem die Ausgabe der mittels COMPRESS zusammengefaßten Felder:

#FULL-NAME	FIRST-NAME	I	NAME
NAME: VIRGINIA J JONES	VIRGINIA	J	JONES
NAME: MARSHA JONES	MARSHA		JONES
NAME: ROBERT B JONES	ROBERT	B	JONES

Bei mehrzeiligen Ausgaben kann es sinnvoll sein, mit einem COMPRESS-Statement mehrere Felder/Text in einer Benutzervariablen zusammenzufassen.

Im folgenden Beispiel werden drei Benutzervariablen benutzt: #FULLSAL, #FULLNAME und #FULLCITY. In #FULLSAL beispielsweise sind der Text 'SALARY:' sowie die Datenbankfelder SALARY und CURR-CODE zusammengefaßt. Das WRITE-Statement referenziert lediglich die komprimierten Variablen.

Beispiel für COMPRESS-Statement:

```

** Example Program 'COMPRX02'
DEFINE DATA LOCAL
1 VIEWEMP VIEW OF EMPLOYEES
  2 NAME
  2 FIRST-NAME
  2 SALARY (1:1)
  2 CURR-CODE (1:1)
  2 CITY
  2 ADDRESS-LINE (1:1)
  2 ZIP
1 #FULLSAL (A25)
1 #FULLNAME (A25)
1 #FULLCITY (A25)
END-DEFINE
READ (3) VIEWEMP BY CITY STARTING FROM 'NEW YORK'
  COMPRESS 'SALARY:' CURR-CODE(1) SALARY(1) INTO #FULLSAL
  COMPRESS FIRST-NAME NAME INTO #FULLNAME
  COMPRESS ZIP CITY INTO #FULLCITY
  DISPLAY 'NAME AND ADDRESS' NAME (EM=X^X^X^X^X^X^X^X^X^X^X)
  WRITE 1/5 #FULLNAME          1/37 #FULLSAL
        2/5 ADDRESS-LINE (1)
        3/5 #FULLCITY

  SKIP 1
END-READ
END

```

Page	1	97-08-19 18:01:17
NAME AND ADDRESS		
<hr/>		
R U B I N		
SYLVIA RUBIN		SALARY: USD 17000
2003 SARAZEN PLACE		
10036 NEW YORK		
W A L L A C E		
MARY WALLACE		SALARY: USD 38000
12248 LAUREL GLADE C		
10036 NEW YORK		
K E L L O G G		
HENRIETTA KELLOGG		SALARY: USD 52000
1001 JEFF RYAN DR.		
19711 NEWARK		

Mathematische Funktionen

Bei der Verarbeitung arithmetischer Statements (ADD, COMPUTE, DIVIDE, MULTIPLY, SUBTRACT) unterstützt Natural die folgenden mathematischen Funktionen:

Funktion	Bedeutung
ABS (Feld)	Absoluter Wert eines <i>Feldes</i> .
ATN (Feld)	Arcustangens eines <i>Feldes</i> .
COS (Feld)	Kosinus eines <i>Feldes</i> .
EXP (Feld)	Potenz eines <i>Feldes</i> (Exponential).
FRAC (Feld)	Bruchteil (hinter dem Komma) eines <i>Feldes</i> .
INT (Feld)	Ganzzahliger Teil eines <i>Feldes</i> .
LOG (Feld)	Natürlicher Logarithmus eines <i>Feldes</i> .
SGN (Feld)	Vorzeichen eines <i>Feldes</i> .
SIN (Feld)	Sinus eines <i>Feldes</i> .
SQRT (Feld)	Quadratwurzel eines <i>Feldes</i> (Square Root).
TAN (Feld)	Tangens eines <i>Feldes</i> .
VAL (Feld)	Numerischer Wert eines alphanumerischen <i>Feldes</i> .

Weitere Einzelheiten zu mathematischen Funktionen finden Sie im *Natural-Referenzhandbuch*.

Weitere Beispiele für COMPUTE-, MOVE- und COMPRESS-Statements:

Siehe Programme WRITEX11, IFX03 und COMPRX03 in Library SYSEXP.

Systemvariablen und Systemfunktionen

Die folgenden Themen werden nachfolgend behandelt:

- Systemvariablen
- Systemfunktionen.

Systemvariablen

Natural-Systemvariablen enthalten Informationen über die laufende Natural-Session, wie z.B.: Namen der gegenwärtig benutzten Library, Benutzerkennung (User ID) und Terminalkennung; gegenwärtiger Status einer Schleifenverarbeitung; gegenwärtiger Status einer Reportverarbeitung; aktuelles Datum und aktuelle Uhrzeit.

Diese Informationen können Sie in Natural-Programmen benutzen, indem Sie die betreffenden Systemvariablen angeben. Zum Beispiel:

Systemvariable	Inhalt
*INIT-USER	Die User-ID des Terminalbenutzers.
*LANGUAGE	Die aktuelle Sprache.
*LIBRARY-ID	Die aktuelle Library-ID.
*INIT-ID	Die Terminal-ID.
*ERROR-NR	Die Natural-Fehlernummer.
*PAGE-NUMBER	Die aktuelle Seitenzahl.
*COUNTER	Die Anzahl der Durchläufe einer Verarbeitungsschleife.
*NUMBER	Die Anzahl der ausgewählten Datensätze.

Datums- und Uhrzeit-Systemvariablen sind unter anderem:

Systemvariable	Inhalt
*DATU	Aktuelles Datum im Format MM/TT/JJ
*DAT4U	Aktuelles Datum im Format MM/TT/JJJJ
*DATE	Aktuelles Datum im Format TT/MM/JJ
*DAT4E	Aktuelles Datum im Format TT/MM/JJJJ
*DATI	Aktuelles Datum im Format JJ-MM-TT
*DAT4I	Aktuelles Datum im Format JJJJ-MM-TT
*DATD	Aktuelles Datum im Format TT.MM.JJ
*DAT4D	Aktuelles Datum im Format TT.MM.JJJJ
*TIME	Aktuelle Uhrzeit im Format HH:MM:SS.Z
*TIMN	Aktuelle Uhrzeit im Format HHMMSSZ

Die Namen von Systemvariablen beginnen alle mit einem Stern (*).

Datums- und Uhrzeit-Systemvariablen können in einem DISPLAY-, WRITE-, PRINT-, MOVE- oder COMPUTE-Statement verwendet werden.

Weitere Informationen zu Systemvariablen finden Sie im *Natural-Referenzhandbuch*.

Systemfunktionen

Natural-Systemfunktionen sind Funktionen, mit denen Sie statistische und mathematische Informationen über die gelesenen Daten erhalten können. Sie können eingesetzt werden, nachdem ein Datensatz gelesen worden ist, aber vor einem Gruppenwechsel.

Systemfunktionen können in WRITE-, DISPLAY-, PRINT-, COMPUTE- oder MOVE-Statements in Verbindung mit AT END OF PAGE-, AT END OF DATA- und AT BREAK-Statements benutzt werden.

Im Falle eines AT END OF PAGE-Statements muß das jeweilige DISPLAY-Statement eine GIVE SYSTEM FUNCTIONS-Klausel enthalten (wie im Beispiel auf der folgenden Seite gezeigt).

Es gibt folgende Systemfunktionen:

Systemfunktion	Ausgegebener Wert
AVER (<i>Feld</i>)	Durchschnittswert (average) aller Werte von <i>Feld</i> .
NAVER (<i>Feld</i>)	Durchschnittswert aller Werte von <i>Feld</i> ohne Berücksichtigung von Nullwerten.
MAX (<i>Feld</i>)	Größter Wert von <i>Feld</i> .
MIN (<i>Feld</i>)	Kleinster Wert von <i>Feld</i> .
NMIN (<i>Feld</i>)	Kleinster Wert von <i>Feld</i> ohne Berücksichtigung von Nullwerten.
OLD (<i>Feld</i>)	Alter Wert von <i>Feld</i> , d.h. Wert des Feldes vor einem Wechsel des Feldwertes (AT BREAK).
SUM (<i>Feld</i>)	Summe aller Werte von <i>Feld</i> (wird nach AT BREAK-Gruppenwechseln wieder auf Null gesetzt).
TOTAL (<i>Feld</i>)	Gesamtsumme aller Werte von <i>Feld</i> (wird nach AT BREAK-Gruppenwechseln nicht wieder auf Null gesetzt).
COUNT (<i>Feld</i>)	Anzahl der Durchläufe einer Verarbeitungsschleife.
NCOUNT (<i>Feld</i>)	Anzahl der Durchläufe einer Verarbeitungsschleife, wobei Durchläufe, bei denen der Wert des Kontrollfeldes Null ist, nicht mitgezählt werden.

Weitere Informationen zu Systemfunktionen finden Sie im *Natural-Referenzhandbuch*.

Beispiel für Systemvariablen und Systemfunktionen:

```
** Example Program 'SYSVAX01'
DEFINE DATA LOCAL
1 MYVIEW VIEW OF EMPLOYEES
  2 CITY
  2 NAME
  2 JOB-TITLE
  2 INCOME (1:1)
    3 CURR-CODE
    3 SALARY
    3 BONUS (1:1)
END-DEFINE
*
WRITE TITLE 'EMPLOYEE SALARY REPORT AS OF' *DAT4E /
READ (3) MYVIEW BY CITY STARTING FROM 'E'
  DISPLAY GIVE SYSTEM FUNCTIONS
    NAME (AL=15) JOB-TITLE (AL=15) INCOME (1:1)
  AT START OF DATA
    WRITE 'REPORT CREATED AT:' *TIME 'HOURS' /
  END-START
  AT END OF DATA
    WRITE / 'LAST PERSON SELECTED:' OLD (NAME) /
  END-ENDDATA
END-READ
AT END OF PAGE
  WRITE 'AVERAGE SALARY:' AVER(SALARY (1))
END-ENDPAGE
END
```

Das obige Programm veranschaulicht die Verwendung von Systemvariablen und Systemfunktionen:

Die Systemvariable *DATU wird mit dem WRITE TITLE-Statement ausgegeben, und die Systemvariable *TIME mit dem AT START OF DATA-Statement.

Die Systemfunktion OLD wird im AT END OF DATA-Statement verwendet, und die Systemfunktion AVER im AT END OF PAGE-Statement.

Beachten Sie, wie die Systemvariablen und Systemfunktionen angezeigt werden:

EMPLOYEE SALARY REPORT AS OF 18/01/1999				
NAME	CURRENT POSITION	CURRENCY CODE	INCOME	
			ANNUAL SALARY	BONUS
REPORT CREATED AT: 11:51:29.3 HOURS				
DUYVERMAN	PROGRAMMER	USD	34000	0
PRATT	SALES PERSON	USD	38000	9000
MARKUSH	TRAINEE	USD	22000	0
LAST PERSON SELECTED: MARKUSH				
AVERAGE SALARY:			31333	

Weitere Beispiele für Systemvariablen:

Siehe Programme EDITMX05, READX04 und WTITLX01 in Library SYSEXPG.

Weitere Beispiele für Systemfunktionen:

Siehe Programme ATBREX06 und ATENPX01 in Library SYSEXPG.

Stack

Der Natural-Stack ist eine Art “Zwischenlager”, in dem Sie Natural-Kommandos, Benutzerkommandos und Daten für ein INPUT-Statement speichern können. So können Sie häufig nacheinander ausgeführte Funktionen, wie beispielsweise eine Abfolge von Logon-Kommandos, die häufig in der gleichen Reihenfolge ausgeführt werden, speichern.

Der Stack ist mit einem Stapel vergleichbar: die Daten/Kommandos werden aufeinander gestapelt und können sowohl oben auf dem Stack wie auch unter dem Stack abgelegt werden. Die gespeicherten Daten/Kommandos können nur in der gestapelten Reihenfolge verarbeitet werden, und zwar von oben nach unten.

Mit der Systemvariable *DATA können Sie sich in einem Programm den Inhalt des Stack anzeigen lassen (weitere Informationen siehe *Natural-Referenzhandbuch*).

Die Größe des Stack wird durch die verbleibende Größe des ESIZE-Puffers bestimmt, nachdem Platz für die Global Data Area und die Source Area zugeteilt wurde.

Die folgenden Themen werden nachfolgend behandelt:

- Verarbeitung des Stack
- Daten im Stack ablegen
- Stack-Inhalt löschen.

Verarbeitung des Stack

Die Verarbeitung der im Stack gespeicherten Kommandos und Daten ist abhängig von der jeweils ausgeführten Funktion.

Falls ein Kommando einzugeben ist, d.h. falls als nächstes die NEXT-Zeile erscheinen müßte, sucht Natural den Stack von oben nach unten nach einem Kommando ab; wird ein Kommando gefunden, so wird die NEXT-Zeile unterdrückt, das Kommando gelesen und aus dem Stack gelöscht. Das Kommando wird ausgeführt, als wäre es von Hand in der NEXT-Zeile eingegeben worden.

Falls ein INPUT-Statement ausgeführt wird, das Eingabefelder enthält, sucht Natural, bevor der INPUT-Schirm angezeigt wird, den Stack nach Eingabedaten ab und übergibt diese automatisch an das INPUT-Statement (und zwar unter Delimiter-Mode-Logik). Natural überprüft, ob es sich um für das betreffende INPUT-Statement gültige Eingabedaten handelt; anschließend löscht es die Daten aus dem Stack.

Falls ein INPUT-Statement mit Stack-Daten ausgeführt wird, und dieses INPUT-Statement durch ein REINPUT-Statement nochmal ausgeführt wird, wird der INPUT-Schirm angezeigt, und zwar mit den gleichen Stack-Daten wie beim erstmal. Bei einem REINPUT-Statement werden keine weiteren Daten vom Stack gelesen.

Wird ein Natural-Programm normal beendet, werden die zuoberst gelagerten Daten im Stack soweit gelöscht, bis sich entweder oben im Stack wieder ein Kommando befindet oder der Stack ganz geleert ist. Wird ein Natural-Programm aufgrund eines Fehlers oder mit dem Terminalkommando “%%” abgebrochen, wird der gesamte Inhalt des Stacks gelöscht.

Daten im Stack ablegen

Es gibt folgende Möglichkeiten, Daten bzw. Kommandos im Stack abzulegen:

STACK-Parameter

Sie können den Natural-Profilparameter STACK benutzen, um Daten oder Kommandos im Stack abzulegen. Der STACK-Parameter, der in Ihrer *Natural Operations Documentation* beschrieben ist, kann vom Natural-Administrator im Natural-Parametermodul bei der Installation von Natural gesetzt werden. Sie können den STACK-Parameter auch als dynamischen Parameter beim Aufruf von Natural angeben.

Werden Daten/Kommandos mit dem STACK-Parameter im Stack abgelegt, so müssen mehrere Kommandos mit einem Semikolon (;) voneinander getrennt werden. Einem Kommando, das innerhalb einer Reihe von Daten- bzw. Kommandoelementen übergeben wird, muß ein Semikolon vorangestellt werden.

Daten für mehrere INPUT-Statements müssen mit einem Doppelpunkt (:) voneinander getrennt werden. Einer Datenkette, die von einem weiteren INPUT-Statement gelesen werden soll, muß jeweils ein Doppelpunkt vorangestellt werden. Soll ein Kommando, das Parameter erfordert, im Stack abgelegt werden, werden das Kommando und die dazugehörigen Parameter nicht durch einen Doppelpunkt voneinander getrennt.

Doppelpunkt und Semikolon dürfen nicht in den für das INPUT-Statement bestimmten Daten selbst auftauchen, da sie als Trennzeichen interpretiert werden.

STACK-Statement

Innerhalb eines Natural-Programms können Sie das STACK-Statement verwenden, um Daten oder Kommandos im Stack abzulegen. Die in einem STACK-Statement angegebenen Datenelemente können nur für ein einziges INPUT-Statement verwendet werden; d.h. Sie müssen mehrere STACK-Statements verwenden, wenn Sie Daten für mehrere INPUT-Statements im Stack ablegen wollen.

Daten können entweder formatiert oder unformatiert im Stack abgelegt werden:

- Werden unformatierte Daten aus dem Stack gelesen, werden sie im Delimiter-Modus interpretiert, wobei die mit den Session-Parametern IA (Input Assign) und ID (Input Delimiter) festgelegten Zeichen als Input-Zuweisungszeichen bzw. -Trennzeichen verarbeitet werden.
- Formatiert im Stack gelagerte Daten werden nach Feldinhalten getrennt und Feld für Feld an die Eingabefelder des betreffenden INPUT-Statements übergeben.

Eine ausführliche Beschreibung des Statements STACK finden Sie im *Natural Statements-Handbuch*.

Statements FETCH und RUN

Werden bei der Ausführung eines FETCH- oder RUN-Statements Parameter an das aufgerufene Programm übergeben, so werden diese Parameter oben auf dem Stack abgelegt.

Stack-Inhalt löschen

Der Inhalt des Stacks kann mit dem Statement RELEASE gelöscht werden. Eine ausführliche Beschreibung des Statements RELEASE finden Sie im *Natural Statements-Handbuch*.

Datumsinformationen verarbeiten

Dieser Abschnitt behandelt verschiedene Aspekte der Behandlung von Datumsinformationen in Ihren Natural-Anwendungen:

- Editiermasken für Datumsfelder und Datumssystemvariablen
- Standard-Editiermaske für Datum — der DTFORM-Parameter
- Datumsformat für alphanumerische Darstellung — der DF-Parameter
- Datumsformat für Ausgabe — der DFOUT-Parameter
- Datumsformat für Stack — der DFSTACK-Parameter
- “Year Sliding Window” — der YSLW-Parameter
- Kombinationen von DFSTACK und YSLW
- Datumsformat für Standard-Seitenüberschriften — der DFTITLE-Parameter.

Editiermasken für Datumsfelder and Datumssystemvariablen

Wenn Sie den Wert eines Datumsfeldes in einer bestimmten Form ausgeben möchten, geben Sie normalerweise für das Feld eine *Editiermaske* an. Mit der Editiermaske bestimmen Sie Zeichen für Zeichen, wie die Ausgabe aussehen soll. Nähere Informationen zu Editiermasken finden Sie auf Seite 153.

Falls Sie das aktuelle Datum in einer bestimmten Form verwenden möchten, brauchen Sie hierfür kein Datumsfeld mit einer entsprechenden Editiermaske zu definieren; stattdessen können Sie einfach eine *Datumssystemvariable* verwenden. Natural bietet verschiedene Datumssystemvariablen, die alle das aktuelle Datum in unterschiedlichen Darstellungsformen enthalten. Bei manchen dieser Darstellungsformen ist die Jahreszahl-Angabe 2-stellig, bei manchen 4-stellig.

Beispiele für Datumssystemvariablen finden Sie auf Seite 79 im *Natural-Referenzhandbuch*. Weitere Informationen sowie eine Liste aller Datumssystemvariablen finden Sie im *Natural-Referenzhandbuch*.

Standard-Editiermaske für Datum — der DTFORM-Parameter

Der Profilparameter DTFORM bestimmt das Standardformat, das für Datumsangaben als Teil von Natural-Standard-Reporttiteln, für Datumskonstanten und für Datumseingaben gilt.

Dieses Datumsformat bestimmt die Reihenfolge der Angaben für Tag, Monat und Jahr sowie die Trennzeichen, die zwischen diesen Angaben stehen müssen.

Mögliche DTFORM-Einstellungen sind:

Einstellung	Datumsformat*	Beispiel
DTFORM=I	<i>yyyy-mm-dd</i>	1997-12-31
DTFORM=G	<i>dd.mm.yyyy</i>	31.12.1997
DTFORM=E	<i>dd/mm/yyyy</i>	31/12/1997
DTFORM=U	<i>mm/dd/yyyy</i>	12/31/1997

* *dd* = Tag (day), *mm* = Monat, *yyyy* = Jahr (year).

Der DTFORM-Parameter kann im Natural-Parametermodul (bzw. der Natural-Parameterdatei) oder dynamisch beim Aufruf von Natural gesetzt werden. Standardmäßig gilt DTFORM=I.

Datumsformat für alphanumerische Darstellung — der DF-Parameter

Der Session-Parameter DF gilt nur für Datumfelder, für die keine Editiermaske angegeben ist.

Wenn eine Editiermaske angegeben ist, wird die Darstellung des Feldwertes durch die Editiermaske bestimmt. Wenn keine Editiermaske angegeben ist, wird die Darstellung des Feldwertes durch den Session-Parameter DF in Kombination mit dem DTFORM-Profilparameter bestimmt.

Mit dem DF-Parameter können Sie eine der folgenden Datumsdarstellungen wählen:

DF=S	8-Byte-Darstellung mit 2-stelliger Jahreskomponente und Delimitern (<i>yy-mm-dd</i>).
DF=I	8-Byte-Darstellung mit 4-stelliger Jahreskomponente ohne Delimiter (<i>yyyymmdd</i>).
DF=L	10-Byte-Darstellung mit 4-stelliger Jahreskomponente und Delimitern (<i>yyyy-mm-dd</i>).

Bei jeder Darstellung wird die Reihenfolge der Tages-, Monats- und Jahreskomponenten sowie die zu verwendenden Delimiterzeichen durch den DTFORM-Parameter bestimmt.

Standardmäßig gilt DF=S (außer bei INPUT-Statements; siehe nächste Seite).

Der DF-Parameter wird bei der Kompilierung ausgewertet. Er kann angegeben werden in einem FORMAT-Statement, den Statements INPUT, DISPLAY, WRITE und PRINT (auf Statement- und Feldebene) sowie den Statements MOVE, COMPRESS, STACK, RUN und FETCH (auf Feldebene).

Der DF-Parameter gilt in folgenden Situationen:

- **DISPLAY, WRITE und PRINT:** wenn der Wert einer Datumsvariablen mit einem dieser Statements ausgegeben wird, wird der Wert in alphanumerische Darstellung umgesetzt, bevor er ausgegeben wird. Der DF-Parameter bestimmt, welche Darstellung dafür verwendet wird.
- **MOVE und COMPRESS:** wenn der Wert einer Datumsvariablen mit einem MOVE- oder COMPRESS-Statement in ein alphanumerisches Feld übertragen wird, wird der Wert in alphanumerische Darstellung umgesetzt, bevor er übertragen wird. Der DF-Parameter bestimmt, welche Darstellung dafür verwendet wird.
- **STACK, FETCH und RUN:** wenn der Wert einer Datumsvariablen auf dem Stack abgelegt wird, wird er in alphanumerische Darstellung umgesetzt, bevor er auf dem Stack abgelegt wird. Der DF-Parameter bestimmt, welche Darstellung dafür verwendet wird. Dasselbe gilt, wenn eine Datumsvariable als Parameter in einem FETCH- oder RUN-Statement angegeben ist (da diese Parameter ebenfalls über den Stack übergeben werden).
- **INPUT:** Wenn eine Datumsvariable in einem INPUT-Statement verwendet wird, bestimmt der DF-Parameter, in welcher Form ein Wert in dieses Feld eingegeben werden muß.
Wenn dagegen eine Datumsvariable, für die *kein* DF-Parameter angegeben ist, in einem INPUT-Statement verwendet wird, kann das Datum entweder mit 2-stelliger Jahresangabe und Delimitern oder mit 4-stelliger Jahresangabe ohne Delimiter eingegeben werden. Auch in diesem Fall werden die Reihenfolge der Tages-, Monats- und Jahreskomponenten sowie die zu verwendenden Delimiterzeichen durch den DTFORM-Parameter bestimmt.

Bei DF=S stehen nur 2 Stellen für die Jahresangabe zur Verfügung; d.h. selbst wenn der Datumswert das Jahrhundert enthielte, gingen diese Informationen bei der Umsetzung verloren. Um die Jahrhundert-Angabe zu behalten, setzen Sie DF=I oder DF=L.

Beispiele für DF-Parameter bei WRITE-Statements:

```

/* DF=S (Standardeinstellung)
WRITE *DATX /* Ausgabe hat folgende Form: dd.mm.yy
END

FORMAT DF=I
WRITE *DATX /* Ausgabe hat folgende Form: ddmmyyyy
END

FORMAT DF=L
WRITE *DATX /* Ausgabe hat folgende Form: dd.mm.yyyy
END

```

Diese Beispiele gehen von DTFORM=G aus.

Beispiel für DF-Parameter bei MOVE-Statement:

```

DEFINE DATA LOCAL
  1 #DATE (D) INIT <D'31/12/1997'>
  1 #ALPHA (A10)
END-DEFINE
...
MOVE #DATE TO #ALPHA /* Ergebnis: #ALPHA enthält 31/12/97
MOVE #DATE (DF=I) TO #ALPHA /* Ergebnis: #ALPHA enthält 31121997
MOVE #DATE (DF=L) TO #ALPHA /* Ergebnis: #ALPHA enthält 31/12/1997
...

```

Dieses Beispiel geht von DTFORM=E aus.

Beispiel für DF-Parameter bei STACK-Statement:

```

DEFINE DATA LOCAL
  1 #DATE (D) INIT <D'1997-12-31'>
  1 #ALPHA1(A10)
  1 #ALPHA2(A10)
  1 #ALPHA3(A10)
END-DEFINE
...
STACK TOP DATA #DATE (DF=S) #DATE (DF=I) #DATE (DF=L)
...
INPUT #ALPHA1 #ALPHA2 #ALPHA3
...
/* Ergebnis: #ALPHA1 enthält 97-12-31
/*           #ALPHA2 enthält 19971231
/*           #ALPHA3 enthält 1997-12-31
...

```

Dieses Beispiel geht von DTFORM=I aus.

Beispiel für DF-Parameter bei INPUT-Statement:

```

DEFINE DATA LOCAL
  1 #DATE1 (D)
  1 #DATE2 (D)
  1 #DATE3 (D)
  1 #DATE4 (D)
END-DEFINE
...
INPUT #DATE1 (DF=S) /* Eingabe muß folgende Form haben: yy-mm-dd
      #DATE2 (DF=I) /* Eingabe muß folgende Form haben: yyyymmdd
      #DATE3 (DF=L) /* Eingabe muß folgende Form haben: yyyy-mm-dd
      #DATE4      /* Eingabe muß folgende Form haben: yy-mm-dd oder yyyymmdd
...

```

Dieses Beispiel geht von DTFORM=I aus.

Datumsformat für Ausgabe — der DFOUT-Parameter

Der Session- bzw. Profilparameter DFOUT gilt nur für Datumsfelder in INPUT-, DISPLAY-, PRINT- und WRITE-Statements, für die keine Editiermaske angegeben ist und für die kein DF-Parameter gilt.

Bei Datumsfeldern, die mit einem INPUT-, DISPLAY-, PRINT- oder WRITE-Statement ausgegeben werden und für die weder eine Editiermaske angegeben noch ein DF-Parameter gesetzt ist, bestimmt der Profil/Session-Parameter DFOUT die Form, in der die Feldwerte angezeigt werden.

Mögliche DFOUT-Einstellungen sind:

DFOUT=S	Datumsvariablen werden mit 2-stelliger Jahreskomponente und mit durch den DTFORM-Parameter bestimmten Delimiterzeichen angezeigt (<i>yy-mm-dd</i>).
DFOUT=I	Datumsvariablen werden mit mit 4-stelliger Jahreskomponente und ohne Delimiterzeichen angezeigt (<i>yyyymmdd</i>).

Standardmäßig gilt DFOUT=S. Bei beiden DFOUT-Einstellungen wird die Reihenfolge der Tages-, Monats- und Jahreskomponenten in den Datumswerten durch den DTFORM-Parameter bestimmt.

Die Länge eines Datumsfeldes wird durch die DFOUT-Einstellung nicht beeinflusst, da jede der beiden Datumswert-Darstellungen in ein 8 Byte langes Feld paßt.

Der DFOUT-Parameter kann im Natural-Parametermodul (bzw. der Natural-Parameterdatei), dynamisch beim Aufrufen von Natural oder mit dem Systemkommando GLOBALS gesetzt werden. Er wird zur Laufzeit ausgewertet.

Beispiel:

```
DEFINE DATA LOCAL
1 #DATE (D) INIT <D'1997-12-31'>
END-DEFINE
...
WRITE #DATE          /* Ausgabe, wenn DFOUT=S gesetzt ist: 97-12-31
                       /* Ausgabe, wenn DFOUT=I gesetzt ist: 19971231
WRITE #DATE (DF=L) /* Ausgabe (unabhängig von DFOUT)... 1997-12-31
...
```

Dieses Beispiel geht von DTFORM=I aus.

Datumsformat für Stack — der DFSTACK-Parameter

Der Session- bzw. Profilparameter DFSTACK gilt nur für Datumsfelder, die in STACK-, FETCH- und RUN-Statements verwendet werden und für die kein DF-Parameter angegeben ist.

Der DFSTACK-Parameter bestimmt die Form, in der die Werte von Datumsvariablen mit einem STACK-, RUN- oder FETCH-Statement auf dem Stack abgelegt werden.

Mögliche DFSTACK-Einstellungen sind:

DFSTACK=S	Datumsvariablen werden mit 2-stelliger Jahreskomponente und mit durch den DTFORM-Parameter bestimmten Delimiterzeichen auf dem Stack abgelegt (<i>yy-mm-dd</i>).
DFSTACK=C	Wie DFSTACK=S. Allerdings wird eine Änderung des Jahrhunderts zur Laufzeit abgefangen.
DFSTACK=I	Datumsvariablen werden mit 4-stelliger Jahreskomponente und ohne Delimiterzeichen auf dem Stack abgelegt (<i>yyyymmdd</i>).

Standardmäßig gilt DFSTACK=S. DFSTACK=S bedeutet, daß die Jahrhundert-Informationen nicht mit abgelegt werden (und verlorengehen), wenn ein Datumswert auf dem Stack abgelegt wird.

Wenn dann der Wert vom Stack in eine andere Datumsvariable eingelesen wird, wird entweder als Jahrhundert das aktuelle Jahrhundert genommen oder das Jahrhundert durch die Einstellung des YSLW-Parameters (siehe unten) bestimmt. Das kann dazu führen, daß das Jahrhundert ein anderes ist als im ursprünglichen Datumswert, ohne daß Natural in diesem Fall einen Fehler ausgibt.

DFSTACK=C funktioniert insofern wie DFSTACK=S, daß ein Datumswert ohne Jahrhundert-Informationen auf dem Stack abgelegt wird. Wenn allerdings der Wert vom Stack gelesen wird und das ermittelte Jahrhundert nicht mit dem des ursprünglichen Datumswerts identisch ist (entweder aufgrund des YSLW-Parameters oder weil das ursprüngliche Jahrhundert nicht das aktuelle ist), gibt Natural einen Laufzeitfehler aus.

Anmerkung:

Dieser Laufzeitfehler wird bereits ausgegeben, sobald der Wert auf dem Stack abgelegt wird.

Mit DFSTACK=I können Sie einen Datumswert in einer Länge von 8 Bytes auf dem Stack ablegen, ohne daß die Jahrhundert-Informationen verlorengehen.

Der DFSTACK-Parameter kann im Natural-Parametermodul (bzw. der -Parameterdatei) dynamisch beim Aufrufen von Natural oder mit dem Systemkommando GLOBALS gesetzt werden. Er wird zur Laufzeit ausgewertet.

Beispiel:

```
DEFINE DATA LOCAL
  1 #DATE (D) INIT <D'1997-12-31'>
  1 #ALPHA1(A8)
  1 #ALPHA2(A10)
END-DEFINE
...
STACK TOP DATA #DATE #DATE (DF=L)
...
INPUT #ALPHA1 #ALPHA2
...
/* Ergebnis, wenn DFSTACK=S oder =C gesetzt ist: #ALPHA1 enthält 97-12-31
/* Ergebnis, wenn DFSTACK=I gesetzt ist .....: #ALPHA1 enthält 19971231
/* Ergebnis (unabhängig von DFSTACK) .....: #ALPHA2 enthält 1997-12-31
...
```

Dieses Beispiel geht von DTFORM=I und YSLW=0 aus.

“Year Sliding Window” — der YSLW-Parameter

Mit dem Profilparameter YSLW können Sie das Jahrhundert eines 2-stelligen Jahr-Wertes bestimmen.

Der YSLW-Parameter kann im Natural-Parametermodul (bzw. der -Parameterdatei) oder dynamisch beim Aufrufen von Natural gesetzt werden. Er wird zur Laufzeit ausgewertet, wenn ein alphanumerischer Datumswert mit einer 2-stelligen Jahreskomponente in eine Datumsvariable übertragen wird. Dies betrifft Datumswerte, die:

- mit der mathematischen Funktion VAL verwendet werden
- mit der Option IS(D) in einer logischen Bedingung verwendet werden
- vom Stack als Eingabedaten gelesen werden
- in ein Feld als Eingabedaten eingegeben werden.

Der YSLW-Parameter bestimmt den Bereich von Jahren, der von einem sogenannten “Year Sliding Window” abgedeckt wird. Der “Sliding Window“-Mechanismus geht davon aus, daß ein Datum mit einem 2-stelligen Jahr innerhalb eines “Fensters” von 100 Jahren liegt. Innerhalb dieser 100 Jahre kann jeder 2-stellige Jahr-Wert eindeutig einem bestimmten Jahrhundert zugeordnet werden.

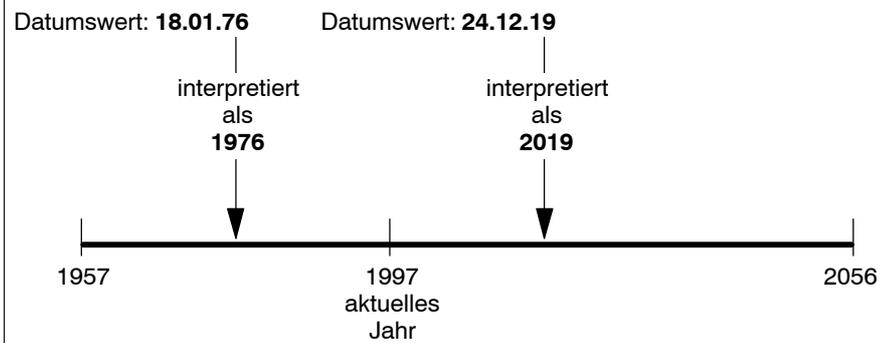
Mit dem YSLW-Parameter legen Sie fest, wieviele Jahre in der Vergangenheit der 100-Jahre-Bereich anfangen soll: das erste Jahr des Fensterbereichs ergibt sich aus dem aktuellen Jahr minus dem YSLW-Wert.

Mögliche Werte des YSLW-Parameters sind 0 bis 99. Der Standardwert ist YSLW=0, d.h. der “Sliding Window“-Mechanismus ist nicht aktiv; bei einem 2-stelligen Jahr wird dann angenommen, daß es im aktuellen Jahrhundert liegt.

Beispiel 1:

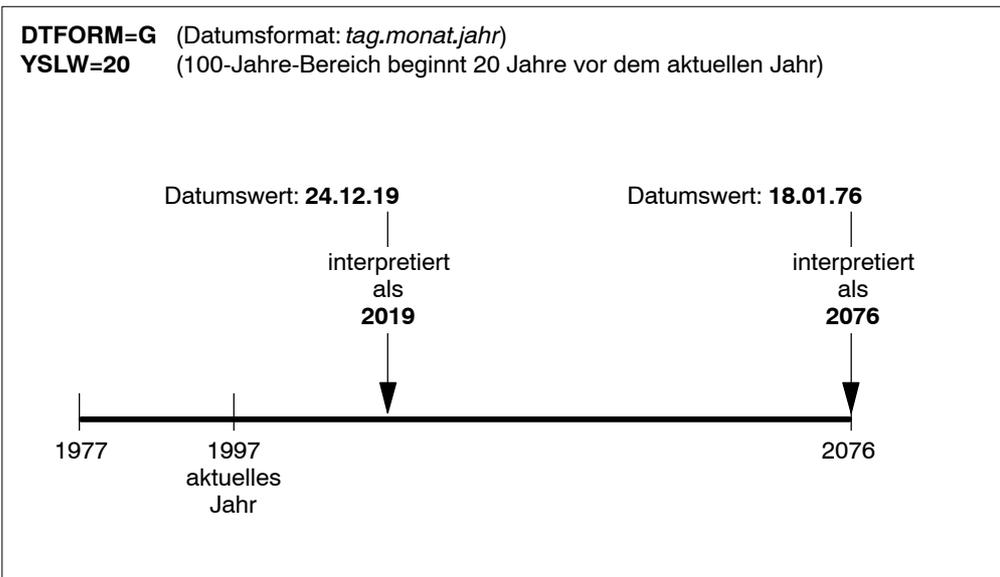
Wenn das aktuelle Jahr 1997 ist und Sie YSLW=40 angeben, deckt das “Sliding Window” die Jahre 1957 bis 2056 ab. Ein 2-stelliger Jahr-Wert nn von 57 bis 99 wird dementsprechend als $19nn$ interpretiert, während ein 2-stelliger Jahr-Wert nn von 00 bis 56 als $20nn$ interpretiert wird.

DTFORM=G (Datumsformat: *Tag.Monat.Jahr*)
YSLW=40 (100-Jahre-Bereich beginnt 40 Jahre vor dem aktuellen Jahr)



Beispiel 2:

Wenn das aktuelle Jahr 1997 ist und Sie YSLW=20 angeben, deckt das “Sliding Window” die Jahre 1977 bis 2076 ab. Ein 2-stelliger Jahr-Wert *nm* von 77 bis 99 wird dementsprechend als *19nm* interpretiert, während ein 2-stelliger Jahr-Wert *nm* von 00 bis 76 als *20nm* interpretiert wird.



Kombinationen von DFSTACK und YSLW

Die folgenden Beispiele veranschaulichen die Wirkungen verschiedener Kombinationen der Parameter DFSTACK und YSLW.

Alle Beispiele gehen von DTFORM=I aus.

Beispiel 1:

Dieses Beispiel geht vom aktuellen Jahr 1997 und folgenden Parametereinstellungen aus:

DFSTACK=S (Standard)

YSLW=20

```

DEFINE DATA LOCAL
  1 #DATE1 (D) INIT <D'1956-12-31'>
  1 #DATE2 (D)
END-DEFINE
...
STACK TOP DATA #DATE1 /* Jahrhundert-Informationen gehen verloren
                       /* (Jahr 56 wird auf dem Stack abgelegt)
...
INPUT #DATE2           /* "Year Sliding Window" berechnet 56 als 2056
...
/* Ergebnis: #DATE2 enthält 2056-12-31

```

In diesem Fall ist das "Year Sliding Window" unpassend gesetzt, so daß die Jahrhundert-Informationen sich (unbeabsichtigterweise) ändern.

Beispiel 2:

Dieses Beispiel geht vom aktuellen Jahr 1997 und folgenden Parametereinstellungen aus:

DFSTACK=S (Standard)

YSLW=50

```
DEFINE DATA LOCAL
  1 #DATE1 (D) INIT <D'1956-12-31'>
  1 #DATE2 (D)
END-DEFINE
...
STACK TOP DATA #DATE1 /* Jahrhundert-Informationen gehen verloren
...                      /* (Jahr 56 wird auf dem Stack abgelegt)
...
INPUT #DATE2            /* "Year Sliding Window" berechnet 56 als 1956
...
/* Ergebnis: #DATE2 enthält 1956-12-31
```

In diesem Fall ist das "Year Sliding Window" passend gesetzt, so daß die ursprünglichen Jahrhundert-Informationen wiederhergestellt werden.

Beispiel 3:

Dieses Beispiel geht vom aktuellen Jahr 1997 und folgenden Parametereinstellungen aus:

DFSTACK=C

YSLW=0 (Standard)

```
DEFINE DATA LOCAL
  1 #DATE1 (D) INIT <D'2056-12-31'>
  1 #DATE2 (D)
END-DEFINE
...
STACK TOP DATA #DATE1 /* Jahrhundert-Informationen gehen verloren
...                    /* (Jahr 56 wird auf dem Stack abgelegt)
...
INPUT #DATE2           /* für 56 wird aktuelles Jahrhundert angenommen -> 1956
...
/* Ergebnis: LAUFZEITFEHLER (UNBEABSICHTIGTE ÄNDERUNG DES JAHRHUNDERTS)
```

In diesem Fall ändern sich (unbeabsichtigtweise) die Jahrhundert-Informationen. Allerdings wird diese Änderung durch die Parametereinstellung **DFSTACK=C** abgefangen.

Beispiel 4:

Dieses Beispiel geht vom aktuellen Jahr 1997 und folgenden Parametereinstellungen aus:

DFSTACK=C
YSLW=20

```
DEFINE DATA LOCAL
  1 #DATE1 (D) INIT <D'1956-12-31'>
  1 #DATE2 (D)
END-DEFINE
...
STACK TOP DATA #DATE1 /* Jahrhundert-Informationen gehen verloren
...                      /* (Jahr 56 wird auf dem Stack abgelegt)
...
INPUT #DATE2           /* "Year Sliding Window" berechnet 56 als 2056
...
/* Ergebnis: LAUFZEITFEHLER (UNBEABSICHTIGTE ÄNDERUNG DES JAHRHUNDERTS)
```

In diesem Fall ändern sich die Jahrhundert-Informationen aufgrund des "Year Sliding Window". Allerdings wird diese Änderung durch die Parametereinstellung DFSTACK=C abgefangen.

Datumsformat für Standard-Seitenüberschriften — der DFTITLE-Parameter

Der Session- bzw. Profilparameter DFTITLE bestimmt die Form des Datums in einer Standard-Seitenüberschrift (wie sie mit einem DISPLAY-, WRITE- oder PRINT-Statement ausgegeben wird).

DFTITLE=S	Das Datum wird mit 2-stelliger Jahreskomponente und Delimitern ausgegeben (<i>yy-mm-dd</i>).
DFTITLE=L	Das Datum wird mit 4-stelliger Jahreskomponente und Delimitern ausgegeben (<i>yyyy-mm-dd</i>).
DFTITLE=I	Das Datum wird mit 4-stelliger Jahreskomponente ohne Delimiter ausgegeben (<i>yyyymmdd</i>).

Bei jeder dieser Ausgabeformen werden die Reihenfolge der Tages-, Monats- und Jahreskomponenten sowie die verwendeten Delimiterzeichen durch den DTFORM-Parameter bestimmt.

Der DFTITLE-Parameter kann im Natural-Parametermodul (bzw. der -Parameterdatei) dynamisch beim Aufrufen von Natural oder mit dem Systemkommando GLOBALS gesetzt werden. Er wird zur Laufzeit ausgewertet.

Beispiel:

```
WRITE 'HELLO'
END
/*
/* Datum in Seitenüberschrift, wenn DFTITLE=S gesetzt ist ...: 98-10-31
/* Datum in Seitenüberschrift, wenn DFTITLE=L gesetzt ist ...: 1998-10-31
/* Datum in Seitenüberschrift, wenn DFTITLE=I gesetzt ist ...: 19981031
```

Dieses Beispiel geht von DTFORM=I aus.

Anmerkung:

Der DFTITLE-Parameter hat keine Auswirkungen auf benutzer-definierte Seitenüberschriften, wie sie mit einem WRITE TITLE-Statement angegeben werden.

REPORTING MODE UND STRUCTURED MODE

Die folgenden Themen werden nachfolgend behandelt:

- Allgemeine Informationen
- Programmiermodus festlegen
- Funktionale Unterschiede
- Verarbeitungsschleife im *Reporting Mode* beenden
- Verarbeitungsschleife im *Structured Mode* beenden
- Datenbank-Referenzierung.

Allgemeine Informationen

Natural bietet zwei Formen der Programmierung: *Reporting Mode* und *Structured Mode*.

Grundsätzlich *empfiehlt es sich, ausschließlich im Structured Mode zu arbeiten*, um klar strukturierte Anwendungen zu erhalten.

Der *Reporting Mode* eignet sich nur für die Erstellung einfacher Reports und Programme, die keine komplexe Daten- und Programmstruktur erfordern. (Falls Sie sich entschließen sollten, ein Programm im *Reporting Mode* zu schreiben, sollten Sie bedenken, daß kleine Programme schnell umfangreicher und komplexer werden können.)

Der *Structured Mode* ist für komplexe Anwendungen gedacht, bei denen es auf eine klare und sinnvoll gegliederte Programmstruktur ankommt. Wesentliche Vorteile des *Structured Mode* sind:

- Die Programme müssen strukturierter geschrieben werden und sind daher leichter zu lesen und folglich auch leichter zu pflegen.
- Da alle in einem Programm verwendeten Felder an einer zentralen Stelle definiert werden müssen (und nicht, wie im *Reporting Mode*, über das ganze Programm verstreut sein dürfen), wird der Überblick über die verwendeten Daten erheblich erleichtert.

Darüber hinaus zwingt der *Structured Mode* zu einer genaueren Anwendungsplanung, bevor es ans eigentliche Programmieren geht. Viele Fehler und Unzulänglichkeiten bei der Programmierung werden dadurch von vorneherein vermieden.

Programmiermodus festlegen

Der Standardprogrammiermodus wird vom Natural-Administrator festgelegt. Sie können den vorgegebenen Modus mit dem Systemkommando GLOBALS ändern:

- **GLOBALS SM=ON** — Structured Mode.
- **GLOBALS SM=OFF** — Reporting Mode.

Funktionale Unterschiede

Die wichtigsten funktionalen Unterschiede zwischen *Reporting Mode* und *Structured Mode* lassen sich wie folgt zusammenfassen:

- Die Programmcode-Syntax zum Beenden von Verarbeitungsschleifen und funktionalen Blöcken ist unterschiedlich:
Im *Structured Mode* muß jede Schleife oder logische Verarbeitungsbedingung ausdrücklich mit einem entsprechenden END-...-Statement abgeschlossen werden. Dadurch wird sofort deutlich, wo welche Schleife bzw. Bedingung aufhört.
Im *Reporting Mode* werden hierzu (CLOSE) LOOP- oder DO ... DOEND-Statements verwendet.
Natural läßt im *Reporting Mode* keine END-...-Statements zu (außer END-DECIDE, END-DEFINE und END-SUBROUTINE), und im *Structured Mode* keine LOOP- und DO/DOEND-Statements.
- Im *Reporting Mode* ist es möglich, Datenbankfelder zu benutzen, ohne daß diese vorher in einem DEFINE DATA-Statement definiert werden müssen; außerdem können Benutzervariablen an jeder Stelle eines Programms, d.h. über das ganze Programm verstreut, definiert werden.
Im *Structured Mode* dagegen müssen *alle* verwendeten Datenelemente an einer zentralen Stelle (entweder im DEFINE DATA-Statement am Anfang des Programms oder in einer externen Data Area) definiert werden.

Im *Natural Statements-Handbuch* wird für Statements, die in beiden Modi verwendet werden können, die jeweils unterschiedliche Programmsyntax in den Syntaxdiagrammen gezeigt.

Die beiden folgenden Beispiele veranschaulichen die je nach Programmiermodus unterschiedliche Konstruktion von Verarbeitungsschleifen und logischen Bedingungen.

Im Reporting-Mode-Beispiel werden die Statements DO und DOEND verwendet, um den Statement-Block einzugrenzen, der an die AT END OF DATA-Bedingung geknüpft ist. Das END-Statement beendet sämtliche aktiven Verarbeitungsschleifen.

Beispiel — Reporting Mode:

```

READ EMPLOYEES BY PERSONNEL-ID
DISPLAY NAME BIRTH POSITION
AT END OF DATA
  DO
    SKIP 2
    WRITE / 'LAST SELECTED:' OLD(NAME)
  DOEND
END

```

Im Structured-Mode-Beispiel wird ein END-ENDDATA-Statement verwendet, um die AT END OF DATA-Bedingung zu beenden, sowie ein END-READ-Statement, um die READ-Schleife zu beenden. Das Ergebnis ist ein deutlicher strukturiertes Programm, in dem Sie sofort sehen können, wo welche Konstruktion anfängt und aufhört:

Beispiel — Structured Mode:

```

DEFINE DATA LOCAL
1 MYVIEW VIEW OF EMPLOYEES
  2 PERSONNEL-ID
  2 NAME
  2 BIRTH
  2 POSITION
END-DEFINE
READ MYVIEW BY PERSONNEL-ID
  DISPLAY NAME BIRTH POSITION
  AT END OF DATA
    SKIP 2
    WRITE / 'LAST SELECTED:' OLD(NAME)
  END-ENDDATA
END-READ
END

```

Verarbeitungsschleife im Reporting Mode beenden

Zum Beenden einer Verarbeitungsschleife können Sie im *Reporting Mode* die Statements END, LOOP (bzw. CLOSE LOOP) oder SORT verwenden.

Mit dem LOOP-Statement können Sie mehrere Schleifen gleichzeitig schließen. Mit dem END-Statement können Sie sämtliche noch nicht beendeten Schleifen schließen. Diese Möglichkeit, mehrere Schleifen mit einem einzigen Statement zu beenden, stellt einen grundlegenden Unterschied zum *Structured Mode* dar.

Ein SORT-Statement beendet alle Schleifen und initiiert gleichzeitig eine neue Schleife.

Beispiel 1 — LOOP:

```
FIND ...
  FIND ...
  ...
  ...
  LOOP (schließt innere FIND-Schleife)
LOOP   (schließt äußere FIND-Schleife)
...
...
```

Beispiel 2 — END:

```
FIND ...
  FIND ...
  ...
  ...
END (schließt alle Schleifen und beendet die Verarbeitung)
```

Beispiel 3 — SORT:

```
FIND ...
  FIND ...
  ...
  ...
SORT ... (schließt alle Schleifen, initiiert neue Schleife)
...
END (schließt SORT-Schleife und beendet die Verarbeitung)
```

Verarbeitungsschleife im Structured Mode beenden

Im *Structured Mode* gibt es zum Beenden jeder Verarbeitungsschleife ein bestimmtes Statement. Mit dem END-Statement werden keine Schleifen geschlossen. Bei Verwendung des SORT-Statements müssen Sie vorher ein END-ALL-Statement verwenden, sowie zum Beenden der SORT-Schleife ein END-SORT-Statement.

Beispiel 1 — FIND:

```
FIND ...
  FIND ...
  ...
  ...
  END-FIND (schließt innere FIND-Schleife)
END-FIND  (schließt äußere FIND-Schleife)
...
```

Beispiel 2 — READ:

```
READ ...
  AT END OF DATA
  ...
  END-ENDDATA
  ...
END-READ (schließt READ-Schleife)
...
...
END
```

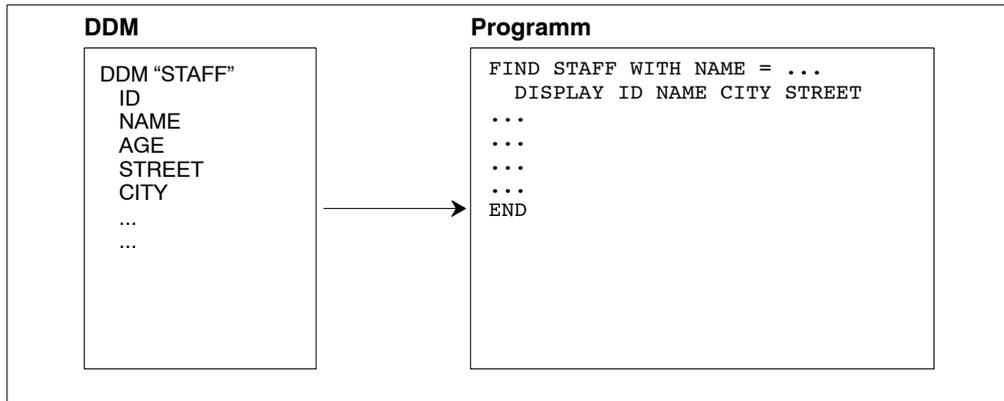
Beispiel 3 — SORT:

```
READ ...
  FIND ...
  ...
  ...
END-ALL (schließt alle Schleifen)
SORT    (erzeugt Schleife)
...
...
END-SORT (schließt SORT-Schleife)
END
```

Datenbank-Referenzierung

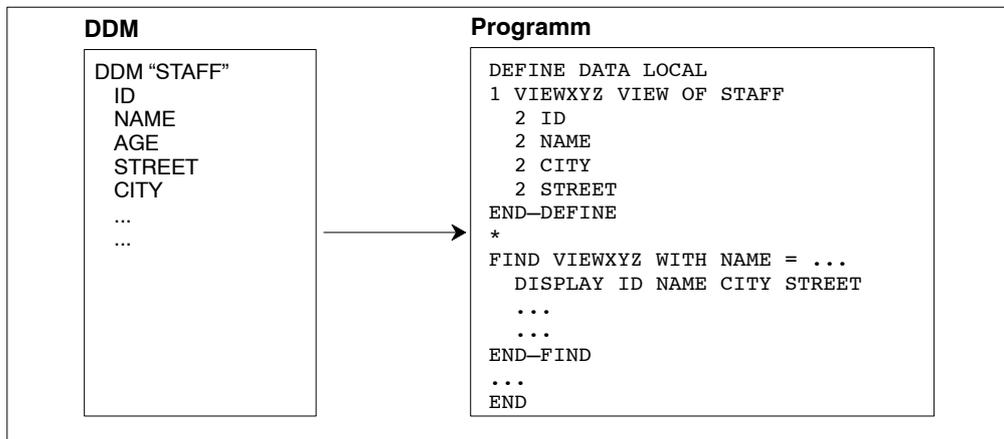
Im *Reporting Mode* ist es möglich, Datenbankfelder oder DDMs zu benutzen, ohne diese in einer Data Area definiert zu haben.

Reporting Mode:



Im *Structured Mode* dagegen muß jedes Datenbankfeld, das benutzt werden soll, in einem DEFINE DATA-Statement angegeben werden (wie in den Kapiteln **Felder definieren** und **Datenbankzugriffe** beschrieben).

Structured Mode:



PORTIERBARE VON NATURAL GENERIERTE PROGRAMME

Ab Natural Version 5 sind von Natural generierte Programme (GPs) über die Plattformen UNIX, OpenVMS und Windows portierbar.

Die folgenden Themen werden nachfolgend behandelt:

- Kompatibilität
- Betrachtungen zum Endian-Modus
- ENDIAN-Parameter
- Von Natural generierte Programme übertragen.

Kompatibilität

Ein GP, das mit Natural Version 5 auf einer von Natural unterstützten UNIX-, OpenVMS- und Windows-Plattform katalogisiert wird, ist dann mit Natural Version 5 auf diesen Plattformen ohne erneute Kompilierung ausführbar. Diese Funktion vereinfacht die Verteilung von Anwendungen über Open Systems-Plattformen hinweg.

Mit Natural Version 4 oder Natural Version 3 generierte Natural-Anwendungen können mit Natural Version 5 ausgeführt werden, ohne die Anwendungen noch einmal zu katalogisieren (Aufwärts-Kompatibilität). In diesem Fall steht die portierbare GP-Funktionalität nicht zur Verfügung. Um das portierbare GP und andere Verbesserungen zu nutzen, muß das Katalogisieren mit Natural Version 5 ausgeführt werden.

Kommando-Prozessor-GPs und Natural-Expert-GPs sind nicht portierbar. Die Funktion zum Portieren von GPs steht für Großrechner-Plattformen nicht zur Verfügung. Dies bedeutet, daß auf Großrechnern generierte Natural-GPs nicht auf UNIX-, OpenVMS- und Windows-Plattformen ohne erneute Kompilierung – und umgekehrt – ausführbar sind.

Betrachtungen zum Endian-Modus

Abhängig davon, auf welcher UNIX-, OpenVMS- oder Windows-Plattform Natural Version 5 abläuft, berücksichtigt Natural Version 5 die Reihenfolge der Bytes, in der mehrere Byte umfassende Zahlen in dem GP gespeichert werden. Die zwei Byte langen Reihenfolge-Modi heißen "Little Endian" und "Big Endian".

- "Little Endian", bedeutet, dass das niedrigstwertigste Byte der Zahl im Hauptspeicher bei der niedrigsten Adresse gespeichert wird, und das höchstwertigste Byte bei der höchsten Adresse.
- "Big Endian" bedeutet, dass das höchstwertigste Byte der Zahl im Hauptspeicher bei der niedrigsten Adresse gespeichert wird, und das niedrigste Byte bei der höchsten Adresse.

Auf UNIX-, OpenVMS- und Windows-Plattformen kommen beide Endian-Modi zum Einsatz: Intel-Prozessoren und AXP-Computer (Natural auf Windows oder OpenVMS) haben als Reihenfolge der Bytes "Little Endian" und auf HP-Maschinen wird der "Big Endian"-Modus verwendet.

Natural Version 5 konvertiert ein portierbares GP automatisch in den Endian-Modus der Ausführungsplattform, wenn erforderlich. Diese Endian-Konvertierung wird nicht ausgeführt, wenn das GP bereits in dem Endian-Modus der Plattform generiert worden ist.

ENDIAN-Parameter

Um die Ausführungsleistung portierbarer GPs zu erhöhen, wurde der Profilparameter ENDIAN eingeführt. ENDIAN legt den Endian-Modus fest, in dem ein GP bei der Kompilierung generiert wird:

Spalte	Erklärung
DEFAULT	Der Endian-Modus der Maschine, auf der das GP generiert wird.
BIG	Big-Endian-Modus (höchstwertigstes Byte zuerst).
LITTLE	Little-Endian-Modus (niedrigstwertigstes Byte zuerst).

Die Werte DEFAULT, BIG und LITTLE sind Alternativen, wobei DEFAULT der Standardwert ist.

Der ENDIAN-Modus-Parameter kann gesetzt werden

- als ein Profilparameter mit der Natural Konfigurations-Utility
- als ein Start-Parameter
- als ein Session-Parameter oder mit dem GLOBALS-Kommando.

Von Natural generierte Programme übertragen

Um das portierbare GP auf unterschiedlichen Plattformen (UNIX, OpenVMS, Windows) zu nutzen, müssen die generierten Natural-Objekte in die Ziel-Plattform übertragen werden oder müssen von der Ziel-Plattform aus aufgerufen werden können, z.B. über NFS.

Wenn der Object Handler SYSOBJH verwendet wird, empfiehlt es sich, von Natural generierte Objekte oder sogar ganze Natural-Anwendungen zu verteilen. Dies wird durch Entladen der Objekte in die Source-Umgebung in eine Arbeitsdatei erreicht, wobei die Arbeitsdatei in die Ziel-Umgebung übertragen und die Objekte von der Arbeitsdatei geladen werden.

Um Ihre von Natural generierten Objekte über Open Systems-Plattformen zu implementieren

- ① Starten Sie den Natural Object Handler.
Entladen Sie alle erforderlichen Objekte in eine Arbeitsdatei des Typs "portable" (portierbar).
Fehlermeldungen, falls erforderlich, können auch in die Arbeitsdatei entladen werden.
Wichtig:
Der angegebene Arbeitsdatei-Typ muß portierbar sein. PORTABLE führt eine automatische Endian-Konvertierung einer Arbeitsdatei aus, wenn sie auf eine andere Maschine übertragen wird.
Siehe auch **Arbeitsdateityp** im Abschnitt **Define Work File** im *Natural Statements Handbuch*.
- ② Übertragen Sie die Arbeitsdatei in die Zielumgebung.
Abhängig vom Übertragungs-Mechanismus (Netzwerk, CD, Diskette, Band, E-Mail, Download, usw.) kann der Einsatz eines komprimierten Archivs wie z.B. einer ZIP-Datei oder einer Verschlüsselung mit UUENCODE/UUDECODE oder von etwas Ähnlichem Sinn machen. Für das Kopieren über FTP ist ein binärer Übertragungstyp erforderlich.
Anmerkung:
Entsprechend der verwendeten Übertragungs-Methode kann es vor dem Fortfahren mit der Lade-Funktion erforderlich sein, das Satz-Format und die Attribute oder Blockgröße der übertragenen Arbeitsdatei in Abhängigkeit von der spezifischen Ziel-Plattform anzupassen. Die Arbeitsdatei sollte dasselbe Format und dieselben Attribute auf der Ziel-Plattform haben wie eine Arbeitsdatei desselben Typs, die auf der Ziel-Plattform selbst generiert wurde. Verwenden Sie Betriebssystem-Werkzeuge, wenn eine Anpassung erforderlich ist.
- ③ Starten Sie den Natural Object Handler in der Zielumgebung.
Wählen Sie "portable" (portierbar) als Arbeitsdatei-Typ.
Laden Sie die Natural-Objekte und Fehlermeldungen von der Arbeitsdatei.

Weitere Einzelheiten zur Bedienung des Natural Object Handlers entnehmen Sie der SYSOBJH Utility-Dokumentation.

Nur für Windows:

Sie können weitere Informationen darüber finden, wie Sie eine Anwendung von einer Natural Development Workstation in eine Natural Runtime Workstation portieren, und zwar im Abschnitt **Porting Procedure Overview** in der Laufzeitversions-Dokumentation von Natural.

Außer der zuvor genannten bevorzugten Methode gibt es noch zahlreiche andere Möglichkeiten, einzelne von Natural generierte Objekte oder sogar ganze Bibliotheken oder Teile davon mittels Betriebssystemswerkzeugen und anderen Übertragungs-Methoden zu “verschieben” oder zu kopieren. Damit die Objekte von Natural ausgeführt werden können, müssen Sie in allen diesen Fällen in die Natural-Systemdatei FUSER importiert werden, so daß die FILEDIR.SAG-Struktur angepaßt wird.

Nur für Windows:

Informationen über das FNAT- oder FUSER Directory (Verzeichnis) entnehmen Sie den Systemdateien FNAT und FUSER im Abschnitt **Host Communication and System File Simulation** in Ihrer *Natural for Windows–Dokumentation*.

Dies kann mit einer der folgenden Methoden ausgeführt werden:

- mittels der Import-Funktion der SYSMAIN-Utility.
- mittels der FTOUCH-Utility.
Diese Utility kann ohne Aufruf von Natural verwendet werden.

Auf Windows-Plattformen: Es ist auch möglich, Dateien vom Windows Explorer in die Natural-Umgebung zu importieren, und zwar mittels der Funktionen Drag & Drop oder Cut, Copy und Paste. Dies bedeutet, wenn Sie Zugriff auf die Natural-Objekte haben, die Sie über den Windows Explorer importieren möchten, können Sie Drag & Drop oder Cut, Copy und Paste benutzen, und die Datei FILEDIR.SAG wird automatisch aktualisiert.

Weitere Einzelheiten siehe **Copying or Moving Objects** und **Importing Objects** in der *Natural Studio-Dokumentation*.

Dasselbe gilt, wenn ein Direktzugriff von einer Ziel-Plattform auf die generierten Objekte in der Quell-Umgebung möglich ist, zum Beispiel, über NSF, Network File Server, usw. In diesem Fall müssen die Objekte auch importiert werden.

Anmerkung:

Mit Natural Version 5.1 wird die Funktion noch nicht unterstützt, die es ermöglicht, dass eine gemeinsame FNAT- oder FUSER-Systemdatei von verschiedenen Open System-Plattformen gemeinsam benutzt werden kann. Die Datei FILEDIR.SAG ist noch nicht plattform-unabhängig.

INDEX

A

ACCEPT-Statement, 93

Addition

 ADD-Statement, 243

 COMPUTE-Statement, 241

ADD-Statement, 243

AL-Session-Parameter, 143

Anwendung

 Ausführung abbrechen, STOP-Statement, 207

 Struktur von, 181

Applikation. *Siehe* Anwendung

Arithmetik, 241

 ADD-Statement, 243

 COMPUTE-Statement, 241

 DIVIDE-Statement, 243

 MULTIPLY-Statement, 243

 SUBTRACT-Statement, 243

Array, 29

Siehe auch Multiples Feld; Periodengruppe

 Ausgangswert für, DEFINE DATA-Statement, 31

 dreidimensional, 38

 in Datenbank, 40, 52

 Index-Notation, 30, 41, 55

 interner Zähler, 58

ASSIGN-Statement. *Siehe* COMPUTE-Statement

AT BREAK-Statement, 223, 231

AT END OF DATA-Statement, 97

AT END OF PAGE-Statement, 128

AT START OF DATA-Statement, 97

AT TOP OF PAGE-Statement, 128

Aufrufen, Natural-Objekte, 182

Ausgabe

 von Daten, 101

 Seitenlayout, 102

 von Nullwerten, ZP-Session-Parameter, 148

Natural Leitfaden zur Programmierung

Ausgabelänge eines Feldes

AL-Parameter, 143

NL-Parameter, 143

Ausgangswert

für Array, DEFINE DATA-Statement, 31

für Benutzervariable, DEFINE DATA-Statement, 23

Ausprägung, 54

Siehe auch Array

B

BACKOUT TRANSACTION-Statement, 88, 90

Bedingte Verarbeitung, 208

BEFORE BREAK PROCESSING-Statement, 235

Beispielprogramme, Library SYSEXPG, 2

Benutzervariablen, 11

Ausgangswert für, DEFINE DATA-Statement, 23

Name von, DEFINE DATA-Statement, 12

Berechnen. *Siehe* Arithmetik

Bildschirmmaske. *Siehe* Map

C

C*-Notation, 58

CALLNAT-Statement, 179, 191

Class, verteilte objekt-basierte Anwendungen, 204

COMPRESS-Statement, 245

COMPUTE-Statement, 243

COMPUTE-Statement, 241

Copycode, 202

D

Data Area, 175

Data Definition Module. *Siehe* DDM

*DATA-Systemvariable, 255

Daten, Ausgabe von, 101

Seitenlayout, 102

- Datenbank
 - Änderung auf, 85
 - Referenzierung von, 281
 - Verarbeitungsschleifen, 79
 - Zugriff auf, 47, 61
 - FIND-Statement, 71
 - HISTOGRAM-Statement, 76
 - READ-Statement, 62
- Datenbereich. *Siehe* Data Area
- Datenblöcke, 43
- Datendefinitionsmodul. *Siehe* DDM
- Datensatz im "Hold", 85
- Datum
 - Ausgabeformat, DFOUT-Parameter, 264
 - Editiermasken, 259
 - Format, DF-Parameter, 260
 - Format für Seitenüberschrift, DFTITLE-Parameter, 274
 - Format für Stack, DFSTACK-Parameter, 265
 - Konstanten, 18
 - Standardformat, DTFORM-Profilparameter, 260
 - Systemvariablen, 251, 259
 - Verarbeitung, 259
- DDM, 48
 - Bestandteile, 49
- DEFINE DATA-Statement, 6
 - View-Definition, 59
- DEFINE SUBROUTINE-Statement, 187
- DELETE-Statement, 85
- Deskriptor, 49
- DF-Parameter, 260
- DFOUT-Parameter, 264
- DFSTACK-Parameter, 265
- DFTITLE-Parameter, 274
- Dialog, ereignisgesteuerte Anwendungen, 204
- DISPLAY-Statement, 105
 - in Kombination mit WRITE-Statement, 159
 - vertikale Ausgaben, 159
- DIVIDE-Statement, 243
- Division
 - COMPUTE-Statement, 241
 - DIVIDE-Statement, 243
- Dreidimensionales Array, 38

Natural Leitfaden zur Programmierung

DTFORM-Profilparameter, 260

E

Editiermaske, 153
Standard für Datum, DF-Parameter, 260
EJECT-Statement, 124
EJ-Session-Parameter, 124
EM-Session-Parameter, 153
END TRANSACTION-Statement, 85
ENDING-Klausel, READ-Statement, 66
END-Statement, 207
Ereignisgesteuerte Anwendungen, Dialog, 204
ESCAPE-Statement, 217
ESCAPE ROUTINE, 193, 194, 197
ES-Session-Parameter, 149

F

FC-Session-Parameter, 136
Feld
Siehe auch Benutzervariablen
multiples. *Siehe* Multiples Feld
FETCH-Statement, 184
FILLER-Option, DEFINE DATA-Statement, 27
FIND-Statement, WHERE-Klausel, 73
FIND-Statement, 71
Format, von Benutzervariablen, 13
Füllzeichen, für Überschriften, 136

G

GC-Session-Parameter, 136
GET-Statement, 88
GET TRANSACTION DATA-Statement, 90
Global Data Area, 175, 177
Gruppe
Siehe auch Periodengruppe
in DEFINE DATA-Statement, 9
in einem View, 49

Gruppenwechsel, 223

H

Hauptprogramm, 182
HC-Session-Parameter, 135
Helproutine, 195
HISTOGRAM-Statement, 76
Hold-Logik, 85
HW-Session-Parameter, 135

I

IA-Session-Parameter, 258
IC-Session-Parameter, 142
ID-Session-Parameter, 258
IF-Statement, 208
INCLUDE-Statement, 202
Index. *Siehe* Array
Interne Satznummer. *Siehe* ISN
ISN, Lesereihenfolge von Datensätzen, 62
IS-Session-Parameter, 146

J

Jahr 2000, Datumsformat für Seitenüberschrift, DFTITLE-Parameter, 274

K

Klasse, 204
Kompatibilität, Portierbare Programme, von Natural generiert, 283
Konstanten, 15
 namentlich definierte, 22

L

Länge, von Benutzervariablen, 13
LC-Session-Parameter, 142

Natural Leitfaden zur Programmierung

Leerstellen zwischen Spalten, SF-Session-Parameter, 110
Leerzeilenunterdrückung, 149
LENGTH-Option, DEFINE DATA-Statement, 23
Level
 aufgerufener Unterprogramme, 182
 in DDM, 49
 in DEFINE DATA-Statement, 8
Limit, für Verarbeitungsschleife
 LIMIT-Statement, 213
 LT-Session-Parameter, 213
LIMIT-Statement, 213
Local Data Area, 6, 175
Logisch sequentielles Lesen von Datensätzen, 62
Logische Transaktion, 85
Loop. *Siehe* Verarbeitungsschleife
LT-Session-Parameter, 213

M

Map, 194
Maske. *Siehe* Editiermaske; Map
Mathematische Berechnungen. *Siehe* Arithmetik
Mathematische Funktionen, 249
Modi, Reporting Mode und Structured Mode, 275
Modulare Anwendungsstruktur, 181
MOVE-Statement, 243
Multiples Feld, 40, 53
 Siehe auch Array
 in DDM, 49
 Referenzierung von, 55
Multiplikation
 COMPUTE-Statement, 241
 MULTIPLY-Statement, 243
MULTIPLY-Statement, 243

N

Neustarten einer Transaktion, 85, 90
NEWPAGE-Statement, 124
Nicht-Natural-Dateien, Ressource, 204
NL-Session-Parameter, 143

Null, Ausgabe von, ZP-Parameter, 148

O

Objekttypen, 173

P

Parameter Data Area, 175, 179

PERFORM BREAK PROCESSING-Statement, 237

PERFORM-Statement, 179, 187

Periodengruppe, 40, 52, 54

Siehe auch Array

in DDM, 49

Referenzierung von, 55

Physisch sequentielles Lesen von Datensätzen, 62

Portierbare Programme, von Natural generiert, 283

Kompatibilität, 283

Programm, 184

Programme, 181

Programmierobjekte, Typen, 173

PS-Session-Parameter, 123

R

READ-Statement

ENDING-Klausel, 66

STARTING-Klausel, 66

WHERE-Klausel, 68

READ-Statement, 62

Rechnen. *Siehe* Arithmetik

REDEFINE-Option, DEFINE DATA-Statement, 26

Redefinition, DEFINE DATA-Statement, 26

REJECT-Statement, 93

RELEASE-Statement, 258

REPEAT-Statement, 215

Report

Erstellung von, 101

Layout von, 102

Reporting Mode, 275

Natural Leitfaden zur Programmierung

RESET-Statement, 25
Ressource, Nicht-Natural-Dateien, 204
Restart einer Transaktion, 85

S

Satz, "Hold"-Logik, 88
Schleifen, Verarbeitung, Datenbank, 79
Schleifenverarbeitung, 212
Seite
 Größe, PS-Session-Parameter, 123
 logische, 122
 physische, 122
 Überschrift, DFTITLE-Parameter, 274
 Überschrift, 118
 WRITE TITLE-Statement, 120
 Unterschrift, 127
 Vorschub, 118
 EJ-Session-Parameter, 124
 EJECT-Statement, 124
 NEWPAGE-Statement, 124
SF-Session-Parameter, 110
SG-Session-Parameter, 143
SKIP-Statement, 120
Sourcecode-Zeilenummer, 219
Spaltenüberschriften, 130
Stack, 255
 Datumsformat für, DFSTACK-Parameter, 265
 schreiben auf
 STACK-Profilparameter, 257
 STACK-Statement, 258
STACK-Profilparameter, 257
STACK-Statement, 255, 258
STARTING-Klausel, READ-Statement, 66
Statements, Label, 219
STOP-Statement, 207
Structured Mode, 275
Stufe. *Siehe* Level
Subprogramm, 191
 Übergabe von Parametern an, 179
Subprogramme, 181

- Subroutine, 187
 - Übergabe von Parametern an, 179
- Subroutinen, 181
- SUBTRACT-Statement, 243
- Subtraktion
 - COMPUTE-Statement, 241
 - SUBTRACT-Statement, 243
- SUSPEND IDENTICAL SUPPRESS-Statement, 146
- SYSEXPB-Library , 2
- SYSEXRM-Library , 2
- Systemfunktionen, 223, 252
- Systemvariablen, 250

T

- Tabelle. *Siehe* Array
- TC-Parameter, 142
- Teilung. *Siehe* Division
- Text, Objekttyp, 203
- Transaktionslogik, 85

U

- UC-Session-Parameter, 138
- Überschriften, von Ausgabespalten, 130
- Uhrzeit. *Siehe* Zeit
- Unterdrückung identischer Werte, 146

V

- Variablen. *Siehe* Benutzervariablen; Systemvariablen
- Verarbeitung, bedingte, 208
- Verarbeitungsschleife, 212
- Verarbeitungsschleifen, Datenbank, 79
- Verteilte objekt-basierte Anwendungen, Class, 204
- Vertikale Ausgabe, 159
- View, Definition von, DEFINE DATA-Statement, 59
- Vorzeichen, SG-Session-Parameter, 143

W

- Wert, Ausgangswert, 23
 - für Array, 31

Natural Leitfaden zur Programmierung

WHERE-Klausel

 FIND-Statement, 73

 READ-Statement, 68

WH-Session-Parameter, 88

WRITE-Statement, 107

 in Kombination mit DISPLAY-Statement, 159

WRITE TITLE-Statement, 120

WRITE TRAILER-Statement, 127

Y

Year Sliding Window, YSLW-Parameter, 267

YSLW-Parameter, 267

Z

Zeilenvorschub

 Schrägstrich-Notation, 113

 SKIP-Statement, 120

Zeit

 Konstanten, 18

 Systemvariablen, 251

Zentrieren von Überschriften, HC-Session-Parameter, 135

ZP-Session-Parameter, 148

Zurücksetzen, Feldwert, RESET-Statement, 25

Natural Leitfaden zur Programmierung