Natural für z/OS

Leitfaden zur Programmierung

Version 9.2.4

Oktober 2025

ADABAS & NATURAL

Dieses Dokument gilt für Natural für z/OS ab Version 9.2.4.

Hierin enthaltene Beschreibungen unterliegen Änderungen und Ergänzungen, die in nachfolgenden Release Notes oder Neuausgaben bekanntgegeben werden.

Dokument-ID: NATMF-NNATPROGRAMMING-924-20251031DE

Inhaltsverzeichnis

Vorwort

Dieses Dokument enthält vertiefende Informationen zu den Natural-Dokumenten, die in der **Dokumentationsübersicht** im Abschnitt **Programmiersprache** aufgeführt sind, und behandelt grundlegende Aspekte der Anwendungsprogrammierung mit Natural.

Natural-Programmiermodi	„Reporting Mode“ und „Structured Mode“.
Objekte zum Erstellen und Pflegen von Natural-Anwendungen	Objekte (z.B. Programme und Datenbereiche), die zum Erstellen und Pflegen von Natural-Anwendungen verwendet werden.
Function Call	Definition von Function Calls.
Felder definieren	Definition von Variablen, Konstanten und Arrays.
Datenbankzugriffe	Zugriff aus Natural auf Daten in einer Adabas-Datenbank und in Nicht-Adabas-Datenbanken.
Steuerung der Ausgabe von Daten	Format und Steuerung von Ausgabe-Report-Daten.
Weitere Programmieraspekte	Verschiedene andere Aspekte der Programmierung mit Natural: **Text-Notation** **Benutzerkommentare** **Datenberechnungen** **Regeln für arithmetische Operationen** **Bedingte Verarbeitung – das IF-Statement** **Logische Bedingungen** **Schleifenverarbeitung** **Gruppenwechsel** **Natural-Stack** **Systemvariablen und Systemfunktionen** **Verarbeitung von Datumsinformationen** **Verarbeitung von Store Clock-Werten** **Ende eines Statements, Programms oder einer Anwendung** **Verarbeitung von Anwendungsfehlern** **Kompilierungsaspekte**
Statements für den Internet-Zugriff und Parsing	Natural-Statements für den Zugriff auf Internet, JSON und XML.
Gestaltung von Benutzungsoberflächen von Anwendungen	Mittel zur Gestaltung der Benutzungsoberflächen und Dialoge Ihrer Natural-Anwendungen.
NaturalX	Objekt-basiertes Programmieren und speziell dafür vorgesehene Natural-Statements.
Für Natural reservierte Schlüsselwörter	Liste aller Natural-Schlüsselwörter und für Natural reservierten Wörter.
Referenzierte Beispielprogramme	Natural-Beispielprogramme, auf die im *Leitfaden zur Programmierung* Bezug genommen wird.

Notation *vrs* bzw. *vr*

Die in diesem Dokument verwendete Notation *vrs* bzw. *vr* steht als Platzhalter für die betreffende Produktversion (siehe auch *Version* im *Glossary*).

1 Über diese Dokumentation

Dokumentationskonventionen

Konvention	Beschreibung
Fettschrift	>Kennzeichnet Elemente auf einem Bildschirm.
`Nichtproportionale Schrift`	Kennzeichnet Namen und Orte von Diensten im Format *`Ordner.Unterordner.Dienst`*, Programmierschnittstellen (APIs), Namen von Klassen, Methoden und Properties in Java.
`Kursivschrift`	Kennzeichnet: Variablen, für die Sie situations- oder umgebungsspezifische Werte angeben müssen. Neue Begriffe, wenn sie erstmals im Text auftreten. Verweise auf andere Dokumentationsquellen.
`Nichtproportionale Schrift`	Kennzeichnet: Text, den Sie eingeben müssen. Meldungen, die vom System angezeigt werden. Programmcode.
{ }	Zeigt eine Reihe von Auswahlmöglichkeiten an, von denen Sie eine auswählen müssen. Geben Sie nur die innerhalb der geschweiften Klammern vorhandenen Informationen ein. Geben Sie nicht die Klammersymbole { } ein.
|	Trennt zwei sich gegenseitig ausschließende Auswahlmöglichkeiten in einer Syntaxzeile voneinander ab. Geben Sie eine der Auswahlmöglichkeiten ein. Geben Sie nicht das Symbol | ein.
[ ]	Zeigt eine oder mehrere Optionen an. Geben Sie nur die innerhalb der eckigen Klammern vorhandenen Informationen ein. Geben Sie nicht die Klammersymbole [ ] ein.
...	Zeigt an, dass Sie mehrere Auswahlmöglichkeiten desselben Typs eingeben können. Geben Sie nur die Informationen ein. Geben Sie nicht die drei Auslassungspunkte (...) ein.

Online-Informationen und Support

Produktdokumentation

Sie finden die Produktdokumentation auf unserer Dokumentationswebsite unter **https://documentation.softwareag.com**.

Zusätzlich können Sie auch über **https://www.softwareag.cloud** auf die Dokumentation für die Cloud-Produkte zugreifen. Navigieren Sie zum gewünschten Produkt und gehen Sie dann, je nach Produkt, zu „Developer Center“, „User Center“ oder „Documentation“.

Produktschulungen

Sie finden hilfreiches Produktschulungsmaterial auf unserem Lernportal unter **https://knowledge.softwareag.com**.

Tech Community

Auf der Website unserer Tech Community unter **https://techcommunity.softwareag.com** können Sie mit Experten der Software AG zusammenarbeiten. Von hier aus können Sie zum Beispiel:

- Unsere umfangreiche Wissensdatenbank durchsuchen.
- In unseren Diskussionsforen Fragen stellen und Antworten finden.
- Die neuesten Nachrichten und Ankündigungen der Software AG lesen.
- Unsere Communities erkunden.
- Unsere öffentlichen Repositories auf GitHub and Docker unter **https://github.com/softwareag** und **https://hub.docker.com/publishers/softwareag** besuchen und weitere Ressourcen der Software AG entdecken.

Produktsupport

Support für die Produkte der Software AG steht lizenzierten Kunden über unser Empower-Portal unter **https://empower.softwareag.com** zur Verfügung. Für viele Dienstleistungen auf diesem Portal benötigen Sie ein Konto. Wenn Sie noch keines haben, dann können Sie es unter **https://empower.softwareag.com/register** beantragen. Sobald Sie ein Konto haben, können Sie zum Beispiel:

- Produkte, Aktualisierungen und Programmkorrekturen herunterladen.
- Das Knowledge Center nach technischen Informationen und Tipps durchsuchen.
- Frühwarnungen und kritische Alarme abonnieren.
- Supportfälle öffnen und aktualisieren.
- Anfragen für neue Produktmerkmale einreichen.

Datenschutz

Die Produkte der Software AG stellen Funktionen zur Verarbeitung von personenbezogenen Daten gemäß der Datenschutz-Grundverordnung (DSGVO) der Europäischen Union zur Verfügung. Gegebenenfalls sind in der betreffenden Systemverwaltungsdokumentation entsprechende Schritte dokumentiert.

I Natural-Programmiermodi

2 Natural-Programmiermodi

Dieses Kapitel beschreibt die zwei von Natural unterstützten Programmiermodi.

Anmerkung: Generell wird empfohlen, ausschließlich im Structured Mode zu programmieren, weil dieser Programmiermodus übersichtlicher strukturierte Anwendungen ergibt. Deshalb gelten die Erläuterungen und Beispiele in diesem *Leitfaden zur Programmierung* normalerweise nur für den Structured Mode.

Zweck der Programmiermodi

Natural bietet zwei Formen der Programmierung:

- Reporting Mode
- Structured Mode

Anmerkung: Grundsätzlich empfiehlt es sich, ausschließlich im Structured Mode zu arbeiten, um klar strukturierte Anwendungen zu erhalten.

Reporting Mode

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.)

Bitte achten Sie darauf, dass manche Natural-Statements nur im Reporting Mode verfügbar sind, wohingegen andere eine spezifische Struktur aufweisen, wenn Sie im Reporting Mode benutzt werden. Eine Übersicht der Statements, die im Reporting Mode verwendet werden können, entnehmen Sie dem Abschnitt *Statements im Reporting Mode* in der *Statements*-Dokumentation.

Structured Mode

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 an das eigentliche Programmieren geht. Viele Fehler und Unzulänglichkeiten bei der Programmierung werden dadurch von vorneherein vermieden.

Eine Übersicht der im Structured Mode zu benutzenden Statements entnehmen Sie dem Abschnitt *Statements nach Funktionen* in der *Statements*-Dokumentation.

Programmiermodus festlegen/ändern

Der Standardprogrammiermodus wird vom Natural-Administrator mit dem Profilparameter `SM` festgelegt.

Weitere Informationen zum Profil- und Session-Parameter `SM` siehe Abschnitt *SM - Programmierung im Structured Mode* in der *Parameter-Referenz*-Dokumentation.

Sie können den vorgegebenen Modus mit dem Systemkommando `GLOBALS` und dem Session-Parameter `SM` ändern:

Structured Mode:	`GLOBALS SM=ON`
Reporting Mode:	`GLOBALS SM=OFF`

Weitere Informationen, wie Sie den Programmiermodus ändern können, finden Sie in folgenden Dokumenten: *Programmiermodus* in *Natural benutzen* und *SM - Programmierung im Structured Mode* in der *Parameter-Referenz*-Dokumentation.

Funktionale Unterschiede

Die wichtigsten funktionalen Unterschiede zwischen Reporting Mode und Structured Mode lassen sich wie folgt zusammenfassen:

- Syntax zum Beenden von Schleifen und funktionalen Blöcken
- Verarbeitungsschleife im Reporting Mode beenden
- Verarbeitungsschleife im Structured Mode beenden
- Platzierung von Datenelementen in einem Programm
- Datenbank-Referenzierung

Anmerkung: Ausführliche Informationen zu funktionalen Unterschieden zwischen den zwei Modi finden Sie in der *`Statements`*-Dokumentation. Sie enthält verschiedene Syntax-Diagramme und Syntax-Elementbeschreibungen für jedes modusabhängig Statement. Eine Funktionsübersicht der im Reporting Mode zu benutzenden Statements finden Sie im Abschnitt *Reporting Mode-Statements* in der *Statements*-Dokumentation.

Syntax zum Beenden von Schleifen und funktionalen Blöcken

Reporting Mode:	`(CLOSE) LOOP` und `DO ... DOEND`-Statements werden hierzu verwendet. `END-...`-Statements (außer `END-DEFINE`, `END-DECIDE` und `END-SUBROUTINE`) können nicht benutzt werden.
Structured Mode:	Jede Schleife oder logische Verarbeitungsbedingung muss ausdrücklich mit einem entsprechenden `END-...`-Statement abgeschlossen werden. Dadurch wird sofort deutlich, wo welche Schleife bzw. Bedingung aufhört. `LOOP` und `DO/DOEND`-Statements können nicht benutzt werden.

Die beiden folgenden Beispiele veranschaulichen die je nach Programmiermodus unterschiedliche Konstruktion von Verarbeitungsschleifen und logischen Bedingungen.

Beispiel — Reporting Mode:

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.

```
READ EMPLOYEES BY PERSONNEL-ID
DISPLAY NAME BIRTH
AT END OF DATA
   DO
      SKIP 2
      WRITE / 'LAST SELECTED:' OLD(NAME)
   DOEND
END
```

Das `END`-Statement beendet sämtliche aktiven Verarbeitungsschleifen.

Beispiel — Structured Mode:

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:

```
DEFINE DATA LOCAL
1 MYVIEW VIEW OF EMPLOYEES
  2 PERSONNEL-ID
  2 NAME
  2 BIRTH

END-DEFINE
READ MYVIEW BY PERSONNEL-ID
   DISPLAY NAME BIRTH
```

```
   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      /* closes inner FIND loop
LOOP         /* closes outer FIND loop
...
...
```

Beispiel 2 — END:

```
FIND ...
   FIND ...
   ...
   ...
END             /* closes all loops and ends processing
```

Beispiel 3 — SORT:

```
FIND ...
   FIND ...
   ...
   ...
SORT ...        /* closes all loops, initiates loop
...
END             /* closes SORT loop and ends processing
```

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       /* closes inner FIND loop
END-FIND          /* closes outer FIND loop
...
```

Beispiel 2 — READ:

```
READ ...
   AT END OF DATA
   ...
   END-ENDDATA
   ...
END-READ           /* closes READ loop
...
...
END
```

Beispiel 3 — SORT:

```
READ ...
   FIND ...
   ...
   ...
END-ALL          /* closes all loops
SORT             /* opens loop
...
...
END-SORT         /* closes SORT loop
END
```

Platzierung von Datenelementen in einem Programm

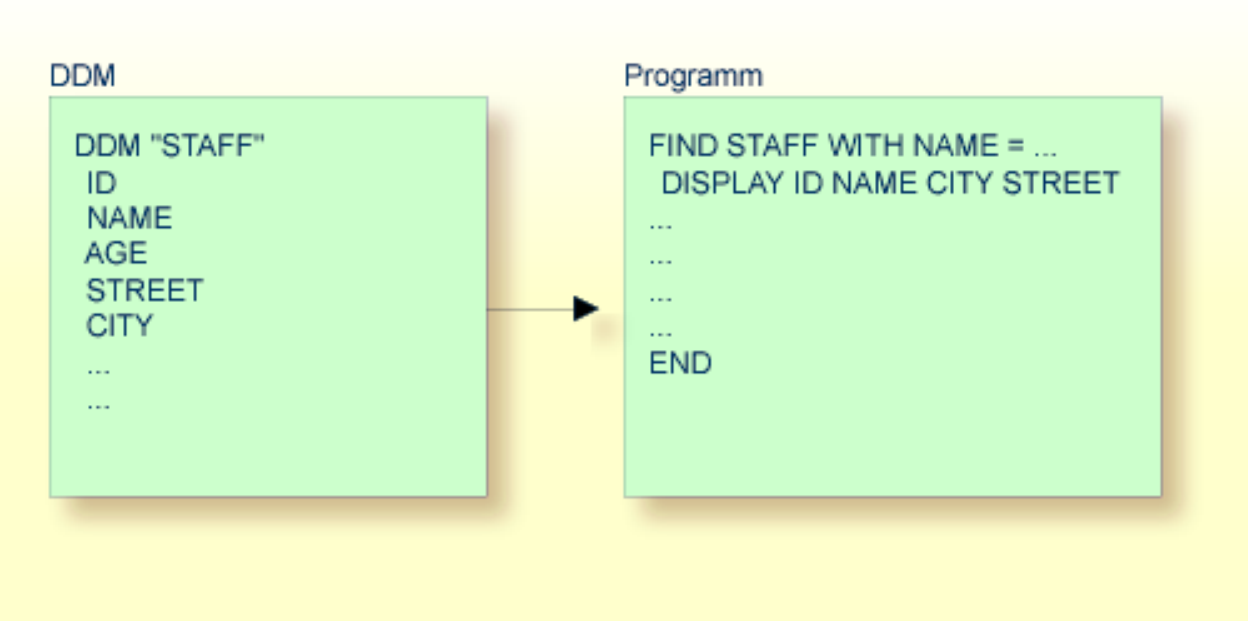
Im Reporting Mode können Datenbankfelder benutzt werden, ohne dass diese vorher in einem `DEFINE DATA`-Statement definiert werden müssen; außerdem können Benutzervariablen an jeder Stelle eines Programms, d.h. über das ganze Programm verstreut, definiert werden.

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.

Datenbank-Referenzierung

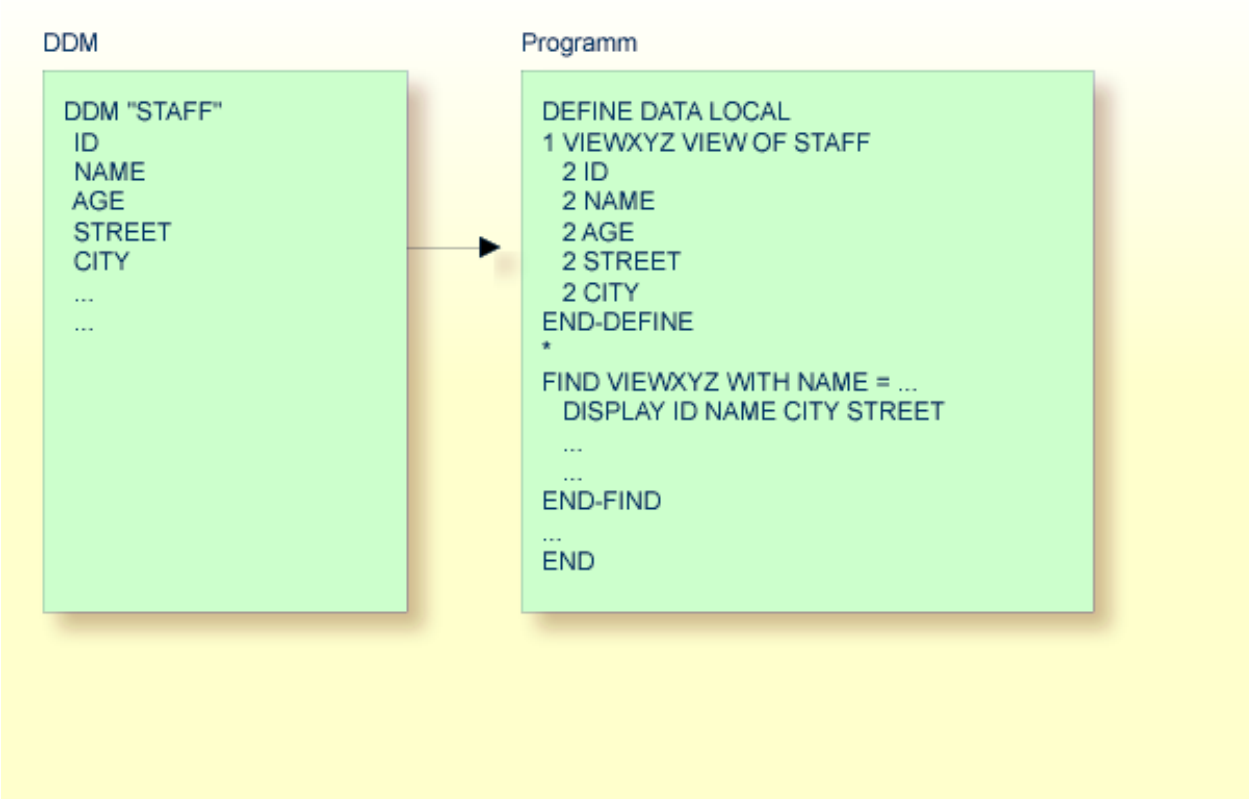
Reporting Mode:

Im Reporting Mode ist es möglich, Datenbankfelder oder DDMs zu benutzen, ohne diese in einer **Data Area** definiert zu haben.



Structured Mode:

Im Structured Mode dagegen muss jedes Datenbankfeld, das benutzt werden soll, in einem `DEFINE DATA`-Statement angegeben werden (wie in den Abschnitten *Felder definieren* und *Datenbankzugriffe* beschrieben).


DDM
DDM "STAFF"
ID
NAME
AGE
STREET
CITY
...
...
Programm
DEFINE DATA LOCAL
1 VIEWXYZ VIEW OF STAFF
2 ID
2 NAME
2 AGE
2 STREET
2 CITY
END-DEFINE
*
FIND VIEWXYZ WITH NAME = ...
DISPLAY ID NAME CITY STREET
...
...
END-FIND
...
END

II Objekte zum Erstellen und Pflegen von Natural-Anwendungen

Diese Dokument beschreibt verschiedene Natural-Objekttypen, die zum Erstellen, Verwalten und Steuern von Natural-Anwendungen zur Verfügung stehen.

Die folgende Tabelle enthält eine Übersicht über Natural-Objekte und Nicht-Natural-Objekte, ihre Verwendung und die Editoren bzw. Utilities, die zum Erstellen und Pflegen benutzt werden können.

Objekttyp	Verwendung	Editor bzw. Utility
Datenbereiche (Data Areas): **Local Data Area** **Global Data Area** **Parameter Data Area**	Definition von Variablen und Parametern für andere Natural-Objekte.	Datenbereich-Editor (Data Area Editor)
Datendefinitionsmodul (DDM)	Natural-Datendefinitionen für den Zugriff auf Datenbankdateien.	DDM-Editor (SYSDDM Utility)
Programme und untergeordnete Routinen: **Programm** **Subroutine** **Subprogramm** **Function**	Hauptprogramme, aufgerufene Routinen und Functions.	Programm-Editor
Helproutine	Hilfe-Anforderungen in Anwendungen.	
Copycode	Mehrfache Verwendung von Quellcode in anderen Natural-Objekten.	
Text	Dokumentation für Natural-Objekte.	
Class	Komponentenbasierte Anwendungen.	Programm-Editor In Natural for z/OS nicht anwendbar (nur Speicherung

Objekttyp	Verwendung	Editor bzw. Utility
		und Anzeige möglich)
Map	Zeichenbasierte Bildschirmmasken-Layouts.	Masken-Editor (Map Editor)
Adapter und GUI-Layout	Komplexe grafische Benutzungsoberflächen und Rich GUI Pages, die mittels eines externen Seiten-Layouts erzeugt werden.	Natural for Ajax Developer (siehe *Natural for Ajax* Dokumentation)
Dialog	Ereignisgesteuerte Anwendungen.	In Natural for z/OS nicht anwendbar (nur Speicherung und Anzeige möglich)
Resource	Nicht-Natural-Objekte, z.B. HTML-Dateien oder Bitmaps.	In Natural for z/OS nicht anwendbar (nur Speicherung und Anzeige möglich)
Recording	Aufgezeichnete Sessions für Test- und Kontrollzwecke.	Recording Utility
Fehlermeldung (Error Message)	Natural-Systemmeldungen und benutzerdefinierte Meldungen.	SYSERR Utility
Command Processor	Navigation mittels Kommandos.	SYSNCP Utility
Editor-Profil	Standardeinstellungen für den Programm-Editor bzw. Datenbereich-Editor (Data Area Editor).	Programm-Editor bzw. Datenbereich-Editor (Data Area Editor)
Map-Profil und Device-Profil	Standardeinstellungen für den Masken-Editor (Map Editor).	Masken-Editor (Map Editor)
Parameter-Profil	Standardparametereinstellungen für den Start einer Session.	SYSPARM Utility
Debug-Umgebung	Kontrolle der Programmausführung.	Debugger
Anwendungsprogrammierschnittstelle	Natural-Anwendungsprogrammierschnittstelle	SYSEXT Utility
	Anwendungsprogrammierschnittstelle eines Natural Add-on-Produkts	SYSAPI Utility

Verwandte Themen:

Weitere Informationen zur Benutzung von Natural-Objekten siehe *Natural-Objekte pflegen und ausführen* und *Namenskonventionen für Objekte* in *Natural benutzen*.

3 Datenbereiche (Data Areas)

Wie im Abschnitt *Felder definieren* erläutert, müssen alle Felder, die in einem Programm verwendet werden sollen, mit 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.

Eine solche separate Data Area ist ein Natural-Objekt, das von mehreren Natural-Programmen, -Subprogrammen, -Subroutinen, Helproutinen oder Klassen benutzt werden kann. Eine Data Area enthält Datenelement-Definitionen, wie z.B. benutzerdefinierte Variablen, Konstanten und Datenbankfelder aus einem Datendefinitionsmodul (DDM).

Alle Data Areas werden mit dem Data Area Editor erstellt und editiert.

Mit Natural können Sie folgende Arten von Data Areas anlegen und referenzieren:

- **Local Data Area**
- **Global Data Area**
- **Parameter Data Area**

Local Data Area (LDA)

Als „lokal“ definierte Variablen können nur von einem einzigen Natural-Objekt benutzt werden.

Sie haben zwei Möglichkeiten, lokale Daten zu definieren:

- Sie können die Daten innerhalb des Programms definieren.
- Sie können die Daten außerhalb des Programms in einem separaten Natural-Objekt, einer Local Data Area (LDA), definieren.

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

Beispiel 1 — Felddefinitionen innerhalb des DEFINE DATA-Statements:

Im folgenden Beispiel sind die Felder direkt innerhalb des `DEFINE DATA`-Statements des Programms 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 — Felddefinitionen in einer separaten Data Area:

Im diesem Beispiel sind dieselben Felder nicht im `DEFINE DATA`-Statement des Programms, sondern in einer LDA mit dem Namen `LDA39` definiert, und das `DEFINE DATA`-Statement enthält lediglich eine Referenz auf diese Data Area.

Programm:

Das Programm selbst enthält keine Felddefinitionen, sondern referenziert die in der `LDA39` enthaltenen Felddefinitionen.

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

Referenzierte Local Data Area `LDA39`:

```
I T L  Name                            F Length     Miscellaneous
All -- --------------------------------- - ---------- ------------------------->
  V  1 VIEWEMP                                       EMPLOYEES
     2 PERSONNEL-ID                    A          8
     2 FIRST-NAME                      A         20
     2 NAME                            A         20
     1 #VARI-A                         A         20
     1 #VARI-B                         N        3.2
     1 #VARI-C                         I          4                          ↩
```

Global Data Area (GDA)

Wenn Sie die Datenelemente in einem separaten Bereich definieren möchten, der von mehreren Natural-Objekten genutzt werden kann, verwenden Sie eine Globa Data Area.

Folgende Themen werden behandelt:

- GDA anlegen und referenzieren

GDA anlegen und referenzieren

GDAs werden mit dem Natural Data Area Editor angelegt und geändert. Weitere Informationen entnehmen Sie dem Abschnitt *Datenbereich-Editor (Data Area Editor)* in der *Editoren*-Dokumentation.

Eine GDA, die von einem Natural-Objekt referenziert wird, muss in derselben Natural-Library (oder in einer für diese Library definierten Steplib) gespeichert werden, in der auch das diese GDA referenzierende Objekt gespeichert ist.

Anmerkung: Verwendung einer GDA namens `COMMON` beim Start:
Wenn eine GDA mit Namen `COMMON` in einer Library vorhanden ist, wird das Programm mit Namen `ACOMMON` automatisch aufgerufen, wenn Sie sich per `LOGON` in dieser Library anmelden.

Wichtig: Wenn Sie eine Anwendung erstellen, bei der mehrere Natural-Objekte eine GDA referenzieren, denken Sie bitte daran, dass Änderungen an den Datenelement-Definitionen in der GDA alle Natural-Objekte betreffen, die diese Data Area referenzieren. Deshalb müssen diese Objekte mittels des Kommandos `CATALOG` oder `STOW` neu kompiliert werden, nachdem die GDA geändert worden ist.

Um eine GDA zu benutzen, muss ein Natural-Objekt sie mit der `GLOBAL`-Klausel des `DEFINE DATA`-Statements referenzieren. Jedes Natural-Objekt kann nur eine GDA referenzieren; d.h. ein `DEFINE DATA`-Statement darf nicht mehr als eine `GLOBAL`-Klausel enthalten.

GDA-Instanzen anlegen und löschen

Die erste Instanz einer GDA wird angelegt und zur Laufzeit initialisiert, wenn das erste, sie referenzierende Natural-Objekt ausgeführt wird.

Sobald eine GDA-Instanz angelegt worden ist, können die Datenwerte, die sie enthält, von allen Natural-Objekten gemeinsam benutzt werden, die diese GDA referenzieren (`DEFINE DATA GLOBAL`-Statement) und die von einem `PERFORM`-, `INPUT`- oder `FETCH`-Statement aufgerufen werden. Alle Objekte, die eine GDA-Instanz gemeinsam benutzen, arbeiten mit dieselben Datenelementen.

Eine neue GDA-Instanz wird angelegt, wenn Folgendes gilt:

- Ein Subprogramm, das eine GDA (eine *beliebige* GDA) referenziert, wird mit einem `CALLNAT`-Statement aufgerufen.
- Ein Subprogramm, das *keine* GDA referenziert, ruft ein Objekt auf, das eine GDA (eine *beliebige* GDA) referenziert.

Beim Anlegen einer neuen Instanz einer GDA wird die aktuelle GDA-Instanz zeitweilig unterbrochen, und die Datenwerte, die sie enthält, werden auf einem Stack abgelegt. Das Subprogramm

referenziert dann die Datenwerte in der neu erstellten GDA-Instanz. Auf die Datenwerte in der/den zeitweilig unterbrochenen GDA-Instanz/Instanzen ist kein Zugriff möglich. Ein Objekt bezieht sich nur auf eine GDA-Instanz und kann nicht auf vorherige GDA-Instanzen zugreifen. Ein GDA-Datenelement kann nur an ein Subprogramm übergeben werden, wenn das Element als ein Parameter im `CALLNAT`-Statement definiert wird.

Wenn das Subprogramm zum aufrufenden Objekt zurückkehrt, wird die von ihm referenzierte GDA-Instanz gelöscht, und die vorher zeitweilig unterbrochene GDA-Instanz wird mit ihren Datenwerten wieder aufgenommen.

Eine GDA-Instanz und ihr Inhalt wird gelöscht, wenn einer der folgenden Punkte gilt:

- Das nächste `LOGON` wird ausgeführt.
- Eine andere GDA wird auf derselben Stufe referenziert (Stufen werden später in diesem Abschnitt beschrieben).
- Ein `RELEASE VARIABLES`-Statement wird ausgeführt.

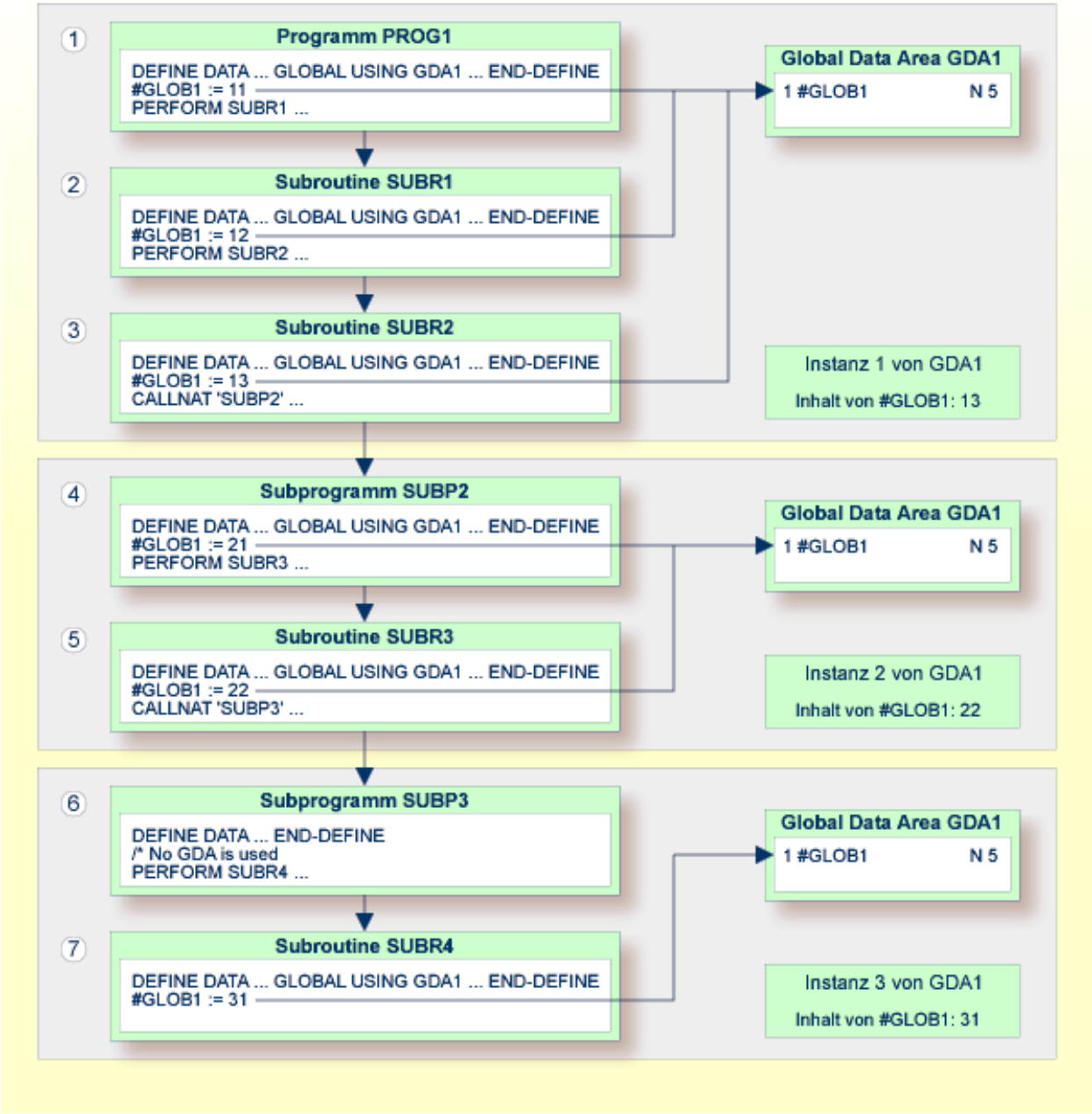
  In diesem Fall werden die Datenwerte in einer GDA-Instanz zurückgesetzt, und zwar entweder wenn ein Programm auf der Stufe 1 seine Ausführung beendet, oder wenn das Programm ein anderes Programm über ein `FETCH`- oder `RUN`-Statement aufruft.

  Die folgende Grafik zeigt, wie Objekte GDAs referenzieren und Datenelemente in GDA-Instanzen gemeinsam nutzen.

Gemeinsame Nutzung von GDA-Instanzen

Die folgende Grafik veranschaulicht, dass ein eine GDA referenzierendes Subprogramm die Datenwerte nicht gemeinsam in einer GDA-Instanz benutzen kann, die von dem aufrufenden Programm referenziert wird. Ein Subprogram, das dieselbe GDA referenziert wie das aufrufende Programm, legt eine neue Instanz dieser GDA an. Die Datenelemente, die in einer GDA definiert sind, die von einem Subprogramm referenziert wird, können aber von einer vom Subprogramm aufgerufenen Subroutine oder Helproutine gemeinsam benutzt werden.

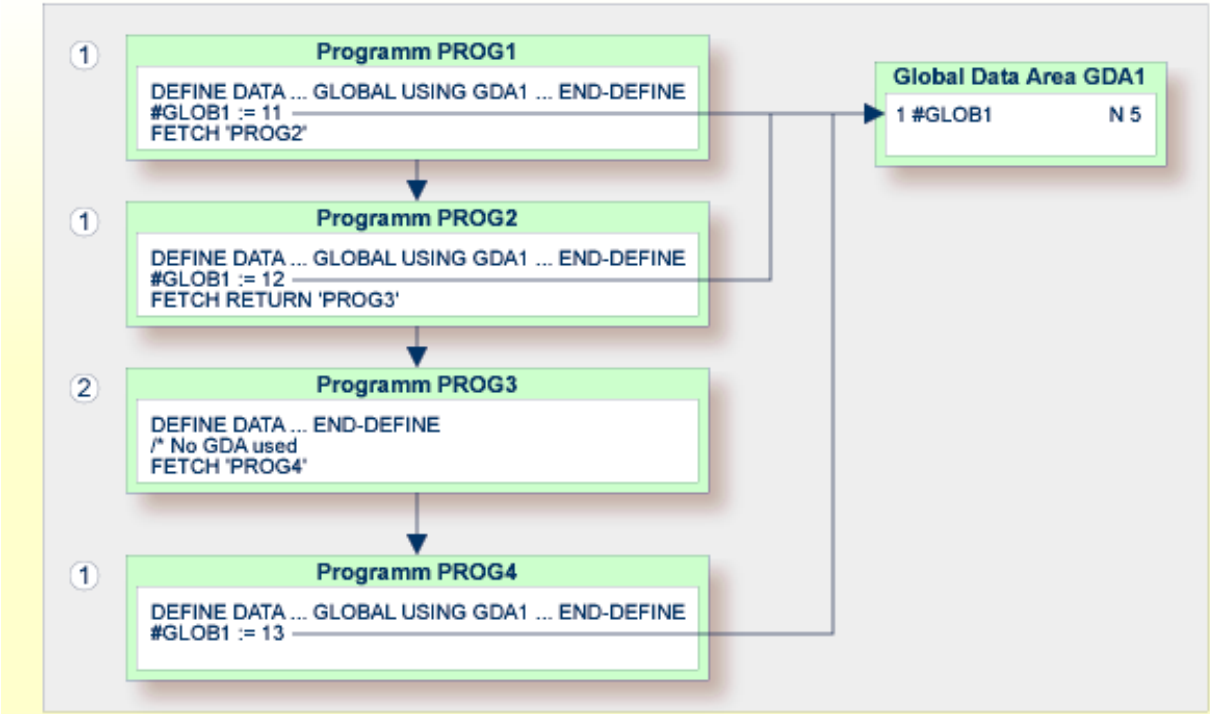
Die folgende Grafik zeigt drei GDA-Instanzen von `GDA1` und die Endwerte, die jeder GDA-Instanz vom Datenelement `#GLOB1` zugewiesen werden. Die Zahlen (1) bis (7) verweisen auf die hierarchischen Stufen der Objekte.



Benutzung von FETCH oder FETCH RETURN

Die folgende Grafik veranschaulicht, dass Programme, die dieselbe GDA referenzieren und sich gegenseitig mit dem `FETCH`- oder `FETCH RETURN`-Statement aufrufen, die in dieser GDA definierten Datenelemente gemeinsam benutzen. Wenn eines dieser Programme keine GDA referenziert, bleibt die Instanz der vorher referenzierten GDA aktiv, und die Werte der Datenelemente werden zurückbehalten.

Die Zahlen (1) und (2) verweisen auf die hierarchischen Stufen der Objekte.



Benutzung von FETCH mit verschiedenen GDAs

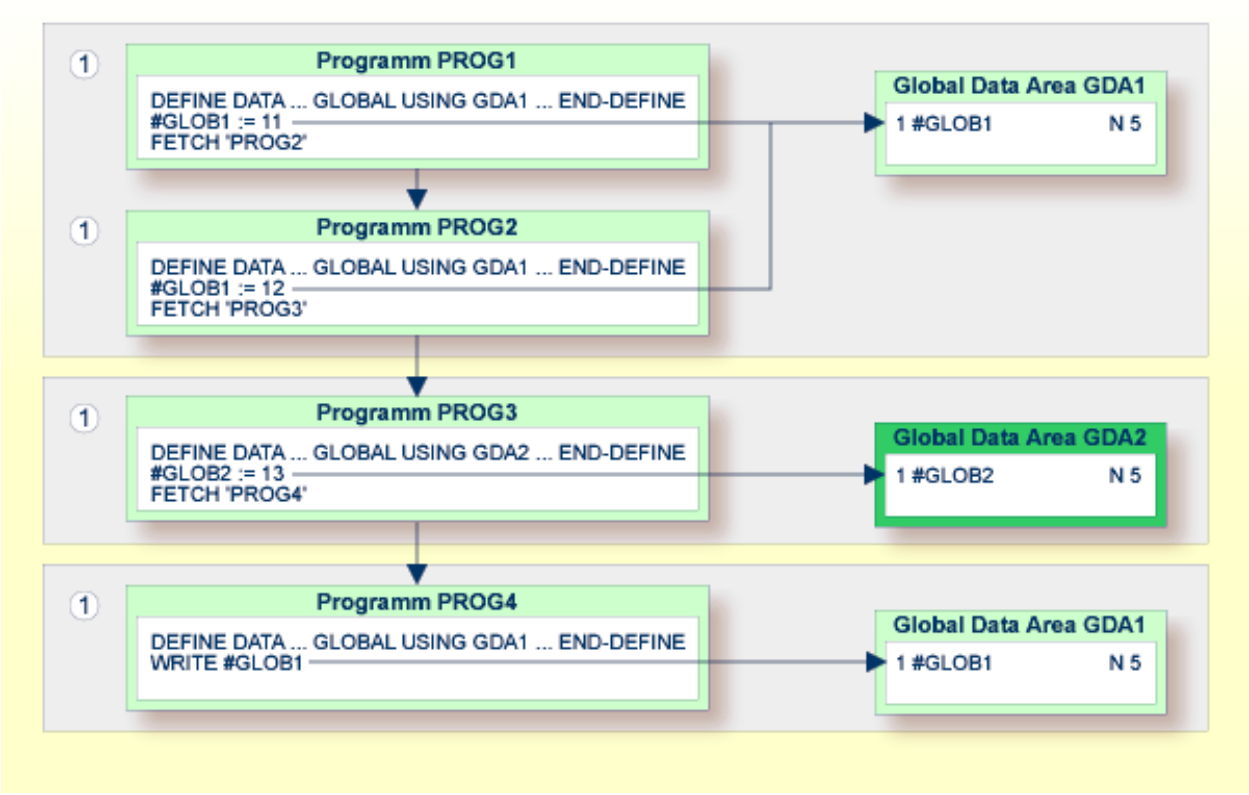
Die folgende Grafik veranschaulicht Folgendes: Wenn ein Programm das `FETCH`-Statement benutzt, um ein anderes Programm aufzurufen, das eine unterschiedliche GDA referenziert, wird die aktuelle, vom aufrufenden Programm referenzierte Instanz der GDA (hier: `GDA1`) gelöscht. Wenn diese GDA dann erneut von einem anderen Programm referenziert wird, dann wird eine neue Instanz dieser GDA angelegt, bei der alle Datenelemente ihre Anfangswerte haben.

Sie können das `FETCH RETURN`-Statement nicht benutzen, um ein anderes Programm aufzurufen, das eine unterschiedliche GDA referenziert.

Die Zahl (1) verweist auf die hierarchische Stufe der Objekte.

Die aufrufenden Programme `PROG3` und `PROG4` beeinflussen die GDA-Instanzen wie folgt:

- Das Statement `GLOBAL USING GDA2` in `PROG3` legt eine Instanz von `GDA2` an und löscht die aktuelle Instanz von `GDA1`.
- Das Statement `GLOBAL USING GDA1` in `PROG4` löscht die aktuelle Instanz von `GDA2` und legt eine neue Instanz von `GDA1` an. Als Ergebnis davon zeigt das `WRITE`-Statement den Wert Null (0) an.



Datenblöcke

Um Datenspeicherplatz zu sparen, können Sie eine GDA mit Datenblöcken erstellen.

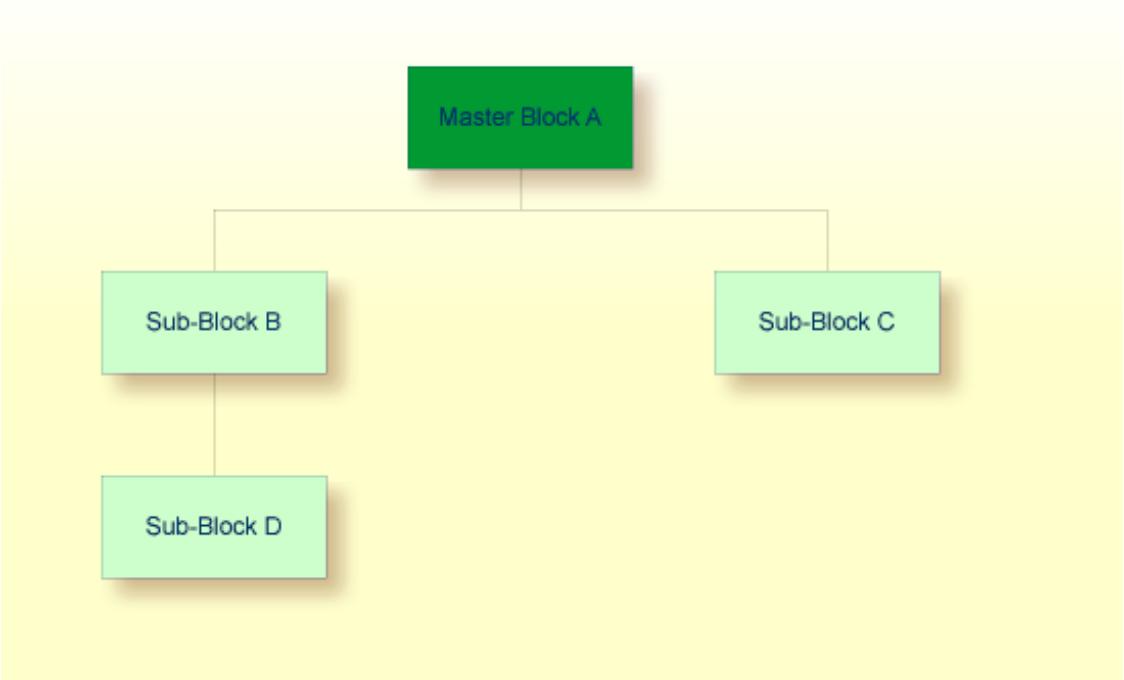
Folgende Themen werden in diesem Abschnitt behandelt:

- Beispiel für die Benutzung von Datenblöcken
- Datenblöcke definieren
- Block-Hierarchien

Beispiel für die Benutzung von Datenblöcken

Datenblöcke können sich bei der Programmausführung gegenseitig überlagern, wodurch Speicherplatz eingespart wird.

Gehen wir beispielsweise davon aus, dass bei der folgenden vorgegebenen Hierarchie den Blöcken B und C derselbe Speicherbereich zugewiesen würde. Somit ist es nicht möglich, dass die Blöcke B und C gleichzeitig in Benutzung sind. Eine Änderung an Block B würde zur Zerstörung des Inhalts von Block C führen.



Datenblöcke definieren

Datenblöcke werden im Data-Area-Editor definiert. Sie bauen die Block-Hierarchie auf, indem Sie angeben, welcher Block welchem anderen untergeordnet ist: geben Sie dazu den Namen des übergeordneten „Parent"-Blocks in das Kommentarfeld der Block-Definition ein.

In dem folgenden Beispiel sind `SUB-BLOCKB` und `SUB-BLOCKC` dem Block `MASTER-BLOCKA` untergeordnet; `SUB-BLOCKD` ist `SUB-BLOCKB` untergeordnet.

Die maximale Anzahl der Block-Stufen ist 8 (einschießlich 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
```

Um die spezifischen Blöcke einem Programm zur Verfügung zu stellen, benutzen Sie die folgende Syntax im `DEFINE DATA`-Statement:

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
```

Mit dieser Struktur kann Programm 1 die Daten in `MASTER-BLOCKA` gemeinsam mit Programm 2, Programm 3 oder Programm 4 benutzen. Allerdings können die Programme 2 und 3 nicht dieselben Data Areas von `SUB-BLOCKB` und `SUB-BLOCKC` gemeinsam benutzen, weil diese Datenblöcke auf derselben Struktur-Stufe definiert sind, und folglich denselben Speicherbereich belegen.

Block-Hierarchien

Besonders sorgfältig müssen Sie vorgehen, wenn Sie Datenblock-Hierarchien verwenden. Gehen wir von folgendem Szenario mit drei Programmen aus, bei denen eine Datenblock-Hierarchie verwendet wird:

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
```

Erläuterung:

- Programm 1 benutzt die Global Data Area `G-BLOCK` mit `MASTER-BLOCKA` und `SUB-BLOCKB`. Das Programm ändert ein Feld in `SUB-BLOCKB` und ruft Programm 2 mit einem `FETCH`-Statement auf, bei dem nur `MASTER-BLOCKA` in der Datendefinition angegeben ist.
- Programm 2 setzt `SUB-BLOCKB` zurück (löscht den Inhalt von `SUB-BLOCKB`). Der Grund dafür ist, dass ein Programm auf Stufe 1 (zum Beispiel ein mit einem `FETCH`-Statement aufgerufenes Programm) alle Datenblöcke zurücksetzt, die den Blöcken untergeordnet sind, die in seiner eigenen Datendefinition definiert sind.
- Programm 2 ruft jetzt mit einem `FETCH`-Statement Programm 3 auf, welches das in Programm 1 geänderte Feld anzeigen soll, aber es gibt einen leeren Bildschirm zurück.

Einzelheiten zu den Programmstufen siehe Abschnitt *Mehrere Stufen (Levels) aufgerufener Objekte*.

Parameter Data Area

Ein Subprogramm wird mit einem `CALLNAT`-Statement aufgerufen. Mit dem `CALLNAT`-Statement können Parameter von dem aufrufenden Objekt an das Subprogramm übergeben werden.

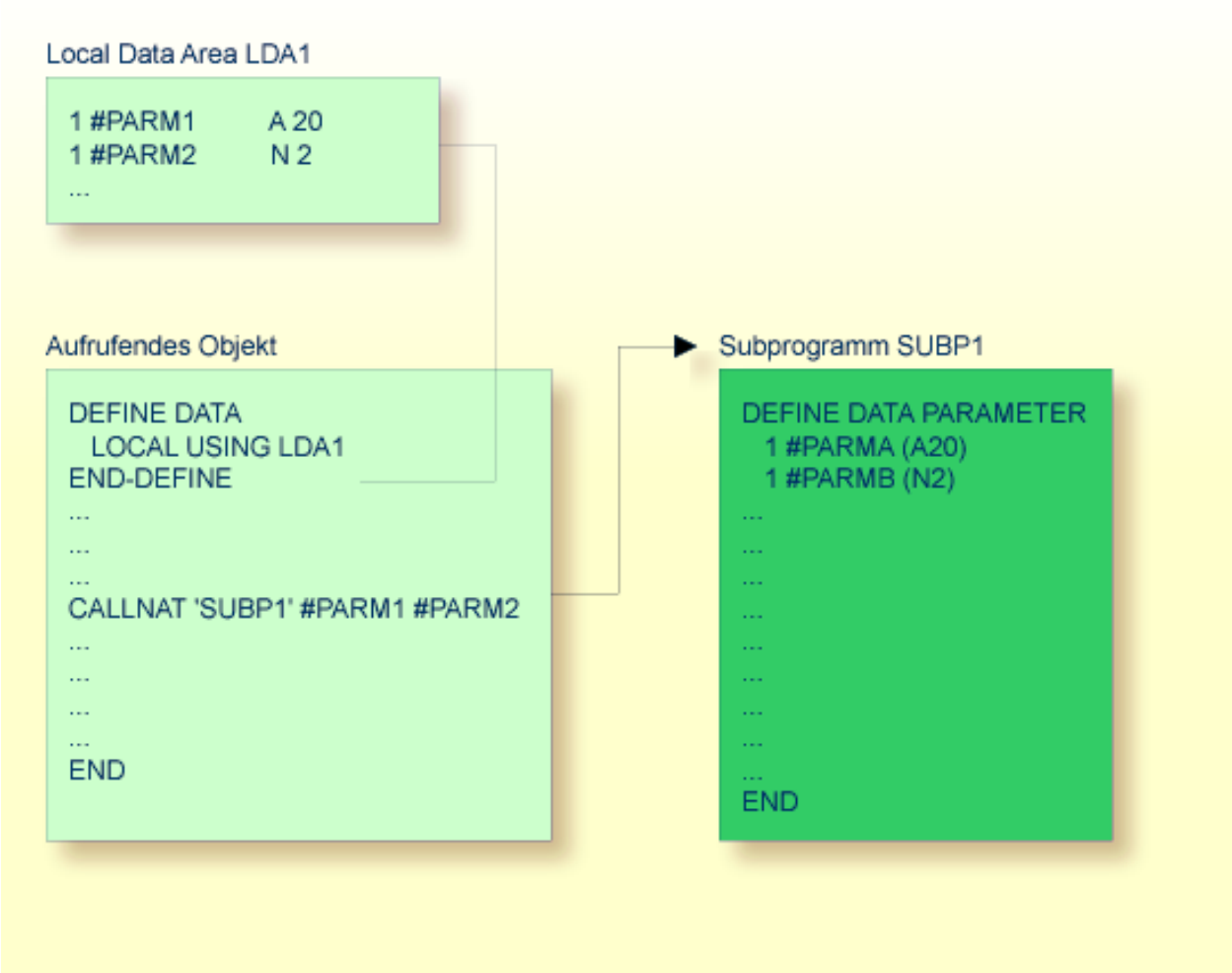
Diese Parameter müssen im Subprogramm mit einem `DEFINE DATA PARAMETER`-Statement definiert werden:

- entweder in der `PARAMETER`-Klausel des `DEFINE DATA`-Statements selbst
- oder in einer separaten Parameter Data Area (PDA), die von dem `DEFINE DATA PARAMETER`-Statement referenziert wird.

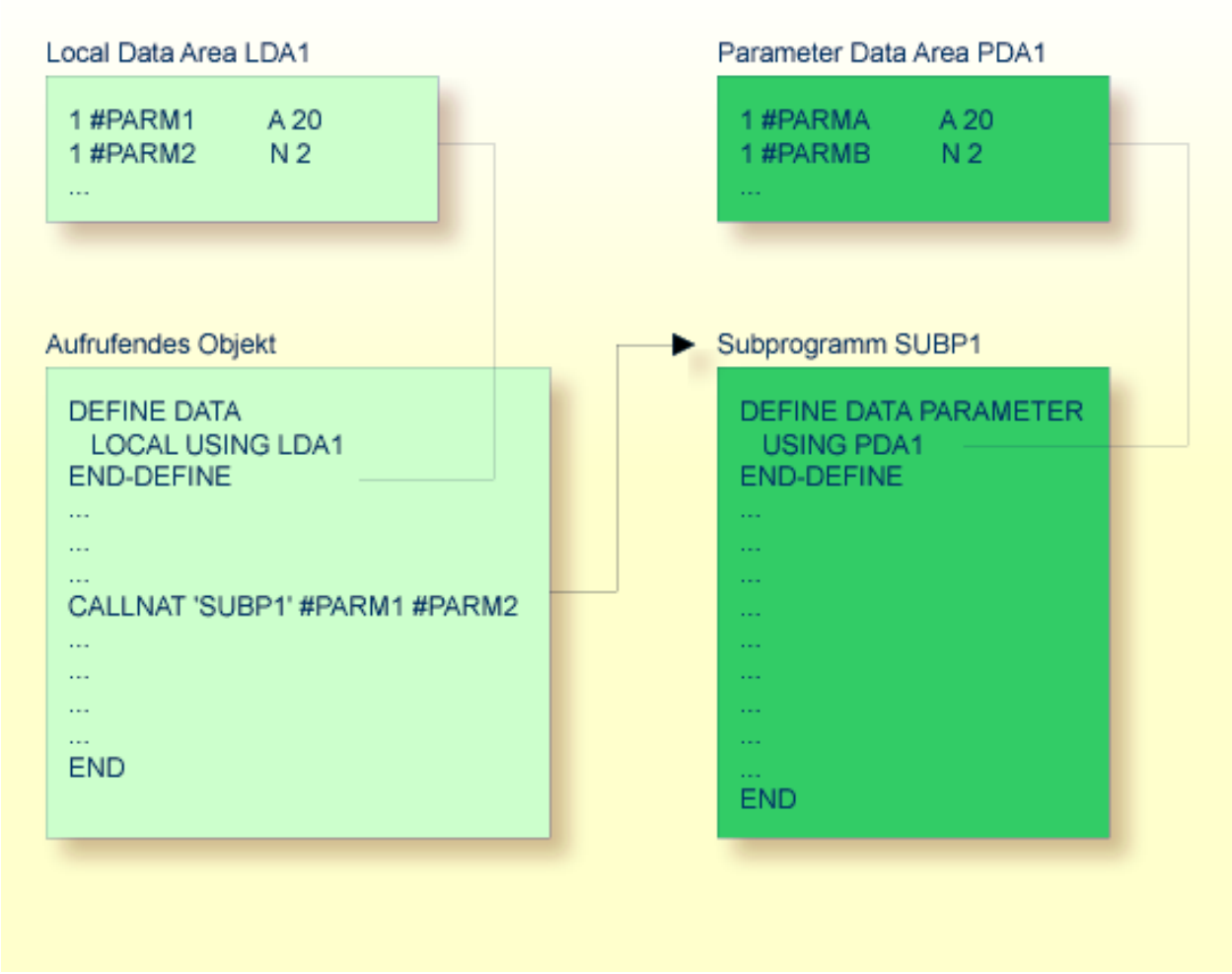
Folgende Themen werden behandelt:

- Übereinstimmende Formatangaben bei Array-Dimensionen

Innerhalb des DEFINE DATA PARAMETER-Statements definierte Parameter



Separat in einer Parameter Data Area definierte Parameter



In der gleichen Weise wie beim `CALLNAT`-Statement 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 müssen genau mit Reihenfolge, Format und Länge der Felder übereinstimmen, die im `DEFINE DATA PARAMETER`-Statement des aufgerufenen Subprogramms bzw. der aufgerufenen Subroutine definiert sind. 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).

Um zu garantieren, dass die im aufrufenden Programm benutzten Datenelement-Definitionen mit den im Subprogramm oder der externen Subroutine benutzten Datenelement-Definitionen identisch sind, können Sie eine PDA in einem `DEFINE DATA LOCAL USING`-Statement angeben. Indem Sie eine PDA als LDA verwenden, können Sie sich den zusätzlichen Aufwand der Erstellung einer LDA ersparen, die dieselbe Struktur wie die PDA hat.

Übereinstimmende Formatangaben bei Array-Dimensionen

Wenn Sie als Parameter ein Array übergeben, müssen Sie darauf achten, dass seine Dimensionen mit den Dimensionen übereinstimmen, die im `DEFINE DATA PARAMETER`-Statement des aufgerufenen Subprogramms bzw. der aufgerufenen Subroutine angegeben sind. Bei nicht übereinstimmenden Dimensionen entsteht selbst dann ein Fehler, wenn die Anzahl der Ausprägungen übereinstimmt.

Beispiel:

Aufgerufenes Subprogramm `SUB`:

```
DEFINE DATA PARAMETER
1 B (A5/1:5)
END-DEFINE
...
```

Aufrufendes Programm mit Compiler-Fehler NAT0937:

```
DEFINE DATA LOCAL
1 A (A5/1:1,1:5)
END-DEFINE
CALLNAT 'SUB' A(1,*)
...
```

Aufrufendes Programm ohne Compiler-Fehler:

```
DEFINE DATA LOCAL
1 A (A5/1:5)
END-DEFINE
CALLNAT 'SUB' A(*)
```

4 Datendefinitionsmodul (DDM)

Ein Datendefinitionsmodul (Data Definition Module/DDM) enthält die Beschreibung einer Datenbankdatei und der darin enthaltenen Felder. Natural benötigt diese Beschreibung, um aus einem Natural-Programm auf die in der Datei enthaltenen Daten zugreifen zu können.

Weitere Informationen siehe ***Datendefinitionsmodule (DDMs)*** und ***Natural und Datenbankzugriff***.

Verwandte Themen:

- ***Daten in einer Adabas-Datenbank aufrufen***
- *Zugriff auf eine Db2-Tabelle* in der *Datenbankmanagementsystem-Schnittstellen*-Dokumentation
- *Natural-Datendefinitionsmodule (DDMs) generieren* - SQL Services - in der *Datenbankmanagement-system-Schnittstellen*-Dokumentation
- *Protecting DDMs On Mainframes* in der *Natural Security*-Dokumentation

5 Programme und untergeordnete Routinen

Dieses Dokument behandelt die Objekttypen, die als Routinen, d.h. als untergeordnete Programme aufgerufen werden können.

Anmerkung: Obwohl sie ebenfalls von anderen Objekten aufgerufen werden, sind Helproutinen und Maps (Masken) genau genommen keine Routinen. Sie werden deshalb in getrennten Dokumenten beschrieben; siehe Abschnitt ***Helproutine*** und ***Map***.

Modulare Anwendungsstruktur

Eine typische Natural-Anwendung besteht nicht aus einem einzigen großen Programm, sondern ist in mehrere Objekt-Module aufgeteilt. Jedes dieser Objekte stellt eine funktionale Einheit von überschaubarer Größe dar, und jedes Objekt ist mit den anderen Objekten der Anwendung auf eine klar definierte Weise verbunden. Dadurch ergibt sich eine übersichtlich strukturierte Anwendung, was die Entwicklung und spätere 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 objekt-orientierte Struktur einer Anwendung äußerst komplex und vielschichtig werden.

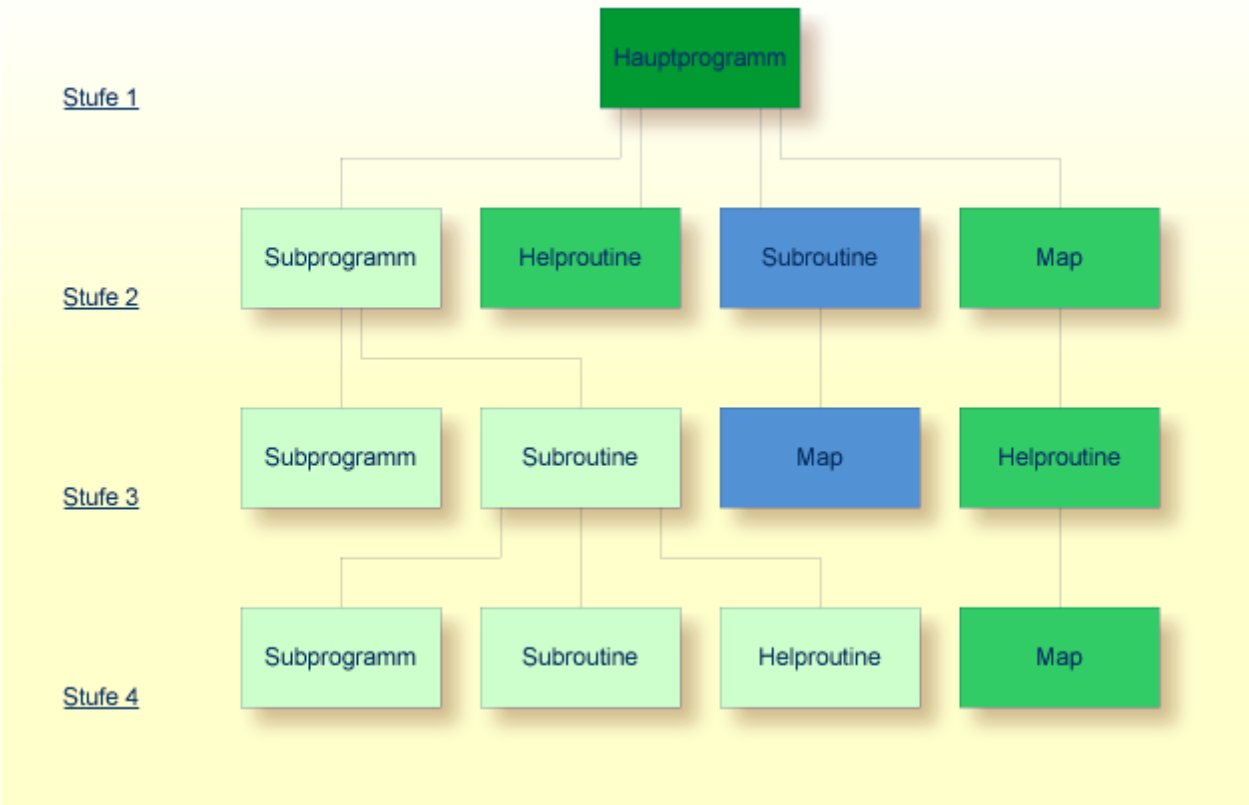
Mehrere Stufen (Levels) aufgerufener Objekte

Ein aufgerufenes Objekt ist jeweils eine Stufe (Level) unter dem Objekt, von dem es aufgerufen wurde; d.h. bei 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 aus einem anderen Objekt über ein `FETCH`-Statement ein Programm aufgerufen, so wird es als Hauptprogramm eingestuft und auf Stufe 1 eingeordnet. Ein Programm, das mit `FETCH RETURN` aufgerufen wird, wird dagegen als untergeordnetes Programm eingestuft und ist eine Stufe unter dem Objekt, von dem es aufgerufen wurde.

Die folgende Abbildung enthält ein Beispiel für mehrere Stufen aufgerufener Objekte und zeigt, wie diese Stufen gezählt werden:



Sie können die Systemvariable `*LEVEL` benutzen (siehe *Systemvariablen*-Dokumentation), um die Level-Nummer des Objekts zu erfahren, das gerade ausgeführt wird.

Verarbeitungsfluss beim Aufruf eines untergeordneten Programms

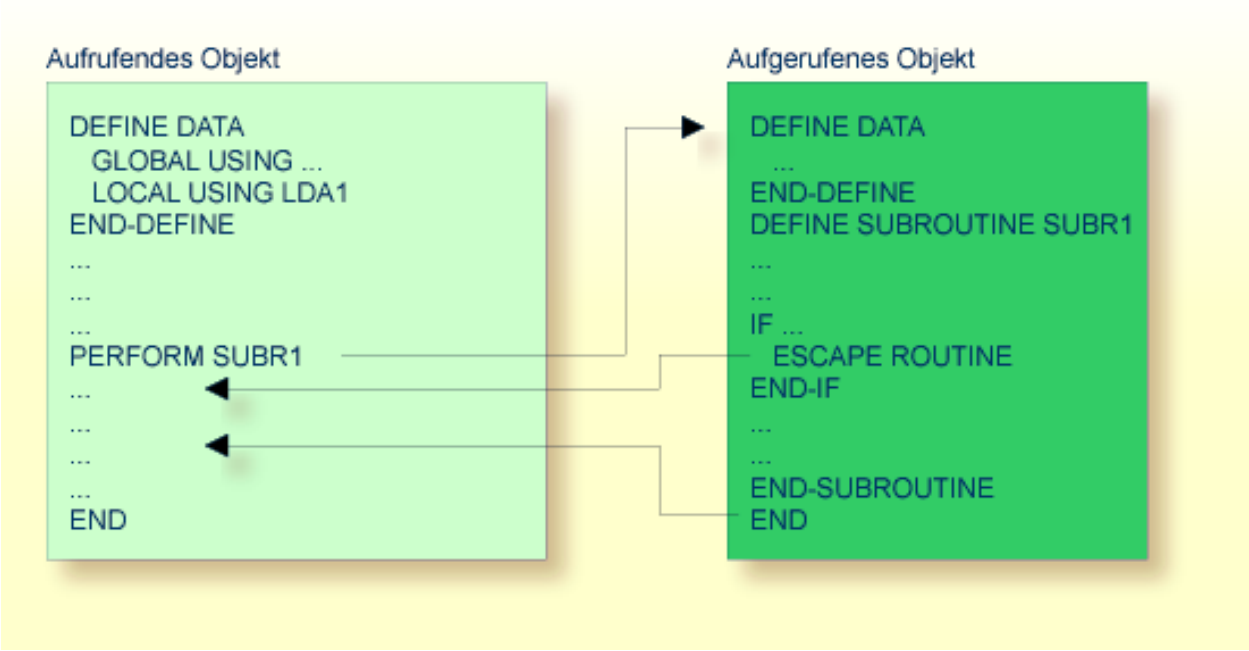
Wenn ein `CALLNAT`-, `PERFORM`- oder `FETCH RETURN`-Statement oder ein Function Call, das eine untergeordnete Routine — ein Subprogramm, eine externe Subroutine, ein Programm bzw. eine Function — aufruft, ausgeführt wird, wird die Ausführung des aufrufenden Objekts unterbrochen, und die Ausführung der untergeordneten Routine beginnt.

Die Ausführung der untergeordneten Routine wird solange fortgesetzt, bis entweder ihr `END`-Statement erreicht ist oder die Verarbeitung der untergeordneten Routine durch die Ausführung eines `ESCAPE ROUTINE`- oder `ESCAPE MODULE`-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 die untergeordnete Routine aufgerufen wurde, fortgesetzt.

Im Falle eines Function Call wird die Verarbeitung des aufrufenden Objekts mit dem Statement fortgesetzt, das den Function Call enthält.

Beispiel:



Programm

Ein Programm kann selbständig ausgeführt — und getestet — werden.

- Um ein Source-Programm zu katalogisieren (kompilieren) und anschließend auszuführen, verwenden Sie das Systemkommando `RUN`.
- Um ein Programm auszuführen, das bereits als katalogisiertes Objekt existiert, verwenden Sie das Systemkommando `EXECUTE`.

Ein Programm kann auch von einem anderen Objekt mit einem `FETCH`- oder `FETCH RETURN`-Statement aufgerufen werden. Das aufrufende Objekt kann ein anderes Programm, eine **Subroutine**, ein **Subprogramm**, eine **Function**, eine **Helproutine** oder eine Verarbeitungsregel in einer Map sein.

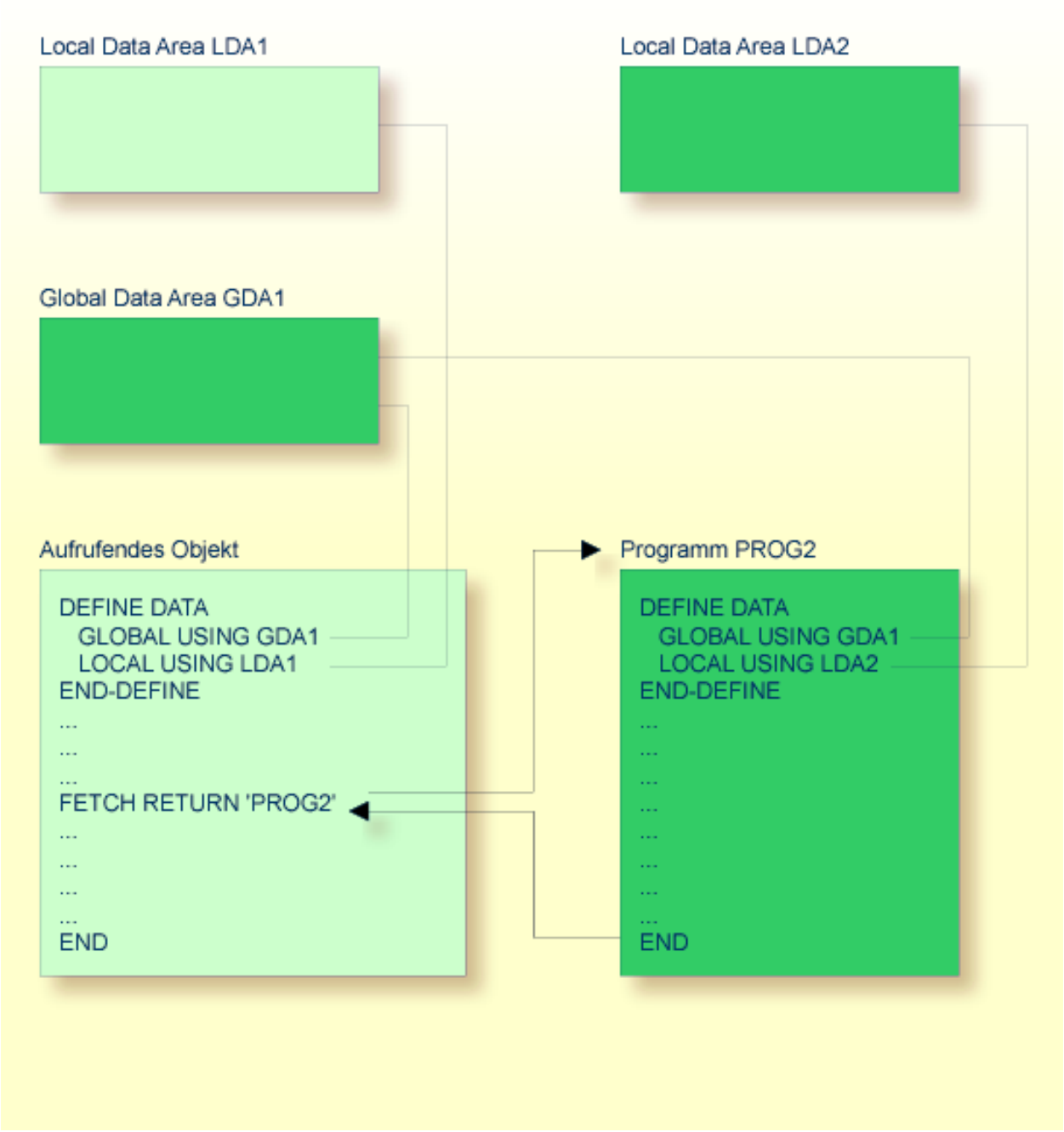
- Wenn ein Programm mit `FETCH RETURN`-Statement aufgerufen wird, wird die Ausführung des aufrufenden Objekts unterbrochen — nicht beendet —, und das aufgerufene Programm wird als *untergeordnetes Programm* aktiviert. Wenn die Ausführung des aufgerufenen Programms beendet ist, wird das aufrufende Objekt 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.

Die folgenden Themen werden nachfolgend erörtert:

- Mit FETCH aufgerufenes Programm

Mit FETCH RETURN aufgerufenes Programm

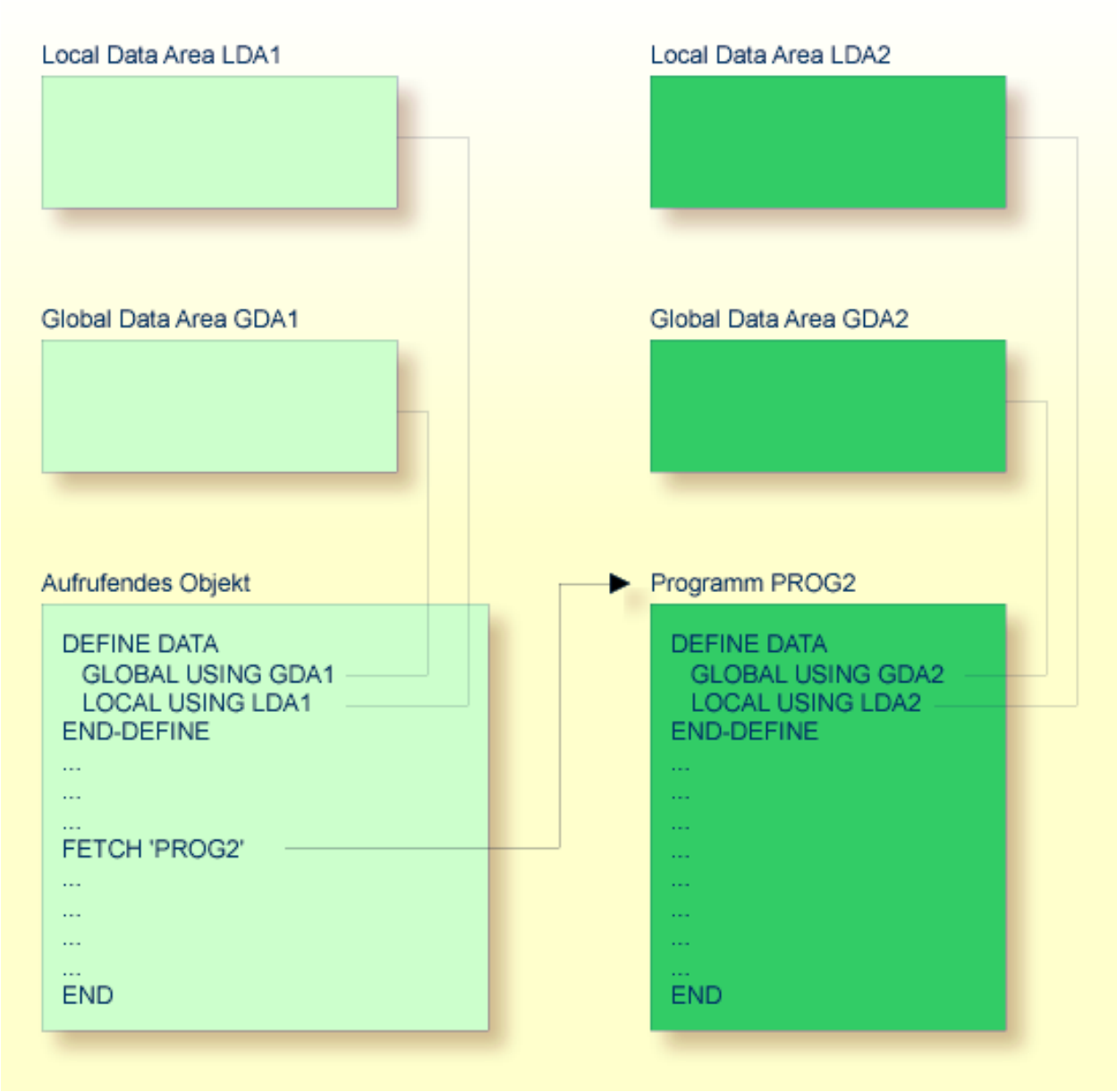


Ein mit `FETCH RETURN` aufgerufenes Programm kann auf die vom aufrufenden Objekt benutzte **Global Data Area** (GDA) zugreifen.

Darüber hinaus kann jedes Programm seine eigene **Local Data Area** (LDA) haben, in der die nur in diesem Programm verwendeten Felder definiert sind. Außerdem kann ein Programm auf anwendungsunabhängige Variable (AIVs) zugreifen. Weitere Informationen siehe *Definition von anwendungsunabhängigen Variablen* in der *Statements*-Dokumentation.

Ein mit `FETCH RETURN` aufgerufenes Programm kann jedoch keine eigene Global Data Area (GDA) 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 der *Statements*-Dokumentation.

Subroutine

In der Regel wird eine Subroutine verwendet, um Funktionalität zu implementieren, die von verschiedenen Objekten in einer Anwendung benutzt wird.

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* („inline subroutine“) oder eine *externe Subroutine* sein:

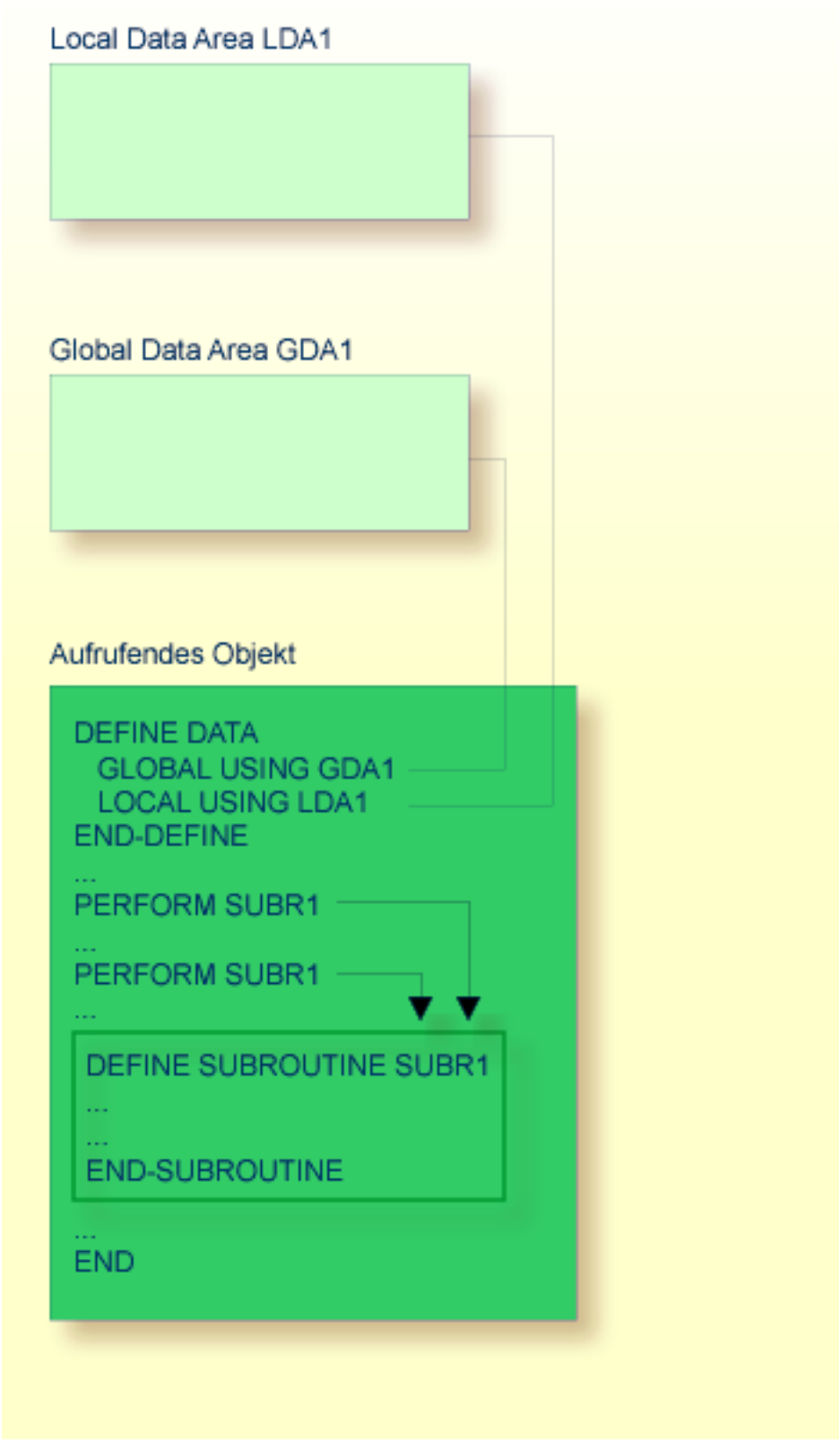
- **Interne Subroutine**
  Eine interne Subroutine wird innerhalb des Objekts, welches das sie aufrufende `PERFORM`-Statement enthält, definiert.
- **Externe Subroutine**
  Eine externe Subroutine wird als separates Objekt — vom Typ Subroutine — außerhalb des Objektes, welches sie aufruft, definiert.

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.

DieseThemen werden nachfolgend erörtert:

- Daten, die einer externen Subroutine zur Verfügung stehen

Interne Subroutine

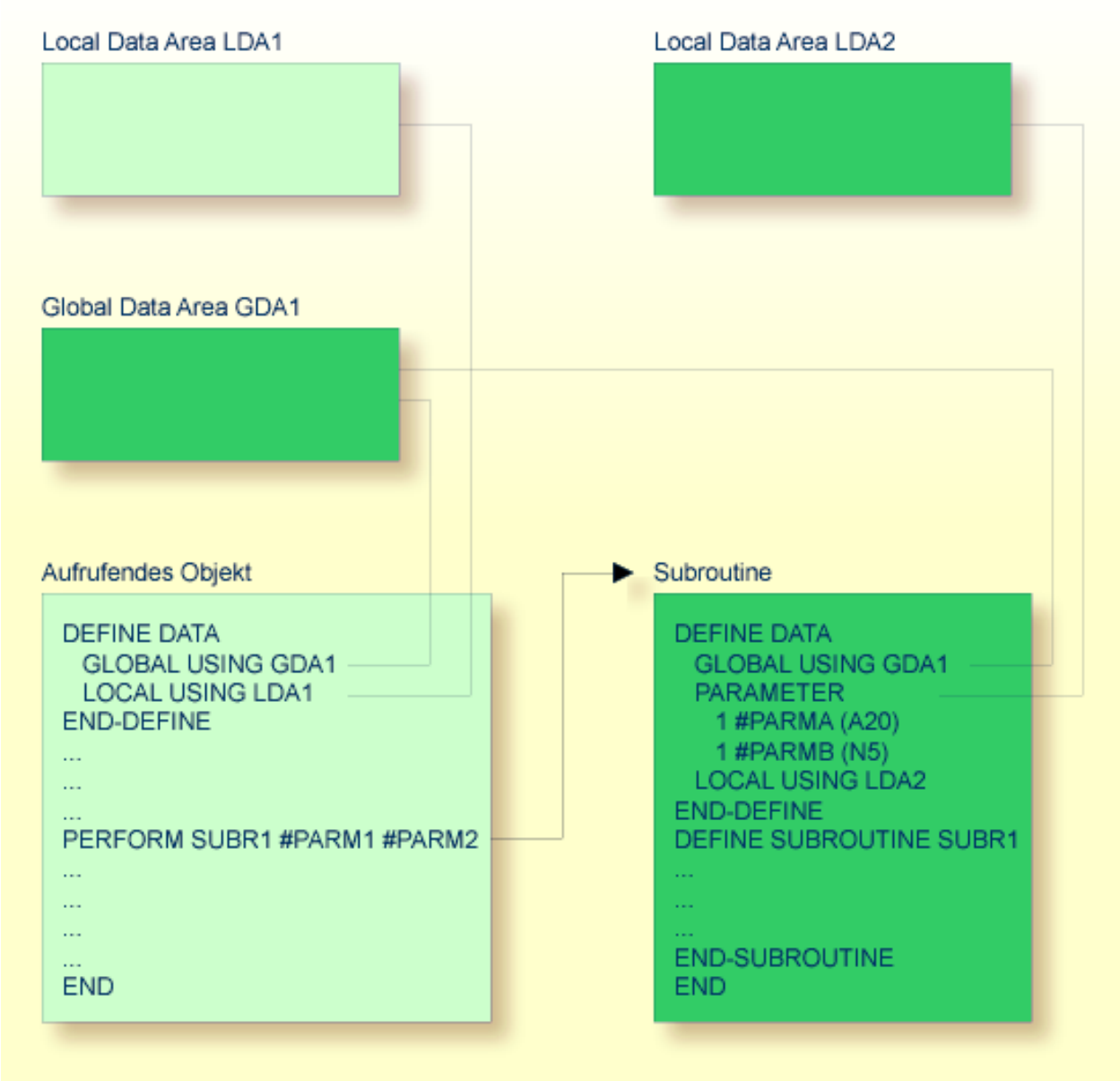

Local Data Area LDA1
Global Data Area GDA1
Aufrufendes Objekt
DEFINE DATA
GLOBAL USING GDA1
LOCAL USING LDA1
END-DEFINE
...
PERFORM SUBR1
...
PERFORM SUBR1
...
DEFINE SUBROUTINE SUBR1
...
...
END-SUBROUTINE
...
END

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

Daten, die einer internen Subroutine zur Verfügung stehen

Eine interne Subroutine hat Zugriff auf alle Datenfelder innerhalb des Objekts, in dem sie enthalten ist.

Externe Subroutine



Eine externe Subroutine — also ein *Objekt* des Typs *Subroutine* — kann nicht selbständig ausgeführt werden. Sie muss von einem anderen Objekt aufgerufen werden. Das aufrufende Objekt kann ein Programm, eine Function, ein Subprogramm, eine Subroutine, eine Helproutine oder eine Verarbeitungsregel in einer Map sein.

Daten, die einer externen Subroutine zur Verfügung stehen

Eine externe Subroutine kann auf die **Global Data Area** (GDA) zugreifen, die vom aufrufenden Objekt benutzt wird.

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 benutzen **Parameter Data Area** (PDA) definiert werden.

Außerdem kann eine externe Subroutine eine eigene **Local Data Area** (LDA) haben, in der die Felder definiert sind, die nur innerhalb der Subroutine verwendet werden sollen. Eine externe Subroutine kann jedoch keine eigene **Global Data Area** (GDA) haben.

Außerdem kann eine externe Subroutine auf anwendungsunabhängige Variable (AIVs) zugreifen. Weitere Informationen siehe *Definition von anwendungsunabhängigen Variablen* in der *Statements*-Dokumentation.

Subprogramm

In der Regel wird ein Subprogramm verwendet, um Funktionalität zu implementieren, die von verschiedenen Objekten in einer Anwendung benutzt werden.

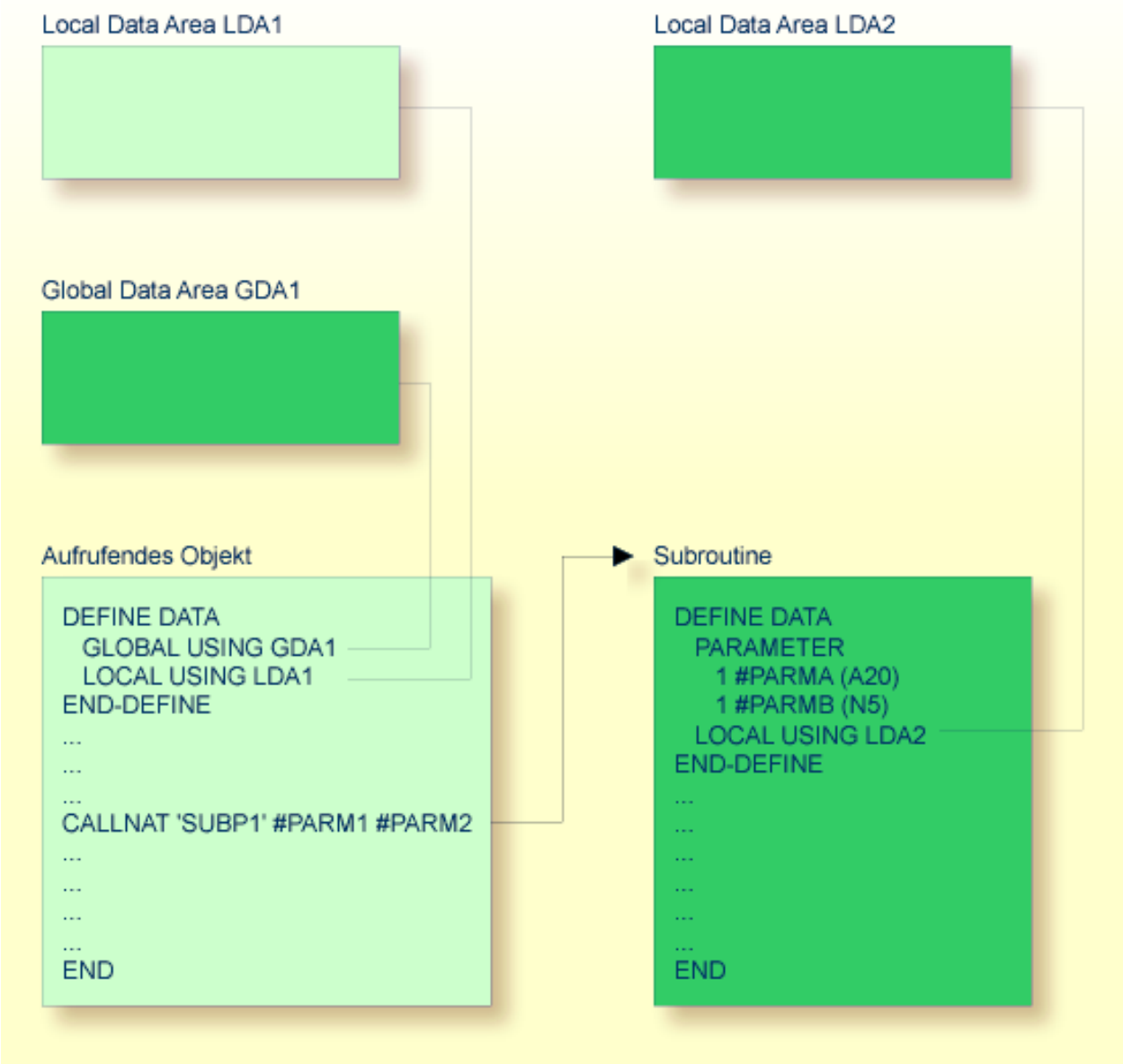
Ein Subprogramm kann nicht selbständig ausgeführt werden. Es muss von einem anderen Objekt aufgerufen werden. Das aufrufende Objekt kann ein Programm, eine Function, ein Subprogramm, eine Subroutine oder eine Helproutine sein.

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.

Daten, die 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 benutzen **Parameter Data Area** (PDA) definiert werden.



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

Wenn ein Subprogramm seinerseits eine Subroutine oder Helproutine aufruft, kann es eine eigene **Global Data Area** (GDA) haben und diese gemeinsam mit der Subroutine bzw. Helproutine nutzen.

Außerdem kann ein Subprogramm auf anwendungsunabhängige Variable (AIVs) zugreifen. Weitere Informationen siehe *Definition von anwendungsunabhängigen Variablen* in der *Statements*-Dokumentation.

Function

In der Regel wird eine Function verwendet, um Funktionalität zu implementieren, die von verschiedenen Objekten in einer Anwendung benutzt werden.

Im Unterschied zu den in Natural eingebauten Systemfunktionen dient eine Function dazu, benutzerdefinierte Funktionalität zur Verfügung zu stellen.

Die Function liefert einen Ergebniswert zurück, der von dem aufrufenden Objekt verwendet wird. Der Ergebniswert wird anhand der Daten errechnet, die der Function zur Verfügung stehen.

Ein Objekt des Typs Function enthält ein `DEFINE FUNCTION`-Statement für die Definition einer einzelnen Funktion und das zugehörige `END`-Statement.

Die Function selbst wird durch einen **Function Call** aufgerufen.

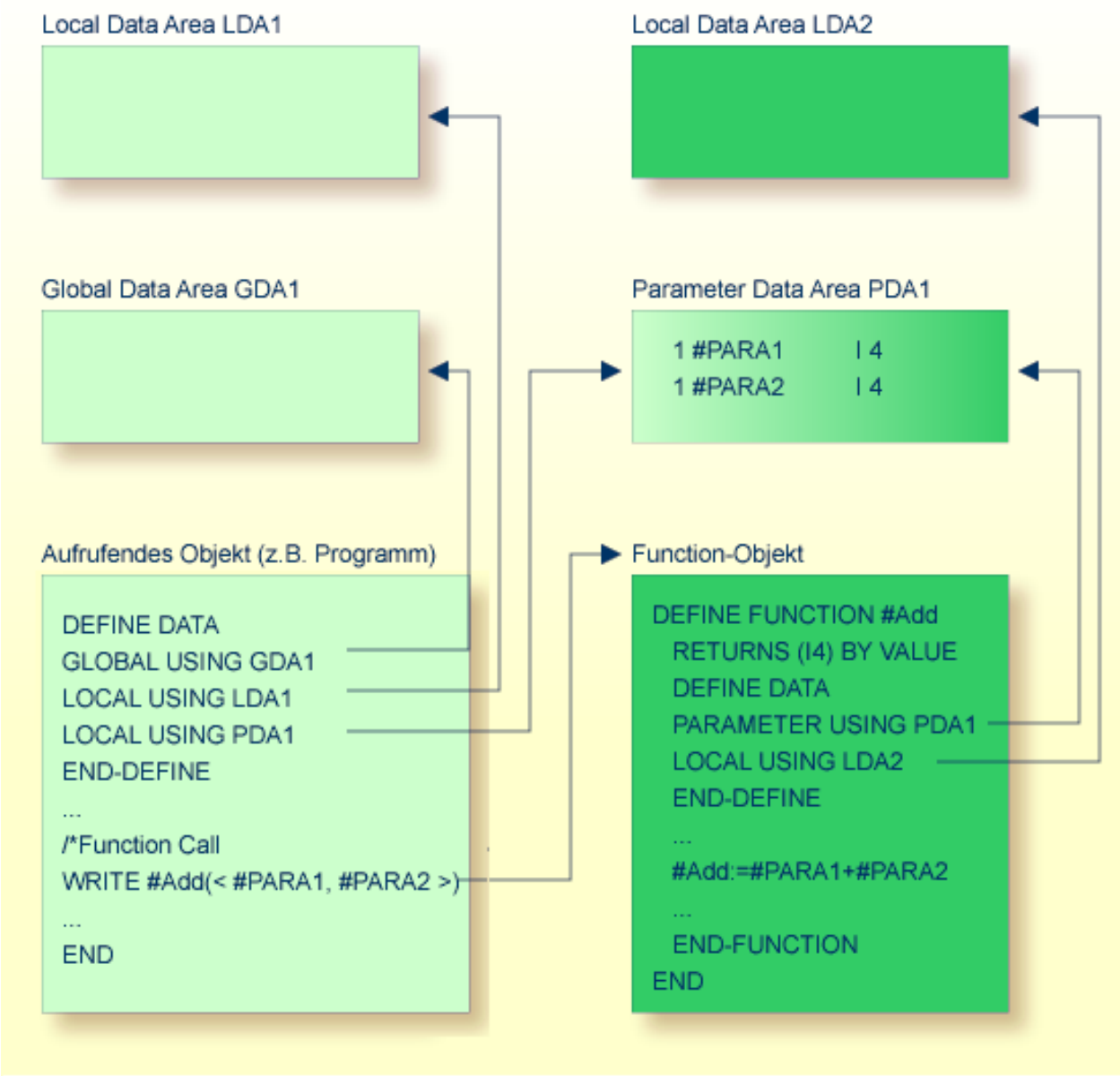
Daten, die einer Function zur Verfügung stehen

Mit dem Function Call können Parameter vom aufrufenden Objekt an die Function übergeben werden. Diese Parameter sind die einzigen Daten, die der Function vom aufrufenden Objekt zur Verfügung stehen. Sie müssen im `DEFINE FUNCTION`-Statement definiert werden.

Zusätzlich kann eine Function eine eigene **Local Data Area** (LDA) haben, in der die Felder definiert sind, die innerhalb der Function verwendet werden sollen. Eine Function kann jedoch keine eigene **Global Data Area** (GDA) haben.

Außerdem kann eine Function auf anwendungsunabhängige Variablen (AIVs) zugreifen. Weitere Informationen siehe *Definition von anwendungsunabhängigen Variablen* in der *Statements*-Dokumentation.

Falls erforderlich, können Sie Ergebnis- und Parameterlayouts für das Objekt definieren, das die Function aufruft. Dazu dient das `DEFINE PROTOTYPE`-Statement.



Weitere Informationen siehe ***Function Call***.

Vergleich zwischen externer Subroutine, Subprogramm und Function

In diesem Abschnitt werden die Merkmale der externen Subroutine, des Subprogramms und der Function zusammengefasst und miteinander verglichen.

Folgende Merkmale sind bei allen vorhanden:

- Der Programmcode, der die Logik der Routine bildet, befindet sich in einem separaten Objekt, das in einer Natural-Library gespeichert wird.
- Parameter werden in dem Objekt mittels eines `DEFINE DATA PARAMETER`-Statement definiert.

Die Unterschiede zwischen externer Subroutine, Subprogramm und Function werden in der folgenden Tabelle dargestellt:

Betrachtungsgegenstand	Externe Subroutine	Subprogramm	Function
Maximale Länge des Namens	32 Zeichen	8 Zeichen	32 Zeichen
Verwendung einer **Global Data Area** (GDA)	Nutzt eine GDA gemeinsam mit dem Aufrufer.	Legt eine **Instanz** einer GDA an.	Eine GDA ist nicht zulässig.
Prüfung von Format/Länge der übergebenen Paramter gegen die Definition im aufgerufen Objekt bei der Kompilierung	Prüfung erfolgt nur, wenn die Compiler Option `PCHECK` auf `ON` gesetzt ist.	Prüfung erfolgt nur, wenn die Compiler Option `PCHECK` auf `ON` gesetzt ist.	Prüfung erfolgt nur, wenn zur Kompilierungszeit ein katalogisiertes Function-Objekt existiert.
Aufruf durch	Aufruf durch `PERFORM`-Statement	Aufruf durch `CALLNAT`-Statement	Aufruf durch einen **Function Call** Ein Function Call kann in Statements anstelle von schreibgeschützten Operanden benutzt werden. Ein Function Call kann auch als Statement benutzt werden.
Bestimmung des aufzurufenden Objekts zur Kompilierungs-/Laufzeit	Bestimmung zur Kompilierungszeit	Bestimmung zur Kompilierungs- bzw. Ausführungszeit, je nach dem Operanden, der beim `CALLNAT`-Statement verwendet wird.	Bestimmung zur Kompilierungs- bzw. Ausführungszeit, je nach dem Operanden, der beim Function Call verwendet wird.
Nutzung eines Ergebniswertes in einem Statement	Ein Ergebniswert muss einem Parameter	Ein Ergebniswert muss einem Parameter	Das Ergebnis eines Function Call wird als Operand in dem

Betrachtungsgegenstand	Externe Subroutine	Subprogramm	Function
	zugewiesen werden, der als Operand in einem Statement verwendet werden soll.	zugewiesen werden, der als Operand in einem Statement verwendet werden soll.	Statement verwendet, das den Function Call enthält.

In den nachfolgenden Beispielen erfolgt ein Vergleich zwischen einem Function Call und dem Aufruf eines Subprogramms.

- Beispiel für einen Function Call
- Beispiel für den Aufruf eines Subprogramms

Beispiel für einen Function Call

Das folgende Beispiel zeigt ein Programm, das eine Function aufruft, und die Definition der Function, die mit einem `DEFINE FUNCTION` erstellt wird.

Programm, das die Function aufruft

```
** Example 'FUNCAX01': Calling a function  (Program)
************************************************************************
*
WRITE 'Function call' F#ADD(< 2,3 >)  /* Function call.
                                      /* No temporary variables needed.
*
END
```

Definition der Function F#ADD

```
** Example 'FUNCAX02': Calling a function  (Function)
************************************************************************
DEFINE FUNCTION F#ADD
  RETURNS #RESULT (I4)
  DEFINE DATA PARAMETER
    1 #SUMMAND1 (I4) BY VALUE
    1 #SUMMAND2 (I4) BY VALUE
  END-DEFINE
  /*
  #RESULT := #SUMMAND1 + #SUMMAND2
  /*
END-FUNCTION
*
END ↩
```

Beispiel für den Aufruf eines Subprogramms

Um die Funktionalität, die im obgigen Beispiel für den Function Call gezeigt wird, stattdessen mittels eines Subprogramm-Aufrufs zu implementieren, müssen Sie temporäre Variablen angeben.

Programm, das das Subprogramm aufruft

Das folgende Beispiel zeigt ein Programm, das ein Subprogramm unter Verwendung einer temporären Variablen aufruft.

```
** Example 'FUNCAX03': Calling a subprogram  (Program)
************************************************************************
DEFINE DATA LOCAL
  1 #RESULT (I4) INIT <0>
END-DEFINE
*
CALLNAT 'FUNCAX04' #RESULT 2 3    /* Result is stored in #RESULT.
*
WRITE '=' #RESULT                 /* Print out the result of the
                                  /* subprogram.
*
END
```

Aufgerufenes Subprogramm FUNCAX04

```
** Example 'FUNCAX04': Calling a subprogram  (Subprogram)
************************************************************************
DEFINE DATA PARAMETER
  1 #RESULT   (I4) BY VALUE RESULT
  1 #SUMMAND1 (I4) BY VALUE
  1 #SUMMAND2 (I4) BY VALUE
END-DEFINE
*
#RESULT := #SUMMAND1 + #SUMMAND2
*
END ↩
```

6 Helproutine

Helproutinen haben besondere Eigenschaften, die die Verarbeitung von Hilfe-Anforderungen erleichtern. Sie ermöglichen den Aufbau komplexer interaktiver Hilfe-Systeme. Helproutinen werden mit dem Programm-Editor erstellt.

Helproutinen aufrufen

In einer Natural-Anwendung fordern die Benutzer Hilfe an, indem Sie das Hilfe-Zeichen — standardmäßig ein Fragezeichen (?) — in einem Feld eingeben oder die Hilfe-Taste (normalerweise PF1) drücken. Dadurch wird eine Helproutine aufgerufen. Dabei gilt Folgendes:

- Das Hilfe-Zeichen ist nur einmal einzugeben.
- Das Hilfe-Zeichen muss das einzige geänderte Zeichen in der Eingabe-Zeichenkette sein.
- Das Hilfe-Zeichen muss das erste Zeichen in der Eingabe-Zeichenkette sein.

Auch in numerischen Feldern kann das Hilfe-Zeichen verwendet werden, wenn für das betreffende Feld eine Helproutine definiert ist. (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 **Programm** 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 Maskenebene 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 Maskenebene angegebene Helproutine aufgerufen.

Eine Helproutine kann auch über ein REINPUT USING HELP-Statement aufgerufen werden — entweder im Programm selbst oder in einer Verarbeitungsregel („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 Bereich Erweiterte Feld-Bearbeitung des Masken-Editors (wie unter ***Maps erstellen*** und in der *Editoren*-Dokumentation 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- oder BACKOUT TRANSACTION-Statement in einer Helproutine die Transaktionslogik des Hauptprogramms beeinflusst.

Parameter-Übergabe an Helproutinen

Eine Helproutine kann auf die zurzeit aktive **Global Data Area** zugreifen (kann aber keine eigene Global Data Area haben). Außerdem kann sie eine eigene **Local Data Area** (LDA) haben.

Darüber hinaus können Daten von der/an die Helproutine mittels Parametern übergeben werden. Eine Helproutine kann bis zu 20 explizite Parameter und einen impliziten Parameter haben. Die expliziten Parameter werden mit dem Operanden HE nach 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` ein expliziter Parameter ist und `#A` der implizite Parameter/das Feld.

Dies wird im `DEFINE DATA PARAMETER`-Statement der Helproutine wie folgt definiert:

```
DEFINE DATA PARAMETER
1 #PARM1 (A3)          /* explicit parameter
1 #PARM2 (A5)          /* implicit parameter
END-DEFINE
```

Bitte beachten Sie, dass im obigen Beispiel der implizite Parameter `#PARM2` 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.

Wird Hilfe angefordert, wird die Helproutine aufgerufen, bevor Daten vom Bildschirm an die Datenbereiche des Programms weitergegeben werden. Das bedeutet, dass Helproutinen nicht auf Daten zugreifen können, die während derselben Bildschirmtransaktion eingegeben wurden.

Nach Beenden der Hilfeverarbeitung 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 einschließlich dem Feld, für das Hilfe angefordert wurde.

Ausnahme: Wenn das Feld, für das Hilfe angefordert wurde, durch dynamische Attribute (Session-Parameter `DY`) in mehrere Teile unterteilt wird und wenn der Teil, in den das Fragezeichen eingegeben wurde, sich *nach* einem vom Benutzer veränderten Teil befindet, wird der Feldinhalt nicht durch die Helproutine verändert.

Attributkontrollvariablen werden nach der Verarbeitung 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 (Format/Länge: A65) verarbeitet, welches den Namen des Feldes (oder, bei Angabe auf Map-Ebene, der Map,) enthält. Die entsprechende Helproutine würde beispielsweise folgendermaßen beginnen:

```
DEFINE DATA PARAMETER
1 FNAME (A65)             /* contains 'PERSONNEL-NUMBER'
1 FVALUE (N8)             /* value of field (optional)
END-DEFINE
```

Diese Option kann benutzt werden, um 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)            /* contains 'A'
1 FVALUE  (N8)             /* value of selected element
1 FINDEX1 (I2)             /* 1st dimension index
1 FINDEX2 (I2)             /* 2nd dimension index
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, zum Beispiel so:

```
********************************************************************************
                          PERSONNEL INFORMATION
PLEASE ENTER NAME: ?__________________
PLEASE ENTER CITY: ___________________
                  +--------------------------+
                  !                          !
                  ! 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 Helproutine haben Sie folgende Möglichkeiten, die Größe eines Fensters zu bestimmen:

- in einem `FORMAT`-Statement (z.B. um die Zeilen- und Seitenlänge anzugeben: `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, errechnet. Natural plaziert das Fenster möglichst nahe an das Feld, ohne es zu überdecken. Mit dem `DEFINE WINDOW`-Statement können Sie diese automatische Positionierung umgehen und die Position des Fensters auch selbst bestimmen.

Weitere Informationen über die Verarbeitung von Bildschirmfenstern finden Sie beim `DEFINE WINDOW`-Statement in der *Statements*-Dokumentation und beim Terminalkommando `%W` in der *Terminalkommando*-Dokumentation.

7 Copycode

Dieses Dokument beschreibt die Vorteile der Nutzung und Verarbeitung von Copycode.

Copycode-Nutzung

Ein Objekt des Typs Copycode enthält ein Stück Quellcode, das mit einem `INCLUDE`-Statement in ein anderes Objekt eingefügt werden kann.

Wenn Sie zum Beispiel 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.

Copycode-Verarbeitung

Der Copycode wird bei der Kompilierung eingefügt; d.h. die Quellcode-Zeilen des Copycode werden nicht physisch in den Quellcode 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 Quellcode eines Copycode verändern, müssen Sie folglich auch alle Objekte, in denen dieser Copycode verwendet wird, mit dem Systemkommando `CATALOG` oder mit `CATALL` katalogisieren.

Achtung:

- 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.
- Ein `END`-Statement darf nicht in einem Copycode untergebracht werden.

Weitere Informationen zu Copycode finden Sie in der Beschreibung des `INCLUDE`-Statements in der *Statements*-Dokumentation.

8 Text

Mit dem Natural-Objekttyp „Text“ können Sie Texte (keine Programme) erstellen.

Verwendung des Natural-Objekttyps Text

Sie können diesen Objekttyp benutzen, um eine Dokumentation für Natural-Objekte zu erstellen, die wesentlich ausführlicher sein kann, als es z.B. durch Kommentare innerhalb des Quellcodes eines Objektes möglich wäre.

Der Objekttyp Text kann auch hilfreich sein, wenn Ihnen Predict nicht zur Dokumentation von Objekten zur Verfügung steht.

Text schreiben

Den Text schreiben Sie im Natural-Programm-Editor.

Der einzige Unterschied zur Programmerstellung liegt darin, dass keine Umsetzung von Klein- in Großbuchstaben vorgenommen wird, d.h. der Text, den Sie schreiben, bleibt unverändert.

Sie können Leerzeilen unterdrücken, indem Sie die Option **Empty Line Suppression for Text** in Ihrem Editorprofil auf `Y` setzen. Siehe auch *Editor-Standardwerte* und *Allgemeine Standardwerte* in der *Editoren*-Dokumentation.

Sie können einen beliebigen Text schreiben (eine Syntax-Prüfung gibt es nicht).

Textobjekte können nur in Source-Form (mit dem Systemkommando `SAVE`) gespeichert werden, aber nicht in Objektform (mit dem Systemkommando `STOW`).

Ein Textobjekt kann nicht mit `RUN` ausgeführt, sondern lediglich im Editor angezeigt werden.

9 Class

Der Natural-Objekttyp Class wird verwendet, um einen objektbasierten Programmierstil anzuwenden.

Weitere Informationen siehe ***NaturalX*** im *Leitfaden zur Programmierung*.

10 Map

Als Alternative zur dynamischen Spezifikation von Bildschirmmasken bietet das `INPUT`-Statement die Möglichkeit, vordefinierte Bildschirmmasken zu benutzen, und verwendet den Natural-Objekttyp Map („Maske“).

Vorteile der Verwendung von Maps

Die Benutzung von vordefinierten Bildschirmmasken im Gegensatz zu dynamischen Bildschirmmasken- Spezifikationen bietet verschiedene Vorteile, zum Beispiel:

- Klar strukturierte Anwendungen als Ergebnis einer konsequenten Trennung von Programm-Logik und Anzeige-Logik.
- Bildschirmmasken-Änderungen sind möglich, ohne Änderungen an Hauptprogrammen vornehmen zu müssen.
- Die Sprache der Benutzerschnittstelle einer Anwendung kann leicht für internationale oder lokale Anforderungen angepasst werden.

Der Vorteil einer modularen Anwendungsstruktur mit Objekten, wie z.B. Maps, wird spätestens bei der Pflege von vorhandenen Natural-Anwendungen offenkundig.

Map-Typen

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

Es gibt folgende Map-Typen (Maskenarten):

- **Eingabemaske (Input Map)**
  Der Dialog mit dem Benutzer erfolgt über Eingabemasken.
- **Ausgabemaske (Output Map)**
  Wenn eine Anwendung einen Ausgabe-Report erzeugt, kann dieser Report mittels einer Ausgabemaske auf dem Bildschirm angezeigt werden.
- **Hilfemaske (Help Map)**
  Hilfemasken sind im Prinzip wie andere Maps, aber wenn sie als Hilfe zugewiesen werden, werden zusätzliche Prüfungen vorgenommen, um ihre Verwendbarkeit für Hilfe-Zwecke zu gewährleisten.

Der Objekttyp Map hat folgende Bestandteile:

- den Map-Hauptteil, in dem die Bildschirmmaske definiert ist, und
- eine zugehörige **Parameter Data Area** (PDA), die als eine Art Schnittstelle Daten-Definitionen enthält, wie z.B. Name, **Format**, Länge jedes in einer spezifischen Map dargestellten Feldes.

Verwandte Themen:

- Informationen zu Auswahlboxen, die an Eingabefelder angehängt werden können, siehe Abschnitt *SB – Auswahlfenster (Selection Box)* in der *Statements*-Dokumentation (`INPUT`-Statement) und im Abschnitt *SB – Auswahlfenster (Selection Box)* in der *Parameter-Referenz*-Dokumentation.
- Informationen zu horizontal geteilten Masken („Split-Screen-Maps“), bei denen der obere Teil als Ausgabemaske und der untere Teil als Eingabemaske benutzt werden kann, entnehmen Sie dem Abschnitt *Split-Screen* in der *Statements*-Dokumentation (`INPUT`-Statement).

Maps erstellen

Maps und Helpmap-Layouts werden mit dem Map-Editor erstellt und bearbeitet.

Die zugehörigen Datendefinitionen können aus einem anderen Natural-Objekt, zum Beispiel aus einer im Data-Area-Editor erstellten und gepflegten **Local Data Area** (LDA), gewählt werden. Siehe auch *Defining Map Fields* in der Map-Editor-Dokumentation.

In Abhängigkeit von der Plattform, auf der Natural betrieben wird, haben diese Editoren entweder eine zeichenorientierte Benutzeroberfläche oder eine grafische Benutzeroberfläche.

Verwandte Themen:

- Informationen zur Benutzung des Masken-Editors entnehmen Sie dem Abschnitt *Masken-Editor* in der plattformspezifischen Natural *Editoren* Dokumentation.
- Informationen zur Benutzung des Datenbereich-Editors (Data-Area-Editor) entnehmen Sie dem Abschnitt *Datenbereich-Editor* in der plattformspezifischen *Editoren*-Dokumentation.
- Eine Beschreibung der gesamten Bandbreite der vom Natural Map-Editor angebotenen Möglichkeiten (Version der zeichenorientierten Benutzeroberfläche) finden Sie im *Map Editor Tutorial*.
- Informationen zur Eingabeverarbeitung mit dynamisch angegebenen Bildschirmmasken siehe Abschnitt *Syntax 1 – Dynamischer generierter Eingabeschirm* in der *Statements*-Dokumentation (`INPUT`-Statement).
- Informationen zur Eingabeverarbeitung mit einer mit dem Map-Editor erstellten Bildschirmmaske siehe Abschnitt *Syntax 2 – Verwendung einer vordefinierten Map* in der *Statements*-Dokumentation.

Map-Verarbeitung starten/stoppen

Eine Eingabemaske („Input Map“) wird mit einem `INPUT USING MAP`-Statement aufgerufen.

Eine Ausgabemaske („Output Map“) wird mit einem `WRITE USING MAP`-Statement aufgerufen.

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

11 Adapter

Den Natural-Objekttyp Adapter können Sie verwenden, um eine Rich-GUI-Page in einer Natural-Anwendung darzustellen. Dieser Objekttyp spielt bei der Verarbeitung einer Rich-GUI-Page eine ähnliche Rolle wie der Objekttyp Map bei der Verarbeitung einer Terminal-Eingabe/Ausgabe. Im Gegensatz zur Map enthält ein Adapter jedoch keine Layout-Informationen.

Ein Objekt vom Typ Adapter wird aus einem externen Page-Layout erzeugt. Es dient als Schnittstelle, über die eine Natural-Anwendung Daten zur Darstellung an ein externes Eingabe-/Ausgabesystem senden kann, und zwar unter Verwendung eines extern definierten und gespeicherten Page-Layouts. Das Objekt Adapter enthält den zum Durchführen dieser Aufgabe erforderlichen Natural-Code.

Eine Anwendung referenziert einen Adapter in einem `PROCESS PAGE USING`-Statement.

Weitere Informationen zum Objekttyp Adapter finden Sie in der *Natural for Ajax*-Dokumentation.

12 Dialog

Dialoge werden bei der ereignisgesteuerten Programmierung für die Erstellung von Natural-Anwendungen mit grafischen Benutzungsoberflächen (GUIs) verwendet.

Anmerkung: Mit Natural for z/OS können keine Objekte des Typs Dialog erstellt werden. Es ist jedoch möglich, Objekte des Typs Dialog in einer Natural-Systemdatei zur Anzeige und für sonstige Zwecke zu speichern.

13 Resource

Dieser Abschnitt beschreibt den Natural-Objekttyp Resource.

Anmerkung: Anders als bei Natural für Offene Systeme, wo es gemeinsam genutzte Ressourcen („Shared Resources“) und nicht gemeinsam genutzte Ressourcen („Private Resources“) verfügbar sind, stehen bei Natural for z/OS zurzeit nur gemeinsam genutzte Resources zur Verfügung.

Was sind Resources?

Resources sind Nicht-Natural-Objekte wie zum Beispiel HTML-Seiten, GIF-Grafiken usw. Gespeichert werden sie in Libraries in der Systemdatei `FNAT` oder `FUSER`, wo sie für Natural-Anwendungen verfügbar sind.

Rein technisch betrachtet handelt es sich um große Datenobjekte im Binärformat oder Zeichenformat, die als Eingabe oder Ergebnis der Ausführung einer Utility oder Benutzeranwendung entweder nur vorübergehend verarbeitet oder dauerhaft gespeichert werden.

Verwendung von Resources

Objekte des Typs Resource werden vom XML Toolkit als Container für DTDs, XML Schemas, Style Sheets usw. benutzt. Das Natural Web Interface verwendet Resource wie zum Beispiel GIFs oder JPEGs. Außerdem können Objekte des Typs Resource zum Speichern von XLIFF-Übersetzungsdatendateien benutzt werden.

Folgende Themen werden im Folgenden behandelt:

- Namenskonventionen für Resources
- Speicherung von Resources

Namenskonventionen für Resources

Objekte des Typs Resource haben einen Langnamen und einen Kurznamen.

Resource-Kurzname

Zu jedem Objekt des Typs Resource existiert ein 8 Byte langer Objektkurzname. Dieser Kurzname besteht aus Großbuchstaben.

Er kann angegeben werden in Systemkommandos wie zum Beispiel `LIST`, `DELETE` und `RENAME` sowie im Object Handler und in den Utilities `INPL` und `SYSMAIN`.

Resource-Langname

Der Langname einer Resource wird in den Verzeichnisdatensätzen der Resource gespeichert. Er hat folgende Struktur:

Bytes	Format	Inhalt
1 - 2	B2	Zeilennummer `H'0000'`
3 - 6	A4	Resource-Typ, in der Regel die Erweiterung des Resource-Namens.
7	A1	Resource-Format, dabei ist A = alphanumerisch, B = binär, U = Unicode
8 - 252	A245	Resource-Name

Der Langname einer Resource kann mit dem Systemkommando `LIST` angezeigt werden. Die Anzeige erfolgt in der *Objekt-Auswahlliste*, wenn Sie dort den Funktionscode `LN` ausführen.

Speicherung von Resources

Objekte des Typs Resource werden ebenso wie andere als Quellcode vorliegende Natural-Objekte in Libraries gespeichert.

Sie können mit den Utilities `SYSMAIN` und `INPL` und mit dem Object Handler gehandhabt werden.

Sie können nicht mit den Natural-Editoren bearbeitet werden.

API zur Verarbeitung von Resources

In der Library `SYSEXT` ist die folgende Anwendungsprogrammierschnittstelle (API) vorhanden, über die eine Benutzeranwendung auf die eindeutigen User Exit-Routinen der Ressources zugreifen kann:

API	Purpose
`USR4208N`	Schreiben, Lesen, Löschen einer Resource unter Verwendung des Kurz- oder Langnamens der Resource.

14 Recording

Ein Objekt des Typs Recording ist ein Natural-Source-Objekt, das Binärdaten einer Natural-Session enthält, die mit der *Recording Utility* aufgezeichnet wurde. Siehe *Debugger und Dienstprogramme*-Dokumentation.

15 Fehlermeldung (Error Message)

Objekte des Typs Error Message werden verwendet, um anwendungsspezifische, benutzerdefinierte Meldungen zu verwalten bzw. um Texte der Natural-Systemmeldungen kundenspezifisch anzupassen.

Fehlermeldungen werden mit der SYSERR Utility erstellt und gepflegt (siehe *Debugger und Dienstprogramme*-Dokumentation). Folgende Optionen stehen zur Verfügung:

- Festlegen von Nummernkreisen für verschiedene Meldungskategorien.
- Vereinheitlichen von Meldungen.
- Übersetzen von Meldungstexten in andere Sprachen.
- Hinzufügen von Langtexten zur ausführlichen Erläuterung der Bedeutung einer Meldung.

16 Kommandoprozessor

Ein Kommandoprozessor wird verwendet, um kommandogesteuerte Navigationssysteme für Natural-Anwendungen als Alternative zur Navigation in hierarchisch organisierten Menüs anbieten zu können.

Der Natural Command Processor (NCP) besteht aus zwei Komponenten: Verwaltung und Laufzeit.

Die SYSNCP Utility bildet die Verwaltungskomponente. Sie enthält alle Funktionen, die für die Definition von Kommandoprozessor-Quellcode und die Steuerung der Navigation innerhalb einer Anwendung benötigt werden.

Das `PROCESS COMMAND`-Statement (siehe *Statements*-Dokumentation) bildet den Laufzeitkomponente, die zum Aufrufen von Natural-Programmen benutzt wird.

17 Editor-Profil

Das Editor-Profil bestimmt die Standardeinstellungen, die beim Erstellen/Bearbeiten eines Programms oder eines Datenbereichs gelten. Beispiele: Umwandlung von Klein- in Großschreibung, Belegung von PF- und PA-Tasten.

Weitere Informationen siehe *Editor-Profil* in der *Editoren*-Dokumentation.

18 Map-Profil und Device-Profil

Das Map-Profil bestimmt die Standardeinstellungen, die bei der Definition und der Ausführung einer Map (Maske) gelten.

Das Device-Profil bestimmt die Standardeigenschaften und -einstellungen, die für ein Device (Gerät) gelten. Damit wird die Kompatibilität zwischen einer Map-Definition und dem zu verwendenden Device sichergestellt.

Sie können das Map-Profil gegen das Device-Profil prüfen.

Weitere Informationen siehe *Profile und Geräte pflegen — Funktion "Maintenance of Profiles & Devices"* im Abschnitt *Masken-Editor* in der *Editoren*-Dokumentation.

19 Parameter-Profil

Ein Parameter-Profil ist ein Satz dynamischer Natural-Profilparameter, die bei jedem Start einer Natural-Session gelten.

Mit der SYSPARM Utility (siehe *Debugger und Dienstprogramme*-Dokumentation) können Sie einen solchen Satz erstellen, ihn unter einem Parameterprofilnamen speichern und danach Natural unter Angabe eines einzigen Parameters aufrufen:

```
PROFILE=profile-name
```

Die Syntax des Profilparameters `PROFILE` und die einzelnen dynamischen Natural-Profilparameter, die Sie in Ihrem Parameterprofil angeben können, sind in der *Parameter-Referenz*-Dokumentation beschrieben.

20 Debug-Umgebung

Eine Debug-Umgebung, die erstellt worden ist, um die Programmausführung zu kontrollieren, kann für zukünftige Verwendung in einer Natural-Systemdatei gespeichert werden.

Die Systemdatei, in der Debug-Umgebungen gespeichert werden sollen, kann mit dem Debugger-Kommando `PROFILE` angegeben werden (siehe *Debugger und Dienstprogramme*-Dokumentation).

Weitere Informationen siehe *Debug Environment Maintenance (Verwaltung der Debug-Umgebung)* in der *Debugger und Dienstprogramme*-Dokumentation.

21 Anwendungsprogrammierschnittstellen

Dieses Kapitel behandelt folgenden Themen:

Verwandte Themen:

- *Application Programming Interfaces* in der *Natural Security*-Dokumentation
- *Application Programming Interfaces* in der *Natural SAF Security*-Dokumentation

Natural-Anwendungsprogrammierschnittstellen (API)

Eine Natural API ist ein Natural-Subprogramm (katalogisiertes Objekt), das dazu benutzt wird, auf Daten zuzugreifen, eventuell Daten zu ändern oder Dienste auszuführen, die für Natural-Statements nicht zugänglich sind. Natural APIs beziehen sich auf Natural, eine Unterkomponente oder ein Unterprodukt.

Mit der SYSEXT Utility können Sie Natural-Anwendungsprogrammierschnittstellen (Application Programming Interfaces/APIs), die in der aktuellen System-Library `SYSEXT` enthalten sind, finden und testen.

Weitere Informationen siehe *SYSEXT Utility*.

Anwendungsprogrammierschnittstellen (API) eines Natural Add-on-Produkts

Die API eines Natural Add-on-Produkts ist ein Natural-Subprogramm (katalogisiertes Objekt), das dazu benutzt wird, auf Daten zuzugreifen, eventuell Daten zu ändern oder Dienste auszuführen, die für ein Natural Add-on-Produkt oder eine Unterkomponente spezifisch sind.

Die API eines Natural Add-on-Produkts wird in der Natural Library und/oder in der Systemdatei geliefert, die für Objekte vorgesehen sind, welche für ein bestimmtes Natural Add-on-Produkt spezifisch sind. Anleitungen zur Verwendung der APIs eines Natural Add-on-Produkts sind in der Dokumentation des betreffenden Add-on-Produkts enthalten.

Die SYSAPI Utility bietet zu jeder API eines Natural Add-on-Produkts eine Funktionsbeschreibung, ein Beispiel-Programm und API-spezifische Schlüsselwörter.

Weitere Informationen siehe *SYSAPI Utility*.

III Function Call

22 Function Call

```
function-name
 (< [([prototype-clause] [intermediate-result-clause])]
   [parameter] [,[parameter]] ... >)
 [array-index-expression]
```

Eine Erläuterung der in dem Syntax-Diagramm verwendeten Symbole entnehmen Sie dem Abschnitt *Syntax-Symbole*.

Verwandte Statements: `DEFINE PROTOTYPE` | `DEFINE FUNCTION`

Verwendung

Ein Function Call wird verwendet, um ein Natural-Objekt des Typs **Function** aufzurufen.

Definiert wird die Function mit dem Natural-Statement `DEFINE FUNCTION`, das Folgendes enthält: die Parameter, lokale und anwendungsunabhängige Variablen, den zu benutzenden Ergebniswert und die Statements, die ausgeführt werden sollen, wenn die Function aufgerufen wird.

Um eine Function aufzurufen können Sie Folgendes angeben:

- entweder den Namen der Function, so wie er im `DEFINE FUNCTION`-Statement definiert ist,
- oder eine alphanumerische Variable, die den Namen der Function zur Laufzeit enthält. In diesem Fall muss die Variable mit dem Schlüsselwort `VARIABLE` in einem `DEFINE PROTOTYPE`-Statement referenziert werden.

Ein Function Call kann anstelle eines schreibgeschützten Operanden in einem Natural-Statement verwendet werden. In diesem Fall muss die Function ein Ergebnis zurückgeben, welches dann von dem Statement wie ein Feld verarbeitet wird, das den gleichen Wert enthält.

Außerdem kann ein Function Call anstelle eines Natural-Statements benutzt werden. In diesem Fall braucht die Function keinen Ergebniswert zurückzugeben; falls doch, wird der Wert des Ergebnisses verworfen.

Einschränkungen

An folgenden Stellen dürfen *keine* Function Calls benutzt werden:

- Stellen, an denen der Operand-Wert durch das Natural-Statement geändert wird, zum Beispiel:

```
MOVE 1 TO #FCT(<..>);
```

- in einem `DEFINE DATA`-Statement;

- alle einem Datenbankzugriff-Statement, zum Beispiel READ, FIND, SELECT, UPDATE und STORE);
- AT BREAK-Statement oder IF BREAK-Statement;
- als Argument von Natural-Systemfunktionen (zum Beispiel AVER, SUM, *TRIM),
- in einem Array-Index-Ausdruck;
- als Parameter eines Function Call.

Wenn ein Function Call in einem INPUT-Statement benutzt wird, dann wird der Rückgabewert wie ein konstanter Wert behandelt. Dies führt zur automatischen Zuweisung des Attributs AD=O, damit dieses Feld schreibgeschützt, d.h. nur zur Ausgabe verfügbar gemacht wird.

Syntax-Beschreibung

Operanden-Definitionstabelle:

Operand	Mögliche Struktur					Mögliche Formate										Referenzierung erlaubt	Dynam. Definition
function-name		S	A			A	U									ja	nein

Syntax-Element-Beschreibung:

Syntax-Element	Beschreibung
function-name	**Function-Name:** Als *function-name* können Sie angeben: ■ entweder den Namen der aufzurufenden Function, so wie er im DEFINE FUNCTION-Statement referenziert wird, ■ oder den Namen einer alphanumerischen Variablen, die den Namen der Function zur Laufzeit enthält. Diese Variable muss in einer Prototyp-Definition mit dem Schlüsselwort VARIABLE des DEFINE PROTOTYPE-Statements referenziert werden. Falls dieser Prototyp nicht die korrekten Parameter- und Ergbnisdefinitionen enthält, kann mit der *prototype-clause* ein anderer Prototyp zugewiesen werden.
prototype-clause	**Prototype-Klausel:** Siehe ***prototype-clause (PT=)***.
intermediate-result-clause	**Intermediate Result-Klausel:** Siehe ***intermediate-result-clause (IR=)***.
parameter	**Parameter-Angabe:** Siehe *parameter*.

Syntax-Element	Beschreibung
array-index-expression	**Array-Index-Notation:** Wenn das vom Function Call zurückgegebene Ergebnis ein Array ist, muss eine Index-Notation vorgesehen werden, um die angeforderten Array-Ausprägungen zu adressieren. Weitere Informationen siehe ***Index-Notation*** in *Benutzervariablen*.

prototype-clause (PT=)

```
PT= prototype-name
```

Um eine Function zur Kompilierungszeit auflösen zu können, benötigt Natural Parameterdefinitionen und das Ergebnis der Function. Wenn kein Prototyp mit dem *function-name*, den Parametern oder dem Ergebnis der Function übereinstimmt, können Sie mit der *prototype-clause* einen passenden Prototyp zuweisen. In diesem Fall wird stattdessen der referenzierte Prototyp benutzt, um die Parameter- und Function-Ergenis-Definitionen aufzulösen. Der in dem referenzierten Prototyp deklarierte *function-name* wird ignoriert.

Syntax-Element-Beschreibung:

Syntax-Element	Beschreibung
prototype-name	**Prototyp-Name:** Als *prototype-name* können Sie angeben: ■ entweder den Namen des Prototyps, dessen Ergebnis- und Parameter-Layouts verwendet werden sollen, ■ den Namen eines alphanumerischen Feldes, der als *function-name* in einem Function Call angegeben ist. Dieses Feld muss den Namen der Funktion enthalten, die zur Aufrufzeit aufgerufen werden soll. Bei dem Feldnamen darf kein Array-Index-Ausdruck angegeben werden.

intermediate-result-clause (IR=)

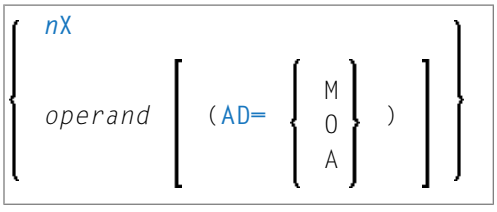
```
IR= ( format-length [/array-definition]
    | [(array-definition)] HANDLE OF OBJECT
    | ( A | U | B ) [/array-definition]) DYNAMIC )
```

Diese Klausel kann benutzt werden, um die *format-length*/*array definition* des Ergebniswertes für einen Function Call anzugeben, falls weder das katalogisierte Objekt der Function noch eine Prototyp-Definition zur Verfügung stehen. Wenn für diesen Function Call ein Prototyp verfügbar ist oder wenn ein katalogisiertes Objekt der aufgerufenen Function existiert, wird das mit der *intermediate-result-clause* angegebene Ergebniswertformat auf Datenübertragungskompatibilität geprüft.

Syntax-Element-Beschreibung:

Syntax-Element	Beschreibung
format-length	**Format/Länge-Definition:** Das Format und die Länge des Feldes. Informationen zur Format/Längen-Definition bei Benutzervariablen siehe ***Format und Länge von Benutzervariablen***.
array-definition	**Array-Dimension-Definition:** In der *array-definition* definieren Sie die untere und die obere Grenze der Dimensionen bei einer Array-Definition. Siehe *Definition von Array-Dimensionen* in der *Statements*-Dokumentation.
HANDLE OF OBJECT	**Object-Handle:** Wird in Verbindung mit NaturalX verwendet. Weitere Informationen siehe *NaturalX* im *Leitfaden zur Programmierung*.
A, B or U	**Datenformat:** Mögliche Formate: alphanumerisch, binär oder Unicode bei dynamischen Variablen.
DYNAMIC	**Dynamische Variable:** Ein Feld kann als DYNAMIC definiert werden. Weitere Informationen zur Verarbeitung dynamischer Variablen siehe ***Dynamische Variablen***.

parameter



Parameter werden entweder im DEFINE DATA PARAMETER-Statement bei der Definition der Function oder in einem DEFINE PROTOTYPE-Statement angegeben. Die semantischen und syntaktischen

Regeln, die bei den Parametern einer Function gelten, sind die gleichen wie diejenigen, die im Parameterabschnitt von Subprogrammen beschrieben werden. Siehe *Parameter* in der Beschreibung des CALLNAT-Statement.

Wenn Sie mehrere Parameter angeben, müssen Sie diese voneinander abtrennen - entweder mit einem Komma oder mit dem Eingabebegrenzungszeichen, das mit dem Session-Parameter ID angegeben wird. Falls in der Liste der Parameter Zahlen vorkommen und wenn das Komma als Dezimalzeichen festgelegt ist (mit dem Session-Parameter DC), müssen Sie entweder das Komma mit einem zusätzlichen Leerzeichen vom Wert trennen oder das Eingabebegrenzungszeichen verwenden.

Beispiel mit Begrenzungszeichendefinition ID=; und DC=,:`WRITE F#ADD (<1 , 2>) F#ADD (<1;2>)`

Operanden-Definitionstabelle:

Operand	Mögliche Struktur					Mögliche Formate												Referenzierung erlaubt	Dynam. Definition
operand	C	S	A	G		A	N	P	I	F	B	D	T	L	C	G	O	ja	nein

Syntax-Element-Beschreibung:

Syntax-Element	Beschreibung	
*n*X	**Zu überspringende Parameter:** Mit der *n*X-Notation können Sie angeben, dass die nächsten *n* Parameter übersprungen werden sollen (zum Beispiel: 1X, um den nächsten Parameter zu überspringen oder 3X, um die nächsten drei Parameter zu überspringen); das bedeutet, dass für die nächsten *n* Parameter keine Werte an die Function übergeben werden. Ein zu überspringender Parameter muss im DEFINE DATA PARAMETER-Statement der Function mit dem Schlüsselwort OPTIONAL definiert werden. OPTIONAL bedeutet, dass der Wert vom aufrufenden Objekt an einen so gekennzeichneten Parameter übergeben werden kann - jedoch nicht übergeben werden muss.	
AD=	**Attribut-Definition:** If *operand* ist eine Variable, die Sie wie folgt markieren können:	
	AD=O	**Nicht änderbar:** Siehe Session-Parameter AD=O. **Anmerkung:** Intern wird AD=O auf die gleiche Weise verarbeitet wie BY VALUE (siehe *parameter-data-definition* in the description of the DEFINE DATA statement).
	AD=M	**Änderbar:** Siehe Session-Parameter AD=M.

Syntax-Element	Beschreibung	
		Dies ist die Standardeinstellung.
	AD=A	**Nur zur Eingabe:** Siehe Session-Parameter AD=A.
	Anmerkung: Wenn *operand* eine Konstante ist, kann die Attribut-Definition AD nicht ausdrücklich angegeben werden. Bei Konstanten gilt immer AD=O.	

Beispiel

Im Beispiel-Programm FUNCEX01 werden die Functions F#ADDITION, F#CHAR, F#EVEN und F#TEXT verwendet.

Alle Beispiele in diesem Abschnitt werden in der Natural-System-Library SYSEXPG als Quellcode-Objekte und als katalogisierte Objekte zur Verfügung gestellt.

- Aufrufendes Programm FUNCEX01:
- Aufgerufene Function F#ADDITION
- Aufgerufene Function F#CHAR
- Aufgerufene Function F#EVEN
- Aufgerufene Function F#TEXT

Aufrufendes Programm FUNCEX01:

```
** Example 'FUNCEX01': Function call  (Program)
************************************************************************
DEFINE DATA LOCAL
  1 #NUM  (I2) INIT <5>
  1 #A    (I2) INIT <1>
  1 #B    (I2) INIT <2>
  1 #C    (I2) INIT <3>
  1 #CHAR (A1) INIT <'A'>
END-DEFINE
*
IF #NUM = F#ADDITION(<#A,#B,#C>)    /* Function with three parameters.
  WRITE 'Sum of #A,#B,#C' #NUM
ELSE
  IF #NUM = F#ADDITION(<1X,#B,#C>)  /* Function with optional parameters.
    WRITE 'Sum of #B,#C' #NUM
  END-IF
END-IF
*
DECIDE ON FIRST #CHAR
  VALUE F#CHAR (<>)(1)              /* Function with result array.
     WRITE 'Character A found'
```

```
  VALUE F#CHAR (<>)(2)
     WRITE 'Character B found'
  NONE
     IGNORE
END-DECIDE
*
IF F#EVEN(<#B>)                     /* Function with logical result value.
  WRITE #B 'is an even number'
END-IF
*
F#TEXT(<'Hello', '*'>)              /* Function used as a statement.
*
WRITE F#TEXT(<(IR=A12) 'Good'>)     /* Function with intermediate result.
*
END
```

Ausgabe des Programms `FUNCEX01`

```
Sum of #B,#C      5
Character A found
     2 is an even number
*** Hello world ***
Good morning
```

Aufgerufene Function F#ADDITION

Die Function `F#ADDITION` ist in der Beispiel-Function `FUNCEX02` definiert.

```
** Example 'FUNCEX02': Function call  (Function)
************************************************************************
DEFINE FUNCTION F#ADDITION
  RETURNS (I2)
  DEFINE DATA PARAMETER
    1 #PARM1 (I2) OPTIONAL
    1 #PARM2 (I2) OPTIONAL
    1 #PARM3 (I2) OPTIONAL
  END-DEFINE
  /*
  RESET F#ADDITION
  IF #PARM1 SPECIFIED
    F#ADDITION := F#ADDITION + #PARM1                                    ↩

  END-IF
  IF #PARM2 SPECIFIED
    F#ADDITION := F#ADDITION + #PARM2                                    ↩

  END-IF
  IF #PARM3 SPECIFIED
    F#ADDITION := F#ADDITION + #PARM3                                    ↩

  END-IF
```

```
  /*
END-FUNCTION
*
END ↩
```

Aufgerufene Function F#CHAR

Die Function F#CHAR ist in der Beispiel-Function FUNCEX03 definiert.

```
** Example 'FUNCEX03': Function call  (Function)
************************************************************************
DEFINE FUNCTION F#CHAR
  RETURNS (A1/1:2)
  /*
  F#CHAR(1) := 'A'
  F#CHAR(2) := 'B'
  /*
END-FUNCTION
*
END  ↩
```

Aufgerufene Function F#EVEN

Die Function F#EVEN ist in der Beispiel-Function FUNCEX04 definiert.

```
** Example 'FUNCEX04': Function call  (Function)
************************************************************************
DEFINE FUNCTION F#EVEN
  RETURNS (L)
  DEFINE DATA
  PARAMETER
    1 #NUM  (N4) BY VALUE
  LOCAL
    1 #REST (I2)
  END-DEFINE
  /*
  DIVIDE 2 INTO #NUM REMAINDER #REST
  /*
  IF #REST = 0
    F#EVEN := TRUE
  ELSE
    F#EVEN := FALSE
  END-IF
  /*
END-FUNCTION
*
END ↩
```

Aufgerufene Function F#TEXT

Die Function F#TEXT ist in der Beispiel-Function FUNCEX05 definiert.

```
** Example 'FUNCEX05': Function call  (Function)
************************************************************************
DEFINE FUNCTION F#TEXT
  RETURNS (A20) BY VALUE
  DEFINE DATA
  PARAMETER
    1 #TEXT1 (A5) BY VALUE
    1 #TEXT2 (A1) BY VALUE OPTIONAL
  LOCAL
    1 #FRAME (A3)
  END-DEFINE
  /*
  IF #TEXT2 SPECIFIED
    MOVE ALL #TEXT2 TO #FRAME
    /*
    COMPRESS #FRAME #TEXT1 'world' #FRAME INTO F#TEXT
    /*
    WRITE F#TEXT
  ELSE
    COMPRESS #TEXT1 'morning' INTO F#TEXT
    /*
  END-IF
  /*
END-FUNCTION
*
END ↩
```

Function-Ergebnis

Entsprechend der Function-Definition kann ein Function Call ein einzelnes Ergebnisfeld zurückgeben. Dies kann ein Skalar-Wert sein oder ein Array-Feld, das in dem Statement, in dem der Function Call eingebettet ist, wie ein temporäres Feld verarbeitet wird. Ist das Ergebnis ein Array, dann muss unmittelbar nach dem Function Call ein *array-index-expression* folgen, mit dem die erforderlichen Ausprägungen adressiert werden.

Zum Beispiel, um auf die erste Ausprägung des zurückgegebenen Array zuzugreifen:

```
#FCT(<#A,#B>)(1)
```

Parameter- und Ergebnisangaben

Um einen Function Call zur Kompilierungszeit korrekt aufzulösen, benötigt der Kompiler die Format-, Längen- und Array-Struktur der Parameter und des Function-Ergebnisses. Die in dem Function Call angegebenen Parameter werden gegen die entsprechenden Definitionen in der Function geprüft um sicherzustellen, dass sie zueinander passen. Wird eine Function anstelle eines Operanden in einem Statement benutzt, muss das Function-Ergebnis hinsichtlich der Format-, Längen- und Array-Struktur mit dem Operanden übereinstimmen.

Sie haben drei Möglichkeiten, diese Informationen zur Verfügung zu stellen:

1. Sie können die Parameter- und Ergebnisangaben implizit aus dem katalogisierten Objekt (falls verfügbar) der aufgerufenen Funktion gewinnen, wenn zuvor kein `DEFINE PROTOTYPE`-Statement ausgeführt wurde.

   Dieses Verfahren erfordert den geringsten Programmieraufwand.
2. Sie können ein `DEFINE PROTOTYPE`-Statement benutzen. Ein `DEFINE PROTOTYPE`-Statement müssen Sie benutzen, wenn das katalogisierte Objekt der aufgerufenen Function nicht verfügbar ist oder wenn die Function zur Kompilierungszeit nicht bekannt, d.h., anstelle eines Function-Namens wird der Name einer alphanumerischen Variablen im Function Call angegeben.
3. Sie können eine explizite **(**`IR=`**)**-Klausel im Function Call angeben.

Bei den ersten beiden Möglichkeiten erfolgt eine vollständige Validierung der Format-, Längen- und Array-Struktur der Parameter und des Function-Ergebnisses.

Zusätzliche Klauseln für den Function Call

Wenn weder ein `DEFINE PROTOTYPE`-Statement noch ein katalogisiertes Function-Objekt existiert, können Sie die folgenden Klauseln in Ihrem Function Call benutzen:

- Die Klausel **(**`IR=`**)** gibt die Format-, Längen- und Array-Struktur des Funktionsergebnisses an.

  Diese Klausel bestimmt, welche Format-, Längen- und Array-Struktur der Kompiler für das Ergebnisfeld (das Zwischenergebnis, so wie es von dem Statement benutzt wird, das den Function Call enthält) nehmen soll. Wenn für einen Function Call eine Prototyp-Definition zur Verfügung steht, dann setzt die Klausel **(**`IR=`**)** die Angaben in dem Prototyp außer Kraft.

  Die Klausel **(**`IR=`**)** erzwingt keine Parameterprüfungen.

- Die Klausel **(PT=)** benutzt einen zuvor definierten Prototyp, dessen Namen anders lautet als der Name der Function. Diese Klausel validiert die Parameter und das Ergebnis der Function mit Hilfe eines DEFINE PROTOTYPE-Statement mit dem referenzierten Namen.

  In dem folgenden Beispiel wird die Function #MULT aufgerufen, es gelten jedoch die Parameter- und Ergebnisangaben aus dem Prototyp, dessen Name #ADD ist:

```
#I := #MULT(<(PT=#ADD) 2 , 3>)
```

Validierung der Parameters und des Ergebnisses der Function

Zur Prüfung der Parameter wird die zuerst gefundene der nachfolgend aufgeführten Definitionen verwendet:

- die Prototyp-Definition, die in der (PT=)-Klausel referenziert wird;
- die Prototyp-Definition im DEFINE PROTOTYPE-Statement, deren Namen mit dem Namen der Function übereinstimmt, der im Function Call benutzt wird;
- die Parameterangaben im katalogisierten Function-Objekt, die mit dem DEFINE FUNCTION-Statement geliefert werden.

Falls keine dieser Optionen angegeben ist, wird keine Parameter-Validierung durchgeführt. Das gestattet es, im Function Call eine beliebige Anzahl Parameter mit beliebigem Layout zu mitliefern, ohne einen Syntaxfehler zu erhalten.

Zur Prüfung des Ergebnisses der Function wird die zuerst gefundene der nachfolgend aufgeführten Definitionen verwendet:

- die Definition, die in der **(IR=)**-Klausel zur Verfügung gestellt wird;
- die RETURNS-Definition im Pprototyp, die in der (PT=)-Klausel referenziert wird;
- die Prototyp-Definition im DEFINE PROTOTYPE-Statement, bei der der Prototyp-Namen mit dem Namen der Function übereinstimmt, der im Function Call benutzt wird;
- die Angabe des Function-Ergebnisses in dem katalogisierten Function-Objekt.

Falls keine der aufgeführten Definitionen angegeben ist, wird ein Syntaxfehler ausgegeben.

Beispiel mit mehreren Definitionen in einem Function Call

Programm:

```
** Example 'FUNCBX01': Declare result value and parameters  (Program)
************************************************************************
*
DEFINE DATA LOCAL
  1 #PROTO-NAME (A20)
  1 #PARM1      (I4)
  1 #PARM2      (I4)
END-DEFINE
*
DEFINE PROTOTYPE VARIABLE #PROTO-NAME
  RETURNS (I4)
  DEFINE DATA PARAMETER
    1 #P1 (I4) BY VALUE OPTIONAL
    1 #P2 (I4) BY VALUE
  END-DEFINE
END-PROTOTYPE
*
#PROTO-NAME := 'F#MULTI'
#PARM1      := 3
#PARM2      := 5
*
WRITE #PROTO-NAME(<#PARM1, #PARM2>)
WRITE #PROTO-NAME(<1X ,5>)
*
WRITE F#MULTI(<(PT=#PROTO-NAME) #PARM1,#PARM2>)
*
WRITE F#MULTI(<(IR=N20) #PARM1, #PARM2>)
*
END
```

Function `F#MULTI`:

```
** Example 'FUNCBX02': Declare result value and parameters  (Function)
************************************************************************
DEFINE FUNCTION F#MULTI
  RETURNS #RESULT (I4) BY VALUE
  DEFINE DATA PARAMETER
    1 #FACTOR1 (I4) BY VALUE OPTIONAL
    1 #FACTOR2 (I4) BY VALUE
  END-DEFINE
  /*
  IF #FACTOR1 SPECIFIED
    #RESULT := #FACTOR1 * #FACTOR2
  ELSE
    #RESULT := #FACTOR2 * 10
  END-IF
  /*
```

```
END-FUNCTION
*
END ↩
```

Reihenfolge bei der Auswertung von Functions in Statements

Alle Function Calls, die in einem Natural-Statement verwendet werden, werden ausgewertet, bevor mit der Ausführung des Statements begonnen wird. Sie werden in derselben Reihenfolge ausgeführt, in der sie im Statement erscheinen. Die Function-Ergebniswerte werden in temporären Feldern gespeichert, die später als Operanden bei der Statement-Ausführung verwendet werden.

Das Aufrufen einer Function, die änderbare Parameter hat, die wiederholt innerhalb desselben Statements benutzt werden, kann unterschiedliche Function-Ergebnisse verursachen, wie im folgenden Beispiel gezeigt wird:

Beispiel:

Bevor die Ausführung des `COMPUTE`-Statement beginnt, hat die Variable `#I` den Wert `1`. Im ersten Schritt wird die Funktion `F#RETURN` ausgeführt. Dadurch ändert sich `#I` auf den Wert `2` und gibt den Wert `2` als das Ergebnis der Funktion zurück. Danach wird mit der `COMPUTE`-Operation begonnen, wobei die erhöhte Variable `#I (2)` und das temporäre Feld `(2)` auf die Summe von `4` addiert werden.

Programm:

```
** Example 'FUNCCX01': Parameter changed within function (Program)
************************************************************************
DEFINE DATA LOCAL
  1 #I      (I2) INIT <1>
  1 #RESULT (I2)
END-DEFINE
*
COMPUTE #RESULT := #I + F#RETURN(<#I>)  /* First evaluate function call,
                                        /* then execute the addition.
*
WRITE '#I     :' #I /
      '#RESULT:' #RESULT
*
END
```

Function:

```
** Example 'FUNCCX02': Parameter changed within function (Function)
************************************************************************
DEFINE FUNCTION F#RETURN
  RETURNS #RESULT (I2) BY VALUE
  DEFINE DATA PARAMETER
    1 #PARM1 (I2) BY VALUE RESULT
  END-DEFINE
  /*
  #PARM1  := #PARM1 + 1       /* Increment parameter.
  #RESULT := #PARM1           /* Set result value.
  /*
END-FUNCTION
*
END
```

Ausgabe des Programms `FUNCCX01`**:**

```
#I     :      2
#RESULT:      4
```

Verwendung einer Function als Statement

Sie können einen Function Call kann auch anstelle eines Natural-Statements benutzen, d.h. ohne den Function Call in ein anderes Statement einzubetten. In diesem Fall braucht der Function Call keinen Ergebniswert zurückzuliefern; falls doch, wird der Ergebniswert ignoriert.

Um eine unerwünschte Verknüpfung zu dem vorangehenden Statement zu verhindern, muss ein Semikolon (;) gesetzt werden, um den Function Call ausdrücklich von diesem Statement zu trennen.

Beispiel:

Programm:

```
** Example 'FUNCDX01': Using a function as a statement (Program)
************************************************************************
DEFINE DATA LOCAL
  1 #A (I4) INIT <1>
  1 #B (I4) INIT <2>
END-DEFINE
*
*
WRITE 'Write:' #A #B
F#PRINT-ADD(< 2,3 >)     /* Function call belongs to operand list
                         /* immediately preceding it.
```

```
*
WRITE // '************************' //
*
WRITE 'Write:' #A #B;    /* Semicolon separates operands and function.
F#PRINT-ADD(< 2,3 >)     /* Function call does not belong to the
                         /* operand list.
*
END
```

Function:

```
** Example 'FUNCDX02': Using a function as a statement (Function)
************************************************************************
DEFINE FUNCTION F#PRINT-ADD
  RETURNS (I4)
  DEFINE DATA PARAMETER
    1 #SUMMAND1 (I4) BY VALUE
    1 #SUMMAND2 (I4) BY VALUE
  END-DEFINE
  /*
  F#PRINT-ADD := #SUMMAND1 + #SUMMAND2    /* Result of function call.
  WRITE 'Function call:' F#PRINT-ADD
  /*
END-FUNCTION
*
END ↩
```

Ausgabe des Programms `FUNCDX01`:

```
Function call:           5
Write:           1           2           5

************************

Write:           1           2
Function call:           5 ↩
```

IV Felder definieren

Dieser Teil beschreibt, wie Sie die Felder definieren, die Sie in einem Programm verwenden möchten. Diese Felder können entweder Datenbankfelder oder benutzerdefinierte Felder sein.

Bitte beachten Sie, dass dieses Kapitel sich auf die wichtigsten Optionen des `DEFINE DATA`-Statements beschränkt. Weitere Optionen bei diesem Statement sind in der *Statements*-Dokumentation beschrieben.

Die Besonderheiten von Datenbankfeldern sind im Kapitel ***Datenbankzugriffe*** beschrieben. Im Prinzip gelten die dort für Adabas beschriebenen Funktionen und Beispiele auch für andere Datenbankverwaltungssysteme. Eventuell vorhandene Unterschiede sind in der betreffenden Datenbankschnittstellen-Dokumentation sowie in der *Statements*- bzw. der *Parameter-Referenz*-Dokumentation beschrieben.

23 Benutzung und Struktur des DEFINE DATA-Statements

Das erste Statement in einem im **Structured Mode** geschriebenen 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.

Informationen zur strukturellen Einrückung eines Source-Programms siehe Natural-Systemkommando `STRUCT`.

Felddefinitionen im DEFINE DATA-Statement

Alle im Programm 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 (siehe **unten**).
- Die Felder können außerhalb des Programms in einer **Local Data Area oder einer Global Data Area** definiert werden, wobei das `DEFINE DATA`-Statement diese Data Area referenziert (siehe **unten**).

Felder, die von mehreren Programmen/Subprogrammen benutzt werden, sollten in einer programmexternen Data Area definiert werden.

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

Data Areas werden mit dem Datenbereich-Editor (Data-Area-Editor) erstellt und gepflegt, der in der *Editoren*-Dokumentation beschrieben ist.

Im ersten der folgenden **Beispiele** sind die Felder innerhalb des `DEFINE DATA`-Statements im Programm selbst definiert. Im zweiten Beispiel sind die gleichen Felder in einer **Local Data Area** (LDA) definiert, und das `DEFINE DATA`-Statement enthält lediglich eine Referenz auf diese Data Area.

Felder innerhalb eines DEFINE DATA-Statements definieren

Das folgende Beispiel veranschaulicht, wie Felder innerhalb des Programms im `DEFINE DATA`-Statements selbst definiert werden können:

```
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
...
```

Felder in einer separaten Data Area definieren

Das folgende Beispiel veranschaulicht, wie Felder außerhalb eines Programms in einer **Local Data Area** (LDA) definiert werden können:

Programm:

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

Vom Programm referenzierte 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
 ↩
```

Diese LDA enthält eine View (Datensicht, siehe auch weiter **unten**) mit Namen `VIEWEMP`, in der die vom Programm verwendeten Felder definiert sind.

Struktur eines DEFINE DATA-Statements — Level-Nummern

Folgende Themen werden behandelt:

Struktur und Gruppierung der Definitionen

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 Null 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 Abschnitt *Datenbankzugriffe*.

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. Siehe *Namenskonventionen für Benutzervariablen* in der Dokumentation *Natural benutzen*.

Beispiel für Level-Nummern in einer Gruppe:

```
DEFINE DATA LOCAL
1 #FIELDA (N2.2)
1 #FIELDB (I4)
1 #GROUPA
  2 #FIELDC (A20)
  2 #FIELDD (A10)
  2 #FIELDE (N3.2)
1 #FIELDF (A2)
...
END-DEFINE ↩
```

In diesem Beispiel sind die Felder `#FIELDC`, `#FIELDD` und `#FIELDE` unter dem gemeinsamen Gruppennamen `#GROUPA` definiert. Die anderen drei Felder sind nicht Teil der Gruppe. Bitte beachten Sie, dass `#GROUPA` 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***.

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.

Speicherplatzausrichtung

Der Speicherbereich, in dem alle Benutzervariablen gespeichert werden, beginnt immer an einer Doppelwortgrenze.

Wird ein `DEFINE DATA`-Statement benutzt, werden alle Datenblöcke (z.B. `LOCAL`-, `GLOBAL`-Blöcke) an einer Doppelwortgrenze ausgerichtet, und alle hierarchischen Strukturen (View -Definitionen und Gruppen) auf Level 1 werden auf Vollwortgrenze ausgerichtet. Redefinitionen, Skalar- und Array-Variablen werden selbst dann nicht ausgerichtet, wenn sie auf Level 1 definiert sind.

Die Ausrichtung an den Grenzen innerhalb des Datenbereichs ist Sache des Benutzers und richtet sich nach der Reihenfolge, in der die Variablen für Natural definiert sind.

24 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.

Siehe auch *Namenskonventionen für Benutzervariablen* in der Dokumentations *Natural benutzen*.

Definition von Benutzervariablen

Sie definieren eine Benutzervariable, indem Sie den Namen sowie das Format und die Länge der Variablen im `DEFINE DATA`-Statement angeben.

Sie definieren die Eigenschaften einer Variable mit der folgenden Notation:

```
(r,format-length/index)
```

Diese Notation folgt auf den Variablennamen, getrennt durch eine oder mehrere Leerstellen, als Option.

Zwischen den einzelnen Elementen der Notation sind keine Leerstellen zulässig.

Die einzelnen Elemente können erforderlichenfalls selektiv angegeben werden. Wenn sie aber zusammen benutzt werden, müssen Sie durch die oben angegebenen Zeichen voneinander getrennt werden.

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
```

Anmerkungen:

1. Im Strucutured Mode oder wenn ein Programm eine `DEFINE DATA LOCAL`-Klausel enthält, können Variablen nicht dynamisch in einem Statement definiert werden.
2. Dies gilt nicht für anwendungsunabhängige Variablen (AIVs). Siehe auch *Anwendungsunabhängige Variablen definieren*.

Datenbankfelder mit der (r)-Notation referenzieren

Ein Statement-Label oder die Quellcode-Zeilennummer kann zum Referenzieren eines vorher eingesetzten Natural-Statements benutzt werden. Damit kann die Standard-Referenzierung von Natural überschrieben werden (wie für jedes einzelne Statement beschrieben, wo zutreffend), oder es wird zu Dokumentationszwecken verwendet. Siehe auch ***Schleifenverarbeitung***, Unterabschnitt ***Statements innerhalb eines Programms referenzieren***.

Folgende Themen werden behandelt:

- Standard-Referenzierung von Datenbankfeldern
- Referenzieren mit Statement-Labels
- Referenzieren mit Quellcode-Zeilennummern

Standard-Referenzierung von Datenbankfeldern

Im Allgemeinen gilt Folgendes, wenn Sie keine Statement-Referenzierung spezifizieren:

- Standardmäßig wird die innerste aktive Datenbank-Schleife (`FIND`, `READ` oder `HISTOGRAM`) referenziert, in der das betreffende Datenbankfeld eingelesen wurde.
- Wenn das Feld in keiner aktiven Datenbank-Schleife eingelesen wurde, wird das letzte vorher verwendete `GET`-Statement (im Reporting Mode auch `FIND FIRST` oder `FIND UNIQUE`-Statement) referenziert, welches das Feld eingelesen hat.

Referenzieren mit Statement-Labels

Ein Natural-Statement, das bewirkt, dass eine Verarbeitungsschleife initiiert wird und/oder dass Datenelemente in der Datenbank aufgerufen werden, kann mit einem symbolischen Label zur nachfolgenden Referenzierung versehen werden.

Ein Label kann entweder in der Form "*label*." vor dem referenzierenden Objekt oder in Klammern "(*label*.)" nach dem referenzierenden Objekt (aber nicht beide gleichzeitig) angegeben werden.

Die Namenskonventionen für Labels sind identisch mit denen für Variablen.

Siehe auch *Namenskonventionen für Benutzervariablen* in der Dokumentations *Natural benutzen*.

Der Punkt nach dem Label-Namen dient zur Identifizierung des Eintrags als ein Label.

Beispiel:

```
...
 RD. READ PERSON-VIEW BY NAME STARTING FROM 'JONES'
   FD. FIND AUTO-VIEW WITH PERSONNEL-ID = PERSONNEL-ID (FD.)
         DISPLAY NAME (RD.) FIRST-NAME (RD.) MAKE (FD.)
    END-FIND
 END-READ
 ...
```

Referenzieren mit Quellcode-Zeilennummern

Ein Statement kann auch referenziert werden, wenn dafür die Nummer der Quellcode-Zeile, in der das Statement sich befindet, benutzt wird.

Alle vier Ziffern der Zeilennummer müssen angegeben werden (führende Nullen dürfen nicht weggelassen werden).

Beispiel:

```
...
0110 FIND EMPLOYEES-VIEW WITH NAME = 'SMITH'
0120   FIND VEHICLES-VIEW WITH MODEL = 'FORD'
0130     DISPLAY NAME (0110) MODEL (0120)
0140   END-FIND
0150 END-FIND
...
```

Quellcode-Zeilennummern umnummerieren

Zeilennummer-Referenzen in einer Source (siehe *Datenbankfelder mit der (r)-Notation referenzieren* und *Statements innerhalb eines Programms referenzieren*), werden geändert, wenn die bezogene Zeilennummer mit dem Systemkommando `RENUMBER` geändert wird. Die Umnummerierung betrifft alle Zeilenreferenz-Zeichenfolgen mit Ausnahme von solchen, die in alphanumerischen Konstanten vorhanden sind, zum Beispiel:

```
#FIELD1 := '(1150)'   /* Wird nicht umnummeriert.
RESET NAME (1150)     /* Wird umnummeriert.
```

Anmerkung: Standardmäßig werden Zeilennummer-Referenzen in alphanumerischen Konstanten nicht umnummeriert. Falls sie auch umnummeriert werden sollen, müssen Sie den Profilparameter `RNCONST` auf `ON` setzen.

Die folgenden Zeichenfolgen werden als gültige Referenz auf eine Quellcode-Zeilennummer erkannt und umnummeriert (*nnnn* ist eine vierstellige Ziffer):

Zeichenfolge	Beispiel-Statement
(*nnnn*)	ESCAPE BOTTOM (0150)
(*nnnn*/	DISPLAY ADDRESS-LINE(0010/1:5)
(*nnnn*,	DISPLAY NAME(0010,A10/1:5)

Wenn unmittelbar nach der linken Klammer kein *nnnn* folgt oder wenn nach *nnnn* keine rechte, schließende Klammer, sondern ein beliebiges anderes Zeichen, ein Komman oder ein Schrägstrich, folgt, wird die Zeichenfolge nicht als Zeilennummer-Referenz angesehen und wird nicht geändert.

Format und Länge von Benutzervariablen

Das Format und die Länge einer Benutzervariablen werden in Klammern hinter dem Namen der Variablen angegeben.

Variablen fester Länge können eine/s der folgenden Formate und Längen haben.

Zur Definition von Format und Länge bei dynamischen Variablen siehe ***Definition dynamischer Variablen***.

Format		Definierbare Länge	Interne Länge (in Bytes)
A	Alphanumerisch	1 - 1073741824 (1 GB)	1 - 1073741824
B	Binär	1 - 1073741824 (1 GB)	1 - 1073741824
C	Attribut-Zuweisung	-	2
D	Datum	-	4
F	Gleitkomma	4 oder 8	4 oder 8
I	Integer (Ganzzahl)	1 , 2 oder 4	1, 2 oder 4
L	Logisch	-	1
N	Numerisch (ungepackt)	1 - 29	1 - 29
P	Numerisch (gepackt)	1 - 29	1 - 15
T	Zeit	-	7
U	Unicode (UTF-16)	1 - 536870912 (0,5 GB)	2 - 1073741824

Die Länge kann nur angegeben werden, wenn das Format angegeben ist. Bei einigen Formaten braucht die Länge nicht explizit angegeben zu werden (wie in der Tabelle oben gezeigt).

Bei mit Format N oder P definierten Feldern können Sie die Dezimalstellen-Notation in der Form *n*.*m* benutzen. *n* stellt die Anzahl der Stellen vor dem Dezimalkomma dar, und *m* stellt die Anzahl der Stellen nach dem Dezimalkomma dar. Die Summe der Werte von *n* und *m* darf 29 nicht überschreiten.

Die maximal definierbare Länge (1 GB für alphanumerische, binäre und Unicode-Felder) stellt die vom Natural-Compiler auferlegte Grenze dar. In Wirklichkeit ist der Speicherplatz, der als Datenspeicher bereitstehen kann, allerdings sehr viel kleiner. Insbesondere in einer „Natural Thread"-basierten Umgebung ist die Größe der sessionabhängigen Benutzerbereiche, also der Umfang der Benutzerfelder in der Data Area auf den Wert des Thread beschränkt.

Anmerkungen:

1. Wenn eine Benutzervariable vom Format P mit einem `DISPLAY`-, `WRITE`- oder `INPUT`-Statement ausgegeben wird, wandelt Natural intern das Format P in das Format N für die Ausgabe um.
2. Wenn im Reporting Mode Format und Länge nicht für eine Benutzervariable angegeben werden, wird das/die Standard-Format/Länge N7 benutzt, es sei denn, diese Standard-Zuweisung wurde durch den Profil/Session-Parameter `FS` ausgeschaltet.

Für ein Datenbankfeld gilt das/die Format/Länge, wie für das Feld in der DDM definiert. (Im Reporting Mode ist es auch möglich, in einem Programm ein/e andere/s Format/Länge für ein Datenbankfeld zu definieren.)

Im Structured Mode kann Format und Länge nur in einer Data Area-Definition oder mit einem `DEFINE DATA`-Statement angegeben werden.

Beispiel für Format/Längen-Definition — Structured Mode:

```
DEFINE DATA LOCAL
1 EMPLOY-VIEW VIEW OF EMPLOYEES
  2 NAME
  2 FIRST-NAME
1 #NEW-SALARY (N6.2)
END-DEFINE
...
FIND EMPLOY-VIEW ...
...
COMPUTE #NEW-SALARY = ...
...
```

Im Reporting Mode kann Format/Länge im Hauptteil des Programms definiert werden, wenn kein `DEFINE DATA`-Statement benutzt wird.

Beispiel für eine Format/Längen-Definition — Reporting Mode:

```
...
...
 FIND EMPLOYEES
... ... COMPUTE #NEW-SALARY(N6.2) = ...
...
```

Spezielle Formate

Zusätzlich zu den standardmäßigen Formaten alphanumerisch (A bzw. U) und numerisch (B, F, I, N, P) unterstützt Natural die folgenden speziellen Formate:

- Format C — Attribut-Kontrolle
- Formate D — Datum und T - Zeit
- Format L — Logisch
- Format Handle of Object

Format C — Attribut-Kontrolle

Eine mit dem Format C definierte Variable kann benutzt werden, um Attribute dynamisch einem in einem `DISPLAY`, `INPUT`-, `PRINT`-, `PROCESS PAGE`- oder `WRITE`-Statement benutzten Feld zuzuweisen.

Für eine Variable des Formats C kann keine Länge angegeben werden. Der Variable wird stets eine Länge von 2 Bytes von Natural zugewiesen.

Beispiel:

```
DEFINE DATA LOCAL
1 #ATTR (C)
1 #A (N5)
END-DEFINE
...
MOVE (AD=I CD=RE) TO #ATTR
INPUT #A (CV=#ATTR)
...
```

Weitere Informationen siehe Session-Parameter `CV`.

Formate D — Datum und T - Zeit

Mit den Formaten D und T definierte Variablen können für die Datums- und Zeit-Arithmetik und -Anzeige benutzt werden. Das Format D kann nur Datums-Informationen enthalten. Das Format T kann sowohl Datums- als auch Zeit-Informationen enthalten; mit anderen Worten sind Datums-Informationen eine Untermenge der Zeit-Informationen. Die Zeit wird in Zehntelsekunden gemessen.

Für Variablen der Formate D und T kann keine Länge angegeben werden. Einer Variable mit Format D wird stets eine Länge von 4 Bytes (P6) und einer Variable mit Format T wird stets eine Länge von 7 Bytes (P12) von Natural zugewiesen. Wenn der Profilparameter `MAXYEAR` auf 9999 gesetzt ist, weist Natural einer Variablen des Formats D immer eine Länge von 4 Bytes (P7) zu, und einer Variablen des Formats T eine Länge von 7 Bytes (P13).

Beispiel:

```
DEFINE DATA LOCAL
1 #DAT1 (D)
END-DEFINE
*
MOVE *DATX TO #DAT1
ADD 7 TO #DAT1
WRITE '=' #DAT1
END
```

Weitere Informationen siehe Session-Parameter `EM` und Systemvariable `*DATX` und `*TIMX`.

Der Wert in einem Datumsfeld muss im Bereich vom 1. Januar 1582 bis 31. Dezember 2699 liegen.

Format L — Logisch

Eine mit Format L definierte Variable kann als ein logisches Bedingungskriterium benutzt werden. Sie kann den Wert `TRUE` (wahr) oder `FALSE` (falsch) annehmen.

Für eine Variable des Formats L kann keine Länge angegeben werden. Einer Variable des Formats L wird stets eine Länge von 1 Byte von Natural zugewiesen.

Beispiel:

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

```
   ...
   MOVE TRUE TO #SWITCH
END-IF
```

Weitere Informationen zur logischen Wertedarstellung siehe Session-Parameter `EM`.

Format Handle of Object

Eine als `HANDLE OF OBJECT` definierte Variable kann als Objekt-Handle benutzt werden.

Weitere Informationen siehe ***NaturalX***.

Index-Notation

Eine Index-Notation wird für Felder benutzt, die ein Array darstellen.

Eine numerische Ganzzahl-Konstante oder Benutzervariable kann bei Index-Notationen eingesetzt werden. Eine Benutzervariable kann unter Benutzung von einem der folgenden Formate angegeben werden: N (numerisch), P (gepackt), I (Ganzzahl) oder B (binär), wobei das Format B nur mit einer Länge von kleiner oder gleich 4 benutzt werden kann.

Eine Systemvariable, Systemfunktion oder qualifizierte Variable kann nicht für Index-Notationen benutzt werden.

Array-Definition – Beispiele:

1. `#ARRAY (3)`

   Definiert ein eindimensionales Array mit drei Ausprägungen.

2. `FIELD (`*`label`*`.,A20/5)` oder *`label`*`.FIELD(A20/5)`

   Definiert ein Array von einem Datenbankfeld, das das durch *`label`* markierte Statement mit dem Format alphanumerisch, der Länge 20 und 5 Ausprägungen referenziert.

3. `#ARRAY (N7.2/1:5,10:12,1:4)`

   Definiert ein Array mit Format/Länge N7.2 und drei Array-Dimensionen mit 5 Ausprägungen (1 bis 5) in der ersten, 3 Ausprägungen (10 bis 12) in der zweiten und 4 Ausprägungen (1 bis 4) in der dritten Dimension.

4. `FIELD (`*`label`*`./i:i + 5)` oder *`label`*`.FIELD(i:i + 5)`

   Definiert ein Array von einem Datenbankfeld, das das durch *`label`*`.` markierte Statement referenziert.

   `FIELD` stellt ein multiples Feld oder ein Feld aus einer Periodengruppe dar, wobei *`i`* den Index für die relative Position innerhalb der Datenbank-Ausprägung angibt. Die Größe des Arrays

innerhalb des Programms ist definiert als 6 Ausprägungen (i:i + 5). Der Datenbank-Index für die relative Position ist angegeben als eine Variable, um eine Positionierung des Programm-Arrays innerhalb der Ausprägungen des multiplen Feldes oder der Periodengruppe zu ermöglichen. Für eine erneute Positionierung von *i* muss über ein `GET`- oder `GET SAME`-Statement ein neuer Aufruf der Datenbank erfolgen.

Natural ermöglicht die Definition von Arrays, bei denen der Index nicht mit `1` anfangen muss. Zur Laufzeit überprüft Natural, ob in der Referenz angegebene Indexwerte nicht die maximale Größe der in der Definition spezifizierten Dimensionen überschreiten.

Anmerkungen:

1. Aus Kompatiblitätsgründen zu früheren Versionen von Natural kann ein Array-Bereich mittels eines Bindestrichs (-) anstatt eines Doppelpunkts (:) angegeben werden.
2. Eine Mischung aus beiden Notationen ist allerdings *nicht* zulässig.
3. Die Bindestrich-Notation ist nur zulässig im Reporting Mode (aber *nicht* in einem `DEFINE DATA`-Statement).

Der maximale Indexwert ist 1.073.741.824 (1 GB).

Einfache arithmetische Ausdrücke können mittels der Operatoren „+“ und „-“ in Index-Referenzen verwendet werden. Wenn arithmetische Ausdrücke als Indizes benutzt werden, muss den Operatoren „+“ oder „-“ ein Leerzeichen vorausgehen und ihnen folgen.

Arrays in Gruppen-Strukturen werden von Natural Feld für Feld aufgelöst, und nicht Gruppen-Ausprägung für Gruppen-Ausprägung.

Beispiel für die Auflösung von Gruppen-Arrays:

```
DEFINE DATA LOCAL
 1 #GROUP (1:2)
   2 #FIELDA (A5/1:2)
   2 #FIELDB (A5)
 END-DEFINE
 ...
```

Wenn die oben definierte Gruppe in einem `WRITE`-Statement ausgegeben würde:

```
WRITE #GROUP (*)
```

würden die Ausprägungen in der folgenden Reihenfolge ausgegeben:

```
#FIELDA(1,1) #FIELDA(1,2) #FIELDA(2,1) #FIELDA(2,2) #FIELDB(1) #FIELDB(2)
```

und *nicht*:

```
#FIELDA(1,1) #FIELDA(1,2) #FIELDB(1) #FIELDA(2,1) #FIELDA(2,2) #FIELDB(2)
```

Referenzieren von Arrays — Beispiele:

1. `#ARRAY (1)`

   Referenziert die erste Ausprägung eines eindimensionalen Arrays.

2. `#ARRAY (7:12)`

   Referenziert die siebente bis zwölfte Ausprägung eines eindimensionalen Arrays.

3. `#ARRAY (i + 5)`

   Referenziert die i+fünfte Ausprägung eines eindimensionalen Arrays.

4. `#ARRAY (5,3:7,1:4)`

   Es erfolgt eine Referenz innerhalb eines dreidimensionalen Arrays auf die Ausprägung 5 in der ersten Dimension, die Ausprägungen 3 bis 7 (5 Ausprägungen) in der zweiten Dimension und 1 bis 4 (4 Ausprägungen) in der dritten Dimension.

5. Stern-Notation (*) kann benutzt werden, um alle Ausprägungen innerhalb einer Dimension zu referenzieren:

```
DEFINE DATA LOCAL
1 #ARRAY1 (N5/1:4,1:4)
1 #ARRAY2 (N5/1:4,1:4)
END-DEFINE
...
ADD #ARRAY1 (2,*) TO #ARRAY2 (4,*)
... ↩
```

Benutzung eines Schrägstrichs vor einer Array-Ausprägung

Wenn auf einen Variablennamen eine vierstellige Zahl in Klammern folgt, interpretiert Natural diese Zahl als eine Zeilennummer-Referenz auf ein Statement. Deshalb muss einer vierstelligen Array-Ausprägung ein Schrägstrich (/) vorausgehen, um anzuzeigen, dass es sich um eine Array-Ausprägung handelt, zum Beispiel:

```
#ARRAY(/1000)
```

und nicht:

```
#ARRAY(1000)
```

weil das Letztere als eine Referenz auf die Quellcode-Zeile 1000 interpretiert würde.

Wenn ein Index-Variablenname als eine Format/Längen-Angabe fehlinterpretiert werden könnte, muss ein Schrägstrich (/) benutzt werden, um anzuzeigen, dass ein Index angegeben wird. Wenn z.B. die Ausprägung eines Arrays durch den Wert der Variable `N7` definiert ist, muss die Ausprägung angegeben werden als:

```
#ARRAY (/N7)
```

und nicht:

```
#ARRAY (N7)
```

weil das Letztere als die Definition eines 7 Bytes umfassenden numerischen Feldes fehlinterpretiert würde.

Datenbank-Array referenzieren

Folgende Themen werden behandelt:

- Multiple Felder und Periodengruppen-Felder referenzieren
- Mit Konstanten definierte Arrays referenzieren
- Mit Variablen definierte Arrays referenzieren
- Mehrfach definierte Arrays referenzieren

Anmerkung: Bevor Sie die folgenden Beispielprogramme ausführen, starten Sie bitte das Programm `INDEXTST` in der Library `SYSEXPG`, um einen Beispielsatz zu erstellen, der 10 verschiedene Sprachcodes verwendet.

Multiple Felder und Periodengruppen-Felder referenzieren

Ein multiples Feld oder ein Periodengruppen-Feld innerhalb eines Views bzw. einer DDM kann mittels verschiedener Index-Notationen definiert werden.

Zum Beispiel, der erste bis zehnte Wert und der Ite bis Ite+10 Wert desselben multiplen Feldes/Periodengruppen-Feldes eines Datenbanksatzes:

```
DEFINE DATA LOCAL
1 I (I2)
1 EMPLOY-VIEW VIEW OF EMPLOYEES
  2 LANG (1:10)
  2 LANG (I:I+10)
END-DEFINE
```

oder:

```
RESET I (I2)
...
READ EMPLOYEES
OBTAIN LANG(1:10) LANG(I:I+10)
```

Anmerkungen:

1. Derselbe Index der unteren Grenze kann nur einmal pro Array benutzt werden (dies gilt für Konstanten-Indizes sowie für Variablen-Indizes).
2. Für eine Array-Definition unter Benutzung eines Variablen-Indexes muss die untere Grenze mittels der Variable selbst angegeben werden, und die obere Grenze muss mittels derselben Variable plus einer Konstante angegeben werden.

Mit Konstanten definierte Arrays referenzieren

Ein mit Konstanten definiertes Array kann entweder mittels Konstanten oder Variablen referenziert werden. Die obere Grenze des Array kann nicht überschritten werden. Die obere Grenze wird von Natural zur Kompilierungszeit geprüft, wenn eine Konstante benutzt wird.

Beispiel für den Reporting Mode:

```
** Example 'INDEX1R': Array definition with constants  (reporting mode)
************************************************************************
*
READ (1) EMPLOYEES WITH NAME = 'WINTER' WHERE CITY = 'LONDON'
  OBTAIN LANG (1:10)
  /*
  WRITE 'LANG(1:10):' LANG (1:10) //
  WRITE 'LANG(1)   :' LANG (1)    /  'LANG(5:9) :' LANG (5:9)
LOOP
```

```
*
END
```

Beispiel für den Structured Mode:

```
** Example 'INDEX1S': Array definition with constants (structured mode)
************************************************************************
DEFINE DATA LOCAL
1 EMPLOY-VIEW VIEW OF EMPLOYEES
  2 NAME
  2 FIRST-NAME
  2 CITY
  2 LANG (1:10)
END-DEFINE
*
READ (1) EMPLOY-VIEW WITH NAME = 'WINTER' WHERE CITY = 'LONDON'
  WRITE 'LANG(1:10):' LANG (1:10) //
  WRITE 'LANG(1)   :' LANG (1)    /  'LANG(5:9) :' LANG (5:9)
END-READ
END
```

Wenn ein multiples Feld oder ein Periodengruppen-Feld mehrmals mittels Konstanten definiert wird und mittels Variablen referenziert werden soll, wird die folgende Syntax benutzt.

Beispiel für den Reporting Mode:

```
** Example 'INDEX2R': Array definition with constants (reporting mode)
**                   (multiple definition of same database field)
************************************************************************
DEFINE DATA LOCAL
1 EMPLOY-VIEW VIEW OF EMPLOYEES
  2 NAME
  2 CITY
  2 LANG (1:5)
  2 LANG (4:8)
END-DEFINE
*
READ (1) EMPLOY-VIEW WITH NAME = 'WINTER' WHERE CITY = 'LONDON'
  DISPLAY 'NAME'         NAME
          'LANGUAGE/1:3' LANG (1.1:3)
          'LANGUAGE/6:8' LANG (4.3:5)
LOOP
*
END
```

Beispiel für den Structured Mode:

```
** Example 'INDEX2S': Array definition with constants (structured mode)
**                   (multiple definition of same database field)
************************************************************************
DEFINE DATA LOCAL
1 EMPLOY-VIEW VIEW OF EMPLOYEES
  2 NAME
  2 CITY
  2 LANG (1:5)
  2 LANG (4:8)
END-DEFINE
*
READ (1) EMPLOY-VIEW WITH NAME = 'WINTER' WHERE CITY = 'LONDON'
  DISPLAY 'NAME'         NAME
          'LANGUAGE/1:3' LANG (1.1:3)
          'LANGUAGE/6:8' LANG (4.3:5)
END-READ
*
END
```

Mit Variablen definierte Arrays referenzieren

Mit Variablen definierte multiple Felder oder Periodengruppen-Felder in Arrays müssen mittels derselben Variable referenziert werden.

Beispiel für den Reporting Mode:

```
** Example 'INDEX3R': Array definition with variables (reporting mode)
************************************************************************
RESET I (I2)
*
I := 1
READ (1) EMPLOYEES WITH NAME = 'WINTER' WHERE CITY = 'LONDON'
  OBTAIN LANG (I:I+10)
  /*
  WRITE 'LANG(I)      :' LANG (I) /
        'LANG(I+5:I+7):' LANG (I+5:I+7)
LOOP
*
END
```

Beispiel für den Structured Mode:

```
** Example 'INDEX3S': Array definition with variables (structured mode)
************************************************************************
DEFINE DATA LOCAL
1 I (I2)
*
1 EMPLOY-VIEW VIEW OF EMPLOYEES
  2 NAME
  2 CITY
  2 LANG (I:I+10)
END-DEFINE
*
I := 1
READ (1) EMPLOY-VIEW WITH NAME = 'WINTER' WHERE CITY = 'LONDON'
  WRITE 'LANG(I)      :' LANG (I) /
        'LANG(I+5:I+7):' LANG (I+5:I+7)
END-READ
END
```

Wenn ein anderer Index benutzt werden soll, muss eine eindeutige Referenz auf die zuerst gefundene Definition des Arrays mit variablem Index erfolgen. Qualifizieren Sie dazu den Index-Ausdruck wie im Folgenden gezeigt.

Beispiel für den Reporting Mode:

```
** Example 'INDEX4R': Array definition with variables (reporting mode)
************************************************************************
RESET I (I2) J (I2)
*
I := 2
J := 3
*
READ (1) EMPLOYEES WITH NAME = 'WINTER' WHERE CITY = 'LONDON'
  OBTAIN LANG (I:I+10)
  /*
  WRITE 'LANG(I.J)  :' LANG (I.J) /
        'LANG(I.1:5):' LANG (I.1:5)
LOOP
*
END
```

Beispiel für den Structured Mode:

```
** Example 'INDEX4S': Array definition with variables (structured mode)
************************************************************************
DEFINE DATA LOCAL
1 I (I2)
1 J (I2)
1 EMPLOY-VIEW VIEW OF EMPLOYEES
  2 NAME
  2 CITY
  2 LANG (I:I+10)
END-DEFINE
*
I := 2
J := 3
READ (1) EMPLOY-VIEW WITH NAME = 'WINTER' WHERE CITY = 'LONDON'
  WRITE 'LANG(I.J)  :' LANG (I.J) /
        'LANG(I.1:5):' LANG (I.1:5)
END-READ
END
```

Der Ausdruck `I.` wird benutzt, um eine eindeutige Referenz auf die Array-Definition und „Positionen“ zum ersten Wert innerhalb des Array-Lesebereichs (`LANG(I.1:5)`) zu erstellen.

Der aktuelle Inhalt von `I` zum Zeitpunkt des Datenbankzugriffs legt die Start-Ausprägung des Datenbank-Arrays fest.

Mehrfach definierte Arrays referenzieren

Für mehrfach definierte Arrays ist gewöhnlich eine Referenz mit Qualifizierung des Index-Ausdrucks erforderlich, um eine eindeutige Referenz auf den gewünschten Array-Bereich sicherzustellen.

Beispiel für den Reporting Mode:

```
** Example 'INDEX5R': Array definition with constants (reporting mode)
**                   (multiple definition of same database field)
************************************************************************
DEFINE DATA LOCAL                   /* For reporting mode programs
1 EMPLOY-VIEW VIEW OF EMPLOYEES     /* DEFINE DATA is recommended
  2 NAME                            /* to use multiple definitions
  2 CITY                            /* of same database field
  2 LANG (1:10)
  2 LANG (5:10)
*
1 I (I2)
1 J (I2)
END-DEFINE
*
I := 1
```

```
J := 2
*
READ (1) EMPLOY-VIEW WITH NAME = 'WINTER' WHERE CITY = 'LONDON'
  WRITE 'LANG(1.1:10) :' LANG (1.1:10) /
        'LANG(1.I:I+2):' LANG (1.I:I+2) //
  WRITE 'LANG(5.1:5)  :' LANG (5.1:5)  /
        'LANG(5.J)    :' LANG (5.J)
LOOP
END
```

Beispiel für den Structured Mode:

```
** Example 'INDEX5S': Array definition with constants (structured mode)
**                    (multiple definition of same database field)
************************************************************************
DEFINE DATA LOCAL
1 EMPLOY-VIEW VIEW OF EMPLOYEES
  2 NAME
  2 CITY
  2 LANG (1:10)
  2 LANG (5:10)
*
1 I (I2)
1 J (I2)
END-DEFINE
*
*
I := 1
J := 2
*
READ (1) EMPLOY-VIEW WITH NAME = 'WINTER' WHERE CITY = 'LONDON'
  WRITE 'LANG(1.1:10) :' LANG (1.1:10) /
        'LANG(1.I:I+2):' LANG (1.I:I+2) //
  WRITE 'LANG(5.1:5)  :' LANG (5.1:5)  /
        'LANG(5.J)    :' LANG (5.J)
END-READ
END
```

Eine ähnliche Syntax wird auch benutzt, wenn multiple Felder oder Periodengruppen-Felder unter Benutzung von Index-Variablen definiert werden.

Beispiel für den Reporting Mode:

```
** Example 'INDEX6R': Array definition with variables (reporting mode)
**                   (multiple definition of same database field)
************************************************************************
DEFINE DATA LOCAL
1 I (I2) INIT <1>
1 J (I2) INIT <2>
1 N (I2) INIT <1>
1 EMPLOY-VIEW VIEW OF EMPLOYEES    /* For reporting mode programs
  2 NAME                           /* DEFINE DATA is recommended
  2 CITY                           /* to use multiple definitions
  2 LANG (I:I+10)                  /* of same database field
  2 LANG (J:J+5)
  2 LANG (4:5)
*
END-DEFINE
*
READ (1) EMPLOY-VIEW WITH NAME = 'WINTER' WHERE CITY = 'LONDON'
*
  WRITE 'LANG(I.I)    :' LANG (I.I) /
        'LANG(1.I:I+2):' LANG (I.I:I+10) //
*
  WRITE 'LANG(J.N)    :' LANG (J.N) /
        'LANG(J.2:4)  :' LANG (J.2:4) //
*
  WRITE 'LANG(4.N)    :' LANG (4.N) /
        'LANG(4.N:N+1):' LANG (4.N:N+1) /
LOOP
END
```

Beispiel für den Structured Mode:

```
** Example 'INDEX6S': Array definition with variables (structured mode)
**                   (multiple definition of same database field)
************************************************************************
DEFINE DATA LOCAL
1 I (I2) INIT <1>
1 J (I2) INIT <2>
1 N (I2) INIT <1>
1 EMPLOY-VIEW VIEW OF EMPLOYEES
  2 NAME
  2 CITY
  2 LANG (I:I+10)
  2 LANG (J:J+5)
  2 LANG (4:5)
*
END-DEFINE
*
READ (1) EMPLOY-VIEW WITH NAME = 'WINTER' WHERE CITY = 'LONDON'
*
```

```
  WRITE 'LANG(I.I)    :' LANG (I.I) /
        'LANG(1.I:I+2):' LANG (I.I:I+10) //
*
  WRITE 'LANG(J.N)    :' LANG (J.N) /
        'LANG(J.2:4)  :' LANG (J.2:4) //
*
  WRITE 'LANG(4.N)    :' LANG (4.N) /
        'LANG(4.N:N+1):' LANG (4.N:N+1) /
END-READ
END
```

Internen Zähler für ein Datenbank-Array referenzieren — C*-Notation

Es ist manchmal erforderlich, ein multiples Feld und/oder eine Periodengruppe zu referenzieren, ohne zu wissen, wie viele Werte/Ausprägungen in einem gegebenen Datensatz vorhanden sind. Adabas unterhält einen internen Zähler der Anzahl für die Werte jedes einzelnen multiplen Feldes und die Anzahl der Ausprägungen jeder einzelnen Periodengruppe. Dieser Zähler kann unter Angabe von `C*` unmittelbar vor dem Feldnamen referenziert werden.

Anmerkung bezüglich andere Datenbanken als Adabas:

SQL	Bei SQL-Datenbanken kann die `C*` Notation nicht verwendet werden.
VSAM	Bei VSAM-Datenbanken gibt die `C*` Notation nicht die Anzahl der Werte/Ausprägungen zurück, sondern die/den maximale/n Ausprägung/Wert, wie in der DDM definiert (`MAXOCC`).

Siehe auch das Zeilenkommando `.*` des Datenbereich-Editors (Data-Area-Editor) in der *Editoren*-Dokumentation.

Das/Die zum Deklarieren eines `C*`-Feldes zulässige explizite Format und Länge ist entweder

- Ganzzahl-Format Integer (I) mit einer Länge von 2 Bytes (I2) oder 4 Bytes (I4),
- numerisch (N) oder gepackt (P) mit einer ganzzahligen Ziffer (aber ohne Dezimalstellen), zum Beispiel (N3).

Wenn kein explizites Format und keine explizite Länge angegeben wird, ist Format/Länge N3 die Voreinstellung.

Beispiele:

C*LANG	gibt den Zähler der Anzahl der Werte für das multiple Feld LANG zurück.
C*INCOME	gibt den Zähler der Anzahl der Ausprägungen für die Periodengruppe INCOME zurück.
C*BONUS(1)	gibt den Zähler der Anzahl der Werte für das multiple Feld BONUS in der Periodengruppen-Ausprägung 1 zurück (wenn man davon ausgeht, dass BONUS ein multiples Fel.d innerhalb einer Periodengruppe ist.)

Beispielprogramm mit C*-Notation bei Variablen:

```
** Example 'CNOTX01': C* Notation
************************************************************************
DEFINE DATA LOCAL
1 EMPL-VIEW VIEW OF EMPLOYEES
  2 NAME
  2 CITY
  2 C*INCOME
  2 INCOME
    3 SALARY  (1:5)
    3 C*BONUS (1:2)
    3 BONUS   (1:2,1:2)
  2 C*LANG
  2 LANG      (1:2)
*
1 #I (N1)
END-DEFINE
*
LIMIT 2
READ EMPL-VIEW BY CITY
  /*
  WRITE NOTITLE 'NAME:' NAME /
       'NUMBER OF LANGUAGES SPOKEN:' C*LANG 5X
       'LANGUAGE 1:' LANG (1) 5X
       'LANGUAGE 2:' LANG (2)
  /*
  WRITE 'SALARY DATA:'
  FOR #I FROM 1 TO C*INCOME
    WRITE 'SALARY' #I SALARY (1.#I)
  END-FOR
  /*
  WRITE 'THIS YEAR BONUS:' C*BONUS(1)  BONUS (1,1) BONUS (1,2)
      / 'LAST YEAR BONUS:' C*BONUS(2)  BONUS (2,1) BONUS (2,2)
  SKIP 1
END-READ
END
```

Ausgabe des Programms CNOTX01:

```
NAME: SENKO
NUMBER OF LANGUAGES SPOKEN:    1     LANGUAGE 1: ENG     LANGUAGE 2:
SALARY DATA:
SALARY  1      36225
SALARY  2      29900
SALARY  3      28100
SALARY  4      26600
SALARY  5      25200
THIS YEAR BONUS:    0          0          0
LAST YEAR BONUS:    0          0          0

NAME: CANALE
NUMBER OF LANGUAGES SPOKEN:    2     LANGUAGE 1: FRE     LANGUAGE 2: ENG
SALARY DATA:
SALARY  1     202285
THIS YEAR BONUS:    1      23000          0
LAST YEAR BONUS:    0          0          0
```

C*-Notation bei multiplen Feldern innerhalb von Periodengruppen

Für ein multiples Feld innerhalb einer Periodengruppe können Sie auch eine mit `C*`-Notation versehene Variable mit einer Indexbereichsangabe definieren.

Bei den folgenden Beispielen wird das multiple Feld `BONUS` benutzt, das Teil der Periodengruppe `INCOME` ist. Alle drei Beispiele liefern dasselbe Ergebnis.

Beispiel 1 — Reporting Mode:

```
** Example 'CNOTX02': C* Notation (multiple-value fields)
************************************************************************
*
LIMIT 2
READ EMPLOYEES BY CITY
  OBTAIN C*BONUS (1:3)
         BONUS   (1:3,1:3)
  /*
  DISPLAY NAME C*BONUS (1:3) BONUS (1:3,1:3)
LOOP
*
END
```

Beispiel 2 — Structured Mode:

```
** Example 'CNOTX03': C* Notation (multiple-value fields)
************************************************************************
DEFINE DATA LOCAL
1 EMPL-VIEW VIEW OF EMPLOYEES
  2 NAME
  2 CITY
  2 INCOME    (1:3)
    3 C*BONUS
    3 BONUS   (1:3)
END-DEFINE
*
LIMIT 2
READ EMPL-VIEW BY CITY
  /*
  DISPLAY NAME C*BONUS (1:3) BONUS (1:3,1:3)
END-READ
*
END
```

Beispiel 3 — Structured Mode:

```
** Example 'CNOTX04': C* Notation (multiple-value fields)
************************************************************************
DEFINE DATA LOCAL
1 EMPL-VIEW VIEW OF EMPLOYEES
  2 NAME
  2 CITY
  2 C*BONUS  (1:3)
  2 INCOME   (1:3)
    3 BONUS  (1:3)
END-DEFINE
*
LIMIT 2
READ EMPL-VIEW BY CITY
  /*
  DISPLAY NAME C*BONUS (*) BONUS (*,*)
END-READ
*
END
```

Vorsicht: Da der Adabas-Formatpuffer keine Bereiche für Zählerfelder zulässt, werden sie als einzelne Felder generiert; deshalb kann ein `C*`-Indexbereich für ein großes Array zu einem Adabas Formatpuffer-Überlauf führen.

Datenstrukturen kennzeichnen

Um ein Feld beim Referenzieren zu identifizieren, können Sie das Feld mit einem Kennzeichner (Qualifier) versehen, indem Sie vor dem Feldnamen den Namen des Level-1-Datenelementes, in dem das Feld sich befindet, und einen Punkt eingeben.

Ist ein Feld nicht eindeutig über den Namen zu identifizieren (z.B. wenn derselbe Feldname in mehreren Gruppen/Views benutzt wird), müssen Sie es beim Referenzieren kennzeichnen.

Die Kombination von Level-1-Datenelement und Feldnamen muss eindeutig sein.

Beispiel:

```
DEFINE DATA LOCAL
1 FULL-NAME
  2 LAST-NAME (A20)
  2 FIRST-NAME (A15)
1 OUTPUT-NAME
  2 LAST-NAME (A20)
  2 FIRST-NAME (A15)
END-DEFINE
...
MOVE FULL-NAME.LAST-NAME TO OUTPUT-NAME.LAST-NAME
...
```

Der Kennzeichner muss ein Level-1-Datenelement sein.

Beispiel:

```
DEFINE DATA LOCAL
1 GROUP1
  2 SUB-GROUP
    3 FIELD1 (A15)
    3 FIELD2 (A15)
END-DEFINE
...
MOVE 'ABC' TO GROUP1.FIELD1
...
```

Datenbankfeld kennzeichnen:

Benutzen Sie denselben Namen für eine Benutzervariable und ein Datenbankfeld (was Sie nicht tun sollten), müssen Sie das Datenbankfeld kennzeichnen, wenn Sie es referenzieren wollen.

Vorsicht: Wenn Sie ein Datenbankfeld mit gleichem Namen wie eine Benutzervariable nicht kennzeichnen, wird die Benutzervariable referenziert.

Beispiele für Benutzervariablen

```
DEFINE DATA LOCAL
1 #A1 (A10)       /* Alphanumeric, 10 positions.
1 #A2 (B4)        /* Binary, 4 positions.
1 #A3 (P4)        /* Packed numeric, 4 positions and 1 sign position.
1 #A4 (N7.2)      /* Unpacked numeric,
                  /* 7 positions before and 2 after decimal point.
1 #A5 (N7.)       /* Invalid definition!!!
1 #A6 (P7.2)     /* Packed numeric, 7 positions before and 2 after decimal point
                  /* and 1 sign position.
1 #INT1 (I1)      /* Integer, 1 byte.
1 #INT2 (I2)      /* Integer, 2 bytes.
1 #INT3 (I3)      /* Invalid definition!!!
1 #INT4 (I4)      /* Integer, 4 bytes.
1 #INT5 (I5)      /* Invalid definition!!!
1 #FLT4 (F4)      /* Floating point, 4 bytes.
1 #FLT8 (F8)      /* Floating point, 8 bytes.
1 #FLT2 (F2)      /* Invalid definition!!!
1 #DATE (D)       /* Date (internal format/length P6).
1 #TIME (T)       /* Time (internal format/length P12).
1 #SWITCH (L)     /* Logical, 1 byte (TRUE or FALSE).
                  /*
END-DEFINE ↩
```

25 Dynamische Variablen

Sinn und Zweck dynamischer Variablen

Insofern als die maximale Größe der großen Datenstrukturen (zum Beispiel Bilder, Audiosignale, Videos) zur Anwendungsentwicklungszeit nicht genau bekannt sein kann, bietet Natural zusätzlich die Definition alphanumerischer und binärer Variablen mit dem Attribut `DYNAMIC`.

Der Wertebereich von Variablen, die mit diesem Attribut definiert sind, wird zur Ausführungszeit dynamisch erweitert, wenn es notwendig wird (zum Beispiel bei einer Zuweisungsoperation: `#picture1 := #picture2`). Dies bedeutet, dass große binäre und alphanumerische Datenstrukturen in Natural verarbeitet werden können, ohne dass Sie eine Beschränkung zur Entwicklungszeit definieren müssen.

Die Zuweisung von dynamischen Variablen zur Ausführungszeit unterliegt natürlich den Beschränkungen des verfügbaren Hauptspeichers. Wenn die Zuweisung dynamischer Variablen dazu führt, dass eine Meldung wegen unzureichenden Hauptspeichers vom zugrundeliegenden Betriebssystem zurückgegeben wird, kann das `ON ERROR`-Statement benutzt werden, um diese Fehlerbedingung abzufangen, sonst wird eine Fehlermeldung von Natural zurückgegeben.

Die Natural-Systemvariable `*LENGTH` kann benutzt werden, um die Länge (in Code- Einheiten) des Wertespeichers zu erhalten, der zur Zeit für eine gegebene dynamische Variable benutzt wird. Für Format A und B ist die Größe einer Code-Einheit 1 Byte. Für Format U ist die Größe einer Code-Einheit 2 Bytes (UTF−16). Natural setzt `*LENGTH` automatisch auf die Länge des Source-Operanden bei Zuweisungen, in denen die dynamische Variable betroffen ist. `*LENGTH(`*`field`*`)` gibt deshalb die aktuell für ein/e dynamische/s Natural-Feld oder eine Variable in Code-Einheiten benutzte Größe zurück.

Wenn der Speicherplatz für die dynamische Variable nicht mehr erforderlich ist, kann das `REDUCE`- oder `RESIZE`-Statement benutzt werden, um den für die dynamische Variable benutzten Speicherplatz auf Null (oder eine andere gewünschte Größe) zu reduzieren. Wenn die obere Grenze der Hauptspeicher-Benutzung für eine spezifische dynamische Variable bekannt ist, kann das `EXPAND`-Statement benutzt werden, um den für die dynamische Variable benutzten Speicherplatz auf diese bestimmte Größe zu setzen.

Große Variablen für alphanumerische und binäre Daten basieren auf den bekannten Natural-Formaten A und B. Die Beschränkungen von 253 Bytes für Format A und 126 für Format B sind nicht mehr gültig. Die neue Größenbeschränkung ist 1 GB. Diese großen statischen Variablen und Felder werden genauso verarbeitet wie traditionelle alphanumerische und binäre Variablen und Felder in Bezug auf Definition, Redefinition, Wertespeicherzuweisung, Konvertierungen, Referenzierungen in Statements, usw. Alle Regeln zu alphanumerischen und binären Formaten gelten für diese großen Formate.

Soll eine dynamische Variable initialisiert werden, sollte das `MOVE ALL UNTIL`-Statement für diesen Zweck benutzt werden.

Definition dynamischer Variablen

Da die wirkliche Größe von großen alphanumerischen und binären Datenstrukturen zur Anwendungsentwicklungszeit nicht genau bekannt sein mag, kann die Definition *dynamischer* Variablen des Formats A, B oder U zur Verwaltung dieser Strukturen benutzt werden. Die dynamische Zuweisung und Erweiterung (Neuzuweisung) großer Variablen ist gegenüber der Anwendungsprogrammierungslogik transparent. Dynamische Variablen werden ohne Längenangabe definiert. Der Hauptspeicher wird entweder implizit zur Ausführungszeit zugewiesen, wenn die dynamische Variable als ein Ziel-Operand benutzt wird, oder explizit mit einem `EXPAND`- oder `RESIZE`-Statement.

Dynamische Variablen können nur in einem `DEFINE DATA`-Statement mittels der folgenden Syntax definiert werden:

```
level variable-name( A ) DYNAMIC
level variable-name( B ) DYNAMIC
level variable-name( U ) DYNAMIC
```

Dabei ist:

- *level* die Stufennummer
- *variable-name* der Name der großen Variablen
- *format* das Format der Variablen (A, U oder B)
- DYNAMIC das Schlüsselwort, durch das die Variable als dynamsiche Variable definiert wird.

Einschränkungen:

Die folgenden Einschränkungen gelten für eine dynamische Variable:

- Eine Redefinition einer dynamischen Variable ist nicht zulässig.
- Eine dynamische Variable darf nicht in einer `REDEFINE`-Klausel enthalten sein.

Zurzeit für eine dynamische Variable benutzter Wertespeicher

Die Größe (in Code-Einheiten) des zurzeit benutzten Wertespeichers einer dynamischen Variable kann aus der Systemvariable `*LENGTH` übernommen werden. `*LENGTH` wird bei Zuweisungen automatisch auf die (verwendete) Länge des Source-Operanden gesetzt.

Vorsicht: Aus Performance-Gründen kann der zur Aufnahme des Wertes der dynamischen Variablen zugewiesene Speicherbereich größer sein als der Wert von `*LENGTH` (benutzte

Größe steht dem Entwickler zur Verfügung). Sie sollten sich nicht auf den Speicherplatz verlassen, der über die benutzte Länge (wie durch `*LENGTH` angegeben) hinaus zugewiesen wird. Dieser Platz kann jederzeit freigegeben werden, auch wenn auf die betreffende dynamische Variable nicht zugegriffen wird. Es ist für den Natural-Programmierer nicht möglich, Informationen über die aktuell zugewiesene Größe zu erhalten. Diese Größe ist ein interner Wert.

Die Systemvariable `*LENGTH(`*`field`*`)` gibt die benutzte Länge eines dynamischen Natural-Feldes oder einer solchen Variable in Code-Einheiten zurück. `*LENGTH` kann nur benutzt werden, um die aktuell verwendete Länge für dynamische Variablen zu erhalten.

Hauptspeicherplatz für eine dynamische Variable zuweisen/freigeben

Die Statements `EXPAND`, `REDUCE` und `RESIZE` werden benutzt, um Hauptspeicherplatz für eine dynamische Variable explizit zuzuweisen und freizugeben.

Syntax:

```
EXPAND [SIZE OF] DYNAMIC [VARIABLE] operand1 TO operand2
REDUCE [SIZE OF] DYNAMIC [VARIABLE] operand1 TO operand2
RESIZE [SIZE OF] DYNAMIC [VARIABLE] operand1 TO operand2
```

dabei ist *operand1* eine dynamische Variable und *operand2* ein nicht-negativer, numerischer Längenwert ist.

EXPAND

Funktion

Das `EXPAND`-Statement wird benutzt, um den zurzeit *zugewiesenen* Speicherplatz der dynamischen Variable (*operand1*) auf die angegebene Größe (*operand2*) zu erweitern.

Angegebene Größe ändern

Die aktuell benutzte Größe (siehe Natural-Systemvariable `*LENGTH` weiter **oben**) für die dynamische Variable wird nicht geändert.

Wenn die angegebene Größe (*operand2*) kleiner ist als die Größe des aktuell zugewiesenen Speicherplatzes der dynamischen Variable, wird das Statement ignoriert.

REDUCE

Funktion

Das `REDUCE`-Statement wird benutzt, um die Größe des zurzeit *zugewiesenen* Speicherplatzes der dynamischen Variable (*operand1*) auf die angegebene Größe (*operand2*) zu reduzieren.

Der über die angegebene Größe (*operand2*) hinausgehende, für die dynamische Variable (*operand1*) zugewiesene Speicherplatz kann jederzeit freigegeben werden, wenn das Statement ausgeführt wird, oder auch später.

Angegebene Länge ändern

Wenn die zurzeit *benutzte* Speichergröße (siehe Natural-Systemvariable `*LENGTH` weiter **oben**) für die dynamische Variable größer ist als die angegebene Speichergröße (*operand2*), wird `*LENGTH` dieser dynamischen Variable auf die angegebene Größe gesetzt. Der Inhalt der Variablen wird abgeschnitten, aber nicht geändert.

Wenn die angegebene Größe größer ist als der zurzeit zugewiesene Speicherplatz der dynamischen Variable, wird das Statement ignoriert.

RESIZE

Funktion

Das `RESIZE`-Statement passt die Größe des zurzeit zugewiesenen Speicherplatzes der dynamischen Variable (*operand1*) an die angegebene Größe (*operand2*) an.

Angegebene Länge ändern

Wenn die angegebene Länge der dynamischen Variable kleiner ist als die benutzte Größe (siehe Natural-Systemvariable `*LENGTH` weiter **above**), wird die benutzte Größe dementsprechend reduziert.

Wenn die *angegebene* Länge größer ist als die Größe des zurzeit zugewiesenen Speicherplatzes der dynamischen Variable, wird die Größe des zugewiesenen Speicherplatzes der dynamischen Variable erhöht. Die aktuell benutzte Größe (siehe `*LENGTH`) der dynamischen Variable ist davon nicht betroffen und bleibt unverändert.

Wenn die angegebene Länge identisch ist mit der Größe des zurzeit zugewiesenen Speicherplatzes der dynamischen Variable, hat die Ausführung des `RESIZE`-Statements keine Auswirkungen.

26 Dynamische und große Variablen benutzen

Allgemeine Informationen zu dynamischen Variablen

Generell gilt Folgendes:

- Eine dynamische alphanumerische Variable kann immer dann benutzt werden, wenn ein alphanumerisches Feld zulässig ist.
- Ein dynamisches binäres Feld kann immer dann benutzt werden, wenn ein binäres Feld erlaubt ist.
- Ein dynamisches Unicode-Feld kann überall dort verwendet werden, wo ein Unicode-Feld erlaubt ist.

Ausnahme:

Dynamische Variablen sind nicht zulässig beim `SORT`-Statement. Um dynamische Variablen in einem `DISPLAY`-, `WRITE`-, `PRINT`-, R`EINPUT`- oder `INPUT`-Statement zu benutzen, müssen Sie entweder den Session-Parameter `AL` oder `EM` benutzen, um die Länge der Variable zu definieren.

Die benutzte Länge (siehe Natural-Systemvariable `*LENGTH`, ***Aktuell für eine dynamische Variable benutzter Wertespeicher***) und die Größe des zugewiesenen Speicherplatzes der dynamischen Variablen sind gleich Null, bis auf die Variable als Ziel-Operand zum ersten Mal zugegriffen wird. Aufgrund von Zuweisungen oder anderen Operationen können dynamische Variablen zuerst zugewiesen oder auf die exakte Größe des Source-Operanden erweitert werden (neu zugewiesen).

Die Größe einer dynamischen Variable kann mit den folgenden Statements erweitert werden, wenn sie als ein änderbarer Operand (Ziel-Operand) in den folgenden Statements benutzt wird:

`ASSIGN`	*`operand1`* (Ziel-Operand in einer Zuweisung).
`CALLNAT`	Siehe *Parameter-Übertragung bei dynamischen Variablen* (außer wenn `AD=O`, oder wenn `BY VALUE` in der entsprechenden PDA vorhanden ist)
`COMPRESS`	*`operand2`*, siehe *Verarbeitung*.
`EXAMINE`	*`operand1`* in der `DELETE REPLACE`-Klausel.
`MOVE`	*`operand2`* (Ziel-Operand), siehe *Funktion*.
`PERFORM`	Außer wenn `AD=O`, oder wenn `BY VALUE` in der betreffenden PDA vorhanden ist.
`READ WORK FILE`	*`operand1`* und *`operand2`*, siehe *Verarbeitung großer und dynamischer Variablen*.
`SEPARATE`	*`operand4`*.
`SELECT` (SQL)	*`parameter`* in der *`INTO`-Klausel*.
`SEND METHOD`	*`operand3`* (außer wenn `AD=O`).

Zurzeit gibt es die folgende Beschränkung in Bezug auf die Verwendung von großen Variablen:

CALL	Parameter-Größe kleiner als 64 KB pro Parameter (keine Beschränkung für `CALL` mit `INTERFACE4`-Option).

In den folgenden Abschnitten wird die Benutzung der dynamischen Variablen detaillierter und mit Beispielen erörtert.

Zuweisungen mit dynamischen Variablen

Im Allgemeinen wird eine Zuweisung in der zurzeit benutzten Länge des Source-Operanden durchgeführt (siehe Natural-Systemvariable `*LENGTH`). Wenn der Ziel-Operand eine dynamische Variable ist, wird seine zurzeit zugewiesene Größe gegebenenfalls erweitert, um den Source-Operanden ohne Abschneiden zu verschieben.

Beispiel:

```
#MYDYNTEXT1 := OPERAND
MOVE OPERAND TO #MYDYNTEXT1
/* #MYDYNTEXT1 IS AUTOMATICALLY EXTENDED UNTIL THE SOURCE OPERAND CAN BE COPIED ↩
```

`MOVE ALL` bzw. `MOVE ALL UNTIL` mit dynamischen Ziel-Operanden wird wie folgt definiert:

- `MOVE ALL` verschiebt den Source-Operanden wiederholt zum Ziel-Operanden, bis die benutzte Länge (`*LENGTH`) des Ziel-Operanden erreicht ist. `*LENGTH` wird nicht geändert. Wenn `*LENGTH` Null ist, wird das Statement ignoriert.
- `MOVE ALL` *operand1* `TO` *operand2* `UNTIL operand3` verschiebt *operand1* wiederholt zu *operand2*, bis die in *operand3* angegebene Länge erreicht ist. Wenn *operand3* größer als `*LENGTH(`*operand2*`)` ist, wird *operand2* erweitert, und `*LENGTH(`*operand2*`)` wird auf *operand3* gesetzt. Wenn *operand3* kleiner als `*LENGTH(`*operand2*`)` ist, wird die benutzte Länge auf *operand3* reduziert. Wenn *operand3* gleich `*LENGTH(`*operand2*`)` ist, entspricht das Verhalten dem bei `MOVE ALL`.

Beispiel:

```
#MYDYNTEXT1 := 'ABCDEFGHIJKLMNO'           /* *LENGTH(#MYDYNTEXT1) = 15
MOVE ALL 'AB' TO #MYDYNTEXT1               /* CONTENT OF #MYDYNTEXT1 = ↩
'ABABABABABABABA';
                                           /* *LENGTH IS STILL 15
MOVE ALL 'CD' TO #MYDYNTEXT1 UNTIL 6       /* CONTENT OF #MYDYNTEXT1 = 'CDCDCD';
                                           /* *LENGTH = 6
MOVE ALL 'EF' TO #MYDYNTEXT1 UNTIL 10     /* CONTENT OF #MYDYNTEXT1 = 'EFEFEFEFEF';
                                           /* *LENGTH = 10
```

`MOVE JUSTIFIED` wird zur Kompilierungszeit zurückgewiesen, wenn der Ziel-Operand eine dynamische Variable ist.

`MOVE SUBSTR` und `MOVE TO SUBSTR` sind zulässig. `MOVE SUBSTR` führt zu einem Laufzeitfehler, wenn ein Substring hinter der benutzten Länge einer dynamischen Variable (`*LENGTH`) referenziert wird. `MOVE TO SUBSTR` führt zu einem Laufzeitfehler, wenn eine Substring-Position hinter `*LENGTH + 1` referenziert wird, weil dies zu einer undefinierten Lücke im Inhalt der dynamischen Variable führen würde. Wenn der Ziel-Operand von `MOVE TO SUBSTR` erweitert werden sollte (zum Beispiel, wenn der zweite Operand auf `*LENGTH+1` gesetzt wird), ist der dritte Operand zwingend.

Gültige Syntax:

```
#OP2 := *LENGTH(#MYDYNTEXT1)
MOVE SUBSTR (#MYDYNTEXT1, #OP2) TO OPERAND                /* MOVE LAST CHARACTER ↩
TO OPERAND
#OP2 := *LENGTH(#MYDYNTEXT1) + 1
MOVE OPERAND TO SUBSTR(#MYDYNTEXT1, #OP2, #1EN_OPERAND)   /* CONCATENATE OPERAND ↩
TO #MYDYNTEXT1
```

Ungültige Syntax:

```
#OP2 := *LENGTH(#MYDYNTEXT1) + 1
MOVE SUBSTR (#MYDYNTEXT1, #OP2, 10) TO OPERAND          /* LEADS TO RUNTIME ERROR; ↩
UNDEFINED SUB-STRING
#OP2 := *LENGTH(#MYDYNTEXT1 + 10)
MOVE OPERAND TO SUBSTR(#MYDYNTEXT1, #OP2, #EN_OPERAND)   /* LEADS TO RUNTIME ERROR; ↩
UNDEFINED GAP
#OP2 := *LENGTH(#MYDYNTEXT1) + 1
MOVE OPERAND TO SUBSTR(#MYDYNTEXT1, #OP2)              /* LEADS TO RUNTIME ERROR; ↩
UNDEFINED LENGTH
```

Zuweisungskompatibilität

Beispiel:

```
#MYDYNTEXT1   := #MYSTATICVAR1
#MYSTATICVAR1 := #MYDYNTEXT2
```

Wenn der Source-Operand eine statische Variable ist, wird die benutzte Länge des dynamischen Ziel-Operanden (`*LENGTH(#MYDYNTEXT1)`) auf die Format-Länge der statischen Variablen gesetzt, und der Source-Wert wird einschließlich nachfolgender Leerzeichen (alphanumerische und Unicode-Felder) oder binärer Nullen (für binäre Felder) in diese Länge kopiert.

Wenn der Ziel-Operand statisch und der Source-Operand dynamisch ist, wird die dynamische Variable in ihre zurzeit benutzte Länge kopiert. Wenn diese Länge kleiner als die Format-Länge der statischen Variable ist, wird der Rest mit Leerzeichen (für alphanumerische und Unicode-Felder) oder binären Nullen (für binäre Felder) aufgefüllt, sonst wird der Wert abgeschnitten. Wenn die aktuell benutzte Länge der dynamischen Variable Null (0) ist, wird der statische Ziel-

Operand mit Leerzeichen (für alphanumerische und Unicode-Felder) oder binären Nullen (für binäre Felder) aufgefüllt.

Initialisierung dynamischer Variablen

Dynamische Variablen können mit einem `RESET`-Statement mit Leerzeichen (alphanumerische und Unicode-Felder) oder Nullen (binäre Felder) bis zur aktuell benutzten Länge (= `*LENGTH`) initialisiert werden. `*LENGTH` wird nicht geändert.

Beispiel:

```
DEFINE DATA LOCAL
  1 #MYDYNTEXT1   (A)  DYNAMIC
END-DEFINE
#MYDYNTEXT1 := 'SHORT TEXT'
WRITE *LENGTH(#MYDYNTEXT1)          /* USED LENGTH = 10
RESET #MYDYNTEXT1                   /* USED LENGTH = 10, VALUE = 10 BLANKS
```

Um eine dynamische Variable mit einem angegebenen Wert in einer angegebenen Länge zu initialisieren, kann das `MOVE ALL UNTIL`-Statement benutzt werden.

Beispiel:

```
MOVE ALL 'Y' TO #MYDYNTEXT1 UNTIL 15     /* #MYDYNTEXT1 CONTAINS 15 'Y'S, USED ↩
LENGTH = 15
```

String-Manipulation mit dynamischen alphanumerischen Variablen

Wenn ein änderbarer Operand eine dynamische Variable ist, wird ihre zurzeit zugewiesene Länge möglicherweise erhöht, um die Operation ohne Abschneidung oder Fehlermeldung auszuführen. Dies gilt für die Verkettung (`COMPRESS`) und Trennung von dynamischen alphanumerischen Variablen (`SEPARATE`).

Beispiel:

```
** Example 'DYNAMX01': Dynamic variables (with COMPRESS and SEPARATE)
************************************************************************
DEFINE DATA LOCAL
1 #MYDYNTEXT1 (A)   DYNAMIC
1 #TEXT       (A20)
1 #DYN1       (A)   DYNAMIC
1 #DYN2       (A)   DYNAMIC
1 #DYN3       (A)   DYNAMIC
END-DEFINE
*
MOVE ' HELLO WORLD ' TO #MYDYNTEXT1
WRITE #MYDYNTEXT1 (AL=25) 'with length' *LENGTH (#MYDYNTEXT1)
/*  dynamic variable with leading and trailing blanks
*
MOVE ' HELLO WORLD ' TO #TEXT
*
MOVE #TEXT TO #MYDYNTEXT1
WRITE #MYDYNTEXT1 (AL=25) 'with length' *LENGTH (#MYDYNTEXT1)
/*  dynamic variable with whole variable length of #TEXT
*
COMPRESS #TEXT INTO #MYDYNTEXT1
WRITE #MYDYNTEXT1 (AL=25) 'with length' *LENGTH (#MYDYNTEXT1)
/*  dynamic variable with leading blanks of #TEXT
*
*
#MYDYNTEXT1 := 'HERE COMES THE SUN'
SEPARATE #MYDYNTEXT1 INTO #DYN1 #DYN2 #DYN3 IGNORE
*
WRITE / #MYDYNTEXT1 (AL=25) 'with length' *LENGTH (#MYDYNTEXT1)
WRITE #DYN1 (AL=25) 'with length' *LENGTH (#DYN1)
WRITE #DYN2 (AL=25) 'with length' *LENGTH (#DYN2)
WRITE #DYN3 (AL=25) 'with length' *LENGTH (#DYN3)
/*  #DYN1, #DYN2, #DYN3 are automatically extended or reduced
*
EXAMINE #MYDYNTEXT1 FOR 'SUN' REPLACE 'MOON'
WRITE / #MYDYNTEXT1 (AL=25) 'with length' *LENGTH (#MYDYNTEXT1)
/*  #MYDYNTEXT1 is automatically extended or reduced
*
END
```

Anmerkung: Im Falle von nicht-dynamischen Variablen kann eine Fehlermeldung zurückgegeben werden.

Logische Bedingungen bei dynamischen Variablen

Im Allgemeinen erfolgt eine Lese-Operation (z.B. ein Vergleich) bei einer dynamischen Variablen mit ihrer zurzeit benutzten Größe. Dynamische Variablen werden wie statische Variablen verarbeitet, wenn sie in einem Lese-Kontext (nicht änderbar) benutzt werden.

Beispiel:

```
IF  #MYDYNTEXT1  =  #MYDYNTEXT2  OR  #MYDYNTEXT1 =  "**"  THEN ...
IF  #MYDYNTEXT1  <  #MYDYNTEXT2  OR  #MYDYNTEXT1 <  "**"  THEN ...
IF  #MYDYNTEXT1  >  #MYDYNTEXT2  OR  #MYDYNTEXT1 >  "**"  THEN ...
```

Auch im Falle von nachfolgenden Leerzeichen oder führenden Nullen zeigen dynamische Variablen ein entsprechendes Verhalten.

Für dynamische Variablen ist der alphanumerische Wert `AA` gleich `AA`, und der binäre Wert `00003031` ist gleich `3031`.

Führende Nullen für alphanumerische und Unicode-Variablen oder führende binäre Nullen für binäre Variablen werden bei statischen und dynamischen Variablen gleich behandelt. Zum Beispiel werden alphanumerische Variable, die die Werte `AA` und `AA ` (d.h. `AA` mit nachfolgendem Leerzeichen) enthalten, als gleich angesehen. Binäre Variablen, die beispielsweise die Werte H'0000031' und H'3031' enthalten, werden ebenso als gleich angesehen. Wenn ein Vergleichsergebnis nur im Falle einer exakten Kopie wahr (`TRUE`) sein sollte, müssen die benutzten Längen der dynamischen Variablen außerdem miteinander verglichen werden. Wenn eine Variable eine exakte Kopie der anderen ist, sind ihre benutzten Längen auch gleich.

Beispiel:

```
#MYDYNTEXT1  :=  'HELLO'                                    /* USED LENGTH IS 5
#MYDYNTEXT2  :=  'HELLO     '                               /* USED LENGTH IS 10
IF   #MYDYNTEXT1  =  #MYDYNTEXT2   THEN  ...                /* TRUE
IF   #MYDYNTEXT1  =  #MYDYNTEXT2 AND
     *LENGTH(#MYDYNTEXT1) = *LENGTH(#MYDYNTEXT2)  THEN  ...  /* FALSE
```

Zwei dynamische Variablen werden Position für Position miteinander verglichen (von links nach rechts bei alphanumerischen Variablen und von rechts nach links bei binären Variablen) bis zum Minimum ihrer benutzten Längen. Die erste Position, an der die Variablen nicht gleich sind, legt fest, ob die erste oder zweite Variable größer, gleich oder kleiner als die andere ist. Die Variablen sind gleich, wenn sie bis zum Minimum ihrer benutzten Längen gleich sind, und der Rest der längeren Variable nur Leerzeichen (bei alphanumerischen dynamischen Variablen) oder binäre Nullen (bei dynamischen binären Variablen) enthält. Um zwei dynamische Unicode-Variablen miteinander zu vergleichen, werden aus beiden Werten die führenden Nullen entfernt, bevor zum

Vergleich der beiden resultierenden Werte der ICU-Collation-Algoritmus benutzt wird. Siehe auch *Logical Condition Criteria* in der *Unicode and Code Page Support*-Dokumentation.

Beispiel:

```
#MYDYNTEXT1  :=  'HELLO1'                    /* USED LENGTH IS 6
#MYDYNTEXT2  :=  'HELLO2'                    /* USED LENGTH IS 10
IF #MYDYNTEXT1  <  #MYDYNTEXT2 THEN  ...     /* TRUE
#MYDYNTEXT2  :=  'HALLO'
IF #MYDYNTEXT1  >  #MYDYNTEXT2 THEN  ...     /* TRUE
```

Vergleichskompatibilität

Vergleiche zwischen dynamischen und statischen Variablen sind gleichwertig mit Vergleichen zwischen dynamischen Variablen. Die Format-Länge der statischen Variable wird als ihre benutzte Länge interpretiert.

Beispiel:

```
#MYSTATTEXT1 :=  'HELLO'                     /* FORMAT LENGTH OF MYSTATTEXT1 IS ↩
A20
#MYDYNTEXT1  :=  'HELLO'                     /* USED LENGTH IS 5
IF  #MYSTATTEXT1  =  #MYDYNTEXT1  THEN  ...  /* TRUE
IF  #MYSTATTEXT1  >  #MYDYNTEXT1  THEN  ...  /* FALSE
```

AT/IF-BREAK dynamischer Kontrollfelder

Der Vergleich des **Gruppenwechsel**-Kontrollfeldes mit seinem alten Wert wird Position für Position von links nach rechts durchgeführt. Wenn der alte und der neue Wert der dynamischen Variable jeweils unterschiedliche Längen hat, dann wird zu Vergleichszwecken der Wert mit der kürzeren Länge nach rechts aufgefüllt (mit Leerzeichen für alphanumerisch und Unicode (dynamische Werte oder binäre Nullen für binäre Werte).

Im Falle eines alphanumerischen oder Unicode-Gruppenwechselkontrollfeldes sind nachfolgende Leerzeichen nicht bedeutend für den Vergleich, d.h. nachfolgende Leerzeichen bedeuten keine Änderung des Wertes, und es tritt kein Gruppenwechsel auf.

Im Falle eines binären Gruppenwechselkontrollfeldes sind nachfolgende binäre Nullen nicht bedeutend für den Vergleich, d.h. nachfolgende binäre Nullen bedeuten keine Änderung des Wertes, und es findet kein Gruppenwechsel statt.

Parameter-Übergabe mit dynamischen Variablen

Dynamische Variablen können als Parameter an aufgerufene Programmobjekte (CALLNAT, PERFORM) übergeben werden. Aufruf über eine Referenz (Call-by-Reference) ist möglich, weil der Wertespeicher einer dynamischen Variable zusammenhängend ist. Aufruf über Wert (Call-by-Value) führt zu einer Zuweisung der Variablen-Definition des Aufrufenden als Source-Operand und der Parameter-Definition als Ziel-Operand. Bei Aufruf über Wert (Ergebnis) (Call-by-Value (Result)) ist es umgekehrt.

Bei Aufruf über eine Referenz (Call-by-Reference) müssen beide Definitionen dynamisch (DYNAMIC) sein. Wenn nur eine von ihnen dynamisch ist, tritt ein Laufzeitfehler auf. Im Falle von Call-by-Value (Result), d.h. Aufruf über Wert (Ergebnis) sind alle Kombinationen möglich. Die folgende Tabelle veranschaulicht die gültigen Kombinationen:

Call-By-Reference

Caller	Parameter	
	Statisch	Dynamisch
Statisch	Ja	Nein
Dynamisch	Nein	Ja

Die Formate der dynamischen Variablen A oder B müssen miteinander im Einklang sein.

Call-by-Value (Result)

Caller	Parameter	
	Statisch	Dynamisch
Statisch	Ja	Ja
Dynamisch	Ja	Ja

Anmerkung: Im Falle von statischen/dynamischen oder dynamischen/statischen Definitionen können gemäß der Datenübertragungsregeln der betreffenden Zuweisungen Werte abgeschnitten werden.

Beispiel 1:

```
** Example 'DYNAMX02': Dynamic variables (as parameters)
************************************************************************
DEFINE DATA LOCAL
1 #MYTEXT (A) DYNAMIC
END-DEFINE
*
#MYTEXT := '123456'      /* extended to 6 bytes, *LENGTH(#MYTEXT) = 6
*
CALLNAT 'DYNAMX03' USING #MYTEXT
*
WRITE *LENGTH(#MYTEXT)          /* *LENGTH(#MYTEXT) = 8
*
END
```

Subprogramm `DYNAMX03`:

```
** Example 'DYNAMX03': Dynamic variables (as parameters)
************************************************************************
DEFINE DATA PARAMETER
1 #MYPARM (A) DYNAMIC  BY VALUE RESULT
END-DEFINE
*
WRITE *LENGTH(#MYPARM)                  /* *LENGTH(#MYPARM) = 6
#MYPARM := '1234567'                    /* *LENGTH(#MYPARM) = 7
#MYPARM := '12345678'                   /* *LENGTH(#MYPARM) = 8
EXPAND DYNAMIC VARIABLE #MYPARM TO 10   /* 10 bytes are allocated
*
WRITE *LENGTH(#MYPARM)                  /* *LENGTH(#MYPARM) = 8
*
/* content of #MYPARM is moved back to #MYTEXT
/* used length of #MYTEXT = 8
*
END
```

Beispiel 2:

```
** Example 'DYNAMX04': Dynamic variables (as parameters)
************************************************************************
DEFINE DATA LOCAL
1 #MYTEXT (A) DYNAMIC
END-DEFINE
*
#MYTEXT := '123456'      /* extended to 6 bytes, *LENGTH(#MYTEXT) = 6
*
CALLNAT 'DYNAMX05' USING #MYTEXT
*
```

```
WRITE *LENGTH(#MYTEXT)          /* *LENGTH(#MYTEXT) = 8
                                /* at least 10 bytes are
                                /* allocated (extended in DYNAMX05)
*
END
```

Subprogramm DYNAMX05:

```
** Example 'DYNAMX05': Dynamic variables (as parameters)
************************************************************************
DEFINE DATA PARAMETER
1 #MYPARM (A) DYNAMIC
END-DEFINE
*
WRITE *LENGTH(#MYPARM)                  /* *LENGTH(#MYPARM) = 6
#MYPARM := '1234567'                    /* *LENGTH(#MYPARM) = 7
#MYPARM := '12345678'                   /* *LENGTH(#MYPARM) = 8
EXPAND DYNAMIC VARIABLE #MYPARM TO 10   /* 10 bytes are allocated
*
WRITE *LENGTH(#MYPARM)                  /* *LENGTH(#MYPARM) = 8
*
END
```

3GL-Progamm aufrufen

Dynamische und große Variablen können mit dem CALL-Statement sinnvoll benutzt werden, wenn die Option INTERFACE4 verwendet wird. Der Einsatz dieser Option führt zu einer Schnittstelle zum 3GL-Programm mit einer unterschiedlichen Parameter-Struktur.

Bevor Sie ein 3GL-Programm mit dynamischen Parametern aufrufen, ist es wichtig sicherzustellen, dass die erforderliche Puffergröße zugewiesen wird. Dies kann explizit über das EXPAND-Statement erfolgen.

Wenn ein initialisierter Puffer erforderlich ist, kann die dynamische Variable mittels des MOVE ALL UNTIL-Statements auf den Ausgangswert und auf die erforderliche Größe gesetzt werden. Natural stellt eine Reihe von Funktionen zur Verfügung, die es dem 3GL-Programm ermöglichen, Informationen über dynamische Parameter zu erhalten und die Länge zu ändern, wenn Parameterdaten zurückgeschrieben werden.

Beispiel:

```
MOVE ALL ' ' TO  #MYDYNTEXT1 UNTIL 10000
  /* a buffer of length 10000 is allocated
  /* #MYDYNTEXT1 is initialized with blanks
  /* and *LENGTH(#MYDYNTEXT1) = 10000
CALL INTERFACE4  'MYPROG'  USING  #MYDYNTEXT1
WRITE *LENGTH(#MYDYNTEXT1)
  /* *LENGTH(#MYDYNTEXT1) may have changed in the 3GL program
```

Eine ausführlichere Beschreibung finden Sie beim `CALL`-Statement in der *Statements*-Dokumentation.

Workfile-Zugriff bei dynamischen und großen Variablen

Es gibt keinen Unterschied in der Behandlung fester Längen von Variablen mit einer Länge kleiner gleich 253 Bytes und großen Variablen mit einer Länge größer als 253 Bytes.

Dynamische Variablen werden in der eingerichteten Länge geschrieben (d.h. der Wert der Systemvariable `*LENGTH` für diese Variable), wenn das `WRITE WORK FILE`-Statement ausgeführt wird. Da die Länge für jede Ausführung desselben `WRITE WORK FILE`-Statements unterschiedlich sein kann, muss das Schlüsselwort `VARIABLE` angegeben werden.

Beim Lesen von Arbeitsdateien des Typs `FORMATTED` wird eine dynamische Variable in der Länge gefüllt, die gültig ist (d.h. der Wert der Systemvariable `*LENGTH` für diese Variable), wenn das `READ WORK FILE`-Statement ausgeführt wird. Wenn die dynamische Variable länger als die verbleibenden Daten im aktuellen Datensatz ist, wird sie bei alphanumerischen und Unicode-Feldern mit Leerzeichen und bei binären Feldern mit binären Nullen aufgefüllt.

Beim Lesen einer Arbeitsdatei des Typs `UNFORMATTED` wird eine dynamische Variable mit dem Rest der Arbeitsdatei aufgefüllt. Ihre Größe wird dementsprechend angepasst und wird im Wert der Systemvariable `*LENGTH` für diese Variable reflektiert.

Performance-Aspekte bei dynamischen Variablen

Wenn eine dynamische Variable in kleinen Beträgen mehrmals (z.B. byteweise) erweitert werden soll, benutzen Sie das `EXPAND`-Statement vor den Schleifen-Iterationen, wenn die obere Grenze des erforderlichen Speichers (ungefähr) bekannt ist. Dadurch vermeiden Sie zusätzlichen Verarbeitungsmehraufwand zum Anpassen des erforderlichen Speicherplatzes.

Benutzen Sie das `REDUCE`- oder `RESIZE`-Statement, wenn die dynamische Variable nicht mehr erforderlich ist, insbesondere für Variablen mit einem hohen Wert von `*LENGTH`. Dadurch wird es Natural ermöglicht, den Speicherplatz wieder zu benutzen oder freizugeben. Somit kann die Gesamtleistung verbessert werden.

Die Größe des einer dynamischen Variable zugewiesenen Hauptspeichers kann mittels des `REDUCE DYNAMIC VARIABLE`-Statements auf eine angegebene Größe reduziert werden. Um eine Variable auf eine angegebene Größe (neu) zuzuweisen, kann das `EXPAND`-Statement benutzt werden.

Wenn die Variable initialisiert werden soll, benutzen Sie das `MOVE ALL UNTIL`-Statement.

Beispiel:

```
** Example 'DYNAMX06': Dynamic variables (allocated memory)
************************************************************************
DEFINE DATA LOCAL
1 #MYDYNTEXT1  (A)  DYNAMIC
1 #LEN         (I4)
END-DEFINE
*
#MYDYNTEXT1 := 'a'     /* used length is 1, value is 'a'
                       /* allocated size is still 1
WRITE *LENGTH(#MYDYNTEXT1)
*
EXPAND DYNAMIC VARIABLE  #MYDYNTEXT1 TO 100
                       /* used length is still 1, value is 'a'
                       /* allocated size is 100
*
CALLNAT 'DYNAMX05' USING #MYDYNTEXT1
WRITE *LENGTH(#MYDYNTEXT1)
                       /* used length and allocated size
                       /* may have changed in the subprogram
*
#LEN := *LENGTH(#MYDYNTEXT1)
REDUCE DYNAMIC VARIABLE  #MYDYNTEXT1  TO  #LEN
                       /* if allocated size is greater than used length,
                       /* the unused memory is released
*
REDUCE DYNAMIC VARIABLE  #MYDYNTEXT1  TO  0
WRITE *LENGTH(#MYDYNTEXT1)
                       /* free allocated memory for dynamic variable
END
```

Regeln:

- Benutzen Sie dynamische Operanden überall dort, wo es sinnvoll ist.
- Benutzen Sie `EXPAND`, wenn die obere Grenze der Hauptspeicher-Benutzung bekannt ist.
- Benutzen Sie `REDUCE`, wenn der dynamische Operand nicht mehr gebraucht wird.

Ausgabe von dynamische Variablen

Dynamische Variablen können innerhalb von Ausgabe-Statements wie folgt benutzt werden:

Statement	Anmerkungen
DISPLAY	Mit diesen Statements müssen Sie das Format der Ausgabe oder Eingabe von dynamischen Variablen setzen, indem Sie die Session-Parameter AL (Alphanumerische Länge für die Ausgabe) oder EM (Editiermaske) benutzen.
WRITE	
INPUT	
REINPUT	--
PRINT	Da die Ausgabe des PRINT-Statements unformatiert ist, muss die Ausgabe der dynamischen Variablen im PRINT-Statement nicht mittels der Parameter AL und EM gesetzt werden. Mit anderen Worten: diese Parameter können weggelassen werden.

Dynamische X-Arrays

Ein dynamisches X-Array kann zugewiesen werden, indem man zuerst die Anzahl der Ausprägungen angibt und dann die Länge der vorher zugewiesenen Array-Ausprägungen erweitert.

Beispiel:

```
DEFINE DATA LOCAL
 1 #X-ARRAY(A/1:*)  DYNAMIC
END-DEFINE
*
EXPAND  ARRAY   #X-ARRAY TO (1:10)  /* Current boundaries (1:10)
#X-ARRAY(*) := 'ABC'
EXPAND  ARRAY   #X-ARRAY TO (1:20)  /* Current boundaries (1:20)
#X-ARRAY(11:20) := 'DEF'
```

27 Benutzerkonstanten

Konstanten können überall in Natural-Programmen benutzt werden. Dieses Dokument behandelt die Arten von Konstanten, die unterstützt werden, und erläutert wie sie benutzt werden.

Numerische Konstanten

Folgende Themen werden behandelt:

Numerische Konstanten

Eine numerische Konstante kann 1 bis 29 numerische Ziffern, ein Sonderzeichen als Dezimalzeichen (Komma oder Punkt) und ein Vorzeichen enthalten.

Beispiele:

```
1234   +1234   -1234

12.34  +12.34  -12.34
```

```
MOVE 3 TO #XYZ
COMPUTE #PRICE = 23.34
COMPUTE #XYZ = -103
COMPUTE #A = #B * 6074
```

Intern werden numerische Konstanten in gepackter Form (Format P) dargestellt. Ausnahme: Wenn eine numerische Konstante in einer arithmetischen Operation benutzt wird, in der der andere Operand eine Ganzzahl-Variable (Format I) ist, wird die numerische Konstante in Ganzzahl-Form (Format I) dargestellt.

Validierung von numerischen Konstanten

Wenn eine numerische Konstante innerhalb eines der Statements `COMPUTE`, `MOVE` oder `DEFINE DATA` mit der `INIT`-Option benutzt werden, überprüft Natural zur Kompilierungszeit, ob ein Konstanten-Wert in das entsprechende Feld passt. Dadurch werden Laufzeitfehler in Situationen vermieden, in denen eine solche Fehlerbedingung bereits bei der Kompilierung entdeckt werden kann.

Alphanumerische Konstanten

Folgende Themen werden behandelt:

- Struktur einer alphanumerischen Konstanten
- Verwendung von Apostrophen in alphanumerischen Konstanten
- Verketten von alphanumerischen Konstanten

Struktur einer alphanumerischen Konstanten

Eine alphanumerische Konstante kann 1 bis 1.073.741.824 Bytes (1 GB) alphanumerische Zeichen enthalten. Eine alphanumerische

Eine alphanumerische Konstante muss entweder in Apostrophen (')

```
'text'
```

oder Anführungszeichen (") stehen.

```
"text"
```

Beispiele:

```
MOVE 'ABC' TO #FIELDX
MOVE '% INCREASE' TO #TITLE
DISPLAY "LAST-NAME" NAME
```

Anmerkung: Eine alphanumerische Konstante, die zur Zuweisung eines Wertes zu einer **Benutzervariable** verwendet wird, darf nicht auf mehrere Statement-Zeilen aufgeteilt werden.

Verwendung von Apostrophen in alphanumerischen Konstanten

Möchten Sie, dass ein Apostroph (') Teil einer in Apostrophen stehenden alphanumerischen Konstante wird, müssen Sie dies durch 2 Apostrophen (' ') oder ein einzelnes Anführungszeichen (") ausdrücken.

Möchten Sie, dass ein Apostroph (') Teil einer in Anführungszeichen (") stehenden alphanumerischen Konstante wird, drücken Sie dies durch einen einzelnen Apostroph (') aus.

Beispiel:

Wenn Sie Folgendes ausgeben möchten

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

können Sie eine der folgenden Notationen benutzen:

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

Anmerkung: Wenn Anführungszeichen nicht in Apostrophen konvertiert werden (siehe oben), dann ist der Grund dafür die Einstellung des Profilparameters `TQ` (Anführungszeichen konvertieren). Einzelheiten dazu erfahren Sie von Ihrem Natural-Administrator.

Verketten von alphanumerischen Konstanten

Alphanumerische Konstanten können mittels eines Bindestrichs miteinander verkettet werden, so dass sie einen einzelnen Wert bilden.

Beispiele:

```
MOVE 'XXXXXX' - 'YYYYYY' TO #FIELD
MOVE "ABC" - 'DEF' TO #FIELD
```

Auf diese Art können alphanumerische Konstanten auch mit **hexadezimalen Konstanten** verkettet werden.

Unicode-Konstanten

Folgende Themen werden behandelt:

- Unicode-Textkonstanten
- Apostroph innerhalb von Unicode-Textkonstanten
- Unicode-Hexadezimalkonstanten

- Verketten von Unicode-Konstanten

Unicode-Textkonstanten

Einer Unicode-Textkonstante muss das Zeichen `U` vorausgehen, und sie muss entweder in Apostrophen (') stehen

```
U'text'
```

oder in Anführungszeichen (''):

```
U"text"
```

Beispiel:

```
U'HELLO'
```

Der Compiler speichert diese Textkonstante in dem generierten Programm im Unicode-Format (UTF-16).

Apostroph innerhalb von Unicode-Textkonstanten

Wenn Sie möchten, dass ein Apostroph (') Teil einer Unicode-Textkonstante wird, die in Apostrophen steht, müssen Sie dies als zwei Apostrophe (' ') oder als ein einzelnes Anführungszeichen ('') kodieren.

Wenn Sie möchten, dass ein Apostroph Teil einer Unicode-Textkonstante wird, die in Anführungszeichen steht, müssen Sie dies als einen einzelnen Apostroph (') kodieren.

Beispiel:

Wenn Sie Folgendes ausgeben möchten

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

können Sie eine der folgenden Notationen benutzen:

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

Anmerkung: Wenn Anführungszeichen nicht in Apostrophe umgesetzt werden (siehe oben), dann liegt dies an der Einstellung des Profilparameters `TQ` (Anführungszeichen umsetzen). Einzelheiten erfahren Sie von Ihrem Natural-Administrator.

Unicode-Hexadezimalkonstanten

Die folgende Syntax wird benutzt, um ein Unicode-Zeichen oder eine Unicode-Zeichenkette in seiner hexadezimale Notation anzugeben:

```
UH'hhhh...'
```

wobei *h* eine hexadezimale Ziffer (0–9, A–F) ist.

Da ein UTF-16-Unicode-Zeichen aus einem Doppelbyte besteht, muss die Anzahl der angegebenen hexadezimalen Zeichen ein Mehrfaches von vier sein.

Beispiel:

Dieses Beispiel definiert die Zeichenkette `45`.

```
UH'00340035'
```

Verketten von Unicode-Konstanten

Die Verkettung von Unicode-Textkonstanten (U) und Unicode-Hexadezimalkonstanten (UH) ist zulässig.

Gültiges Beispiel:

```
MOVE U'XXXXXX' - UH'00340035' TO #FIELD
```

Unicode-Textkonstanten oder Unicode-Hexadezimalkonstanten können nicht mit alphanumerischen Codepage-Konstanten oder H-Konstanten verkettet werden.

Ungültiges Beispiel:

```
MOVE U'ABC' - 'DEF' TO #FIELD
MOVE UH'00340035' - H'414243' TO #FIELD
```

Weiteres gültiges Beispiel:

```
DEFINE DATA LOCAL
1 #U10 (U10)               /* Unicode variable with 10 (UTF-16) characters,
                           /* total byte length = 20
1 #UD  (U)  DYNAMIC        /* Unicode variable with dynamic length
END-DEFINE
*
#U10 := U'ABC'            /* Constant is created as X'004100420043' in the object,
                           /* the UTF-16 representation for string 'ABC'.
```

```
#U10 := UH'004100420043'   /* Constant supplied in hexadecimal format only,
                           /* corresponds to U'ABC'

#U10 := U'A'-UH'0042'-U'C' /* Constant supplied in mixed formats, corresponds to ↩
U'ABC'.
END
```

Datums- und Zeitkonstanten

Folgende Themen werden behandelt:

- Datumskonstante
- Zeitkonstante
- Erweiterte Zeitkonstante

Datumskonstante

Eine Datumskonstante kann in Verbindung mit einer Variablen des Formats D benutzt werden.

Datumskonstanten können folgende Formate haben:

D'*yyyy-mm-dd*'	Internationales Datumsformat
D'*dd.mm.yyyy*'	Deutsches Datumsformat
D'*dd/mm/yyyy*'	Europäisches Datumsformat
D'*mm/dd/yyyy*'	USA-Datumsformat

Dabei bezeichnet *dd* den Tag, *mm* den Monat und *yyyy* das Jahr.

Beispiel:

```
DEFINE DATA LOCAL
  1 #DATE (D)
END-DEFINE
...
MOVE D'2004-03-08' TO #DATE
...
```

Das standardmäßige Datumsformat wird von dem vom Natural-Administrator gesetzten Profilparameter DTFORM (Datumsformat) gesteuert.

Zeitkonstante

Eine Zeitkonstante kann in Verbindung mit einer Variablen des Formats T benutzt werden.

Eine Zeitkonstante hat das folgende Format:

```
T'hh:ii:ss'
```

Dabei bezeichnet *hh* Stunden, *ii* Minuten und *ss* Sekunden.

Beispiel:

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

Erweiterte Zeitkonstante

Eine Zeitvariable (Format T) kann Datums- und Zeit-Informationen enthalten, wobei die Datumsinformationen eine Untermenge der Zeitinformationen sind. Allerdings können bei einer „normalen“ Zeitkonstante (Präfix `T`) nur die Zeit-Informationen einer Zeitvariablen verarbeitet werden:

```
T'hh:ii:ss'
```

Bei einer erweiterten Zeitkonstante (Präfix `E`) ist es möglich, den vollständigen Inhalt einer Zeitvariablen einschließlich der Datums-Informationen zu verarbeiten:

```
E'yyyy-mm-dd hh:ii:ss'
```

Einmal abgesehen davon ist die Benutzung einer erweiterten Zeitkonstanten in Verbindung mit einer Zeitvariablen identisch mit der für eine normale Zeitkonstante.

Anmerkung: Das Format, in dem Datums-Informationen angegeben werden müssen, ist bei einer erweiterten Zeitkonstante abhängig von der Einstellung des Profilparameters `DTFORM`. Bei der oben angegebenen erweiterten Zeitkonstante wird von `DTFORM=I` (internationales Datumsformat) ausgegangen.

Hexadezimale Konstanten

Folgende Themen werden behandelt:

- Verwendung und Verarbeitung von hexadezimalen Konstanten
- Verketten von hexadezimalen Konstanten

Verwendung und Verarbeitung von hexadezimalen Konstanten

Eine hexadezimale Konstante kann benutzt werden, um einen Wert einzugeben, der nicht als ein standardmäßiges Tastaturzeichen eingegeben werden kann.

Eine hexadezimale Konstante kann 1 bis 1.073.741.824 bytes (1 GB) alphanumerische Zeichen enthalten.

Einer hexadezimalen Konstante geht ein Präfix `H` voraus. Die Konstante selbst muss in Apostrophen (') stehen und kann aus den hexadezimalen Zeichen 0 − 9, A − F bestehen. Zwei hexadezimale Zeichen sind erforderlich, um ein Daten-Byte darzustellen.

Die hexadezimale Darstellung eines Zeichens variiert, je nachdem, ob Ihr Computer einen ASCII- oder EBCDIC-Zeichensatz verwendet. Wenn Sie hexadezimale Konstanten auf einen anderen Computer übertragen, müssen Sie deshalb vielleicht die Zeichen konvertieren.

ASCII-Beispiele:

```
H'313233'    (equivalent to the alphanumeric constant '123')
H'414243'    (equivalent to the alphanumeric constant 'ABC')
```

EBCDIC-Beispiele:

```
H'F1F2F3'    (equivalent to the alphanumeric constant '123')
H'C1C2C3'    (equivalent to the alphanumeric constant 'ABC')
```

Wenn eine hexadezimale Konstante in ein anderes Feld übertragen wird, wird sie als ein alphanumerischer Wert behandelt (Format A).

Die Datenübertragung eines alphanumerischen Werts (Format A) in ein Feld, das nicht in einem der Formate A, U oder B definiert ist, ist nicht zulässig. Deshalb wird eine hexadezimale Konstante als Ausgangswert in einem `DEFINE DATA`-Statement mit einem Syntaxfehler NAT0094 zurückgewiesen, wenn die entsprechende Variable nicht vom Format A, U oder B ist.

Beispiel:

```
DEFINE DATA LOCAL
1 #I(I2) INIT <H'000F'>      /* causes a NAT0094 syntax error
END-DEFINE
```

Verketten von hexadezimalen Konstanten

Hexadezimale Konstanten können mittels eines Bindestrichs zwischen den Konstanten miteinander verkettet werden.

ASCII-Beispiel:

```
H'414243' - H'444546' (equivalent to 'ABCDEF')
```

EBCDIC-Beispiel:

```
H'C1C2C3' - H'C4C5C6' (equivalent to 'ABCDEF')
```

Auf diese Weise können hexadezimale Konstanten auch mit alphanumerischen Konstanten verkettet werden.

Logische Konstanten

Die logischen Konstanten `TRUE` (wahr) und `FALSE` (falsch) können benutzt werden, um einen logischen Wert einem mit Format L definierten Feld zuzuweisen.

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 im Format F definierten Variablen benutzt werden.

Beispiel:

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

Attribut-Konstanten

Attribut-Konstanten können mit im Format C (Kontroll-Variablen) definierten Variablen benutzt werden. Diese Art von Konstante muss in Klammern stehen.

Die folgenden Attribute können benutzt werden:

Attribut	Beschreibung
AD=D	Standard
AD=B	blinkend
AD=I	hell hervorgehoben
AD=N	keine Anzeige
AD=V	invers
AD=U	unterstrichen
AD=C	kursiv
AD=Y	dynamisches Attribut
AD=P	geschützt
CD=BL	blau
CD=GR	grün
CD=NE	neutral
CD=PI	rosa
CD=RE	rot
CD=TU	türkis
CD=YE	gelb

Weitere Informationen zu diesen Attributen finden Sie bei den Session-Parametern AD und CD.

Beispiel:

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

Handle-Konstanten

Die Handle-Konstante NULL-HANDLE kann mit Objekt-Handles benutzt werden.

Weitere Informationen zu Objekt-Handles siehe *NaturalX*.

Namens-Konstanten definieren

Wenn Sie denselben Konstanten-Wert mehrmals in einem Programm benutzen müssen, können Sie den Pflegeaufwand durch Definition einer Namens-Konstante reduzieren:

- Definieren Sie ein Feld im DEFINE DATA-Statement,
- weisen Sie ihm einen Konstanten-Wert zu und
- benutzen Sie im Programm den Feldnamen anstatt des Konstanten-Werts.

Wenn der Wert geändert werden muss, müssen Sie ihn somit nur einmal im DEFINE DATA-Statement und nicht überall in dem Programm ändern, in dem er vorkommt.

Geben Sie den Konstanten-Wert in viereckigen Klammern mit dem **Schlüsselwort** CONSTANT hinter der Feld-Definition im DEFINE DATA-Statement an.

- Wenn der Wert alphanumerisch ist, muss er in Apostrophen (') stehen.
- Wenn der Wert Text im Unicode-Format ist, muss ihm das Zeichen U vorangehen, und er muss in Apostrophen (') stehen.
- Wenn der Wert im hexadezimalen Unicode-Format ist, müssen ihm die Zeichen UH vorangehen, und er muss in Apostrophen (') stehen.

Beispiel:

```
DEFINE DATA LOCAL
  1 #FIELDA (N3) CONSTANT <100>
  1 #FIELDB (A5) CONSTANT <'ABCDE'>
  1 #FIELDC (U5) CONSTANT <U'ABCDE'>
  1 #FIELDD (U5) CONSTANT <UH'00410042004300440045'>
END-DEFINE
...
```

Während der Ausführung des Programms kann der Wert einer solchen Namens-Konstante nicht geändert werden.

28 Ausgangswerte (und das RESET-Statement)

Dieses Kapitel beschreibt die standardmäßigen Ausgangswerte von Benutzervariablen, erläutert, wie Sie einer Benutzervariable einen Ausgangswert zuweisen können und wie Sie das `RESET`-Statement zum Zurücksetzen des Feldwertes auf seinen Standard-Ausgangswert oder den für diese Variable im `DEFINE DATA`-Statement definierten Ausgangswert benutzen können.

Standard-Ausgangswert einer Benutzervariablen/ eines Arrays

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

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

Ausgangswert einer Benutzervariablen/einem Array zuweisen

Im `DEFINE DATA`-Statement können Sie einer Benutzervariable einen Ausgangswert zuweisen. Wenn der Ausgangswert alphanumerisch ist, muss er in Apostrophen (') stehen.

Folgende Themen werden behandelt:

- Änderbaren Ausgangswert zuweisen
- Konstanten-Ausgangswert zuweisen
- Natural-Systemvariable als Ausgangswert zuweisen

- Zeichen als Ausgangswert für alphanumerische Variable zuweisen

Änderbaren Ausgangswert zuweisen

Wenn der Variablen bzw. dem Array ein änderbarer Ausgangswert zugewiesen werden soll, geben Sie den Ausgangswert in spitzen Klammern mit dem Schlüsselwort `INIT` nach der Variablen-Definition im `DEFINE DATA`-Statement an. Die zugewiesenen Werte werden jedesmal benutzt, wenn die Variable bzw. das Array referenziert wird. Die zugewiesenen Werte können während der Programmausführung geändert werden.

Beispiel:

```
DEFINE DATA LOCAL
1 #FIELDA (N3) INIT <100>
1 #FIELDB (A20) INIT <'ABC'>
END-DEFINE
...
```

Konstanten-Ausgangswert zuweisen

Wenn die Variable bzw. das Array als eine Namens-Konstante behandelt werden soll, geben Sie den Ausgangswert in spitzen Klammern mit dem Schlüsselwort `CONSTANT` nach der Variablen-Definition im `DEFINE DATA`-Statement an. Die zugewiesenen Konstanten-Werte werden jedesmal benutzt, wenn die die Variable bzw. das Array referenziert wird. Die zugewiesenen Werte können während der Programmausführung *nicht* geändert werden.

Beispiel:

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

Natural-Systemvariable als Ausgangswert zuweisen

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

Beispiel:

Hier liefert die Systemvariable `*DATX` den Ausgangswert.

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

Zeichen als Ausgangswert für alphanumerische Variable zuweisen

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

- **Feld komplett füllen:**
  Mit der Option `FULL LENGTH <character-string>` wird das gesamte Feld mit dem/den angegebenen Zeichen gefüllt.

  Hier wird das gesamte Feld mit Sternen (*) gefüllt:

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

- **Erste*n* Stellen eines Feldes füllen:**
  Mit der Option `LENGTH n <character(s)>` werden die ersten *n* Stellen des Feldes mit dem/den angegebenen Zeichen gefüllt.

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

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

Benutzervariable auf ihren Ausgangswert zurücksetzen

Das `RESET`-Statement dient dazu, den Wert eines Feldes zurückzusetzen. Zwei Optionen stehen dabei zur Verfügung:

- Auf Standard-Ausgangswert zurücksetzen
- Auf den im DEFINE DATA-Statement definierten Ausgangswert zurücksetzen

Anmerkungen:

1. Ein mit einer `CONSTANT`-Klausel im `DEFINE DATA`-Statement deklariertes Feld kann nicht in einem `RESET`-Statement referenziert werden, weil sein Inhalt nicht geändert werden kann.

2. Im Reporting Mode dient das RESET-Statement auch zur Definition einer Variablen, vorausgesetzt das Programm enthält kein DEFINE DATA LOCAL-Statement.

Auf Standard-Ausgangswert zurücksetzen

RESET (ohne INITIAL) setzt den Inhalt jedes angegebenen Feldes je nach Format auf seinen Standard-Ausgangswert zurück.

Beispiel:

```
DEFINE DATA LOCAL
1 #FIELDA (N3)  INIT <100>
1 #FIELDB (A20) INIT <'ABC'>
1 #FIELDC (I4)  INIT <5>
END-DEFINE
...
...
RESET #FIELDA                    /* resets field value to default initial value
... ↩
```

Auf den im DEFINE DATA-Statement definierten Ausgangswert zurücksetzen

RESET INITIAL setzt den Inhalt jedes angegebenen Feldes auf den Ausgangswert zurück, der für das Feld im DEFINE DATA-Statement definiert wurde.

Für ein ohne INIT-Klausel im DEFINE DATA-Statement deklariertes Feld hat RESET INITIAL denselben Effekt wie RESET (ohne INITIAL).

Beispiel:

```
DEFINE DATA LOCAL
1 #FIELDA (N3)  INIT <100>
1 #FIELDB (A20) INIT <'ABC'>
1 #FIELDC (I4)  INIT <5>
END-DEFINE
...
RESET INITIAL #FIELDA #FIELDB #FIELDC /* resets field values to initial values as ↩
defined in DEFINE DATA
...
```

29 Felder redefinieren

Die Redefinition dient dazu, das Format eines Feldes zu ändern oder ein einzelnes Feld in mehrere Teile aufzuteilen.

REDEFINE-Option des DEFINE DATA-Statements

Mit der `REDEFINE`-Option des `DEFINE DATA`-Statements kann ein einzelnes Feld — entweder eine Benutzervariable oder ein Datenbankfeld — als ein neues Feld oder mehrere neue Felder redefiniert werden. Eine Gruppe kann ebenfalls redefiniert werden.

⚠ **Wichtig:** Dynamische Variablen sind bei einer Redefinition 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 neudefinierten Feldes bzw. der neudefinierten Felder müssen einander entsprechen.

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

Beispiel 1:

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
   ...
```

Beispiel 2:

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.)

Beispiel 3:

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
...
```

Beispielprogramm für eine Redefinition

Das folgende Programm veranschaulicht die Anwendung einer Redefinition:

```
** Example 'DDATAX01': DEFINE DATA
************************************************************************
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
```

Ausgabe des Programms DDATAX01:

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

```
Page      1                                                  04-11-11  14:15:54

        NAME              FIRST-NAME          #PAY    #USD #000
-------------------- -------------------- ---------- ---- ----

JONES                VIRGINIA                46000   46    0
JONES                MARSHA                  50000   50    0
JONES                ROBERT                  31000   31    0
           ↩
```

30 Arrays

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.

Arrays definieren

Ein Natural-Array kann ein-, zwei- oder dreidimensional sein. Es kann eine unabhängige Variable, Teil einer größeren Datenstruktur oder Teil einer Datenbanksicht sein.

Wichtig: Dynamische Variablen sind in einer Array-Definition nicht zulässig.

Um ein eindimensionales Array zu definieren

- Geben Sie hinter Format und Länge einen Schrägstrich (/) und danach eine Index-Notation, d.h. die Anzahl der Ausprägungen des Arrays, an.

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

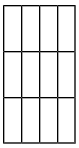
```
DEFINE DATA LOCAL
1 #ARRAY (A10/1:3)
END-DEFINE
...
```

Um ein zweidimensionales Array zu definieren

- Geben Sie für beide Dimensionen eine Index-Notation an:

```
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 (siehe folgende Beispiele), 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

Dieser Abschnitt zeigt, wie einem zweidimensionalen Array Ausgangswerte zugewiesen werden. Die folgenden Themen werden behandelt:

Vorbemerkung

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)
```

Vertikal: Erste Dimension (1:3), Horizontal: 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.

Gleichen Wert zuweisen

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

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

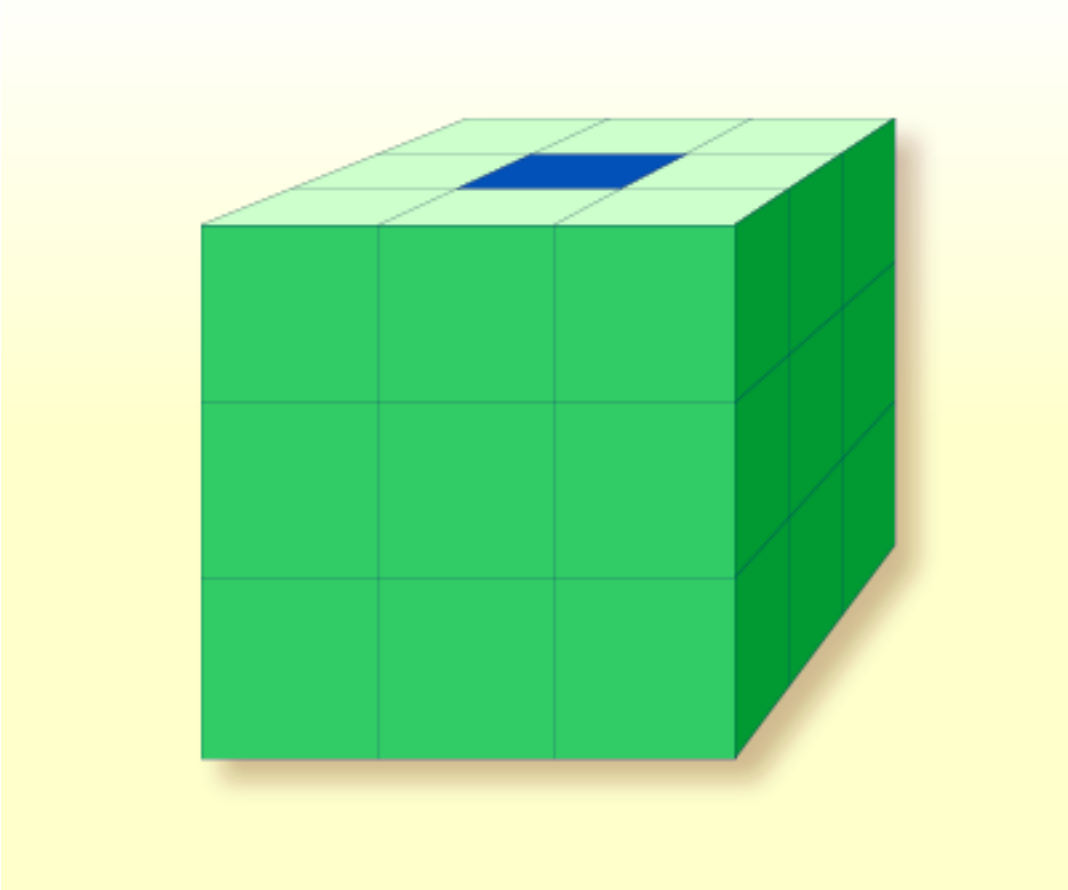
	B		
A	B		
	B	C	D

- `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, hat 10 Ausprägungen 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 hat 4 Ausprägungen.

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 Abschnitt *Datenbank-Arrays* 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) /* <--- MULTIPLE-VALUE FIELD
END-DEFINE
...
```

Dieselbe View in einer programmexternen Local Data Area:

```
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 Bereiches von Ausprägungen in einem Array 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(*,*)`

   Der Ergebnisoperand, Array `#B`, enthält die elementweise Addition des Arrays `#A` und des ursprünglichen Werts von Array `#B`.

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

   Die zweite Spalte des Arrays `#A` wird durch den ursprünglichen Wert plus 4 ersetzt.

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

   Die zweite Zeile des Arrays `#A` wird durch den ursprünglichen Wert plus 2 ersetzt.

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

   Der Wert der zweiten Zeile des Arrays `#A` wird zur vierten Zeile des Arrays `#B` addiert.

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 zweiten Zeile des Arrays #A werden zu dem Skalarwert #C addiert.

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

   Die Werte der 5. bis 7. Spalte der zweiten Zeile des Arrays #A werden zum Skalarwert #C addiert.

31 X-Arrays

Wenn Sie ein normales Array definieren, müssen sie die Indexgrenzen und folglich die Anzahl der Ausprägungen für jede Dimension genau angeben. Zur Laufzeit existiert standardmäßig das vollständige Array-Feld; auf jede seiner Ausprägungen kann ohne zusätzliche Zuweisungsoperationen zugegriffen werden. Das Größen-Layout kann nicht mehr geändert werden; Sie können Feldausprägungen weder hinzufügen noch entfernen.

Zur Anwendungsentwicklungszeit kennen Sie wahrscheinlich nicht die genaue Anzahl der Ausprägungen eines Arrays. Sie möchten vielleicht aber die Größe eines Arrays zur Laufzeit ändern können.

Zu diesem Zweck können Sie ein sogenanntes X-Array (*eXtensible array* = erweiterbares Array) definieren. Die Größe eines X-Arrays kann zur Laufzeit geändert werden, wodurch Sie darin unterstützt werden, Ihren Hauptspeicher effizienter zu verwalten. Sie können z.B. eine große Anzahl von Array-Ausprägungen für einen kurzen Zeitraum benutzen und dann den Hauptspeicherplatz reduzieren, wenn die Anwendung den Array nicht mehr benutzt.

Definition

Ein X-Array ist ein Array, dessen Anzahl an Ausprägungen zur Kompilierungszeit nicht bekannt ist. Ein X-Array kann nur in einem `DEFINE DATA`-Statement definiert werden, wenn Sie einen Stern (*) für mindestens eine Grenze von wenigstens einer Dimension des Arrays angeben. Der Stern (*) in der Grenzen-Definition verweist darauf, dass die entsprechende Grenze verlängerbar ist. Nur eine Grenze — entweder die obere oder die untere — kann verlängert werden, aber nicht beide.

Ein X-Array kann immer dann definiert werden, wenn ein (fester) Array definiert werden kann, d.h. auf jeder Ebene oder sogar als indizierte Gruppe. Es kann nicht verwendet werden, um auf MU-PE-Felder zuzugreifen. Ein mehrdimensionales Array kann eine Mischung aus konstanten und verlängerbaren Grenzen haben.

Beispiel:

```
DEFINE DATA LOCAL
1 #X-ARR1 (A5/1:*)          /* lower bound is fixed at 1, upper bound is variable
1 #X-ARR2 (A5/*)            /* shortcut for (A5/1:*)
1 #X-ARR3 (A5/*:100)        /* lower bound is variable, upper bound is fixed at 100
1 #X-ARR4 (A5/1:10,1:*)     /* 1st dimension has a fixed index range with (1:10)
END-DEFINE                  /* 2nd dimension has fixed lower bound 1 and variable ↩
upper bound
```

Speicherverwaltung von X-Arrays

Die Ausprägungen eines X-Arrays müssen ausdrücklick zugewiesen werden, bevor auf sie zugegriffen werden kann. Sie können für die Statements `EXPAND`, `RESIZE` und `REDUCE` die Anzahl der Ausprägungen für jede Dimension ändern.

Die Anzahl der Dimension des X-Arrays (1, 2 oder 3 Dimensionen) kann aber nicht geändert werden.

Beispiel:

```
DEFINE DATA LOCAL
1 #X-ARR(I4/10:*)
END-DEFINE
EXPAND ARRAY #X-ARR TO (10:10000)
/* #X-ARR(10) to #X-ARR(10000) are accessible
WRITE *LBOUND(#X-ARR)                  /* is 10
   *UBOUND(#X-ARR)                     /* is 10000
   *OCCURRENCE(#X-ARR)                 /* is 9991
#X-ARR(*) := 4711                      /* same as #X-ARR(10:10000) := 4711
/* resize array from current lower bound=10 to upper bound =1000
RESIZE ARRAY #X-ARR TO (*:1000)
/* #X-ARR(10) to #X-ARR(1000) are accessible
/* #X-ARR(1001) to #X-ARR(10000) are released
WRITE *LBOUND(#X-ARR)                  /* is 10
   *UBOUND(#X-ARR)                     /* is 1000
   *OCCURRENCE(#X-ARR)                 /* is 991
/* release all occurrences
REDUCE ARRAY #X-ARR TO  0
WRITE *OCCURRENCE(#X-ARR)              /* is 0
```

Speicherverwaltung von X-Gruppen-Arrays

Wenn Sie Ausprägungen der X-Gruppen-Arrays erhöhen oder reduzieren möchten, müssen Sie zwischen unabhängigen und abhängigen Dimensionen unterscheiden.

Eine Dimension, die direkt (nicht weitergegeben) für einen X-Gruppen-Array angegeben wird, ist *unabhängig*.

Eine Dimension, die nicht *direkt* für einen X-Gruppen-Array angegeben, sondern weitergegeben wird, ist *abhängig*.

Nur die unabhängigen Dimensionen können in den Statements EXPAND, RESIZE und REDUCE geändert werden. Die Dimensionen müssen unter Verwendung des entsprechenden Namens des X-Gruppen-Arrays, zu dem diese Dimension als unabhängige Dimension gehört, geändert werden.

Beispiel - Unabhängige/abhängige Dimensionen:

```
DEFINE  DATA LOCAL
1 #X-GROUP-ARR1(1:*)                  /* (1:*)
  2 #X-ARR1    (I4)                   /* (1:*)
  2 #X-ARR2    (I4/2:*)               /* (1:*,2:*)
  2 #X-GROUP-ARR2                     /* (1:*)
    3 #X-ARR3  (I4)                   /* (1:*)
    3 #X-ARR4  (I4/3:*)               /* (1:*,3:*)
    3 #X-ARR5  (I4/4:*, 5:*)          /* (1:*,4:*,5:*)
END-DEFINE
```

Die folgende Tabelle zeigt, ob die Dimensionen in dem obengezeigten Programm unabhängig oder abhängig sind.

Name	Abhängige Dimension	Unabhängige Dimension
#X-GROUP-ARR1		(1:*)
#X-ARR1	(1:*)	
#X-ARR2	(1:*)	(2:*)
#X-GROUP-ARR2	(1:*)	
#X-ARR3	(1:*)	
#X-ARR4	(1:*)	(3:*)
#X-ARR5	(1:*)	(4:*,5:*)

Als Index-Notation für die abhängige Dimension ist entweder ein einzelner Stern (*), ein mit einem Stern definierter Bereich oder die Definition der Ober- und Untergrenze zulässig. Diese Angabe soll darauf verweisen, dass die Grenzen der abhängigen Dimension so bleiben müssen wie sie sind und nicht geändert werden können.

Die Ausprägungen der abhängigen Dimensionen können nur durch Manipulation der entsprechenden Array-Gruppen geändert werden.

```
EXPAND ARRAY #X-GROUP-ARR1 TO (1:11)       /* #X-ARR1(1:11) are allocated
                                           /* #X-ARR3(1:11) are allocated
EXPAND ARRAY #X-ARR2 TO (*:*, 2:12)        /* #X-ARR2(1:11, 2:12) are allocated
EXPAND ARRAY #X-ARR2 TO (1:*, 2:12)        /* same as before
EXPAND ARRAY #X-ARR2 TO (*  , 2:12)        /* same as before
EXPAND ARRAY #X-ARR4 TO (*:*, 3:13)        /* #X-ARR4(1:11, 3:13) are allocated
EXPAND ARRAY #X-ARR5 TO (*:*, 4:14, 5:15)  /* #X-ARR5(1:11, 4:14, 5:15) are allocated
```

Die EXPAND-Statements können in beliebiger Reihenfolge kodiert werden.

Die folgende Verwendung des EXPAND-Statements ist nicht zulässig, da die Arrays nur abhängige Dimensionen haben.

```
EXPAND ARRAY #X-ARR1 TO ...
EXPAND ARRAY #X-GROUP-ARR2 TO ...
EXPAND ARRAY #X-ARR3 TO ...
```

X-Array referenzieren

Die Ausprägungen eines X-Arrays müssen über ein EXPAND- oder RESIZE-Statement zugewiesen werden, bevor sie aufgerufen werden können.

In der Regel gilt: der Versuch, eine nicht existierende X-Array-Ausprägung zu adressieren führt zu einem Laufzeitfehler. Bei einigen Statements verursacht der Zugriff auf ein nicht verwirklichtes X-Array-Feld jedoch keinen Fehler, wenn alle Ausprägungen eines X-Arrays mit der kompletten Bereichsnotation referenziert werden, zum Beispiel #X-ARR(*). Dies gilt für

- Parameter, die in einem CALL-Statement benutzt werden,
- Parameter, die bei CALLNAT, PERFORM, SEND EVENT oder OPEN DIALOG benutzt werden, wenn sie als OPTIONAL definiert sind,
- Source-Felder, die in einem COMPRESS-Statement benutzt werden,
- Ausgabefelder, die mit einem PRINT-Statement angegeben werden,
- Felder, die in einem RESET-Statement referenziert werden.

Wenn individuelle Ausprägungen eines nicht verwirklichten X-Arrays in einem dieser Statements referenziert werden, wird eine entsprechende Fehlermeldung ausgegeben.

Beispiel:

```
DEFINE DATA LOCAL
1 #X-ARR (A10/1:*) /* X-array only defined, but not allocated
END-DEFINE
RESET #X-ARR(*)    /* no error, because complete field referenced with (*)
RESET #X-ARR(1:3)  /* runtime error, because individual occurrences (1:3) are ↩
referenced
END
```

Die Stern-Notation (*) in einer Array-Referenz steht für den kompletten Bereich einer Dimension. Wenn das Array ein X-Array ist, dann steht der Stern für den Indexbereich der aktuell zugewiesenen Unter- und Obergrenze, die mit *LBOUND und *UBOUND festgelegt wird.

Parameter-Übertragung mit X-Arrays

Als Parameter benutzte X-Arrays werden im Hinblick auf die Überprüfung folgender Elemente wie Konstanten-Arrays behandelt:

- Format
- Länge
- Dimension

  oder
- Anzahl der Ausprägungen

Außerdem können X-Array-Parameter auch die Anzahl der Ausprägungen ändern, wenn Sie die Statements `RESIZE`, `REDUCE` oder `EXPAND` benutzen. Die RESIZE-Funktion eines X-Array-Parameters ist von drei Faktoren abhängig:

- der Art der benutzten Parameter-Übertragung, d.h. By Reference oder By Value
- der Definition des Caller oder des X-Array-Parameters
- dem Typ des weitergegebenen X-Array-Bereichs (kompletter Bereich oder Unterbereich)

Die folgenden Tabellen zeigen, wann ein `EXPAND`, `RESIZE` oder `REDUCE` eines X-Array-Parameters zulässig ist.

Beispiel mit CALL By Value

CALLER	PARAMETER		
	Statisch	Variable (1:V)	`X-Array`
Statisch	Nein	Nein	Ja
X-Array, Unterbereich, z.B. `CALLNAT...#XA(1:5)`	Nein	Nein	Ja
X-Array, kompletter Bereich, z.B. `CALLNAT...#XA(*)`	Nein	Nein	Ja

CALL By Reference/CALL By Value Result

CALLER	PARAMETER

	Statisch	Variable (1:V)	X-Array mit einer festen Untergrenze, kompletter Bereich, z.B.
			
			`DEFINE DATA ↩`
			`PARAMETER`
			`1 #PX (A10/1:*)`
				X-Array mit einer festen Obergrenze, kompletter Bereich, z.B.
				
				`DEFINE DATA ↩`
				`PARAMETER`
				`1 #PX (A10/*:1)`

Statisch	Nein	Nein	Nein	Nein
X-Array, Unterbereich, z.B. `CALLNAT...#XA(1:5)`	Nein	Nein	Nein	Nein

X-Array mit einer festen Untergrenze, kompletter Bereich, z.B.
 	
`DEFINE DATA LOCAL`
`1 #XA(A10/1:*)`
`...`
`CALLNAT...#XA(*)`
	Nein	Nein	Ja	Nein

X-Array mit einer festen Obergrenze, kompletter Bereich, z.B.
 	
`DEFINE DATA LOCAL`
`1 #XA(A10/*:1)`
`...`
`CALLNAT...#XA(*)`
	Nein	Nein	Nein	Ja

Parameter-Übertragung mit X-Group-Arrays

Die Deklaration eines X-Group-Arrays impliziert, dass jedes Element der Gruppe dieselben Werte für die Ober- und Untergrenze hat. Aus diesem Grund kann die Anzahl der Ausprägungen von abhängigen Felddimensionen eines X-Group-Arrays nur geändert werden, wenn der Gruppenname des X-Group-Arrays mit dem Statement `RESIZE`, `REDUCE` und `EXPAND` angegeben wird (siehe *Speicherverwaltung von X-Gruppen-Arrays* oben).

Bestandteile von X-Group-Arrays können als Parameter an X-Group-Arrays übertragen werden, die in einer Parameter Data Area definiert sind. Die Gruppenstrukturen des Aufrufers und des Aufgerufenen müssen nicht unbedingt identisch sein. Ein `RESIZE`, `REDUCE` und `EXPAND`, das vom Aufgerufenen gemacht wird, ist nur möglich solange das X-Group-Array des Aufrufers gleich bleibt.

Beispiel - Elemente eines X-Group Array als Parameter übertragen:

Programm:

```
DEFINE DATA LOCAL
1 #X-GROUP-ARR1(1:*)                    /* (1:*)
   2 #X-ARR1 (I4)                       /* (1:*)
   2 #X-ARR2 (I4)                       /* (1:*)
1 #X-GROUP-ARR2(1:*)                    /* (1:*)
   2 #X-ARR3 (I4)                       /* (1:*)
   2 #X-ARR4 (I4)                       /* (1:*)
END-DEFINE
...
CALLNAT ... #X-ARR1(*) #X-ARR4(*)
...
END
```

Subprogramm:

```
DEFINE DATA PARAMETER
1 #X-GROUP-ARR(1:*)                     /* (1:*)
   2 #X-PAR1 (I4)                       /* (1:*)
   2 #X-PAR2 (I4)                       /* (1:*)
END-DEFINE
...
RESIZE ARRAY #X-GROUP-ARR to (1:5)
...
END
```

Das `RESIZE`-Statement ist im Subprogramm nicht möglich. Das Ergebnis wäre eine uneinheitliche Anzahl von Ausprägungen der Felder, die in den X-Group-Arrays des Programms definiert sind.

X-Array mit dynamischen Variablen

Ein X-Array mit dynamischen Variablen kann zugewiesen werden, indem man zuerst die Anzahl der Ausprägungen mit Hilfe des `EXPAND`-Statements angibt und dann den zuvor zugewiesenen Array-Ausprägungen einen Wert zuweist.

Beispiel:

```
DEFINE DATA LOCAL
 1 #X-ARRAY(A/1:*)  DYNAMIC
END-DEFINE
EXPAND  ARRAY   #X-ARRAY TO (1:10)
  /* allocate #X-ARRAY(1) to #X-ARRAY(10) with zero length.
  /* *LENGTH(#X-ARRAY(1:10)) is zero
#X-ARRAY(*) := 'abc'
  /* #X-ARRAY(1:10) contains 'abc',
  /* *LENGTH(#X-ARRAY(1:10)) is 3
EXPAND  ARRAY   #X-ARRAY TO (1:20)
  /* allocate #X-ARRAY(11) to #X-ARRAY(20) with zero length
  /* *LENGTH(#X-ARRAY(11:20)) is zero
#X-ARRAY(11:20) := 'def'
  /* #X-ARRAY(11:20) contains 'def'
  /* *LENGTH(#X-ARRAY(11:20)) is 3
```

Unter- und Obergrenze eines Arrays

Die Systemvariablen `*LBOUND` und `*UBOUND` enthalten die aktuelle Unter- und Obergrenze eines Arrays für die angegebene(n) Dimension(en) (1, 2 oder 3).

Wenn keine Ausprägungen eines X-Arrays zugewiesen worden sind, ist der Aufruf von `*LBOUND` oder `*UBOUND` für die variablen Indexgrenzen nicht definiert, d.h. für die durch einen Stern (*) in der Indexdefinition dargestellten Grenzen, und führt zu einem Laufzeitfehler. Um einen Laufzeitfehler zu vermeiden, kann `*OCCURRENCE` benutzt werden, um auf Null-Ausprägungen zu überprüfen, bevor `*LBOUND` oder `*UBOUND` ausgewertet wird:

Beispiel:

```
IF *OCCURRENCE (#A) NE 0  AND  *UBOUND(#A) < 100 THEN ...
```

V Datenbankzugriffe

Dieser Teil beschreibt verschiedene Aspekte des Zugriffs auf Daten in einer Datenbank mit Natural.

Anmerkung: Grundsätzlich gelten die im Folgenden bei Adabas beschriebenen Merkmale und Funktionen auch für die übrigen, von Natural unterstützten Datenbankverwaltungssysteme. Unterschiede werden gegebenenfalls in den betreffenden Abschnitten der *Datenbankmanagementsystem-Schnittstellen*-Dokumentation, der *Statements*-Dokumentation bzw. der *Parameter-Referenz*-Dokumentation beschrieben.

Siehe auch *DBMS-Schnittstelle - Datenbankzugriff* in der *System-Architektur*-Dokumentation. Dieses Dokument beschreibt die Natural-Datenbank-Schnittstellen für die verschiedenen Arten von Datenbankverwaltungssystemen.

32 Natural und Datenbankzugriff

Dieses Kapitel gibt einen Überblick über die Funktionen, die Natural zum Zugriff auf verschiedene Typen von Datenbankverwaltungssystemen und Dateiverwaltungssystemen bietet.

Von Natural unterstützte Datenbankverwaltungssysteme

Natural bietet spezielle Datenbank-Schnittstellen für die folgenden Arten von Datenbankverwaltungssystemen (DBMS):

- Geschachtelte relationale DBMS (Adabas)
- DBMS vom Typ SQL (Db2), Oracle, Sybase, Informix, MS SQL Server)
- Dateisysteme (VSAM)

Die folgenden Themen werden im Folgenden erörtert:

Adabas

Über seine integrierte Adabas-Schnittstelle kann Natural auf Adabas-Datenbanken entweder auf einer lokalen Maschine oder auf entfernten Computern zugreifen. Beim entfernten Zugriff ist eine zusätzliche Weiterleitungs- und Kommunikationssoftware, wie z.B. Entire Net-Work, erforderlich. Auf jeden Fall ist die Art von Host-Maschine, auf der die Adabas-Datenbank läuft, für den Natural-Benutzer transparent.

SQL-Datenbanken

Natural for Db2 bietet einen gemeinsamen Satz Statements zum Zugriff auf Db2-Datenbankverwaltungssysteme. Weitere Informationen entnehmen Sie der entsprechenden Add-On-Produktbeschreibung in der *Database Management System Interfaces*-Dokumentation.

- *Natural for Db2*

VSAM

Mit der Natural-Schnittstelle für VSAM kann ein Natural-Benutzer auf in VSAM-Dateien gespeicherte Daten zugreifen. Detaillierte Informationen und besondere Erwägungen bei der Benutzung der Natural-Statements und Systemvariablen mit VSAM entnehmen Sie der *Natural for VSAM*-Dokumentation.

Profilparameter zur Beeinflussung der Datenbankzugriffe

Es gibt zahlreiche Profilparameter, die Einfluß darauf haben, wie Datenbankzugriffe in Natural behandelt werden.

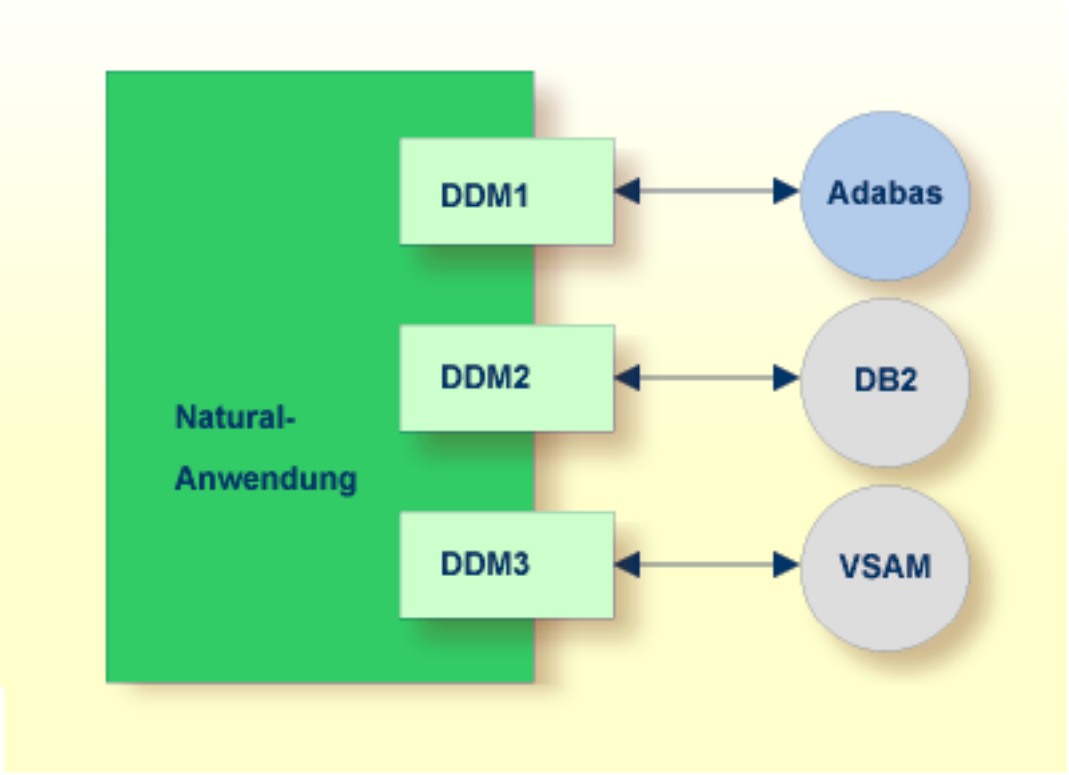
Eine Übersicht über diese Profilparameter finden Sie im Abschnitt *Datenbankverwaltung* unter *Profilparameter sortiert nach Kategorien* in der *Parameter-Referenz*-Dokumentation.

Eine ausführliche Beschreibung der dort aufgeführten Profilparameter finden Sie in den entsprechenden Abschnitten der *Parameter-Referenz*-Dokumentation.

Zugriff über Datendefinitionsmodule

Um einen komfortablen und klaren Zugriff auf die verschiedenen Datenbankverwaltungssysteme zu ermöglichen, wird ein spezifisches Objekt, das Datendefinitionsmodul (Data Definition Module/DDM), in Natural benutzt. Dieses DDM stellt die Verbindung zwischen Natural-Datenstrukturen und den Datenstrukturen im zu benutzenden Datenbanksystem her. Eine solche Datenbankstruktur könnte eine Tabelle in einer SQL-Datenbank oder eine Datei in einer Adabas-Datenbank sein. Folglich verbirgt das DDM die wahre Struktur der von der Natural-Anwendung aufgerufenen Datenbank. DDMs werden mit dem Natural DDM-Editor erstellt.

Natural kann von aus einer einzelnen Anwendung heraus auf verschiedene Arten von Datenbanken zugreifen (Adabas, Db2, oder VSAM), indem es Referenzen auf die DDMs benutzt, die die spezifischen Datenstrukturen im spezifischen Datenbanksystem darstellen. Die Abbildung weiter unten zeigt eine Anwendung, die eine Verbindung zu verschiedenen Arten von Datenbanken herstellt.



Eingebaute Datenmanipulationssprache

Natural hat eine eingebaute Datenmanipulationssprache (DML), die es Natural-Anwendungen ermöglicht, auf alle von Natural unterstützten Datenbanksysteme mittels derselben Sprach-Statements, wie z.B. `FIND`, `READ`, `STORE` oder `DELETE` zuzugreifen. Diese Statements können in einer Natural-Anwendung benutzt werden, ohne dass Sie die Art der Datenbank kennen, auf die gerade zugegriffen wird.

Natural ermittelt in seiner Konfigurationsdatei das Datenbanksystem und übersetzt die DMLStatements in datenbank-spezifische Kommandos; d.h. Natural generiert Direktkommandos für Adabas, SQL Statement-Strings und Hostvariablen-Strukturen für SQL-Datenbanken.

Da einige der Natural DML-Statements eine Funktionalität anbieten, die nicht für alle Datenbank-Arten unterstützt werden kann, ist die Benutzung dieser Funktionalität auf spezifische Datenbanksysteme beschränkt. Bitte beachten Sie die betreffenden datenbankspezifischen Erwägungen in der *Statements*-Dokumentation.

Spezielle SQL-Statements in Natural

Außer den „normalen“ Natural DML-Statements bietet Natural eine Menge von SQLStatements für einen spezifischeren Einsatz in Verbindung mit SQL-Datenbanksystemen; siehe die *Natural-SQL-Statements benutzen* in der *Statements*-Dokumentation).

Flexible SQL und Funktionen zum Verarbeiten von *Stored Procedures* vervollständigen die SQL-Kommandos. Diese Statements können nur für den Zugriff auf SQL-Datenbanken benutzt werden und gelten nicht für Adabas oder andere Nicht-SQL-Datenbanken.

33 Daten in einer Adabas-Datenbank aufrufen

Dieses Dokument beschreibt verschiedene Aspekte des Aufrufs von Daten in einer Adabas-Datenbank mit Natural.

Siehe auch *Datenbankverwaltung* in *Profilparameter sortiert nach Kategorien* in der *Parameter-Referenz*-Dokumentation. Dieses Dokument enthält eine Übersicht der Natural-Profilparameter, die gelten, wenn Natural mit einer Adabas-Datenbank benutzt wird.

Datendefinitionsmodule (DDMs)

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

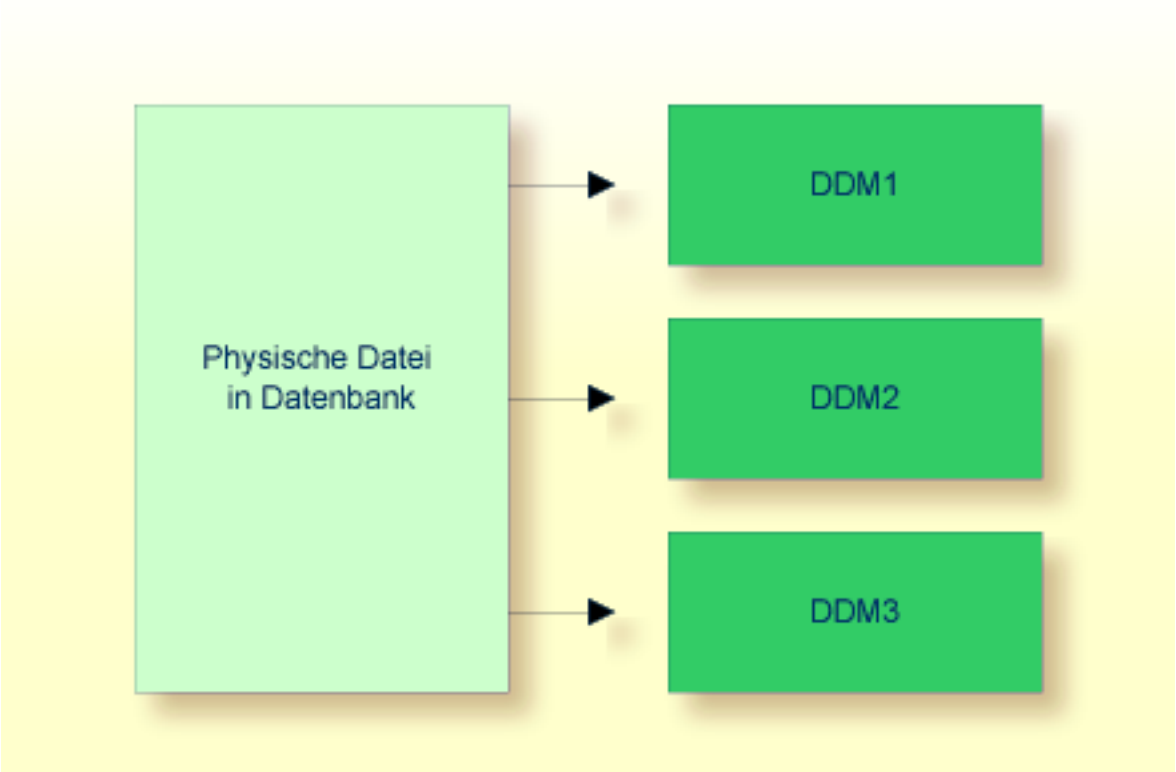
Dieser Abschnitt behandelt folgende Themen:

- Datendefinitionsmodule benutzen
- DDMs verwalten
- DDMs auflisten/anzeigen

Datendefinitionsmodule benutzen

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. Und für jedes DDM können ein oder mehrere Datensichten definiert werden (siehe *View-Definition* in der `DEFINE DATA`-Statement-Dokumentation).



DDMs werden vom Natural-Administrator mit Predict definiert (oder, falls Predict nicht vorhanden ist, mit der entsprechenden Natural-Funktion zum Verwalten von DDMs).

DDMs verwalten

Benutzen Sie das Systemkommando `SYSDDM`, um die Utility SYSDDM aufzurufen. Diese Utility bietet alle Funktionen zum Erstellen und Pflegen von Natural-Datendefinitionsmodulen.

Weitere Informationen über die SYSDDM-Utility siehe *DDM-Editor (SYSDDM Utility)* in der Natural-*Editoren*-Dokumentation.

Ein DDM enthält die datenbankinternen 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.).

Informationen zu den in einem DDM definierten Feldattributen entnehmen Sie dem Abschnitt *Spalten für Feld-Attribute* unter *DDM-Editor (SYSDDM Utility)* in der Natural *Editoren*-Dokumentation.

DDMs auflisten/anzeigen

Falls Sie den Namen des von Ihnen benötigten DDMs nicht wissen, können Sie sich mit dem Systemkommando `LIST DDM` (siehe *DDMs (Views) anzeigen* in der *Systemkommandos*-Dokumentation) eine Liste aller in der aktuellen Library 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 mit dem Kommando eine Liste aller in dem DDM `EMPLOYEES` definierten Felder mit verschiedenen Angaben zu jedem Feld.

Informationen zu den in einem DDM definierten Feldattributen entnehmen Sie dem Abschnitt *Spalten für Feld-Attribute* unter *DDM-Editor (SYSDDM Utility)* in der *Editoren*-Dokumentation.

Datenbank-Arrays

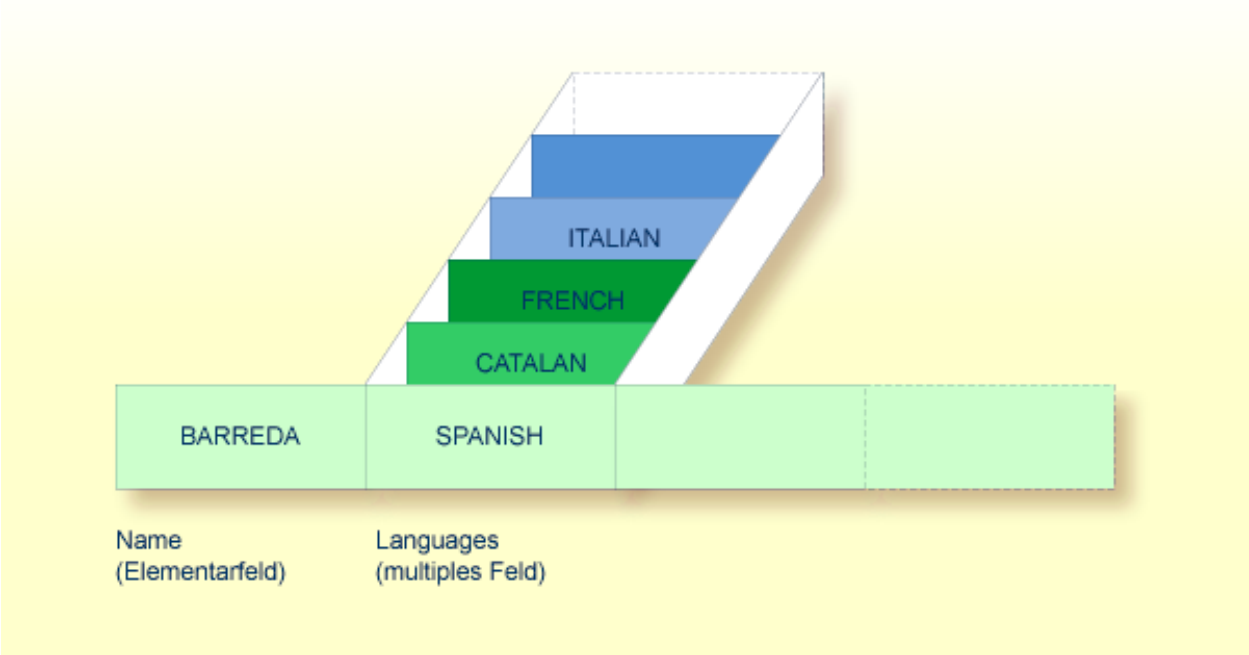
Adabas unterstützt Array-Strukturen innerhalb der Datenbank in Form von *multiplen Feldern* und *Periodengruppen*.

Dieser Abschnitt behandelt folgende Themen:

Multiple Felder

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

Beispiel:



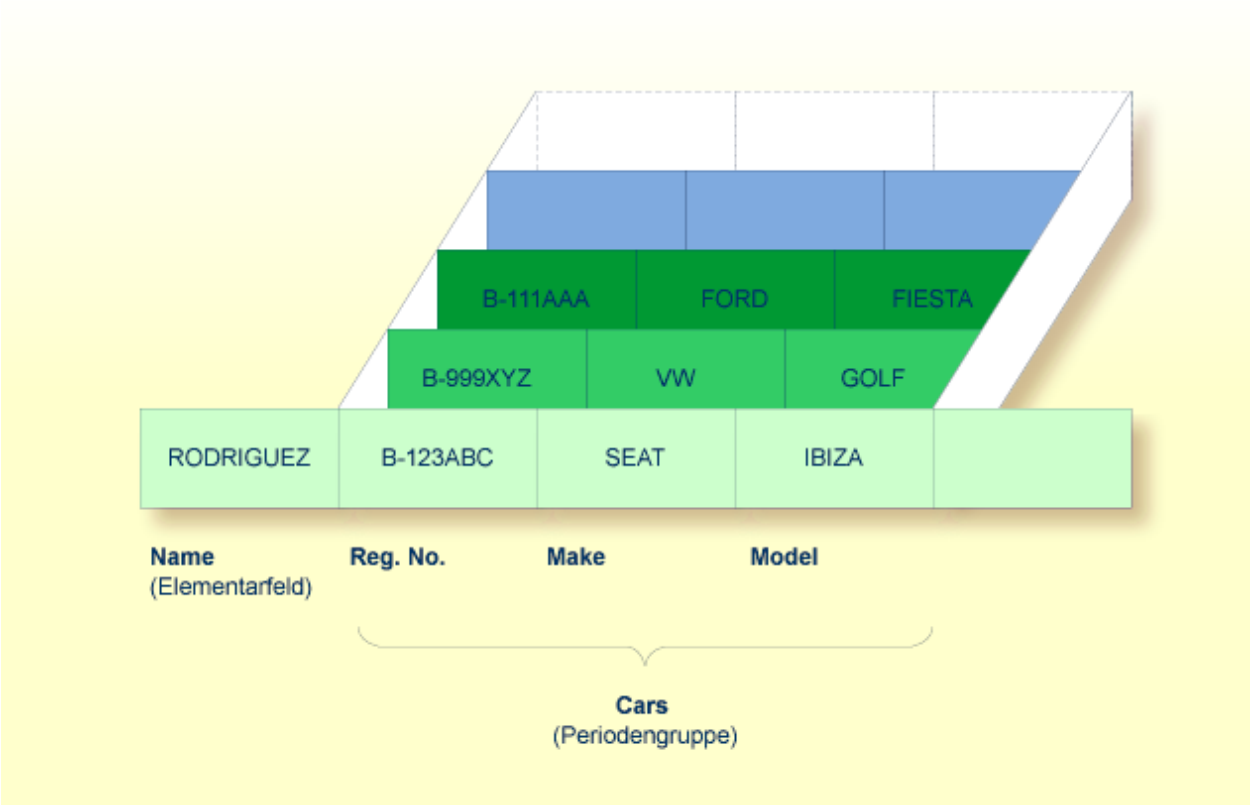
Angenommen, obige Abbildung zeigt einen Datensatz aus einer Personaldatei: das erste Feld (`Name`) ist ein Elementarfeld, das nur einen Wert enthalten kann, nämlich den Namen der Person; das zweite Feld (`Languages`) enthält die Sprachen, die die Person spricht, und ist ein multiples Feld, da eine Person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

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.

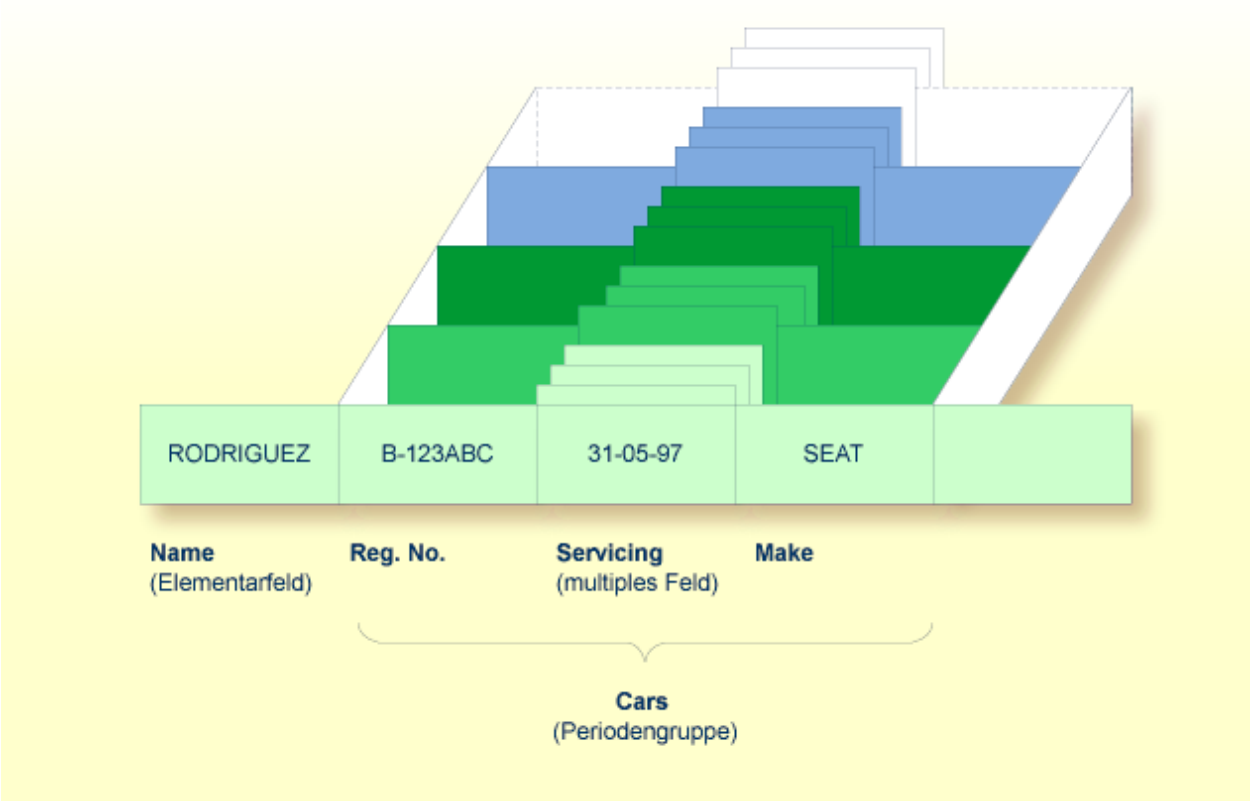
Die verschiedenen Ausprägungen der Periodengruppe `CARS` können in der gleichen Weise referenziert werden:

`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. Weitere Informationen entnehmen Sie dem Abschnitt *Internen Zähler für ein Datenbank-Array referenzieren — C*-Notation*.

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).

Datenbank-View definieren

Um Datenbankfelder in einem Natural-Programm verwenden zu können, müssen Sie sie in einer sogenannten View (Datenbanksicht) angeben.

In dem View geben Sie Folgendes an: den Namen des Datendefinitionsmoduls (siehe *Datendefinitionsmodule (DDMs)*), aus dem die Felder stammen, und die **Namen der Datenbankfelder** selbst (d.h. ihre Langnamen, nicht ihre datenbankinternen Kurznamen).

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

Wie im Abschnitt *Statements für Datenbankzugriffe* noch gezeigt wird, wird der View-Name in den Statements `READ`, `FIND`, `HISTOGRAM` verwendet, um zu bestimmen, auf welche Datenbank zugegriffen werden soll.

Weitere Informationen bezüglich der vollständigen Syntax der View-Definition oder über die Definition/Redifinition einer Gruppe von Feldern siehe *View-Definition* in der Beschreibung des `DEFINE DATA`-Statements in der *Statements*-Dokumentation.

Sie haben folgende Möglichkeiten, um eine Datenbank-View zu definieren:

- **Innerhalb des Programms**
  Sie können eine Datenbank-View innerhalb des Programms, d.h. direkt im `DEFINE DATA`-Statement des Programms definieren.
- **Außerhalb des Programms**
  Sie können eine Datenbank-View außerhalb des Programms, d.h. in einem separaten Programmierobjekt definieren: entweder in einer Local Data Area (LDA) oder in einer Global Data Area (GDA), wobei das `DEFINE DATA`-Statement dann diese Data Area referenziert.

Um eine Datenbank-View innerhalb des Programms zu definieren

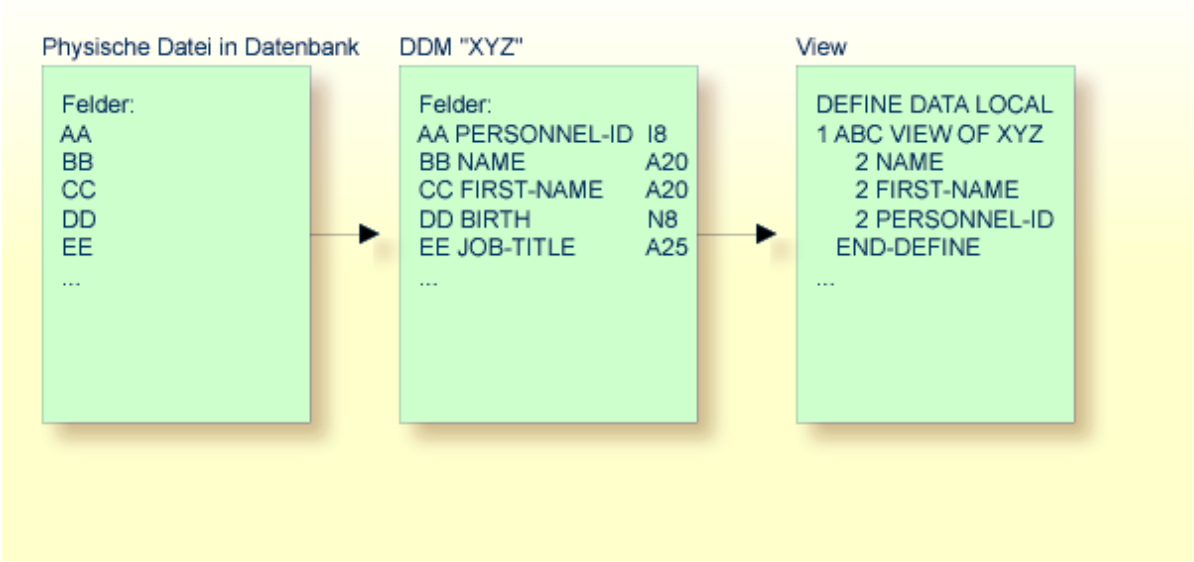
1 Auf Level 1 geben Sie den View-Namen wie folgt an:

```
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.

2 Darunter, auf Level 2, geben Sie die Namen der Datenbankfelder aus dem DDM an.

In der folgenden Abbildung hat die View den Namen `ABC` und umfasst die Felder `NAME`, `FIRST-NAME` und `PERSONNEL-ID` aus dem DDM `XYZ`.



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

Beispiel-Programm:

In diesem Beispiel lautet der View-Name `VIEWEMP`, der DDM-Name ist `EMPLOYEES` und die Namen der aus dem DDM stammenden Felder lauten `NAME`, `FIRST-NAME` und `PERSONNEL-ID`.

```
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
...
```

Um eine Datenbank-View außerhalb des Programms zu definieren

1 Im Programm selbst geben Sie an:

```
DEFINE DATA LOCAL
       USING <data-area-name>
END-DEFINE
  ...
```

wobei *data-area-name* der von Ihnen gewählte Name für die Local or Global Data Area ist, zum Beispiel `LDA39`.

2 In der im Programm referenzierten Data Area geben Sie Folgendes an:

1. Auf Level 1 in der Spalte `Name` den Namen, den Sie für die View gewählt haben, und in der Spalte `Miscellaneous` den Namen des DDM, aus dem die in der View angegebenen Felder stammen.
2. Darunter, auf Level 2, geben Sie die Namen der Datenbankfelder aus dem DDM an.

   Beispiel-Data-Area `LDA39`:

   In diesem Beispiel lautet der View-Name `VIEWEMP`, der DDM-Name ist `EMPLOYEES` und die Namen der aus dem DDM stammenden Felder lauten `PERSONNEL-ID`, `FIRST-NAME` und `NAME`.

```
I T L  Name                            F Length     Miscellaneous          ↩

All -- ------------------------------- - ---------- ------------------------>
  V  1 VIEWEMP                                      EMPLOYEES              ↩

     2 PERSONNEL-ID                    A          8                        ↩

     2 FIRST-NAME                      A         20                        ↩

     2 NAME                            A         20                        ↩

     1 #VARI-A                         A         20                        ↩

     1 #VARI-B                         N        3.2                        ↩

     1 #VARI-C                         I          4                        ↩
  ↩
```

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.

Das READ-Statement

Folgende Themen werden behandelt:

- Verwendung des READ-Statements
- Syntax-Grundform des READ-Statements
- Beispiel für READ-Statement:
- Anzahl der zu lesenden Datensätze begrenzen
- STARTING- und ENDING-Klausel beim READ-Statement
- WHERE-Klausel beim READ-Statement

Verwendung des READ-Statements

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 in der *Statements*-Dokumentation.

Syntax-Grundform des READ-Statements

Die Grundform des READ-Statements ist:

```
READ view IN LOGICAL SEQUENCE BY descriptor
```

oder kürzer:

```
READ view LOGICAL BY descriptor
```

- dabei ist

view	der Name eines im DEFINE DATA-Statement definierten Views (wie im Abschnitt *Datenbank-View definieren* beschrieben).
descriptor	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 **Schlüsselwortes** 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 für READ-Statement:

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

Ausgabe des Programms READX01:

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                                                 04-11-11  14:15:54

        NAME         PERSONNEL          CURRENT
                        ID             POSITION
-------------------- --------- -------------------------

ABELLAN              60008339  MAQUINISTA
ACHIESON             30000231  DATA BASE ADMINISTRATOR
ADAM                 50005800  CHEF DE SERVICE
ADKINSON             20008800  PROGRAMMER
ADKINSON             20009800  DBA
ADKINSON             2001100
```

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, Hyperdeskriptor, phonetischer Deskriptor oder Nicht-Deskriptor 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 beim 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-Währung (USD) bezahlt werden, lesen wollen, dann geben Sie Folgendes an:

```
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/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 in einer WHERE-Klausel verwendet werden:

EQUAL	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 'READX02': READ  (with STARTING, ENDING and WHERE clause)
************************************************************************
DEFINE DATA LOCAL
1 MYVIEW 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
STARTING FROM 'TRAINEE' ENDING AT 'TRAINEE'
                WHERE CURR-CODE (*) = 'USD'
  DISPLAY NOTITLE NAME / JOB-TITLE 5X INCOME (1:2)
  SKIP 1
END-READ
END
```

Ausgabe des Programms `READX02`:

```
          NAME                            INCOME
        CURRENT
       POSITION              CURRENCY   ANNUAL     BONUS
                               CODE     SALARY
-------------------------     -------- ---------- ----------

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 folgendes Beispiel-Programm:

- *READX03 - READ-Statement*

Das FIND-Statement

Folgende Themen werden behandelt:

- Verwendung des FIND-Statements
- Syntax-Grundform des FIND Statements
- Anzahl der zu verarbeitenden Datensätze begrenzen
- WHERE-Klausel beim FIND-Statement
- Beispiel für FIND-Statement mit WHERE-Klausel:
- IF NO RECORDS FOUND-Bedingung
- Weitere Beispiele zum FIND-Statement

Verwendung des FIND-Statements

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

Syntax-Grundform des FIND Statements

Die Grundform des `FIND`-Statements ist:

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

oder kürzer:

```
FIND view WITH field = value
```

- dabei ist

view	der Name eines im DEFINE DATA-Statement definierten Views (wie im Abschnitt *Datenbank-View definieren* beschrieben).
field	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, phonetischer Deskriptor oder ein Nicht-Deskriptor sein).

Die vollständige Syntax entnehmen Sie der FIND-Statement-Dokumentation.

Anzahl der zu verarbeitenden Datensätze begrenzen

Ähnlich wie beim READ-Statement (siehe **oben**) 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 FIND-Statement mit WHERE-Klausel:

```
** Example 'FINDX01': FIND (with WHERE)
************************************************************************
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
```

Anmerkung: 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.

Ausgabe des Programms FINDX01:

```
       CITY                 CURRENT          PERSONNEL         NAME
                           POSITION             ID
------------------- ------------------------ --------- --------------------

PARIS               INGENIEUR COMMERCIAL      50007300  CAHN
PARIS               INGENIEUR COMMERCIAL      50006500  MAZUY
PARIS               INGENIEUR COMMERCIAL      50004700  FAURIE
PARIS               INGENIEUR COMMERCIAL      50004400  VALLY
PARIS               INGENIEUR COMMERCIAL      50002800  BRETON
PARIS               INGENIEUR COMMERCIAL      50001000  GIGLEUX
PARIS               INGENIEUR COMMERCIAL      50000400  KORAB-BRZOZOWSKI
```

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 FIND-Statement mit IF NO RECORDS FOUND-Bedingung:

```
** Example 'FINDX02': FIND (with IF NO RECORDS FOUND)
************************************************************************
DEFINE DATA LOCAL
1 MYVIEW VIEW OF EMPLOYEES
  2 NAME
  2 FIRST-NAME
END-DEFINE
*
FIND MYVIEW WITH NAME = 'BLACKSMITH'
  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 `BLACKSMITH` enthält. Von jedem ausgewählten Datensatz werden der Name (`NAME`) und der Vorname (`FIRST-NAME`) angezeigt. Falls in der Datei kein Datensatz mit `NAME = 'BLACKSMITH'` gefunden wird, wird das in der `IF NO RECORDS FOUND`-Klausel angegebene `WRITE`-Statement ausgeführt:

Ausgabe des Programms `FINDX02`:

```
Page      1                                                 04-11-11  14:15:54

        NAME              FIRST-NAME
-------------------- --------------------

NO PERSON FOUND.
```

Weitere Beispiele zum FIND-Statement

Siehe die folgenden Beispiel-Programme:

- *FINDX07 - FIND (mit mehreren Klauseln)*
- *FINDX08 - FIND (mit LIMIT)*
- *FINDX09 - FIND (unter Verwendung von *NUMBER, *COUNTER, *ISN)*
- *FINDX10 - FIND (in Kombination mit READ)*
- *FINDX11 - FIND NUMBER (mit *NUMBER)*

Das HISTOGRAM-Statement

Folgende Themen werden behandelt:

- Verwendung des HISTOGRAM-Statements
- Syntax-Grundform des HISTOGRAM-Statements
- Anzahl der zu lesenden Werte begrenzen
- STARTING- und ENDING-Klausel beim HISTOGRAM-STATEMENT
- WHERE-Klausel beim HISTOGRAM-Statement

- Beispiel für HISTOGRAM-Statement

Verwendung des HISTOGRAM-Statements

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.

Syntax-Grundform des HISTOGRAM-Statements

Die Grundform des HISTOGRAM-Statements ist:

```
HISTOGRAM VALUE IN view FOR field
```

oder kürzer:

```
HISTOGRAM view FOR field
```

- dabei ist

view	der Name eines im DEFINE DATA-Statement definierten Views (wie im Abschnitt *Datenbank-View definieren* beschrieben).
field	der Name des in diesem View definierten Datenbankfeldes.

Die vollständige Syntax entnehmen Sie der HISTOGRAM-Statement-Dokumentation.

Anzahl der zu lesenden Werte begrenzen

Ähnlich wie beim READ-Statement (siehe **oben**) 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 beim HISTOGRAM-STATEMENT

Wie das READ-Statement (siehe **oben**) 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 'HISTOX01': HISTOGRAM
************************************************************************
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
```

In diesem 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.

Ausgabe des Programms HISTOX01:

```
        CITY          NUMBER OF     CNT
                       PERSONS
-------------------- ----------- -----------

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
```

Multi-Fetch-Klausel

Dieser Abschnitt behandelt die *Multi-Fetch*-Datensatz-Retrieval-Funktionalität für Adabas-Datenbanken.

Die Multi-Fetch-Funktionalität wird nur bei Adabas unterstützt. Informationen zur *Multi-Fetch*-Datensatz-Retrieval-Funktionalität für Db2-Datenbanken siehe *Multiple Row Processing* in der *Natural for Db2*-Dokumentation.

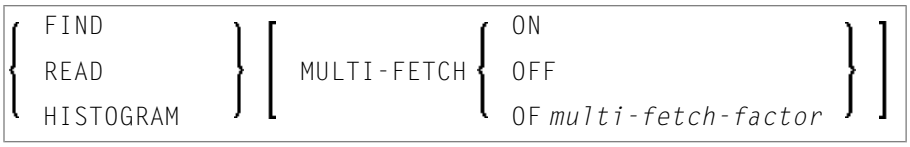
Folgende Themen werden behandelt:

- Zweck der Multi-Fetch-Funktion
- Anmerkungen zur Multi-Fetch-Benutzung
- Größe des Multi-Fetch-Puffers
- Unterstützung von TEST DBLOG

Zweck der Multi-Fetch-Funktion

Im Standardmodus liest Natural mit einem einzigen Datenbank-Aufruf nicht mehrere Datensätze ein, sondern stets nur einen Datensatz pro Fetch-Modus. Diese Art von Betrieb ist solide und stabil, es kann aber einige Zeit dauern, wenn eine große Anzahl von Datenbank-Sätzen verarbeitet wird.

Um die Verarbeitungszeit dieser Programme zu verbessern, können Sie die Multi-Fetch-Klausel in den `FIND`-, `READ`- oder `HISTOGRAM`-Statements benutzen. Mit der Multi-Fetch-Klausel können Sie den Multi-Fetch-Faktor definieren. Dies ist ein numerischer Wert, der die Anzahl der pro Datenbank-Zugriff eingelesenen Datensätze angibt.



Dabei ist der *multi-fetch-factor* entweder eine

- numerische Konstante im Wertebereich (0 : 2147483647) oder
- eine Variable mit dem Format integer/binär (B1:B4) oder gepackt/numerisch nur mit Ganzzahlen.

Bei der Ausführung des Statements überprüft die Laufzeitumgebung, ob für das Datenbank-Statement ein Multi-Fetch-Faktor größer als 1 angegeben ist.

Wenn der Multi-Fetch-Faktor

ein negativer Wert ist,	wird ein Laufzeitfehler ausgegeben.
0 oder 1 ist,	wird der Datenbank-Aufruf im normalen Modus mit einem Datensatz pro Zugriff fortgesetzt.
2 oder größer ist,	wird der Datenbank-Aufruf dynamisch aufbereitet, um mehrere Datensätze (z.B. 10) mit einem einzigen Datenbank-Zugriff in einen Hilfspuffer (Multi-Fetch-Puffer) einzulesen. Bei Erfolg wird der erste Satz in den zugrunde liegenden Daten-View übertragen. Bei Ausführung der nächsten Schleife wird der Daten-View ohne Datenbank-Zugriff direkt vom Multi-Fetch-Puffer aufgefüllt. Nachdem alle Datensätze vom Multi-Fetch-Puffer eingelesen sind, führt die nächste Schleife dazu, dass der nächste Datensatz von der Datenbank eingelesen wird. Wird die Datenbank-Schleife beendet (entweder durch End-of-Records, `ESCAPE`, `STOP` usw.), wird der Inhalt des Multi-Fetch-Puffers freigegeben.

Anmerkungen zur Multi-Fetch-Benutzung

- Ein Multi-Fetch-Zugriff wird nur für eine Auflist-Schleife („Browse Loop“) unterstützt, mit anderen Worten, wenn die Datensätze ohne Sperrung („No Hold“) gelesen werden.
- Das Programm empfängt von der Datenbank keine „frischen“ Datensätze für jede Schleife, sondern bearbeitet die beim letzten Multi-Fetch-Zugriff eingelesenen Abbilder.
- Wird eine Repositionierung für ein `READ` oder `HISTOGRAM`-Statement ausgelöst, dann wird der Inhalt des Multi-Fetch-Puffers zu diesem Zeitpunkt freigegeben.
- In folgenden Fällen ist die Multi-Fetch-Funktion nicht möglich und führt zu einem entsprechenden Syntax-Fehler bei der Kompilierung:
  - Wenn eine dynamische Änderung der Richtung (`IN DYNAMIC...SEQUENCE`) für ein `READ` oder `HISTOGRAM`-Statement kodiert ist, oder
  - wenn eine `IN SHARED HOLD`-Klausel in einem `READ`- oder `FIND`-Statement benutzt wird.

- Der erste Datensatz einer `FIND`-Schleife wird mit dem einleitenden `S1`-Kommando eingelesen. Da ein Adabas Multi-Fetch für alle Arten von `Lx`-Kommandos definiert wird, kann es erst ab dem zweiten Datensatz benutzt werden.
- Die von der Datenbank-Schleife im Multi-Fetch-Puffer eingenommene Größe wird entsprechend der folgenden Regel festgelegt:

```
((record-buffer-length + isn-buffer-entry-length) * multi-fetch-factor ) + 4 + header-length
=
((size-of-view-fields + 20) * multi-fetch-factor) + 4 + 128
```

Der Multi-Fetch-Faktor wird zur Laufzeit automatisch reduziert, wenn

- das „Loop-Limit“ (z.B. `READ (2) ..`) kleiner ist, jedoch nur wenn keine `WHERE`-Klausel beteiligt ist,
- die „ISN-Menge“ kleiner ist (gilt nur beim `FIND`-Statement),
- die Größe des Multi-Fetch-Puffers nicht ausreicht, um die vom Multi-Fetch-Faktor angeforderte Anzahl an Datensätzen aufzunehmen.

Des Weiteren wird die Multi-Fetch-Option zur Laufzeit völlig ignoriert, wenn

- der Multi-Fetch-Faktor einen Wert kleiner gleich 1 enthält;
- der Multi-Fetch-Puffer nicht verfügbar ist, oder nicht genügend freien Speicherplatz hat (weitere Einzelheiten siehe *Größe des Multi-Fetch-Puffers* weiter unten).

Beispiel:

Das ausgeführte Statement lautet `READ MULTI-FETCH 100 EMPL-VIEW BY NAME`. Die Datensatzgröße (Länge der Felder im View `EMPL-VIEW` beträgt 1000 Bytes. Der Multi-Fetch-Puffer hat eine Größe von 64 KB. Zur Laufzeit wird der Multi-Fetch-Faktor automatisch von 100 auf 64 verringert, so dass der gesamte Datensatzpuffer in den Multi-Fetch-Puffer passt.

Größe des Multi-Fetch-Puffers

Um die für Multi-Fetch-Zwecke verfügbare Speichermenge zu steuern, können Sie die maximale Größe des Multi-Fetch-Puffers begrenzen.

Im Natural-Parametermodul können Sie mittels des Parameter-Makros `NTDS` eine statische Zuweisung durchführen:

```
NTDS MULFETCH,nn
```

Beim Start der Session können Sie auch den Profilparameter DS benutzen:

```
DS=(MULFETCH,nn)
```

wobei *nn* die vollständige Größe darstellt, die zur Zuweisung für Multi-Fetch-Zwecke (in KB) zulässig ist. Der Wert kann im Bereich 0 – 1024 mit einem Standardwert von 64 gesetzt werden. Das Setzen eines hohen Wertes bedeutet nicht unbedingt, dass ein Puffer dieser Größe zugewiesen wird, da der Multi-Fetch-Handler dynamische Zuweisungen und Größenänderungen vornimmt, und zwar in Abhängigkeit davon, was wirklich erforderlich ist, um den Multi-Fetch-Datenbankaufruf auszuführen. Wenn kein Multi-Fetch-Datenbankaufruf in einer Natural-Session ausgeführt wird, wird der Multi-Fetch-Puffer ungeachtet des gesetzten Wertes auf keinen Fall erstellt.

Wenn der Wert 0 angegeben wird, wird die Multi-Fetch-Verarbeitung vollständig ausgeschaltet, ganz gleich ob ein Datenbankzugriffs-Statement eine MULTI-FETCH OF ..-Klausel enthält oder nicht. Dadurch wird es Ihnen ermöglicht, alle Multi-Fetch-Aktivitäten komplett auszuschalten, wenn nicht genügend Speicherplatz in der aktuellen Umgebung verfügbar ist, oder für Fehlerbeseitigungszwecke.

Anmerkungen:

1. Zur Ausführung eines Multi-Fetch-Aufrufs ist ein Benutzer-Zwischenspeicherbereich in Adabas erforderlich. Die Größe dieses Zwischenspeichers wird mit dem Adabas-Parameter LU gesetzt, wobei der Standardwert 65535 (64 KB) ist. Falls der Wert des Adabas-Parameters LU kleiner als die Größe des MULFETCH-Zwischenspeichers ist, kann bei einer Multi-Fetch-Verarbeitung der Natural-Laufzeitfehler NAT3152 (interner Benutzer-Zwischenspeicherbereich zu klein) auftreten.

   Derartige Fehler können Sie vermeiden, indem Sie die Größe des MULFETCH-Zwischenspeichers (Standardeinstellung ist 64 KB) auf den gleichen Wert oder einen kleineren Wert als die Größe des Adabas-Zwischenspeichers (Standardeinstellung ist 64 KB, gesetzt mit dem Adabas-Parameter LU) setzen. Das bedeutet, dass Sie, falls Sie den MULFETCH-Zwischenspeicher vergrößern, auch den Adabas-Zwischenspeicher entsprechend vergrößern müssen. Beispiel: Setzen Sie LU=102400 (100 KB) wenn DS=(MULFETCH,100).
2. Der Natural-MULFETCH-Zwischenspeicher hat einen nicht benutzten Reservespeicherplatz von 6 KB. Dadurch werden NAT3152-Laufzeitfehler vermieden, die auftreten können, wenn der Adabas-Parameter LU auf 64 KB (Standardwert) oder höher gesetzt wird und gleich groß wie oder größer als der MULFETCH-Zwischenspeicher ist.
3. Ein Multi-Fetch-Aufruf wird gewöhnlich im ACB-Layout ausgeführt (ACB = Adabas Control Block).

   Ein Multi-Fetch-Aufruf wird jedoch im ACBX-Layout ausgeführt (ACBX = erweiterter Adabas Control Block), wenn Folgendes zutrifft:

   - Der Natural-Profilparameter ADAACBX ist auf ON gesetzt.

- Die Gesamtgröße des Record Buffer (= Einzel-Satzlänge * Multi-Fetch-Faktor) des Multi-Fetch-Aufrufs ist größer als 32 KB.
- Der Natural-Profilparameter `DB` ist beim Katalogisieren des Programms für die Datenbank, auf die zugegriffen wird, auf Adabas Version 8 (oder höher) gesetzt.
- Der Natural-`MULFETCH`-Zwischenspeicher ist größer als 32 KB (Standardeinstellung ist 64 KB).

Unterstützung von TEST DBLOG

Informationen, wie Multi-Fetch-Datenbankaufrufe von der Utility `TEST DBLOG` unterstützt werden, entnehmen Sie der Beschreibung der *DBLOG Utility*, *Adabas-Kommandos anzeigen, die Multi-Fetch benutzen* in der *Debugger und Dienstprogramme*-Dokumentation.

Datenbank-Verarbeitungsschleifen

Dieser Abschnitt erörtert Verarbeitungsschleifen, die zum Abarbeiten von Daten erforderlich sind, welche von einer Datenbank als Ergebnis eines `FIND`-, `READ`- oder `HISTOGRAM`-Statements ausgewählt wurden.

Folgende Themen werden behandelt:

Erstellung von 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.

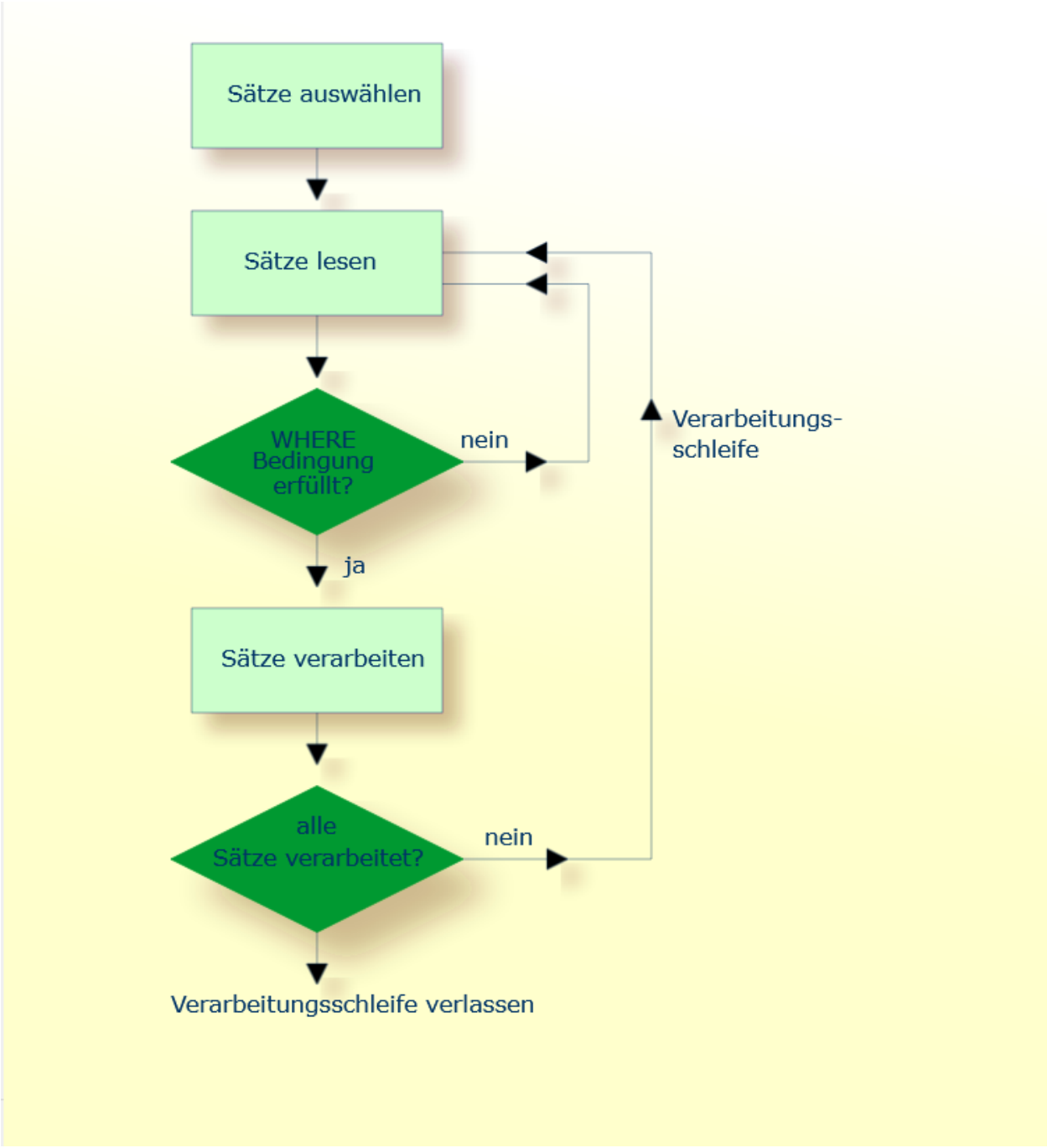
Beispiel:

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.

```
** Example 'FINDX03': FIND
************************************************************************
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
```

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:


Sätze auswählen
Sätze lesen
WHERE Bedingung erfüllt?
nein
ja
Verarbeitungs-schleife
Sätze verarbeiten
alle Sätze verarbeitet?
nein
Verarbeitungsschleife verlassen

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:

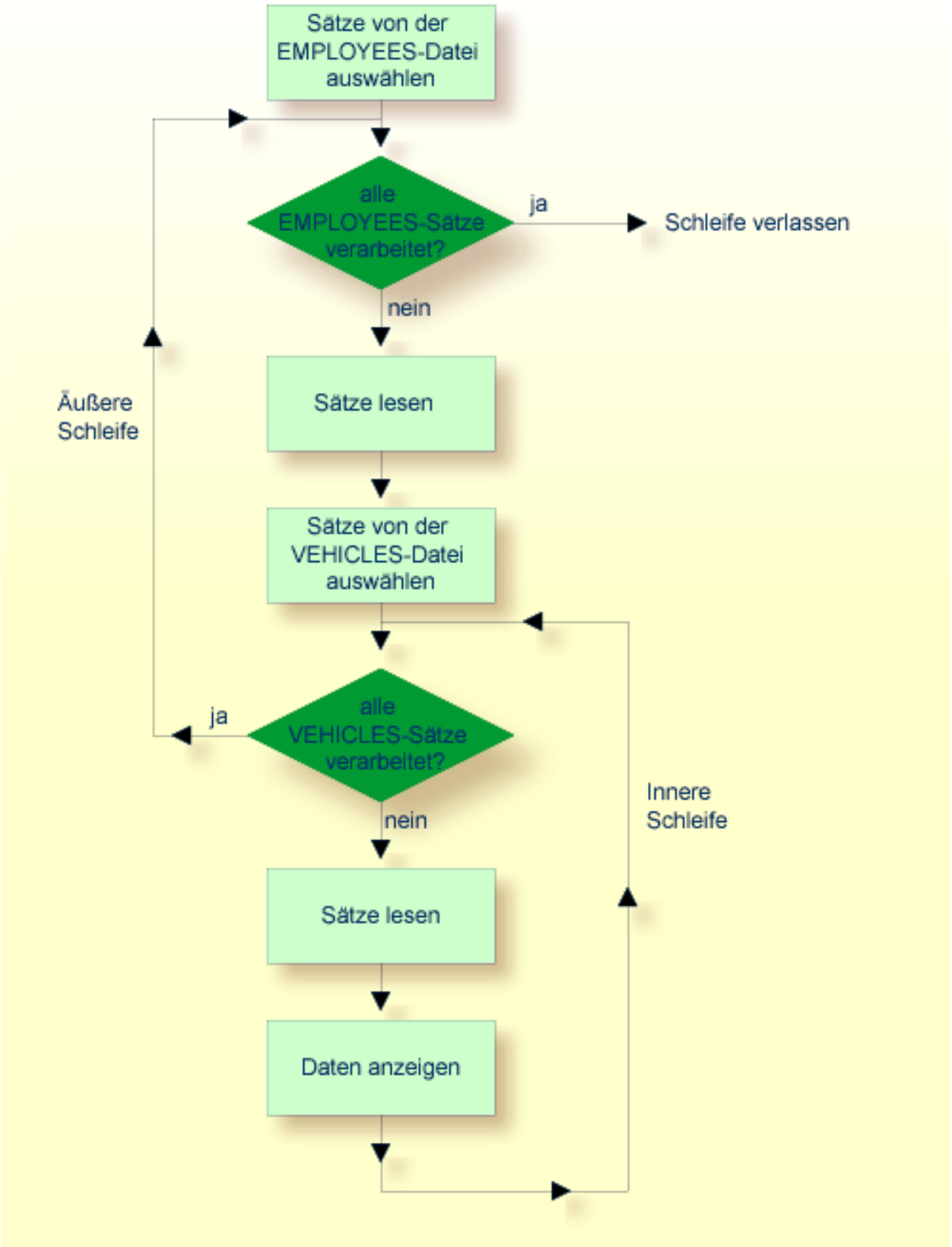
```
** Example 'FINDX04': FIND  (two FIND statements nested)
************************************************************************
DEFINE DATA LOCAL
1 PERSONVIEW VIEW OF EMPLOYEES
  2 PERSONNEL-ID
  2 NAME
1 AUTOVIEW VIEW OF VEHICLES
  2 PERSONNEL-ID
  2 MAKE
  2 MODEL
END-DEFINE
*
EMP. FIND PERSONVIEW 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 (jede Person) 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 mit DISPLAY folgende Werte angezeigt: 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 folgende Diagramm zeigt den logischen Ablauf in der Datenbank-Verarbeitungsschleifen-Hierarchie in dem obigen Beispiel-Programm:


Sätze von der EMPLOYEES-Datei auswählen
alle EMPLOYEES-Sätze verarbeitet?
ja
Schleife verlassen
nein
Äußere Schleife
Sätze lesen
Sätze von der VEHICLES-Datei auswählen
alle VEHICLES-Sätze verarbeitet?
ja
nein
Innere Schleife
Sätze lesen
Daten anzeigen

Beispiel für geschachtelte FIND-Schleifen, die dieselbe Datei aufrufen

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

```
** Example 'FINDX05': FIND (two FIND statements on same file nested)
************************************************************************
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.

Ausgabe des Programms `FINDX05`:

```
PEOPLE IN SAME CITY AS: JONES LAUREL
CITY: BALTIMORE

        NAME              FIRST-NAME              CITY
-------------------- -------------------- --------------------

JENSON               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 die folgenden Beispiel-Programme:

- *READX04 - READ-Statement (in Kombination mit FIND und den Systemvariablen *NUMBER und *COUNTER)*
- *LIMITX01 - LIMIT-Statement (für READ- und FIND-Schleifenverarbeitung)*

Datenänderungen - Transaktionsverarbeitung

Dieser Abschnitt beschreibt, wie Natural Datenbankänderungsoperationen mittels Transaktionen durchführt.

Folgende Themen werden behandelt:

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 Natural-Profilparameter `ETEOP` 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 ein STORE-Statement:

In dem folgenden Beispiel-Programm werden neue Datensätze der `EMPLOYEES`-Datei hinzugefügt.

Vorsicht: Wenn Sie dieses Beispielprogramm ausführen, verändern Sie Datensätze in der Datenbank.

```
** Example 'STOREX01': STORE  (Add new records to EMPLOYEES file)
*
** CAUTION: Executing this example will modify the database records!
************************************************************************
DEFINE DATA LOCAL
1 EMPLOYEE-VIEW VIEW OF EMPLOYEES
  2 PERSONNEL-ID(A8)
  2 NAME        (A20)
  2 FIRST-NAME  (A20)
  2 MIDDLE-I    (A1)
  2 SALARY      (P9/2)
  2 MAR-STAT    (A1)
  2 BIRTH       (D)
  2 CITY        (A20)
  2 COUNTRY     (A3)
*
1 #PERSONNEL-ID (A8)
1 #NAME         (A20)
1 #FIRST-NAME   (A20)
1 #INITIAL      (A1)
1 #MAR-STAT     (A1)
1 #SALARY       (N9)
```

```
1 #BIRTH        (A8)
1 #CITY         (A20)
1 #COUNTRY      (A3)
1 #CONF         (A1)   INIT <'Y'>
END-DEFINE
*
REPEAT
  INPUT 'ENTER A PERSONNEL ID AND NAME (OR ''END'' TO END)' //
        'PERSONNEL-ID : ' #PERSONNEL-ID    //
        'NAME         : ' #NAME  /
        'FIRST-NAME   : ' #FIRST-NAME
  /*********************************************************************
  /*  validate entered data
  /*********************************************************************
  IF #PERSONNEL-ID = 'END' OR #NAME = 'END'
    STOP
  END-IF
  IF #NAME = ' '
    REINPUT WITH TEXT 'ENTER A LAST-NAME'
            MARK 2 AND SOUND ALARM
  END-IF
  IF #FIRST-NAME = ' '
    REINPUT WITH TEXT 'ENTER A FIRST-NAME'
            MARK 3 AND SOUND ALARM
  END-IF
  /*********************************************************************
  /*  ensure person is not already on file
  /*********************************************************************
  FIP2. FIND NUMBER EMPLOYEE-VIEW WITH PERSONNEL-ID =  #PERSONNEL-ID
  /*
  IF *NUMBER (FIP2.) > 0
    REINPUT 'PERSON WITH SAME PERSONNEL-ID ALREADY EXISTS'
             MARK 1 AND SOUND ALARM
  END-IF
  /*********************************************************************
  /*  get further information
  /*********************************************************************
  INPUT
    'ENTER EMPLOYEE DATA'                          ////
    'PERSONNEL-ID             :' #PERSONNEL-ID (AD=IO) /
    'NAME                     :' #NAME         (AD=IO) /
    'FIRST-NAME               :' #FIRST-NAME   (AD=IO) ///
    'INITIAL                  :' #INITIAL              /
    'ANNUAL SALARY            :' #SALARY               /
    'MARITAL STATUS           :' #MAR-STAT             /
    'DATE OF BIRTH (YYYYMMDD) :' #BIRTH                /
    'CITY                     :' #CITY                 /
    'COUNTRY (3 CHARS)        :' #COUNTRY              //
    'ADD THIS RECORD (Y/N)    :' #CONF         (AD=M)
  /*********************************************************************
  /*  ENSURE REQUIRED FIELDS CONTAIN VALID DATA
  /*********************************************************************
```

```
  IF #SALARY < 10000
    REINPUT TEXT 'ENTER A PROPER ANNUAL SALARY' MARK 2
  END-IF
  IF NOT (#MAR-STAT = 'S'  OR = 'M' OR = 'D' OR = 'W')
    REINPUT TEXT 'ENTER VALID MARITAL STATUS  S=SINGLE ' -
                 'M=MARRIED D=DIVORCED W=WIDOWED' MARK 3
  END-IF
  IF NOT(#BIRTH = MASK(YYYYMMDD) AND #BIRTH = MASK(1582-2699))
    REINPUT TEXT 'ENTER CORRECT DATE' MARK 4
  END-IF
  IF #CITY  = ' '
    REINPUT TEXT 'ENTER A CITY NAME' MARK 5
  END-IF
  IF #COUNTRY = ' '
    REINPUT TEXT 'ENTER A COUNTRY CODE' MARK 6
  END-IF
  IF NOT (#CONF = 'N' OR= 'Y')
    REINPUT TEXT 'ENTER Y (YES) OR N (NO)' MARK 7
  END-IF
  IF #CONF = 'N'
    ESCAPE TOP
  END-IF
  /*********************************************************************
  /*  add the record with STORE
  /*********************************************************************
  MOVE #PERSONNEL-ID TO EMPLOYEE-VIEW.PERSONNEL-ID
  MOVE #NAME         TO EMPLOYEE-VIEW.NAME
  MOVE #FIRST-NAME   TO EMPLOYEE-VIEW.FIRST-NAME
  MOVE #INITIAL      TO EMPLOYEE-VIEW.MIDDLE-I
  MOVE #SALARY       TO EMPLOYEE-VIEW.SALARY (1)
  MOVE #MAR-STAT     TO EMPLOYEE-VIEW.MAR-STAT
  MOVE EDITED #BIRTH TO EMPLOYEE-VIEW.BIRTH (EM=YYYYMMDD)
  MOVE #CITY         TO EMPLOYEE-VIEW.CITY
  MOVE #COUNTRY      TO EMPLOYEE-VIEW.COUNTRY
  /*
  STP3. STORE RECORD IN FILE EMPLOYEE-VIEW
  /*
  /*********************************************************************
  /*  mark end of logical transaction
  /*********************************************************************
  END OF TRANSACTION
  RESET INITIAL #CONF
END-REPEAT
END
```

Ausgabe des Programms `STOREX01`:

```
ENTER A PERSONNEL ID AND NAME (OR 'END' TO END)

PERSONNEL ID :

NAME         :
FIRST NAME   :
```

Datensatz-Kontrolle während einer Transaktion (Hold-Logik)

Wird Natural zusammen mit Adabas eingesetzt, so wird jeder Datensatz, der verändert werden soll, solange in den Hold-Status 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-Status 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 verhindern, dass ein Benutzer im Wartestatus verbleibt, ist es möglich den Session-Parameter `WH` (Wait Hold) entsprechend zu setzen (siehe *Parameter-Referenz*-Dokumentation).

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

- Die Zeit, für die ein Datensatz höchstens in den Hold-Status 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 erfolgten, werden rückgängig gemacht.
- Die Anzahl der ISNs im Hold-Status 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.
- Das „Halten“ wird von Datensätzen auch vom Natural-Profilparameter `RI` (Release ISNs) gesteuert, der vom Natural-Administrator gesetzt wird.

Beispiel für Hold-Logik:

Vorsicht: Wenn Sie dieses Beispielprogramm ausführen, verändern Sie Datensätze in der Datenbank.

```
** Example 'GETX01': GET (put single record in hold with UPDATE stmt)
**
** CAUTION: Executing this example will modify the database records!
************************************************************************
DEFINE DATA LOCAL
1 EMPLOY-VIEW VIEW OF EMPLOYEES
  2 NAME
  2 SALARY (1)
END-DEFINE
*
RD. READ EMPLOY-VIEW BY NAME
  DISPLAY EMPLOY-VIEW
  IF SALARY (1) > 1500000
    /*
    GE. GET EMPLOY-VIEW *ISN (RD.)
    /*
    WRITE '=' (50) 'RECORD IN HOLD:' *ISN(RD.)
    COMPUTE SALARY (1) = SALARY (1) * 1.15
    UPDATE (GE.)
    END TRANSACTION
  END-IF
END-READ
END
```

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.

Vorsicht: Wenn Sie dieses Beispielprogramm ausführen, verändern Sie Datensätze in der Datenbank.

```
** Example 'GETTRX01': GET TRANSACTION
*
** CAUTION: Executing this example will modify the database records!
************************************************************************
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
*
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'
        ESCAPE BOTTOM
      END-NOREC
      IF *COUNTER (FIND2.) = 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 IP=OFF) //////// 28T MAKE / 28T 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
```

Datensätze mit ACCEPT/REJECT auswählen

Dieser Abschnitt behandelt die Statements ACCEPT und REJECT, die Sie zur Auswahl von Datensätzen anhand von Ihnen definierter logischer Auswahlkriterien verwenden können.

Folgende Themen werden behandelt:

Mit ACCEPT und REJECT verwendbare Statements

Sie können ACCEPT und REJECT zusammen mit den folgenden Datenbankzugriffs-Statements verwenden:

- READ
- FIND
- HISTOGRAM

Beispiel für ein ACCEPT-Statement

```
** Example 'ACCEPX01': ACCEPT IF
************************************************************************
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
```

Ausgabe des Programms ACCEPX01:

```
 Page      1                                                  04-11-11  11:11:11

        NAME                  CURRENT             ANNUAL
                             POSITION             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
```

Logische Bedingungen in ACCEPT/REJECT-Statements

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 siehe ***Logische Bedingungen***):

EQUAL	EQ	:=
NOT EQUAL TO	NE	¬=
LESS THAN	LT	<
LESS EQUAL	LE	<=
GREATER THAN	GT	>
GREATER EQUAL	GE	>=

Außerdem können Sie die Boole'schen Operatoren AND, OR und NOT zur Verknüpfung logischer Bedingungen in ACCEPT / REJECT-Statements einsetzen; mit Klammern können Sie die Bedingungen in logische Einheiten unterteilen, siehe folgende Beispiele.

Beispiel für ACCEPT-Statement mit AND-Operator

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

```
** Example 'ACCEPX02': ACCEPT IF ... AND ...
************************************************************************
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
```

Ausgabe des Programms `ACCEPX02`:

```
Page      1                                                 04-12-14  12:22:01

        NAME                  CURRENT            ANNUAL
                             POSITION            SALARY
-------------------- ------------------------- ----------

AFANASSIEV           DBA                            42800
ATHERTON             ANALYST                        43000
ATHERTON             MANAGER                        40000
```

Beispiel für REJECT-Statement mit OR-Operator

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

```
** Example 'ACCEPX03': REJECT IF ... OR ...
************************************************************************
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
```

Ausgabe des Programms `ACCEPX03`:

```
Page      1                                                 04-12-14  12:26:27

        NAME                  CURRENT            ANNUAL
                             POSITION            SALARY
-------------------- ------------------------ ----------

AFANASSIEV           DBA                           42800
ATHERTON             ANALYST                       43000
ATHERTON             MANAGER                       40000
```

Weitere Beispiele für ACCEPT- und REJECT-Statements

Siehe die folgenden Beispiel-Programme:

- ***ACCEPX04 - ACCEPT IF ... LESS THAN ...***
- ***ACCEPX05 - ACCEPT IF ... AND ...***
- ***ACCEPX06 - REJECT IF ... OR ...***

AT START/END OF DATA-Statements

Dieser Abschnitt behandelt die Verwendung der Statements `AT START OF DATA` und `AT END OF DATA`.

Folgende Themen werden behandelt:

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.

Beispiel für AT START OF DATA- und AT END OF DATA-Statement

Das folgende Beispielprogramm 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.

```
** Example 'ATSTAX01': AT START OF DATA
************************************************************************
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
```

Ausgabe des Programms `ATSTAX01`:

```
                  XYZ EMPLOYEE ANNUAL SALARY AND BONUS REPORT

     NAME           CURRENT                  INCOME
                    POSITION
                                  CURRENCY   ANNUAL     BONUS
                                    CODE     SALARY
--------------- --------------- -------- ---------- ----------

RUN TIME: 12:43:19.1

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-Statement

Siehe die folgenden Beispiel-Programme.

- *ATENDX01 - AT END OF DATA*
- *ATSTAX02 - AT START OF DATA*
- *WRITEX09 - WRITE-Statement (in Kombination mit AT END OF DATA)*

Unicode-Daten

Natural ermöglicht es den Benutzern, auf Wide-Character-Fields mit dem Format W in einer Adabas-Datenbank zuzugreifen.

In diesem Abschnitt werden folgende Themen behandelt:

Datendefinitionsmodul

Adabas Wide-Character-Fields (W) werden auf Natural-Format U (Unicode) abgebildet.

Die Längen-Definition für ein Natural-Feld vom Format U entspricht der Hälfte der Größe des Adabas-Feldes mit dem Format W. Ein Adabas Wide-Character-Field der Länge 200 wird zum Beispiel auf `(U100)` in Natural abgebildet.

Zugriffskonfiguration

Natural erhält Daten aus Adabas und sendet Daten zurück an Adabas mittels UTF-16 als gemeinsam benutzte Kodierung.

Diese Kodierung wird mit dem Profilparameter `OPRB` angegeben und an Adabas mit der offenen Anforderung versandt. Sie wird für Wide-Character-Fields benutzt und gilt für die gesamte Adabas Benutzer-Session.

Einschränkungen

Sortierfolgen-Deskriptoren werden nicht unterstützt.

Weitere Informationen zu Adabas und Unicode-Unterstützung können Sie der spezifischen Adabas Produkt-Dokumentation entnehmen.

34 Daten in einer SQL-Datenbank aufrufen

Im Prinzip gelten die in dem Dokument *Daten in einer Adabas-Datenbank aufrufen* enthaltenen Funktionen und Beispiele auch für die von Natural unterstützten SQL-Datenbanken.

Falls es Unterschiede gibt, sind diese in den Dokumenten für die einzelnen Datenbankzugriffs-Statements beschrieben (siehe *Statements*-Dokumentation), in Abschnitten namens *Datenbank-spezifische Bemerkungen* oder in den Beschreibungen zu den einzelnen Natural-Parametern (siehe *Parameter-Referenz*-Dokumentation).

Des Weiteren bietet die Natural-Db2-Schnittstelle eine Reihe von speziellen Statements zum Zugriff auf Db2-Datenbankverwaltungssysteme. Informationen hierzu können Sie den entsprechenden Produkt-Beschreibungen in der *Datenbankmanagementsystem-Schnittstellen*-Dokumentation entnehmen:

- *Natural for Db2*

35 Daten in einer VSAM-Datenbank aufrufen

Im Prinzip gelten die in dem Dokument ***Daten in einer Adabas-Datenbank aufrufen*** enthaltenen Funktionen und Beispiele auch für VSAM-Datenbanken.

Falls es Unterschiede gibt, sind diese in den Dokumenten für die einzelnen Datenbankzugriffs-Statements beschrieben (siehe *Statements*-Dokumentation), in Abschnitten namens *Datenbank-spezifische Bemerkungen* oder in den Beschreibungen zu den einzelnen Natural-Parametern (siehe *Parameter-Referenz*-Dokumentation).

Informationen hierzu können Sie der entsprechenden Produkt-Beschreibung in der *Datenbankmanagementsystem-Schnittstellen*-Dokumentation entnehmen:

- *Natural for VSAM*

VI Steuerung der Ausgabe von Daten

Dieser Teil beschreibt, wie Sie vorgehen müssen, wenn ein Natural-Programm mehrere Reports erzeugen soll. Außerdem behandelt es 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.

Folgende Themen werden behandelt:

36 Report-Spezifikation — (rep)-Notation

`(rep)` ist der Ausgabereport-Identifikator, für den ein Statement anwendbar ist.

Report-Spezifikationen benutzen

Wenn ein Natural-Programm mehrere Reports erzeugen soll, muss die Notation (*rep*) bei jedem Ausgabe-Statement angegeben werden (siehe *Betroffene Statements*, weiter unten), das zum Erzeugen von Ausgaben für einen Report außer dem ersten Report (Report 0) benutzt werden soll.

Es kann ein Wert von 0 − 31 angegeben werden.

Diese Notation gilt nur für Reports, die im Batch-Modus erzeugt wurden, für Reports unter Complete und IMS TM; oder wenn Sie das Produkt Natural Advanced Facilities unter CICS oder TSO verwenden.

Der Wert für `(rep)` kann auch ein logischer Name sein, der mittels des `DEFINE PRINTER`-Statements zugewiesen wurde, siehe ***Beispiel 2*** weiter unten.

Betroffene Statements

Die Notation `(rep)` kann mit den folgenden Ausgabe-Statements benutzt werden:

`AT END OF PAGE` | `AT TOP OF PAGE` | `COMPOSE` | `DISPLAY` | `EJECT` | `FORMAT` | `NEWPAGE` | `PRINT` | `SKIP` | `SUSPEND IDENTICAL SUPPRESS` | `WRITE` | `WRITE TITLE` | `WRITE TRAILER`

Beispiele für Report-Spezifikation

Beispiel 1 – Mehrere Reports

```
DISPLAY (1) NAME ...
WRITE (4) NAME ...
```

Beispiel 2 – Logische Namen benutzen

```
DEFINE PRINTER (LIST=5) OUTPUT 'LPT1'
WRITE (LIST) NAME ...
```

37 Layout einer Ausgabeseite

Dieses Kapitel gibt eine Übersicht über die Statements, die zur Definition eines spezifischen Layouts für einen Report benutzt werden können.

Statements mit Auswirkungen auf das Aussehen eines Report-Layouts

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. Standardmäßig sind Seiten-Kopfzeilen zentriert und nicht unterstrichen.
`WRITE TRAILER`	Mit diesem Statement können Sie eine Seiten-Fußzeile angeben, d.h. Text, der am Ende einer Seite ausgegeben werden soll. Standardmäßig sind Seiten-Fußzeilen zentriert und nicht unterstrichen.
`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. Siehe Abschnitt ***Statements DISPLAY und WRITE***.

Die Statements `AT START OF DATA` und `AT END OF DATA` sind im Kapitel *Datenbankzugriffe*, ***AT START/END OF DATA Statements***, beschrieben. Die anderen oben aufgeführten Statements sind in den folgenden Abschnitten des vorliegenden Dokuments beschrieben.

Allgemeines Layout-Beispiel

Das folgende Beispiel-Programm veranschaulicht die allgemeine Form einer Ausgabeseite:

```
** Example 'OUTPUX01': Several sections of output
************************************************************************
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
```

Ausgabe des Programms `OUTPUX01`:

```
                    ********** 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 =====
```

38 Statements DISPLAY und WRITE

Dieses Kapitel beschreibt, wie Sie die Statements DISPLAY und WRITE zur Ausgabe von Daten und zur Steuerung der Art und Weise benutzen, in der die Informationen ausgegeben werden.

Das 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 'DISPLX01': DISPLAY
************************************************************************
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
```

Ausgabe des Programms DISPLX01:

```
Page      1                                                  04-11-11  14:15:54

PERSONNEL         NAME                  CURRENT
   ID                                   POSITION
--------- -------------------- -------------------------

30020013  GARRET               TYPIST
30016112  TAILOR               WAREHOUSEMAN
20017600  PIETSCH              SECRETARY
```

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 'DISPLX02': DISPLAY
************************************************************************
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
```

Ausgabe des Programms DISPLX02:

```
Page      1                                              04-11-11  14:15:54

        NAME                  CURRENT           PERSONNEL
                             POSITION             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 im Abschnitt *Spaltenüberschriften* beschrieben.

Das 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 Werten eines multiplen Feldes 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 unter ***DISPLAY VERT-Statement*** im Abschnitt ***Vertikale Ausgabe durch Kombination von DISPLAY und WRITE*** beschrieben.

Beispiel für ein DISPLAY-Statement

```
** Example 'DISPLX03': DISPLAY
************************************************************************
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
```

Ausgabe des Programms `DISPLX03`:

```
Page      1                                                 04-11-11  14:15:54

        NAME              FIRST-NAME        ANNUAL
                                            SALARY
-------------------- -------------------- ----------

JONES                VIRGINIA                 46000
                                              42300
                                              39300
JONES                MARSHA                   50000
                                              46000
                                              42700
```

Beispiel für ein WRITE-Statement

```
** Example 'WRITEX01': WRITE
************************************************************************
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
```

Ausgabe des Programms `WRITEX01`:

```
Page      1                                          04-11-11  14:15:55

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) 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 Element-Ebene.

Mit der Notation *n*`X` können Sie die Anzahl der Leerstellen (*n*) zwischen zwei bestimmten Spalten angeben. Eine *n*`X`-Notation hat Vorrang vor einer `SF`-Parameterangabe.

Beispiel:

```
** Example 'DISPLX04': DISPLAY (with nX)
************************************************************************
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
```

Ausgabe des Programms `DISPLX04`:

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:

```
Page      1                                                 04-11-11  14:15:54

PERSONNEL           NAME                      CURRENT
   ID                                         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 *n*T, 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 'DISPLX05': DISPLAY (with nT)
************************************************************************
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
```

Ausgabe des Programms DISPLX05:

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                                                 04-11-11  14:15:54

            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 1 - Zeilenvorschub bei einem DISPLAY-Statement:

```
** Example 'DISPLX06': DISPLAY (with slash '/')
************************************************************************
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
```

Ausgabe des Programms `DISPLX06`:

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

```
Page      1                                              04-11-11  14:15:54

        NAME         DEPART-
     FIRST-NAME       MENT
-------------------- -------

JONES                SALE
VIRGINIA
JONES                MGMT
MARSHA
JONES                TECH
ROBERT
```

Beispiel 2 - Zeilenvorschub bei einem WRITE-Statement:

```
** Example 'WRITEX02': WRITE (with line advance)
************************************************************************
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
```

Ausgabe des Programms `WRITEX02`:

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                                                  04-11-11  14:15:55

JONES
VIRGINIA             DEPART-/MENT SALE


JONES
MARSHA               DEPART-/MENT MGMT


JONES
ROBERT               DEPART-/MENT TECH
```

Beispiel 3 - Zeilenvorschub in DISPLAY- und WRITE-Statements:

```
** Example 'DISPLX21': DISPLAY (usage of slash '/' in DISPLAY and WRITE)
************************************************************************
DEFINE DATA LOCAL
1 EMPLOY-VIEW VIEW OF EMPLOYEES
  2 CITY
  2 NAME
  2 FIRST-NAME
  2 ADDRESS-LINE (1)
END-DEFINE
*
WRITE TITLE LEFT JUSTIFIED UNDERLINED
       *TIME
   5X  'PEOPLE LIVING IN SALT LAKE CITY'
   21X 'PAGE:' *PAGE-NUMBER /
   15X 'AS OF' *DAT4E //
*
WRITE TRAILER UNDERLINED 'REGISTER OF' / 'SALT LAKE CITY'
*
READ (2) EMPLOY-VIEW WITH CITY = 'SALT LAKE CITY'
  DISPLAY  NAME /
           FIRST-NAME
```

```
           'HOME/CITY' CITY
           'STREET/OR BOX NO.' ADDRESS-LINE (1)
  SKIP 1
END-READ
END
```

Ausgabe des Programms `DISPLX21`:

```
14:15:54.6     PEOPLE LIVING IN SALT LAKE CITY                    PAGE:      1
               AS OF 11/11/2004


------------------------------------------------------------------------------
        NAME                 HOME                STREET
     FIRST-NAME              CITY              OR BOX NO.
-------------------- -------------------- --------------------

ANDERSON             SALT LAKE CITY       3701 S. GEORGE MASON
JENNY

SAMUELSON            SALT LAKE CITY       7610 W. 86TH STREET
MARTIN

                                  REGISTER OF
                                SALT LAKE CITY
------------------------------------------------------------------------------
```

Weitere Beispiele für DISPLAY- und WRITE-Statements

Siehe die folgenden Beispiel-Programme:

- *DISPLX13 - DISPLAY-Statement (zum Vergleich mit WRITEX08 mit WRITE)*
- *WRITEX08 - WRITE-Statement (zum Vergleich mit DISPLX13 mit DISPLAY)*
- *DISPLX14 - DISPLAY-Statement (mit AL, SF und nX)*
- *WRITEX09 - WRITE-Statement (in Kombination mit AT END OF DATA)*

39 Index-Notation für multiple Felder und Periodengruppen

Dieses Kapitel beschreibt, wie Sie die Index-Notation (*n*:*n*) benutzen können, um anzugeben, wieviele Werte eines multiplen Feldes oder wieviele Ausprägungen einer Periodengruppe ausgegeben werden sollen.

Index-Notation benutzen

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 im DISPLAY-Statement

```
** Example 'DISPLX07': DISPLAY (with index notation)
************************************************************************
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
```

Ausgabe des Programms DISPLX07:

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

```
Page      1                                              04-11-11  14:15:54

PERSONNEL         NAME                     INCOME
   ID
                                 CURRENCY   ANNUAL     BONUS
                                   CODE     SALARY
--------- -------------------- -------- ---------- ----------

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, d.h. einen Schrägstrich (/), einfügen, wird der Report übersichtlicher:

Beispiel für Index-Notation im WRITE-Statement

```
** Example 'WRITEX03': WRITE (with index notation)
************************************************************************
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
```

Ausgabe des Programms `WRITEX03`:

```
Page      1                                              04-11-11  14:15:55

30020013 GARRET
TYPIST                          4200       4150          0

30016112 TAILOR
WAREHOUSEMAN                    7450       7350       6700

20017600 PIETSCH
SECRETARY                      22000      20200      18700
```

40 Seitenüberschriften, Seitenvorschübe und Leerzeilen

Dieses Kapitel beschreibt verschiedene Möglichkeiten, wie Sie den Seitenumbruch in einem Report sowie die Ausgabe von Seitenüberschriften am Anfang jeder Seite des Reports und die Erzeugung von Leerzeilen in einem Ausgabe-Report beeinflussen können.

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.

Beispiel:

```
WRITE 'HELLO'
END
```

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

```
Page      1                                                  04-12-14  13:19:33

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.

Beispiel - DISPLAY mit NOTITLE:

```
** Example 'DISPLX20': DISPLAY (with NOTITLE)
************************************************************************
DEFINE DATA LOCAL
1 EMPLOY-VIEW VIEW OF EMPLOYEES
  2 CITY
  2 NAME
  2 FIRST-NAME
END-DEFINE
*
READ (5) EMPLOY-VIEW BY CITY FROM 'BOSTON'
  DISPLAY NOTITLE NAME FIRST-NAME CITY
END-READ
END
```

Ausgabe des Programms `DISPLX20`:

```
        NAME              FIRST-NAME              CITY
-------------------- -------------------- --------------------

SHAW                 LESLIE               BOSTON
STANWOOD             VERNON               BOSTON
CREMER               WALT                 BOSTON
PERREAULT            BRENDA               BOSTON
COHEN                JOHN                 BOSTON
```

Beispiel - WRITE mit NOTITLE:

```
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`.

In diesem Abschnitt werden folgende Themen behandelt:

- Text für Ihre Überschrift angeben
- Leerzeilen nach der Überschrift angeben
- Überschriften-Ausrichtung und/oder -Unterstreichung
- Überschrift mit Seitenzahl

Text für Ihre Überschrift angeben

Mit dem Statement `WRITE TITLE` geben Sie (in Apostrophen) den Text Ihrer Kopfzeile an.

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

Mit dem obigen Programm wird die folgende Ausgabe erzeugt:

```
                          THIS IS MY PAGE TITLE
HELLO
```

Leerzeilen nach der Überschrift angeben

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
```

Mit dem obigen Programm wird die folgende Ausgabe erzeugt:

```
                          THIS IS MY PAGE TITLE


HELLO
```

`SKIP` kann nicht nur als Option in einem `WRITE TITLE`-Statement, sondern auch als eigenständiges Statement verwendet werden (siehe *Leerzeilen erzeugen — das SKIP-Statement*).

Überschriften-Ausrichtung und/oder -Unterstreichung

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.

Option	Auswirkung
`LEFT JUSTIFIED`	Bewirkt, dass die Überschrift linksbündig angezeigt wird.
`UNDERLINED`	Bewirkt, dass die Überschrift unterstrichen angezeigt wird. Das Unterstreichen legt die Breite der Zeile fest (siehe auch Natural Profil- und Session-Parameter `LS`). Standardmäßig werden Überschriften mit einem Bindestrich (-) unterstrichen. Mit dem Session-Parameter `UC` können Sie jedoch ein anderes Zeichen angeben, das als Zeichen

Option	Auswirkung
	zum Unterstreichen benutzt werden soll (siehe *Unterstreichungszeichen für Überschriften — der UC-Parameter*).

Das folgende Beispiel zeigt die Auswirkungen der Optionen `LEFT JUSTIFIED` und `UNDERLINED`:

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

Mit dem obigen Programm wird die folgende Ausgabe erzeugt:

```
THIS IS MY PAGE TITLE
------------------------------------------------------------------------------


HELLO
```

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

Überschrift mit Seitenzahl

In den folgenden Beispielen wird die Systemvariable `*PAGE-NUMBER` in Verbindung mit dem Statement `WRITE TITLE` zur Ausgabe der Seitenzahl in der Überschriftenzeile benutzt.

```
** Example 'WTITLX01': WRITE TITLE (with *PAGE-NUMBER)
************************************************************************
DEFINE DATA LOCAL
1 VEHIC-VIEW VIEW OF VEHICLES
  2 MAKE
  2 YEAR
  2 MAINT-COST (1)
END-DEFINE
*
LIMIT 5
*
READ VEHIC-VIEW
END-ALL
SORT BY YEAR USING MAKE MAINT-COST (1)
  DISPLAY NOTITLE YEAR MAKE MAINT-COST (1)
  AT BREAK OF YEAR
    MOVE 1 TO *PAGE-NUMBER
    NEWPAGE
  END-BREAK
  /*
```

```
  WRITE TITLE LEFT JUSTIFIED
        'YEAR:' YEAR 15X 'PAGE' *PAGE-NUMBER
END-SORT
END
```

Ausgabe des Programms `WTITLX01`:

```
YEAR:  1980               PAGE      1
YEAR          MAKE          MAINT-COST
----- -------------------- ----------

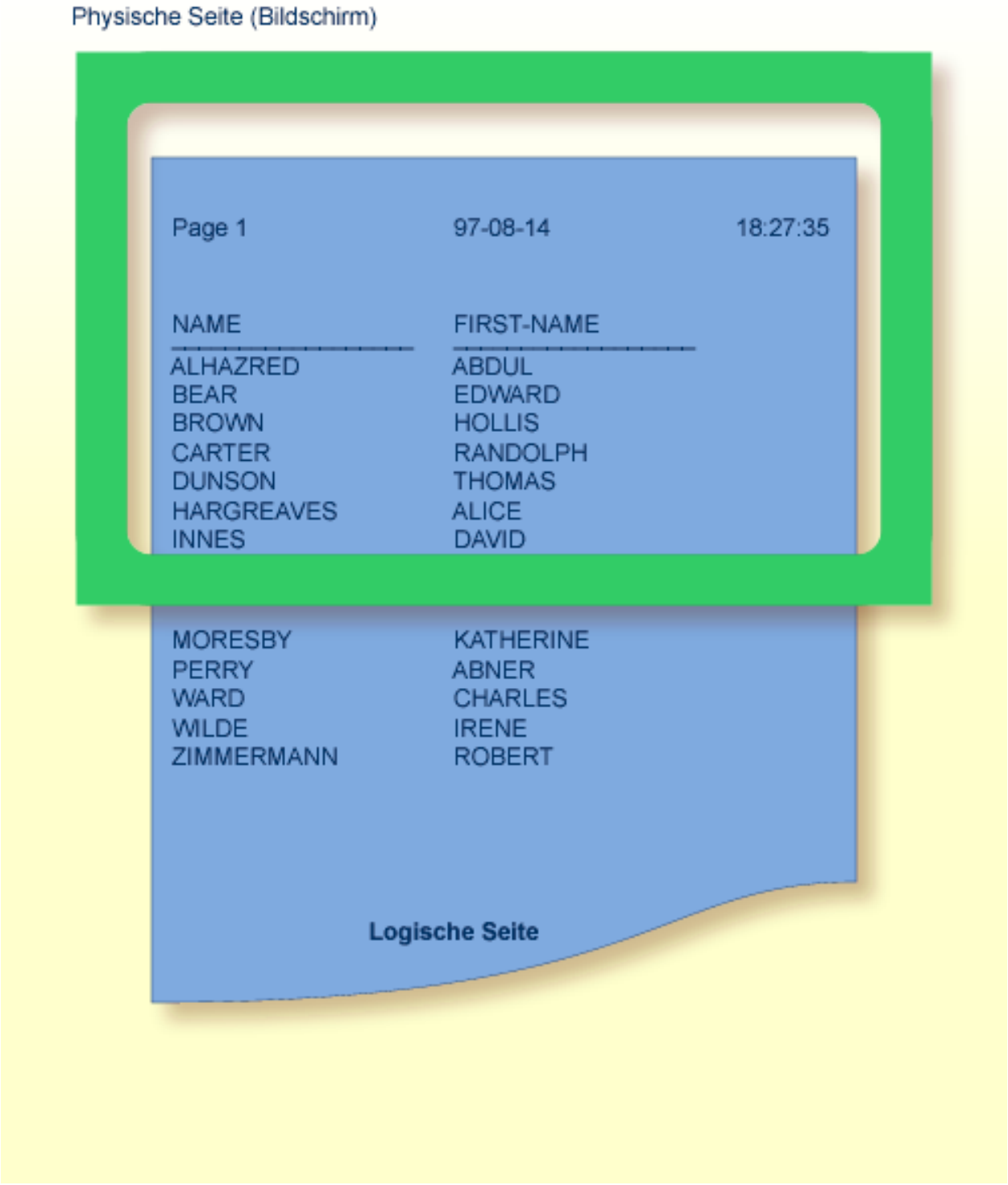
 1980 RENAULT                   20000
 1980 RENAULT                   20000
 1980 PEUGEOT                   20000
```

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.

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.



Anmerkung: 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, können Sie die logische Seitenlänge mit dem Session-Parameter `PS` (wie unten beschrieben) entsprechend verkleinern.

Seitenlänge — der PS-Parameter

Mit dem Session-Parameter PS können Sie die maximale Anzahl der Zeilen einer (logischen) Ausgabeseite bestimmen.

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:

```
GLOBALS PS=nn
```

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

Ein Seitenvorschub kann durch eine der folgenden Methoden erreicht werden:

- Seitenvorschub durch EJ-Parameter
- Seitenvorschub durch EJECT oder NEWPAGE-Statement
- EJECT/NEWPAGE bei weniger als n restlichen Zeilen auf der Seite

Diese Methoden werden im Folgenden erörtert.

Seitenvorschub durch 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 kann auf Session-Ebene mit dem Session-Parameter EJ unter Verwendung des Systemkommandos GLOBALS gesetzt werden:

```
GLOBALS EJ=OFF
```

Die Einstellung des Session-Parameters EJ wird durch das EJECT-Statement überschrieben.

Seitenvorschub durch EJECT oder NEWPAGE-Statement

Folgende Themen werden behandelt:

- Seitenvorschub ohne Überschrift/Fußzeile auf der nächsten Seite
- Seitenvorschub mit Verarbeitung am Ende/Anfang der Seite

Seitenvorschub ohne Überschrift/Fußzeile auf der nächsten Seite

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 TRAILER oder AT END OF PAGE, WRITE TITLE, AT TOP OF PAGE oder *PAGE-NUMBER werden nicht ausgeführt.

Das EJECT-Statement hat Vorrang vor dem EJ-Parameter.

Seitenvorschub mit Verarbeitung am Ende/Anfang der Seite

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 Session-Parameter PS definierten Seitenlänge (siehe ***Seitenlänge - der PS-Parameter*** oben).

EJECT/NEWPAGE bei weniger als n restlichen Zeilen auf der Seite

Das NEWPAGE- ebenso 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 1:

```
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 Session-Parameter PS, also nach 55 Zeilen.

Beispiel 2:

```
** Example 'NEWPAX02': NEWPAGE (in combination with EJECT and
**                     parameter PS)
************************************************************************
DEFINE DATA LOCAL
1 EMPLOY-VIEW VIEW OF EMPLOYEES
  2 CITY
  2 NAME
  2 JOB-TITLE
END-DEFINE
*
FORMAT PS=15
*
READ (9) EMPLOY-VIEW BY CITY STARTING FROM 'BOSTON'
  AT START OF DATA
    EJECT
    WRITE /// 20T '%' (29) /
              20T '%%'                          47T '%%' /
              20T '%%' 3X 'REPORT OF EMPLOYEES' 47T '%%' /
              20T '%%' 3X '  SORTED BY CITY   ' 47T '%%' /
              20T '%%'                          47T '%%' /
              20T '%' (29) /
    NEWPAGE
  END-START
  AT BREAK OF CITY
    NEWPAGE WHEN LESS 3 LINES LEFT
```

```
  END-BREAK
  DISPLAY CITY (IS=ON) NAME JOB-TITLE
END-READ
END
```

Neue Seite mit Titel

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 Session-Parameter PS und des Statements NEWPAGE. Außerdem wird hier die Natural-Systemvariable *PAGE-NUMBER verwendet, die jeweils die aktuelle Seitenzahl enthält.

```
** Example 'NEWPAX01': NEWPAGE
************************************************************************
DEFINE DATA LOCAL
1 VIEWEMP VIEW 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
```

Ausgabe des Programms NEWPAX01:

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

```
Page      1                                                  04-11-11  14:15:54

        NAME           DEPT        LOCATION
-------------------- ------ --------------------

FICKEN               TECH10 MADISON
KELLOGG              TECH10 MADISON
ALEXANDER            SALE20 MADISON
```

Seite 2:

```
EMPLOYEES BY CITY - PAGE:      2
        NAME           DEPT        LOCATION
-------------------- ------ --------------------

DE JUAN              SALE03 MADRID
DE LA MADRID         PROD01 MADRID
```

Seite 3:

```
EMPLOYEES BY CITY - PAGE: 3
```

Seiten-Fußzeile — das WRITE TRAILER-Statement

Folgende Themen werden behandelt:

- Seiten-Fußzeile angeben
- Logische Seitenlänge berücksichtigen
- Ausrichtung und/oder Unterstreichung der Seiten-Fußzeile

Seiten-Fußzeile angeben

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 einem NEWPAGE-Statement oder am Ende einer logischen Seite.

Logische Seitenlänge berücksichtigen

Die Prüfung, ob das Ende einer logischen Seite erreicht ist, erfolgt erst, *nachdem* ein WRITE-Statement oder ein 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.

Ausrichtung und/oder Unterstreichung der Seiten-Fußzeile

Standardmäßig wird die Seiten-Fußzeile zentriert auf der Seite und nicht unterstrichen ausgegeben.

Das WRITE TRAILER-Statement bietet Ihnen aber auch die Möglichkeit, eine Fußzeile linksbündig (LEFT JUSTIFIED) und/oder unterstrichen (UNDERLINED) auszugeben:

Option	Auswirkung
LEFT JUSTIFIED	Bewirkt, dass die Fußzeile linksbündig angezeigt wird.
UNDERLINED	Bewirkt, dass die Fußzeile unterstrichen angezeigt wird. Das Unterstreichen erfolgt über die ganze Breite der Zeile fest (siehe auch Natural-Profil- und Session-Parameter LS). Standardmäßig werden Überschriften mit einem Bindestrich (-) unterstrichen. Mit dem Session-Parameter UC können Sie aber auch ein anderes Zeichen angeben, das als Zeichen zum Unterstreichen benutzt werden soll (siehe *Unterstreichungszeichen für Überschriften*).

Die folgenden Beispiele zeigen die Verwendung der Optionen LEFT JUSTIFIED und UNDERLINED des WRITE TRAILER-Statements:

Beispiel 1:

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

Beispiel 2:

```
** Example 'WTITLX02': WRITE TITLE AND WRITE TRAILER
************************************************************************
DEFINE DATA LOCAL
1 EMPLOY-VIEW VIEW OF EMPLOYEES
  2 CITY
  2 NAME
  2 FIRST-NAME
  2 ADDRESS-LINE (1)
END-DEFINE
*
WRITE TITLE LEFT JUSTIFIED UNDERLINED
       *TIME
   5X  'PEOPLE LIVING IN SALT LAKE CITY'
   21X 'PAGE:' *PAGE-NUMBER /
   15X 'AS OF' *DAT4E //
*
WRITE TRAILER UNDERLINED 'REGISTER OF' / 'SALT LAKE CITY'
*
READ (2) EMPLOY-VIEW WITH CITY = 'SALT LAKE CITY'
  DISPLAY  NAME /
           FIRST-NAME
           'HOME/CITY' CITY
           'STREET/OR BOX NO.' ADDRESS-LINE (1)
  SKIP 1
END-READ
END
```

Leerzeilen erzeugen — das SKIP-Statement

Das `SKIP`-Statement wird zum Erzeugen von einer oder mehrerer Leerzeilen in einem Ausgabe-Report benutzt.

Beispiel 1 - SKIP in Verbindung mit WRITE und DISPLAY:

```
** Example 'SKIPX01': SKIP (in conjunction with WRITE and DISPLAY)
************************************************************************
DEFINE DATA LOCAL
1 EMPLOY-VIEW VIEW OF EMPLOYEES
  2 CITY
  2 NAME
  2 FIRST-NAME
  2 ADDRESS-LINE (1)
END-DEFINE
*
WRITE TITLE LEFT JUSTIFIED UNDERLINED
     'PEOPLE LIVING IN SALT LAKE CITY AS OF' *DAT4E 7X
     'PAGE:' *PAGE-NUMBER
SKIP 3
*
READ (2) EMPLOY-VIEW WITH CITY = 'SALT LAKE CITY'
  DISPLAY NAME / FIRST-NAME CITY ADDRESS-LINE (1)
  SKIP 1
END-READ
END
```

Beispiel 2 - SKIP in Verbindung mit DISPLAY VERT:

```
** Example 'SKIPX02': SKIP (in conjunction with DISPLAY VERT)
************************************************************************
DEFINE DATA LOCAL
1 EMPLOY-VIEW VIEW OF EMPLOYEES
  2 NAME
  2 FIRST-NAME
  2 CITY
  2 JOB-TITLE
END-DEFINE
*
READ (2) EMPLOY-VIEW WITH JOB-TITLE = 'SECRETARY'
  DISPLAY NOTITLE VERT
          NAME FIRST-NAME / CITY
  SKIP 3
END-READ
*
NEWPAGE
*
READ (2) EMPLOY-VIEW WITH JOB-TITLE = 'SECRETARY'
  DISPLAY NOTITLE
          NAME FIRST-NAME / CITY
  SKIP 3
END-READ
END
```

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.

Beispiel:

```
** Example 'ATTOPX01': AT TOP OF PAGE
************************************************************************
DEFINE DATA LOCAL
1 EMPLOY-VIEW VIEW OF EMPLOYEES
  2 PERSONNEL-ID
  2 NAME
  2 MAR-STAT
  2 BIRTH
  2 CITY
  2 JOB-TITLE
  2 DEPT
END-DEFINE
*
LIMIT 10
READ EMPLOY-VIEW BY PERSONNEL-ID FROM '20017000'
  DISPLAY NOTITLE (AL=10)
          NAME DEPT JOB-TITLE CITY 5X
          MAR-STAT 'DATE OF/BIRTH' BIRTH (EM=YY-MM-DD)
  /*
  AT TOP OF PAGE
    WRITE /   '-BUSINESS INFORMATION-'
          26X '-PRIVATE INFORMATION-'
  END-TOPPAGE
END-READ
END
```

Ausgabe des Programms `ATTOPX01`:

```
-BUSINESS INFORMATION-                          -PRIVATE INFORMATION-
   NAME    DEPARTMENT  CURRENT      CITY         MARITAL   DATE OF
              CODE     POSITION                   STATUS    BIRTH
---------- ---------- ---------- ----------     ---------- --------

CREMER     TECH10     ANALYST    GREENVILLE     S          70-01-01
MARKUSH    SALE00     TRAINEE    LOS ANGELE     D          79-03-14
GEE        TECH05     MANAGER    CHAPEL HIL     M          41-02-04
KUNEY      TECH10     DBA        DETROIT        S          40-02-13
NEEDHAM    TECH10     PROGRAMMER CHATTANOOG     S          55-08-05
JACKSON    TECH10     PROGRAMMER ST LOUIS       D          70-01-01
PIETSCH    MGMT10     SECRETARY  VISTA          M          40-01-09
PAUL       MGMT10     SECRETARY  NORFOLK        S          43-07-07
HERZOG     TECH05     MANAGER    CHATTANOOG     S          52-09-16
DEKKER     TECH10     DBA        MOBILE         W          40-03-03
```

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 Anmerkungen bezüglich logischer und physischer Seitenlängen, die für das `DISPLAY`- und `WRITE`-Statement gelten (vgl. **oben**), treffen auch auf das `AT END OF PAGE`-Statement zu.

Beispiel:

```
** Example 'ATENPX01': AT END OF PAGE (with system function available
**                     via GIVE SYSTEM FUNCTIONS in DISPLAY)
************************************************************************
DEFINE DATA LOCAL
1 EMPLOY-VIEW VIEW OF EMPLOYEES
  2 PERSONNEL-ID
  2 NAME
  2 JOB-TITLE
  2 SALARY (1)
END-DEFINE
*
READ (10) EMPLOY-VIEW BY PERSONNEL-ID = '20017000'
  DISPLAY NOTITLE GIVE SYSTEM FUNCTIONS
          NAME JOB-TITLE 'SALARY' SALARY(1)
```

```
  /*
  AT END OF PAGE
    WRITE / 24T 'AVERAGE SALARY: ...' AVER(SALARY(1))
  END-ENDPAGE
END-READ
END
```

Ausgabe des Programms `ATENPX01`:

```
        NAME                  CURRENT            SALARY
                             POSITION
-------------------- ------------------------- ----------

CREMER               ANALYST                        34000
MARKUSH              TRAINEE                        22000
GEE                  MANAGER                        39500
KUNEY                DBA                            40200
NEEDHAM              PROGRAMMER                     32500
JACKSON              PROGRAMMER                     33000
PIETSCH              SECRETARY                      22000
PAUL                 SECRETARY                      23000
HERZOG               MANAGER                        48500
DEKKER               DBA                            48000


AVERAGE SALARY: ...      34270
            ↩
```

Weiteres Beispiel

Siehe das folgende Beispielprogramm:

- ***DISPLX21 - DISPLAY-Statement (mit Schrägstrich '/' und zum Vergleich mit WRITE)*** im Kapitel *Referenzierte Beispielprogramme*.

41 Spaltenüberschriften

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

Standard-Spaltenüberschriften

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

```
** Example 'DISPLX01': DISPLAY
************************************************************************
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
```

Ausgabe des Programms DISPLX01:

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

```
Page      1                                                 04-11-11  14:15:54

PERSONNEL         NAME                  CURRENT
   ID                                   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, zum Beispiel:

```
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 einen Text in Apostrophen (') an, wobei `'text'` die für das Feld zu verwendende Spaltenüberschrift ist.

```
** Example 'DISPLX08': DISPLAY  (with column title in 'text')
************************************************************************
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
```

Ausgabe des Programms `DISPLX08`:

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                                                  04-11-11  14:15:54

PERSONNEL       EMPLOYEE               POSITION
   ID
--------- -------------------- -------------------------

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 Session-Parameter `HC` (Überschriften-Zentrierung) können Sie die Ausrichtung der Spaltenüberschriften beeinflussen:

`HC=L`	Spaltenüberschriften werden linksbündig ausgerichtet.
`HC=R`	Spaltenüberschriften werden rechtsbündig ausgerichtet.
`HC=C`	Spaltenüberschriften werden zentriert. Dies ist die Standardeinstellung.

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 Elementebene.

Beispiel für die Angabe des HC-Parameters auf Statement-Ebene, d.h. für die linksbündige Ausrichtung *aller* Spaltenüberschriften:

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

Breite von Spaltenüberschriften — der HW-Parameter

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

HW=ON	Die Breite einer DISPLAY-Spalte wird entweder durch die Länge des Feldes oder durch die Länge der Spaltenüberschrift bestimmt, je nachdem was länger ist. Dies ist die Standardeinstellung.
HW=OFF	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.

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 Elementebene.

Füllzeichen für Überschriften — die Parameter FC und GC

Mit dem Session-Parameter FC 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 und Beschreibung **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 Session-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 Elementebene.

```
** Example 'FORMAX01': FORMAT (with parameters FC, GC)
************************************************************************
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
```

Ausgabe des Programms `FORMAX01`:

```
Page      1                                                 04-11-11  14:15:54

========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 Session-Parameter `UC` (Unterstreichungszeichen) 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 Elementebene.

```
** Example 'FORMAX02': FORMAT (with parameter UC)
************************************************************************
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.

Ausgabe des Programms `FORMAX02`:

```
EMPLOYEES REPORT
================================================================================

PERSONNEL         NAME                 CURRENT
   ID                                  POSITION
********* ==================== =========================

30020013  GARRET               TYPIST
30016112  TAILOR               WAREHOUSEMAN
20017600  PIETSCH              SECRETARY
```

Spaltenüberschriften unterdrücken — die Schrägstrich-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 Sie die Ausgabe von Standard-Spaltenüberschriften für *alle* Spalten unterdrücken können, können Sie mit der `'/'`-Notation die Überschrift für *eine einzelne* Spalte unterdrücken.

Dazu geben Sie die Notation '/' in einem DISPLAY-Statement jeweils unmittelbar vor dem Namen des Feldes an, dessen Spaltenüberschrift unterdrückt werden soll.

Zwei Beispiele zum Vergleich:

Beispiel 1:

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

In diesem Beispiel werden die Standardüberschriften aller drei Spalten ausgegeben:

```
Page      1                                                  04-11-11  14:15:54

        NAME         PERSONNEL          CURRENT
                          ID             POSITION
-------------------- --------- -------------------------

ABELLAN              60008339  MAQUINISTA
ACHIESON             30000231  DATA BASE ADMINISTRATOR
ADAM                 50005800  CHEF DE SERVICE
ADKINSON             20008800  PROGRAMMER
ADKINSON             20009800  DBA
ADKINSON             20011000  SALES PERSON
```

Beispiel 2:

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

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

```
Page      1                                                  04-11-11  14:15:54

                     PERSONNEL          CURRENT
                        ID             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 die folgenden Beispiel-Programme im Kapitel *Referenzierte Beispielprogramme*:

- ***DISPLX15 – DISPLAY-Statement (mit FC, UC)***
- ***DISPLX16 – DISPLAY-Statement (mit '/', 'text', 'text/text')***

42 Parameter zur Beeinflussung der Ausgabe von Feldern

Dieses Kapitel erörtert die Benutzung der Natural Profil- und/oder Session-Parameter, die Sie zum Steuern des Ausgabe-Formats von Feldern verwenden können.

Eine Übersicht der Natural-Profilparameter, die während der Erstellung der Natural-Reports benutzte Standard-Attribute steuern, finden Sie im Abschnitt *Ausgabe-Reports und Arbeitsdateien* in der *Operations*-Dokumentation.

Übersicht über Feldausgabe-relevante Parameter

Natural bietet eine Reihe von Profil- und/oder Session-Parametern, mit denen Sie die Art, in der Felder ausgegeben werden, beeinflussen können:

Parameter	Funktion
LC, IC und TC	Mit diesen Session-Parametern können Sie Zeichen angeben, die vor bzw. nach einem Feld bzw. vor einem Feldwert angezeigt werden sollen.
AL und NL	Mit diesen Session-Parametern können Sie die Ausgabelänge von Feldern vergrößern oder verkleinern.
DL	Mit diesem Session-Parameter können Sie die Standard-Ausgabelänge für ein alphanumerisches Maskenfeld des Formats U angeben.
SG	Mit diesem Session-Parameter können Sie bestimmen, ob negative Werte mit oder ohne Minuszeichen angezeigt werden sollen.
IS	Mit diesem Session-Parameter können Sie die Anzeige von Feldwerten, die mit dem jeweils vorigen Feldwert identisch sind, unterdrücken.
ZP	Mit diesem Profil- und Session-Parameter können Sie bestimmen, ob Feldwerte, die `0` sind, angezeigt werden sollen oder nicht.
ES	Mit diesem Session-Parameter können Sie die Anzeige von Leerzeilen, die von einem `DISPLAY`- oder `WRITE`-Statement erzeugt werden, unterdrücken.

Diese Parameter werden im Folgenden behandelt.

Vorangestellte Zeichen — der LC-Parameter

Mit dem Session-Parameter `LC` (Vorangestellte Zeichen) 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 Session-Parameter `AD` (Attribut-Definition) können Sie diese Ausrichtung ändern. 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 folgenden Statements verwendet werden:

- FORMAT
- DISPLAY

Er kann dort sowohl auf Statement- als auch auf Elementebene gesetzt werden.

Vorangestellte Zeichen im Unicode-Format — der LCU Parameter

Der Session-Parameter LCU (Vorangestellte Unicode-Zeichen) ist identisch mit dem Session-Parameter LC. Der Unterschied ist, dass alle vorangestellten Zeichen immer im Unicode-Format gespeichert werden.

Das gibt Ihnen die Möglichkeit, vorangestellte Zeichen mit Zeichen von verschiedenen Codepages gemischt anzugeben, und stellt sicher, dass unabhängig von der installierten Codepage immer das richtige Zeichen angezeigt wird.

Weitere Informationen siehe *Unicode and Code Page Support in the Natural Programming Language*, *Session Parameters*, Abschnitt *EMU, ICU, LCU, TCU im Vergleich zu EM, IC, LC, TC*.

Die Parameter LCU und ICU dürfen nicht beide gleichzeitig für ein Feld angegeben werden.

Einfügungszeichen — der IC-Parameter

Mit dem Session-Parameter IC (Einfügungszeichen) 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 folgenden Statements verwendet werden:

- FORMAT
- DISPLAY

Der IC-Parameter kann dort sowohl auf Statement- als auch auf Elementebene gesetzt werden.

Einfügungszeichen im Unicode-Format — der ICU Parameter

Der Session-Parameter `ICU` ist identisch mit dem Session-Parameter `IC`. Der Unterschied ist, dass im Fall von `ICU` alle vorangestellten Zeichen immer im Unicode-Format gespeichert werden.

Das gibt Ihnen die Möglichkeit, Einfügungszeichen mit Zeichen von verschiedenen Codepages gemischt anzugeben, und stellt sicher, dass unabhängig von der installierten Codepage immer das richtige Zeichen angezeigt wird.

Weitere Informationen siehe *Unicode- und Codepage-Unterstützung in der Natural-Programmiersprache*, *Session-Parameter*, Abschnitt *EMU, ICU, LCU, TCU versus EM, IC, LC, TC*.

Die Parameter `LCU` und `ICU` dürfen nicht beide gleichzeitig für ein Feld angegeben werden.

Nachgestellte Zeichen — der TC-Parameter

Mit dem Session-Parameter `TC` (Nachgezogene Zeichen) 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 folgenden Statements verwendet werden:

- `FORMAT`
- `DISPLAY`

Der `TC`-Parameter kann dort sowohl auf Statement- als auch auf Elementebene gesetzt werden.

Vorangestellte Zeichen im Unicode-Format — der TCU Parameter

Der Session-Parameter `TCU` ist identisch mit dem Session-Parameter `TC`. Der Unterschied ist, dass im Fall von `TCU` alle vorangestellten Zeichen immer im Unicode-Format gespeichert werden.

Das gibt Ihnen die Möglichkeit, Einfügungszeichen mit Zeichen von verschiedenen Codepages gemischt anzugeben, und stellt sicher, dass unabhängig von der installierten Codepage immer das richtige Zeichen angezeigt wird.

Weitere Informationen siehe *Unicode- und Codepage-Unterstützung in der Natural-Programmiersprache*, *Session-Parameter*, Abschnitt *EMU, ICU, LCU, TCU im Vergleich zu EM, IC, LC, TC*.

Ausgabelänge — der AL- und der NL-Parameter

Mit dem Session-Parameter AL bestimmen Sie die *Ausgabelänge eines alphanumerischen Feldes*. Mit dem Session-Parameter NL 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 folgenden Statements verwendet werden:

- FORMAT
- DISPLAY
- WRITE
- PRINT
- INPUT

Sie können dort sowohl auf Statement- als auch auf Elementebene gesetzt werden.

Anmerkung: Wenn eine Editiermaske angegeben ist, hat diese Vorrang vor einer AL- bzw. NL-Angabe. Editiermasken werden im Abschnitt ***Editiermasken - der EM-Parameter*** beschrieben.

Ausgabelänge — der DL Parameter

Anmerkung: Um die Möglichkeiten, die der DL-Parameter bietet, voll nutzen zu können, müssen Sie das Web I/O Interface benutzen. Wenn Sie eine Terminal-Emulation verwenden, ist es nicht möglich, in der Anzeige in einem Feld zu „blättern“, wenn der mit dem DL-Parameter angegebene Wert kleiner als die Länge des Feldinhaltes ist.

Mit dem Session-Parameter DL bestimmen Sie die *Ausgabelänge eines Felds des Formats A oder U*, weil die Ausgabelänge einer Unicode-Zeichenkette doppelt so lang sein kann wie die Zeichenkette und der Benutzer die Möglichkeit haben muss, die ganze Zeichenkette anzuzeigen. Der Standardwert ist zum Beispiel die Länge für ein Format/Länge U10, die Anzeigelänge kann 10 bis 20 sein, während die Standardlänge (ohne DL-Angabe) 10 ist.

Der DL-Parameter kann mit den folgenden Statements verwendet werden:

- FORMAT
- DISPLAY
- WRITE

- PRINT
- INPUT

Der DL-Parameter kann sowohl auf Statement- als auch auf Elementebene gesetzt werden.

Der Unterschied zwischen den Session-Parametern AL und DL besteht darin, dass der AL-Parameter die Datenlänge eines Feldes bestimmt, während der DL-Parameter die Anzahl der Spalten bestimmt, die zur Anzeige des Feldes auf dem Schirm benutzt werden. Der Benutzer kann in Einabefeldern „blättern“, um den ganzen Inhalt eines Feldes zu sehen, wenn der mit dem DL-Parameter angegebene Wert kleiner als die Länge der Felddaten ist.

Die Benutzung des DL-Parameters mit einer Länge, die kürzer als die Feldlänge ist, wird nur empfohlen, wenn Sie mit dem Web I/O Interface arbeiten. Wenn Sie Natural in einer Terminal-Emulation ausführen, ist das Blättern in einem Feld nicht möglich; Sie haben denselben Effekt wie mit dem AL-Parameter. Wenn Sie den Feldinhalt ändern, gehen außerdem alle Zeichen verloren, die sich hinter der Anzeigelänge befinden.

Anmerkung: Der DL-Parameter ist auch bei Feldern des Formats A zulässig. In Verbindung mit dem Web I/O Interface kann die Edit-Control-Size kleiner als der Inhalt eines Feldes gemacht werden.

Beispiel:

```
DEFINE DATA LOCAL
1    #U1 (U10)
1    #U2 (U10)
END-DEFINE
*
#U1 := U'latintxt00'
#U2 := U'特別是伺服器都需要支'
*
INPUT (AD=M) #U1 #U2
END
```

Das obige Programm liefert die folgende Ausgabe, in der der Inhalt des Feldes #U2 unvollständig ist:

```
#U1 latintxt00 #U2 特別是伺服
```

Wird bei dem Feld #U2 der Session-Parameter DL verwendet, dann wird der Inhalt dieses Feldes korrekt angezeigt.

```
DEFINE DATA LOCAL
1    #U1 (U10)
1    #U2 (U10)
END-DEFINE
*
#U1 := U'latintxt00'
#U2 := U'特別是伺服器都需要支'
*
INPUT (AD=M) #U1 #U2 (DL=20)
END
```

Ergebnis:

```
#U1 latintxt00 #U2 特別是伺服器都需要支
```

Vorzeichen-Stelle — der SG-Parameter

Mit dem Session-Parameter SG (Vorzeichen-Stelle) 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 folgenden Statements verwendet werden:

- FORMAT
- DISPLAY
- PRINT
- WRITE
- INPUT

Er kann dort sowohl auf Statement- als auch auf Elementebene gesetzt werden.

Anmerkung: Wenn eine Editiermaske angegeben ist, hat diese Vorrang vor einer SG-Angabe. **Editiermasken** sind im Abschnitt *Editiermasken - der EM-Parameter* beschrieben.

Beispielprogramm ohne Parameter

```
** Example 'FORMAX03': FORMAT (without FORMAT and compare with 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
*
READ (5) VIEWEMP BY NAME STARTING FROM 'JONES'
  DISPLAY NAME
          FIRST-NAME
          SALARY (1:1)
          BONUS  (1:1,1:1)
END-READ
END
```

Ausgabe des Programms `FORMAX03`:

Das obige Programm enthält keine Parameterangaben und erzeugt folgende Ausgabe:

```
  Page      1                                              04-11-11  11:11:11

          NAME              FIRST-NAME        ANNUAL     BONUS
                                              SALARY
  -------------------- -------------------- ---------- ----------

  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

In diesem Beispiel werden die Session-Parameter `AL`, `NL`, `LC`, `IC` und `TC` benutzt.

```
** Example 'FORMAX04': FORMAT (with parameters AL, NL, LC, TC, IC and
**                      compare with 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
*
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:

```
  Page      1                                                 04-11-11  11:11:11

     NAME      FIRST-NAME   ANNUAL   BONUS
                            SALARY
  ----------- ----------- -------- --------

  *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- und 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 Session-Parameter `IS` (Unterdrückung identischer Werte) 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.

Der`IS`-Parameter kann angegeben werden:

- in einem `FORMAT`-Statement; er gilt dann für den gesamten Report.
- in einem `DISPLAY`- oder `WRITE`-Statement, und zwar sowohl auf Statement- als auch auf Elementebene.

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

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 'FORMAX05': FORMAT (without parameter IS
**                     and compare with FORMAX06)
************************************************************************
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
```

Ausgabe des Programms `FORMAX05`:

```
 Page      1                                                  04-11-11  11:11:11

         NAME              FIRST-NAME
 ------------------- --------------------

 JONES                VIRGINIA
 JONES                MARSHA
 JONES                ROBERT
```

Beispielprogramm mit IS-Parameter

```
** Example 'FORMAX06': FORMAT (with parameter IS
**                      and compare with FORMAX05)
************************************************************************
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
```

Ausgabe des Programms `FORMAX06`:

```
Page      1                                                04-11-11  11:54:01

        NAME              FIRST-NAME
-------------------- --------------------

JONES                VIRGINIA
                     MARSHA
                     ROBERT
```

Nullwerte anzeigen — der ZP-Parameter

Mit dem Profil- und Session-Parameter ZP (Anzeige von Nullwerten) 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.

Der ZP-Parameter kann angegeben werden:

- in einem FORMAT-Statement; er gilt dann für den gesamten Report.
- in einem DISPLAY- oder WRITE-Statement, und zwar sowohl auf Statement- als auch auf Elementebene.

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

Leerzeilen unterdrücken — der ES-Parameter

Mit dem Parameter ES (Leerzeilenunterdrückung) 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.

Der ES-Parameter kann angegeben werden:

- in einem FORMAT-Statement; er gilt dann für den gesamten Report.
- in einem DISPLAY- oder WRITE-Statement, 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 'FORMAX07': FORMAT (without parameter ES and ZP
**                     and compare with FORMAX08)
************************************************************************
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
```

Ausgabe des Programms `FORMAX07`:

```
Page      1                                                 04-11-11  11:58:23

        NAME              FIRST-NAME        BONUS
-------------------- -------------------- ----------

JONES                VIRGINIA                  9000
                                               6750
JONES                MARSHA                       0
                                                  0
JONES                ROBERT                       0
                                                  0
JONES                LILLY                        0
                                                  0
```

Beispielprogramm mit den Parametern ZP und ES

```
** Example 'FORMAX08': FORMAT (with parameters ES and ZP
**                     and compare with FORMAX07)
************************************************************************
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
```

Ausgabe des Programms `FORMAX08`:

```
Page      1                                                  04-11-11  11:59:09

        NAME              FIRST-NAME        BONUS
-------------------- -------------------- ----------

JONES                VIRGINIA                  9000
                                               6750
JONES                MARSHA
JONES                ROBERT
JONES                LILLY
```

Weitere Beispiele für Feldausgabe-relevante Parameter

Weitere Beispiele für die Parameter `LC`, `IC`, `TC`, `AL`, `NL`, `IS`, `ZP` und `ES` und das `SUSPEND IDENTICAL SUPPRESS`-Statement finden Sie in den folgenden Beispielprogrammen im Kapitel *Referenzierte Beispielprogramme*.

- ***DISPLX17 - DISPLAY-Statement (mit NL, AL, IC, LC, TC)***
- ***DISPLX18 – DISPLAY-Statement (Benutzung von Voreinstellungen für SF, AL, UC, LC, IC, TC und zum Vergleich mit DISPLX19)***
- ***DISPLX19 - DISPLAY-Statement (mit SF, AL, LC, IC, TC und zum Vergleich mit DISPLX18)***
- ***SUSPEX01 - SUSPEND IDENTICAL SUPPRESS-Statement (in Verbindung mit den Parametern IS, ES, ZP bei DISPLAY)***
- ***SUSPEX02 - SUSPEND IDENTICAL SUPPRESS-Statement (in Verbindung mit den Parametern IS, ES, ZP in DISPLAY). Identisch mit SUSPEX01, aber mit IS=OFF.***
- ***COMPRX03 - COMPRES-Statement (in Verbindung mit LC und TC)***

43 Codepage-Editiermasken — der EM-Parameter

Dieses Kapitel beschreibt, wie Sie eine Editiermaske für ein alphanumerisches oder numerisches Feld angeben können.

Verwendung des EM-Parameters

Der Session-Parameter EM wird verwendet, um 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. Wenn Sie den Session-Parameter EMU verwenden, können Sie Unicode-Zeichen auf die gleiche Weise benutzen wie für den Session-Parameter EM beschrieben.

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 Circumflex (^) für eine Leerstelle. Bei Anzeige mittels DISPLAY-Statement würde der Name JOHNSON in diesem Fall wie folgt ausgegeben:

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

Sie können den Session-Parameter EM an folgenden Stellen angeben:

- auf Report-Ebene (in einem FORMAT-Statement),
- auf Statement-Ebene (in einem DISPLAY-, WRITE-, INPUT- oder PRINT-Statement)
- oder auf Elementebene, d.h. Feldebene, (in einem DISPLAY-, WRITE-, INPUT-, COMPRESS-, DOWNLOAD PC FILE, MOVE EDITED- oder WRITE WORK FILE-Statement).

Eine mit dem EM-Parameter definierte Editiermaske hat Vorrang vor einer im **DDM** für das betreffende Feld definierten Standard-Editiermaske. Siehe auch *Spalten für Feld-Attribute*, *Erweiterte Feld-Attribute angeben*.

Falls EM=OFF gesetzt worden ist, wird überhaupt keine Editiermaske verwendet.

Eine auf Statement-Ebene definierte Editiermaske hat Vorrang vor einer auf Report-Ebene definierten Editiermaske.

Eine auf Elementebene definierte Editiermaske hat Vorrang vor einer auf Statement-Ebene definierten Editiermaske.

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 `Z` wird benutzt, um anzuzeigen, dass die Ausgabe-Position nur ausgefüllt wird, wenn die verfügbare Zahl nicht Null 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.

Weitere Informationen siehe Session-Parameter `EM`, *Editiermasken für numerische Felder* in der *Parameter-Referenz*-Dokumentation.

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 einem Circumflex (^) gekennzeichnet.

Weitere Informationen siehe Session-Parameter `EM`, *Editiermasken für alphanumerische Felder* in der *Parameter-Referenz*-Dokumentation.

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:

Editiermaske	Ausgabe
EM=X.X.X.X.X	J.O.H.N.S
EM=****XXXXXX****	****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.

Trennzeichen-Angaben an lokale Standards anpassen

Natural-Programme werden in der ganzen Welt in Geschäftsanwendungen eingesetzt. Je nach den lokalen Gegebenheiten ist es üblich, numerische Datenfelder und Felder mit einer Datums- oder Zeitangabe bei der Anzeige in Eingabe/Ausgabe-Statements in einem ganz bestimmten Format auszugeben. Die unterschiedliche Erscheinungsform sollte nicht durch einen anderen Programmcode realisiert werden, der selektiv als eine Funktion des Bereichs verarbeitet wird, in dem das Programm ausgeführt wird, sondern sollte mit demselben Programmtyp in Verbindung mit einer Reihe von Laufzeit-Parametern ausgeführt werden, um das Dezimalpunkt-Zeichen und das „Tausender-Trennzeichen“ anzugeben.

Folgende Themen werden behandelt:

- Dezimaltrennzeichen
- Dynamisches Tausendertrennzeichen

- Beispiele

Dezimaltrennzeichen

Der Natural-Parameter `DC` (Dezimaltrennzeichen) steht zur Verfügung, um das Zeichen anzugeben, das anstelle von Zeichen eingefügt wird, die zur Darstellung des Dezimal-Trennzeichens (auch als „Basiszeichen“ bezeichnet) in Editiermasken benutzt werden. Dieser Parameter ermöglicht es den Benutzern eines Natural-Programms oder einer Natural-Anwendung, beliebige Zeichen oder Sonderzeichen zu wählen, um die Ganzzahl-Stellen von den Dezimalstellen eines numerischen Datenelements zu trennen, und ermöglicht es zum Beispiel US-Unternehmen, den Dezimalpunkt (.) zu verwenden, und europäischen Unternehmen, das Komma (,) zu benutzen.

Dynamisches Tausendertrennzeichen

Um die Ausgabe von großen Ganzzahl-Werten zu strukturieren, ist es üblich, Trennzeichen zwischen jeder dritten Ziffer einer Ganzzahl einzufügen, um Tausender voneinander zu trennen. Dieses Trennzeichen wird als Tausendertrennzeichen bezeichnet. Beispielsweise kann in den Vereinigten Staaten und Großbritannien ein Komma für diesem Zweck benutzt (`1,000,000.00`) werden, wohingegen in Deutschland und Österreich das Leerzeichen (`1'000'000,00`) oder der Punkt (`1.000.000,00`) und in der Schweiz und Liechtenstein das Hochkomma (`1'000'000,00`) verwendet werden kann.

In einer Natural-Editiermaske ist ein dynamisches Tausendertrennzeichen ein Komma (oder Punkt), welches die Position anzeigt, an der (mit dem Parameter `THSEPCH` definierte) Tausendertrennzeichen zur Laufzeit eingefügt werden. Zur Kompilierungszeit aktiviert oder deaktiviert die Option `THSEP` des Systemkommandos `COMPOPT` oder der Subparameter `THSEP` des Profileparameters `CMPO` bzw. des Macros `NTCCMPO` die Interpretation des Kommas (oder Punktes) als ein dynamisches Tausendertrennzeichen.

Wenn `THSEP` auf `OFF` (Voreinstellung) gesetzt ist, wird jedes in der Editiermaske als Tausendertrennzeichen benutzte Zeichen als Literal behandelt und zur Laufzeit unverändert angezeigt. Diese Einstellung gewährleistet die Abwärtskompatibilität.

Wenn `THSEP` auf `ON` gesetzt ist, wird ein Komma (oder Punkt) in der Editiermaske als dynamisches Tausendertrennzeichen interpretiert. Im Allgemeinen ist das dynamische Tausendertrennzeichen ein Komma, aber wenn das Komma bereits als Dezimalzeichen (`DC`) vergeben ist, wird der Punkt als dynamisches Trennzeichen verwendet.

Zur Laufzeit werden die dynamischen Tausendertrennzeichen durch den aktuellen Wert des Parameters `THSEPCH` (Tausendertrennzeichen) ersetzt.

Beispiele

Ein Natural-Programm, das mit den Parameter-Einstellungen `DC='.'` und `THSEP=ON` katalogisiert ist, benutzt die Editiermaske `(EM=ZZ,ZZZ,ZZ9.99)`.

Parameter-Einstellungen zur Laufzeit	Wird angezeigt als
`DC='.'` und `THSEPCH=','`	`1,234,567.89`
`DC=','` und `THSEPCH='.'`	`1.234.567,89`
`DC=','` und `THSEPCH='/'`	`1/234/567,89`
`DC=','` und `THSEPCH=' '`	`1 234 567,89`
`DC=','` und `THSEPCH=''''`	`1'234'567,89`

Beispiele für Editiermasken

Im folgenden sehen Sie einige Beispiele für Editiermasken und die Ausgaben, die sie erzeugen.

Zusätzlich ist die jeweilige Kurzschreibweise angegeben. Sie können die 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^XXXXX`	`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` in der *Parameter-Referenz*-Dokumentation.

Beispielprogramm ohne EM-Parameter

```
** Example 'EDITMX01': Edit mask (using default edit masks)
************************************************************************
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
```

Ausgabe des Programms `EDITMX01`:

Es erzeugt die folgende Ausgabe unter Verwendung von Standard-Editiermasken (soweit vorhanden):

```
Page      1                                                 04-11-11  14:15:54

         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 'EDITMX02': Edit mask (using EM)
************************************************************************
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) /
                       FIRST-NAME   (EM=...X(10)...)
         'OCCUPATION' JOB-TITLE    (EM=' ___ 'X(12))
         'SALARY'     SALARY (1:3) (EM=' USD 'ZZZ,999)
  SKIP 1
END-READ
END
```

Ausgabe des Programms `EDITMX02`:

Vergleichen Sie sie mit der des vorigen Programms (*Beispielprogramm ohne EM-Parameter*), um zu sehen, wie sich die `EM`-Angaben auf die Anzeige der Felder auswirken:

```
Page      1                                                 04-11-11  14:15:54

          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 die folgenden Beispiel-Programme im Kapitel *Referenzierte Beispielprogramme*.

- ***EDITMX03 - Editiermaske (unterschiedliche EM-Angabe bei alphanumerischen Feldern)***
- ***EDITMX04 - Editiermaske (unterschiedliche EM-Angaben bei numerischen Feldern)***
- ***EDITMX05 - Editiermaske (EM-Angaben für Datums- und Uhrzeit-Systemvariablen)***

44 Unicode-Editiermasken — EMU-Parameter

Unicode-Editiermasken können auf die gleiche Weise wie Codepage-Editiermasken verwendet werden. Der Unterschied besteht darin, dass die Editiermaske immer im Unicode-Format gespeichert wird.

Das gibt Ihnen die Möglichkeit, Editiermasken mit Zeichen von verschiedenen Codepages gemischt anzugeben, und stellt sicher, dass unabhängig von der installierten Codepage immer das richtige Zeichen angezeigt wird.

Für die Benutzung der Unicode-Editiermasken gilt dasselbe wie im Abschnitt ***Codepage-Editiermasken — der EM-Parameter*** beschrieben.

Informationen zum Session-Parameter `EMU`, finden Sie im Abschnitt *EMU - Unicode-Editiermaske* (in der *Parameter-Referenz*).

45 Vertikale Ausgabe von Feldwerten

Dieses Kapitel beschreibt, wie Sie die Funktionen der Statements DISPLAY und WRITE miteinander kombinieren können, um vertikale Ausgaben von Feldwerten zu erzeugen.

Vertikale Ausgaben erzeugen

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

- mit einer Kombination von DISPLAY- und WRITE-Statement,
- mit der VERT-Klausel im DISPLAY-Statement.

Vertikale Ausgabe durch Kombination von DISPLAY und WRITE

Wie weiter oben 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 'WRITEX04': WRITE (in combination with DISPLAY)
************************************************************************
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 22T 'DEPT:' DEPT
  SKIP 1
```

```
END-READ
END
```

Ausgabe des Programms WRITEX04:

```
Page      1                                                  04-11-11  14:15:55

        NAME                  CURRENT
                             POSITION
-------------------- ------------------------

KOLENCE              MANAGER
                     DEPT: TECH05

GOSDEN               ANALYST
                     DEPT: TECH10

WALLACE              SALES PERSON
                     DEPT: SALE20
```

Tabulator-Notation — T*field

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

Durch die Notation T**field* können Sie die WRITE-Ausgabe nach der Position eines im vorangegangenen DISPLAY-Statement ausgegebenen Feldes ausrichten (wobei *field* der Name des Feldes ist, nach dem die Ausrichtung erfolgen soll).

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 'WRITEX05': WRITE  (in combination with DISPLAY)
************************************************************************
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
```

Ausgabe des Programms `WRITEX05`:

```
Page      1                                                  04-11-11  14:15:55

        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.

Die Positionierungsnotation bewirkt, dass das nächste Element in dem `DISPLAY`- oder `WRITE`-Statement *x* Zeilen unterhalb der letzten Ausgabe platziert wird, beginnend in Spalte *y* der Ausgabe.

Das folgende Programm veranschaulicht die Verwendung dieser Notation:

```
** Example 'WRITEX06': WRITE  (with n/n)
************************************************************************
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
```

Ausgabe des Programms `WRITEX06`:

```
Page      1                                                 04-11-11  14:15:55

  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. Siehe **Beispiel 1**.

- `AS CAPTIONED` bewirkt die Ausgabe der Standard-Spaltenüberschriften. Siehe **Beispiel 2**.
- `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. Siehe **Beispiel 3**.
- `AS 'text' CAPTIONED` bewirkt, dass Text als Spaltenüberschrift ausgegeben wird und außerdem die Standard-Spaltenüberschrift in jeder Ausgabezeile dem jeweiligen Feldwert direkt vorangestellt wird. Siehe **Beispiel 4**.

Beispiel 1 - DISPLAY VERT ohne AS-Klausel

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

```
** Example 'DISPLX09': DISPLAY  (without column title)
************************************************************************
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
```

Ausgabe des Programms `DISPLX09`:

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

```
Page      1                                                 04-11-11  14:15:54


RUBIN
SYLVIA

NEW YORK


WALLACE
MARY

NEW YORK
```

```
KELLOGG
HENRIETTA

NEWARK
```

Beispiel 2 - 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 'DISPLX10': DISPLAY (with VERT as CAPTIONED and HORIZ clause)
************************************************************************
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
```

Ausgabe des Programms `DISPLX10`:

```
Page      1                                                 04-11-11  14:15:54

        NAME                  CURRENT            ANNUAL
     FIRST-NAME              POSITION            SALARY
-------------------- ------------------------- ----------

RUBIN                SECRETARY                      17000
SYLVIA

WALLACE              ANALYST                        38000
MARY

KELLOGG              DIRECTOR                       52000
HENRIETTA
```

Beispiel 3 - DISPLAY VERT AS 'text'

Das folgende Programm enthält eine `AS 'text'`-Klausel, die bewirkt, dass der angegebene `'text'` als Spaltenüberschrift ausgegeben wird.

Anmerkung: Ein Schrägstrich (/) in dem Textelement in einem `DISPLAY`-Statement bewirkt einen Zeilenumbruch.

```
** Example 'DISPLX11': DISPLAY  (with VERT AS 'text' clause)
************************************************************************
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
```

Ausgabe des Programms `DISPLX11`:

```
Page      1                                                 04-11-11  14:15:54

     EMPLOYEES                CURRENT            ANNUAL
                             POSITION           SALARY
-------------------- ------------------------ ----------

RUBIN                SECRETARY                     17000
SYLVIA

WALLACE              ANALYST                       38000
MARY

KELLOGG              DIRECTOR                      52000
HENRIETTA
```

Beispiel 4 - DISPLAY VERT AS 'text' CAPTIONED

Die Klausel `AS 'text' CAPTIONED` bewirkt, dass der angegebene Text als Spaltenüberschrift angezeigt wird und dass die Standard-Spaltenüberschriften direkt vor dem Feldwert in jeder ausgegebenen Zeile angezeigt werden:

Das folgende Programm enthält eine `AS 'text' CAPTIONED`-Klausel.

```
** Example 'DISPLX12': DISPLAY (with VERT AS 'text' CAPTIONED clause)
************************************************************************
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
```

Ausgabe des Programms `DISPLX12`:

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

```
Page      1                                              04-11-11  14:15:54

           EMPLOYEES                    CURRENT           ANNUAL
                                       POSITION           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*field

Bei einer Kombination von DISPLAY VERT-Statement mit nachfolgendem WRITE-Statement können Sie mit der Notation P*_field-name_ 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 'WRITEX07': WRITE  (with P*field)
************************************************************************
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
```

Ausgabe des Programms WRITEX07:

```
Page      1                                                  04-11-11  14:15:55

        NAME                  CURRENT            INCOME
                             POSITION
-------------------- ------------------------ ----------

SMITH                                                  0
                                                       0
                                                ***SALARY PLUS BONUS***
                                                (IN US DOLLARS)

POORE JR             SECRETARY                     25000
```

```
                                                  0
                                              ***SALARY PLUS BONUS***
                                              (IN US DOLLARS)

PREPARATA            MANAGER                       46000
                                                    9000
                                              ***SALARY PLUS BONUS***
                                              (IN US DOLLARS)
```

Weiteres Beispiel für DISPLAY VERT- mit WRITE-Statement

Siehe das folgende Beispiel-Programm im Kapitel *Referenzierte Beispielprogramme*:

- ***WRITEX10 - WRITE-Statement (mit nT, T*field und P*field)***

VII Weitere Programmieraspekte

In diesem Teil werden folgende Themen behandelt:

46 Text-Notation

In den Statements INPUT, DISPLAY, WRITE, WRITE TITLE oder WRITE TRAILER können Sie Text-Notation benutzen, um einen in Verbindung mit einem solchen Statement zu benutzenden Text zu definieren.

Mit einem Statement zu benutzenden Text definieren — die 'text'-Notation

Der mit dem Statement zu benutzende Text (z.B. eine Aufforderungsmeldung für den Benutzer) muss entweder in Apostrophen (') oder in Anführungszeichen (") stehen.

Vorsicht: Verwechseln Sie doppelte Apostrophe ('') nicht mit dem Anführungszeichen (").

In Anführungszeichen stehender Text kann automatisch von Klein- in Großbuchstaben konvertiert werden. Sie können die automatische Konvertierung ausschalten, indem Sie die Einstellungen im Editor-Profil ändern. Einzelheiten siehe Option **Dynamic Conversion of Lower Case** in *Allgemeine Standardwerte - General Defaults* in der *Editoren*-Dokumentation.

Der '*text*' darf 1 bis 72 Zeichen lang sein und darf nicht über das Ende einer Quellcode-Zeile hinausgehen.

Textelemente können mittels eines Bindestriches (-) verkettet werden.

Beispiele:

```
DEFINE DATA LOCAL
1 #A(A10)
END-DEFINE

INPUT 'Input XYZ' (CD=BL) #A
WRITE '=' #A
WRITE 'Write1 ' - 'Write2 ' - 'Write3' (CD=RE)
END
```

Apostrophe als Teil eines Textelements benutzen

Folgende gilts, wenn der Natural-Profilparameter TQ (Translate Quotation Marks = Anführungszeichen konvertieren) bzw. der Schlüsselwort-Subparameter TQMARK (Umsetzung von Anführungszeichen) des Natural-Profilparameters CMPO auf ON gesetzt ist. Dies ist die Standardstellung.

Für ein Apostroph, das Teil eines in Apostrophen stehenden *text*-Elements ist, können Sie entweder doppelte Apostrophe ('') oder ein einzelnes Anführungszeichen (") schreiben, beides wird dann bei der Ausgabe in ein einzelnes Apostroph umgesetzt.

Für ein Apostroph ('), das Teil eines in Anführungszeichen (") stehenden *text*-Elements ist, schreiben sie ein einzelnes Apostroph.

Beispiele für Apostrophe:

```
#FIELDA = 'O''CONNOR'
#FIELDA = 'O"CONNOR'
#FIELDA = "O'CONNOR"
```

In allen drei Fällen erhalten Sie folgende Ausgabe:

```
O'CONNOR
```

Anführungszeichen als Teil eines Textelements benutzen

Es gilt Folgendes, wenn der Natural-Profilparameter `TQ` (Translate Quotation Marks) oder der Schlüsselwort-Subparameter `TQMARK` des Natural-Profilparameters `CMPO` auf `OFF` gesetzt ist. Die Standardeinstellung ist `ON`.

Für ein Anführungszeichen, das Teil eines in einzelnen Apostrophen stehenden *`text`*-Elements ist, schreiben Sie *ein* Anführungszeichen.

Für ein Anführungszeichen, das Teil eines in Anführungszeichen stehenden *`text`*-Elements ist, schreiben sie *doppelte* Anführungszeichen (`""`).

Beispiel für Anführungszeichen:

```
#FIELDA = 'O"CONNOR'
#FIELDA = "O""CONNOR"
```

In beiden Fällen erhalten Sie folgende Ausgabe:

```
O"CONNOR
```

Vor einem Feldwert n-mal anzuzeigendes Zeichen definieren — die 'c'(n)-Notation

Soll als Text ein einzelnes Zeichen mehrmals wiederholt werden, verwenden Sie folgende Notation:

```
'c'(n)
```

c steht hierbei für das auszugebende Zeichen, und mit *n* geben Sie an, wie oft das Zeichen generiert werden soll. *n* darf maximal 249 betragen.

Beispiel:

```
WRITE '*'(3)
```

Statt der Apostrophe (') vor und nach dem Zeichen *c* können Sie auch Anführungszeichen (") verwenden.

47 Benutzerkommentare

Benutzerkommentare sind zu den Statements des Quellcodes hinzugefügte oder in ihnen verteilte Beschreibungen oder erläuternde Anmerkungen. Solche Informationen können insbesondere dann hilfreich sein, wenn es um das Verstehen und die Pflege von Quellcode geht, der von einem anderen Programmierer geschrieben oder editiert wurde.

Des Weiteren können die den Anfang eines Kommentars markierenden Zeichen benutzt werden, um die Funktion eines Statements oder mehrere Quellcode-Zeilen zu Test-Zwecken zeitweilig auszuschalten.

Sie haben in Natural verschiedene Möglichkeiten, im Quellcode Kommentare einzufügen.

Ganze Quellcode-Zeile als Kommentarzeile benutzen

Falls Sie eine ganze Quellcode-Zeile als Kommentarzeile verwenden möchten, geben Sie am Anfang der Zeile Folgendes ein:

- einen Stern und ein Leerzeichen (`*`),
- zwei Sterne (`**`) oder
- einen Schrägstrich und einen Stern (`/*`).

```
*  USER COMMENT
** USER COMMENT
/* USER COMMENT
```

Beispiel:

Wie dem folgenden Beispiel zu entnehmen ist, können Kommentarzeilen auch benutzt werden, um den Quellcode klar zu strukturieren.

```
** Example 'LOGICX03': BREAK option in logical condition
************************************************************************
DEFINE DATA LOCAL
1 EMPLOY-VIEW VIEW OF EMPLOYEES
  2 NAME
  2 FIRST-NAME
  2 BIRTH
*
1 #BIRTH (A8)
END-DEFINE
*
LIMIT 10
READ EMPLOY-VIEW BY BIRTH
  MOVE EDITED BIRTH (EM=YYYYMMDD) TO #BIRTH
  /*
```

```
  IF BREAK OF #BIRTH /6/
    NEWPAGE IF LESS THAN 5 LINES LEFT
    WRITE / '-' (50) /
  END-IF
  /*
  DISPLAY NOTITLE BIRTH (EM=YYYY-MM-DD) NAME FIRST-NAME
END-READ
END
```

Teil einer Quellcode-Zeile als Kommentarzeile benutzen

Falls Sie nur einen Teil einer Quellcode-Zeile für einen Kommentar verwenden möchten, geben Sie ein Leerzeichen, einen Schrägstrich und einen Stern (/*) ein. Der Rest der Zeile ab dieser Markierung ist damit als Kommentar gekennzeichnet:

```
ADD 5 TO #A         /* USER COMMENT
```

Beispiel:

```
** Example 'LOGICX04': IS option as format/length check
************************************************************************
DEFINE DATA LOCAL
1 #FIELDA (A10)          /* INPUT FIELD TO BE CHECKED
1 #FIELDB (N5)           /* RECEIVING FIELD OF VAL FUNCTION
1 #DATE   (A10)          /* INPUT FIELD FOR DATE
END-DEFINE
*
INPUT #DATE #FIELDA
IF #DATE IS(D)
  IF #FIELDA IS (N5)
    COMPUTE #FIELDB = VAL(#FIELDA)
    WRITE NOTITLE 'VAL FUNCTION OK' //  '=' #FIELDA '=' #FIELDB
  ELSE
    REINPUT 'FIELD DOES NOT FIT INTO N5 FORMAT'
            MARK *#FIELDA
  END-IF
ELSE
  REINPUT 'INPUT IS NOT IN DATE FORMAT (YY-MM-DD) '
          MARK *#DATE
END-IF
*
END
```

48 Datenberechnungen

Dieses Kapitel behandelt die zur Berechnung von Daten verwendeten arithmetischen Statements:

- COMPUTE
- ADD
- SUBTRACT
- MULTIPLY
- DIVIDE

Außerdem werden die folgenden Statements behandelt, die zur Übertragung des Wertes eines Operanden in eines oder mehrere Felder benutzt werden:

- MOVE
- COMPRESS

Wichtig: 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 das Feld `LEAVE-DUE` gestellt.

Beispiel 2:

```
COMPUTE #A = SQRT (#B)
```

Die Quadratwurzel des Feldwertes von `#B` wird errechnet und dem Feld `#A` zugewiesen. `SQRT` (Square Root) ist eine von mehreren in Natural eingebauten mathematischen Funktionen, die mit den folgenden Statements verwendet werden können:

- `COMPUTE`
- `ADD`
- `SUBTRACT`
- `MULTIPLY`
- `DIVIDE`

Einen Überblick über die in Natural eingebauten mathematischen Funktionen finden Sie im Abschnitt **Mathematische Funktionen**.

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 **Konstante** (z.B. Text oder eine Zahl), eine **Benutzervariable**, ein Datenbankfeld, 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 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 4 Statements haben eine `ROUNDED`-Option, mit der Sie gerundete Werte erhalten können.

Informationen zu Rundungsregeln entnehmen Sie dem Abschnitt ***Regeln für arithmetische Operationen***. Weitere Informationen zu diesen Statements finden Sie in der *Statements*-Dokumentation.

Beispiel für MOVE-, SUBTRACT- und COMPUTE-Statements

Das folgende Programm veranschaulicht die Verwendung von **Benutzervariablen** in arithmetischen Statements. Es berechnet Alter und Einkommen von drei Mitarbeitern und gibt die Ergebnisse aus.

```
** Example 'COMPUX01': COMPUTE
************************************************************************
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
```

Ausgabe des Programms `COMPUX01`:

```
Page      1                                              04-11-11  14:15:54

        NAME                 POSITION          #AGE  #INCOME
-------------------- ------------------------- ---- ----------

JONES                MANAGER                     63      55000
JONES                DIRECTOR                    58      50000
JONES                PROGRAMMER                  48      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 in der *Statements*-Dokumentation unter der `COMPRESS`-Statement-Option `LEAVING NO SPACE` 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 finden Sie in der *Statements*-Dokumentation.

Beispiel für die Statements COMPRESS und MOVE

Das folgende Beispiel veranschaulicht die Benutzung der Statements MOVE und COMPRESS.

```
** Example 'COMPRX01': COMPRESS
************************************************************************
DEFINE DATA LOCAL
1 MYVIEW VIEW 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
```

Ausgabe des Programms COMPRX01:

Beachten Sie vor allem die Ausgabe der mittels COMPRESS-Statement zusammengefassten Felder.

```
Page      1                                              04-11-11  14:15:54

          #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.

Beispiele für COMPRESS-Statement

Beispiel 1:

Im folgenden Programm werden drei **Benutzervariablen** benutzt: #FULL-SALARY, #FULL-NAME und #FULL-CITY. In #FULL-SALARY beispielsweise sind der Text 'SALARY:' sowie die Datenbankfelder SALARY und CURR-CODE zusammengefasst. Das WRITE-Statement referenziert lediglich die komprimierten Variablen.

```
** Example 'COMPRX02': COMPRESS
************************************************************************
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 #FULL-SALARY   (A25)
1 #FULL-NAME     (A25)
1 #FULL-CITY     (A25)
END-DEFINE
*
READ (3) VIEWEMP BY CITY STARTING FROM 'NEW YORK'
  COMPRESS 'SALARY:' CURR-CODE(1) SALARY(1) INTO #FULL-SALARY
  COMPRESS FIRST-NAME NAME                  INTO #FULL-NAME
  COMPRESS ZIP CITY                         INTO #FULL-CITY
  /*
  DISPLAY 'NAME AND ADDRESS' NAME (EM=X^X^X^X^X^X^X^X^X^X^X^X^X)
  WRITE 1/5  #FULL-NAME
        1/37 #FULL-SALARY
```

```
       2/5  ADDRESS-LINE (1)
       3/5  #FULL-CITY
  SKIP 1
END-READ
END
```

Ausgabe des Programms COMPRX02:

```
Page      1                                                 04-11-11  14:15:54

   NAME AND ADDRESS
-----------------------

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
```

Beispiel 2:

Das folgende Programm ist wie das Programm COMPRX02, jedoch werden drei Felder, NAME, SALARY und COUNTRY im COMPRESS-Statement mit Editiermasken behandelt.

```
** Example 'COMPRX04': COMPRESS with Edit Mask (EM)
** see similar example COMPRX02 with DISPLAY statement etc.
************************************************************************
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
  2 COUNTRY
*
1 #FULL-SALARY   (A25)
1 #FULL-NAME     (A25)
1 #FULL-CITY     (A25)
```

```
1 #COUNTRY       (A10)
1 #NAME          (A25)
END-DEFINE
*
READ (3) VIEWEMP BY CITY STARTING FROM 'NEW YORK'
  COMPRESS NAME (EM=X^X^X^X^X^X^X^X^X^X^X^X^X^X) INTO #NAME
  COMPRESS FIRST-NAME NAME                        INTO #FULL-NAME
  COMPRESS 'SALARY:' CURR-CODE(1)
           SALARY(1) (EM=ZZZ,ZZ9)                 INTO #FULL-SALARY
  COMPRESS ZIP CITY                               INTO #FULL-CITY
  COMPRESS COUNTRY (EM=X'-'X'-'X)                 INTO #COUNTRY
  /*
  DISPLAY 'NAME AND ADDRESS' #NAME
  WRITE 1/5  #FULL-NAME
        1/37 #FULL-SALARY
        2/5  ADDRESS-LINE (1)
        3/5  #FULL-CITY
        4/5  #COUNTRY
  SKIP 1
END-READ
END
```

Ausgabe des Programms `COMPRX04`:

```
Page      1                                                  21-12-20  17:25:24

    NAME AND ADDRESS
-------------------------

R U B I N
    SYLVIA RUBIN                    SALARY: USD  17,000
    2003 SARAZEN PLACE
    10036 NEW YORK
    U-S-A

W A L L A C E
    MARY WALLACE                    SALARY: USD  38,000
    12248 LAUREL GLADE C
    10036 NEW YORK
    U-S-A

K E L L O G G
    HENRIETTA KELLOGG               SALARY: USD  52,000
    1001 JEFF RYAN DR.
    19711 NEWARK
    U-S-A
```

Mathematische Funktionen

Bei der Verarbeitung arithmetischer Statements (`ADD`, `COMPUTE`, `DIVIDE`, `SUBTRACT`, `MULTIPLY`) unterstützt Natural die folgenden mathematischen Funktionen:

Mathematische Funktion	Natural-Systemfunktion
Absoluter Wert eines Feldes.	ABS(*field*)
Arcustangens eines Feldes.	ATN(*field*)
Kosinus eines Feldes.	COS(*field*)
Potenz eines Feldes (Exponential).	EXP(*field*)
Bruchteil (hinter dem Komma) eines Feldes.	FRAC(*field*)
Ganzzahliger Teil eines Feldes.	INT(*field*)
Natürlicher Logarithmus eines Feldes.	LOG(*field*)
Vorzeichen eines Feldes.	SGN(*field*)
Sinus eines Feldes.	SIN(*field*)
Quadratwurzel eines Feldes (Square Root).	SQRT(*field*)
Tangens eines Feldes.	TAN(*field*)
Numerischer Wert eines alphanumerischen Feldes.	VAL(*field*)

Weitere Einzelheiten zu mathematischen Funktionen finden Sie in der *Systemfunktionen*-Dokumentation.

Weitere Beispiele für COMPUTE-, MOVE- und COMPRESS-Statements

Siehe folgende Beispielprogramme:

- ***WRITEX11 – WRITE-Statement (mit nX, n/n und COMPRESS)***
- ***IFX03 - IF-Statement***
- ***COMPRX03 - COMPRESS-Statement (mit Parametern LC and TC)***

49 Regeln für arithmetische Operationen

Initialisierung von Feldern

Ein Feld (Datenbankfeld oder Benutzervariable), das in einer arithmetischen Operation als Operand verwendet werden soll, muss mit einem der folgenden Formate definiert werden:

Format	
N	Numerisch ungepackt
P	Numerisch gepackt
I	Integer (Ganzzahl)
F	Floating Point (Gleitkomma)
D	Datum
T	Zeit

Anmerkung: Reporting Mode: Ein Feld, das in einer arithmetischen Operation als Operand verwendet werden soll, muss vorher definiert werden. Benutzervariablen oder Datenbankfelder, die in einer arithmetischen Operation als Ergebnisfeld verwendet werden, müssen nicht vorher definiert werden.

Sobald ein Programm zur Ausführung aufgerufen wird, werden alle im `DEFINE DATA`-Bereich definierten Benutzervariablen und Datenbankfelder mit den entsprechenden Leer- bzw. Nullwerten initialisiert.

Kompatibilitätsregeln zur Datenübertragung

Die Datenübertragung erfolgt mit einem `MOVE`- oder `COMPUTE`-Statement. Die folgende Tabelle fasst die Kompatibilitätsregeln zur Datenübertragung der Formate zusammen, die ein Operand annehmen kann.

Ausgangsfeld	Zielfeld													
	N oder P	A	U	B*n* (*n*<5)	B*n* (*n*>4)	I	L	C	D	T	F	G	O
N oder P	Y	[2]	[14]	[3]	-	Y	-	-	-	Y	Y	-	-
A	-	Y	[13]	[1]	[1]	-	-	-	-	-	-	-	-
U	-	[11]	Y	[12]	[12]	-	-	-	-	-	-	-	-
B*n* (*n*<5)	[4]	[2]	[14]	[5]	[5]	Y	-	-	-	Y	Y	-	-
B*n* (*n*>4)	-	[6]	[15]	[5]	[5]	-	-	-	-	-	-	-	-
I	Y	[2]	[14]	[3]	-	Y	-	-	-	Y	Y	-	-

L	-	[9]	[16]	-	-	-	Y	-	-	-	-	-	-		
C	-	-	-	-	-	-	-	Y	-	-	-	-	-		
D	Y	[9]	[16]	Y	-	Y	-	-	Y	[7]	Y	-	-		
T	Y	[9]	[16]	Y	-	Y	-	-	[8]	Y	Y	-	-		
F	Y	[9]	[10]	[10]	[16]	[3]	-	Y	-	-	-	Y	Y	-	-
G	-	-	-	-	-	-	-	-	-	-	-	-	Y	-	
O	-	-	-	-	-	-	-	-	-	-	-	-	-	Y	

Legende:

Y	Übertragung möglich.
-	Übertragung nicht möglich.
[ ]	Übertragung möglich. Die Ziffern in eckigen Klammern [ ] beziehen sich auf die entsprechende Regel für die Übertragung (siehe unten).

Umsetzung von Daten in ein anderes Format

Bei der Umsetzung von Werten in ein anderes Format gelten folgende Regeln:

1. **Von Alphanumerisch (A) in Binär (B):**
   Der Wert wird Byte für Byte von links nach rechts übertragen. Je nach Länge des Zielfeldes und der Anzahl der Bytes wird der Wert entweder abgeschnitten oder der Rest des Feldes mit Leerzeichen aufgefüllt.
2. **Von Numerisch (N), Gepackt (P), Ganzzahl (I) und Binär (B) mit 1-4 Bytes Länge in Alphanumerisch:**
   Der Wert wird in ungepacktes Format umgesetzt und linksbündig in das Zielfeld übertragen, wobei vorangestellte Nullen weggelassen werden und der Rest des Feldes mit Leerzeichen aufgefüllt wird. Bei negativen numerischen Werten wird das Vorzeichen in die Hexadezimalnotation `Dx` umgesetzt. Ein Komma (Dezimalpunkt) im Ausgangswert wird nicht berücksichtigt, und alle Stellen vor und nach dem Komma werden als ganze Zahl interpretiert.
3. **Von Numerisch (N), Gepackt (P), Ganzzahl (I), Gleitkomma (F) in Binär (B) mit 1-4 Bytes Länge:**
   Der Wert wird in binäres Format umgesetzt (4 Bytes). Ein Komma (Dezimalpunkt) wird ignoriert, die Stellen vor und nach dem Komma werden als ganze Zahl behandelt. Je nach Vorzeichen ist die Binärzahl entweder positiv oder das Zweierkomplement des Wertes.
4. **Von Binär (B) mit 1-4 Bytes Länge in Numerisch (N):**
   Der Wert wird umgesetzt und rechtsbündig übertragen, der Rest des Feldes wird mit Nullen aufgefüllt. Binäre Werte von 1 bis 3 Bytes Länge werden immer als positiv interpretiert. Bei 4 Byte langen binären Werten bestimmt das erste (linke) Bit das Vorzeichen: 1 = negativ, 0 = positiv. Ein Komma (Dezimalpunkt) im Zielfeld wird nicht berücksichtigt, und alle Stellen vor und nach dem Komma werden als ganze Zahl interpretiert.

5. **Von Binär (B) in Binär (B):**
   Der Wert wird Byte für Byte von rechts nach links übertragen, und der Rest des Feldes wird mit Nullen aufgefüllt.
6. **Von Binär (B) mit mehr als 4 Bytes in Alphanumerisch (A):**
   Der Wert wird Byte für Byte von links nach rechts übertragen. Je nach Länge des Zielfeldes und der Anzahl der Bytes wird der Wert entweder abgeschnitten oder der Rest des Feldes mit Leerzeichen aufgefüllt.
7. **Von Datum (D) in Zeit (T):**
   Das Datum wird in Zeit umgesetzt, und zwar ausgehend von der Zeit `00:00:00:0`.
8. **Von Zeit (T) in Datum (D):**
   Die Zeitkomponente wird abgeschnitten und nur die Datumskomponente des Zeitfeldes wird in das Datumsfeld übertragen.
9. **Von Logisch (L), Datum (D), Zeit (T), Gleitkomma (F) in Alphanumerisch (A):**
   Der Wert wird in Anzeigeform umgesetzt und linksbündig übertragen.
10. **Gleitkomma (F):**
    Wird ein Gleitkomma-Wert in ein alphanumerisches oder Unicode-Feld übertragen, das zu kurz ist, wird die Mantisse entsprechend gekürzt.
11. **Von Unicode (U) in Alphanumerisch (A):**
    Der Unicode-Wert wird anhand der Library ICU (International Components for Unicode) entsprechend der voreingestellten (Default-)Codepage (Wert der Systemvariablen `*CODEPAGE`) in alphanumerische Zeichencodes umgesetzt. Je nach Länge des Zielfeldes und der Anzahl der Bytes wird das Ergebnis entweder abgeschnitten oder der Rest des Feldes mit Leerzeichen aufgefüllt. Wenn die Zeichen des Unicode-Wertes in der voreingestellten (Default-)Codepage nicht definiert sind, wird ein Laufzeitfehler ausgegeben, oder an die Stelle der Zeichen tritt je nach der Einstellung des Profil/Session-Parameters `CPCVERR` das Ersetzungszeichen.
12. **Von Unicode (U) in Binär (B):**
    Der Wert wird Code Unit für Code Unit von links nach rechts verschoben. Je nach Länge des Zielfeldes und der Anzahl der Bytes wird das Ergebnis entweder abgeschnitten oder der Rest des Feldes mit Leerzeichen aufgefüllt. Die Länge des binären Zielfeldes muss geradezahlig sein.
13. **Von Alphanumerisch (A) in Unicode (U):**
    Der alphanumerische Wert wird unter Benutzung der Library ICU (International Components for Unicode) von der voreingestellten (Default-)Codepage in einen Unicode-Wert umgesetzt. Je nach Länge des Zielfeldes und der Anzahl der Code Units wird das Ergebnis entweder abgeschnitten oder der Rest des Feldes mit Leerzeichen aufgefüllt.
14. **Von N, P, I und Binär (Länge 1–4) in Unicode (U):**
    Der Wert wird in ungepacktes Format konvertiert, aus dem dann ein alphanumerischer Wert durch die Unterdrückung von führenden Nullen erhalten werden kann. Bei negativen numerischen Werten wird das Vorzeichen in die hexadezimale Notation `Dx` umgesetzt. Ein Dezimalpunkt im numerischen Wert wird ignoriert. Alle Ziffern vor und nach dem Dezimalpunkt werden als ein Ganzzahlwert (Integer) behandelt. Der Ergebniswert wird von alphanumerisch

in Unicode umgesetzt. Je nach Länge des Zielfeldes und der Anzahl der Code Units wird das Ergebnis entweder abgeschnitten oder der Rest des Feldes mit Leerzeichen aufgefüllt.

15. **Von Binär (B) mit mehr als 4 Bytes in Unicode (U):**
    Der Wert wird Byte für Byte von links nach rechts verschoben. Je nach Länge des Zielfeldes und der Anzahl der Bytes wird das Ergebnis entweder abgeschnitten oder der Rest des Feldes mit Leerzeichen aufgefüllt. Die Länge des binären Ausgangsfeldes muss geradzahlig sein.
16. **Von L, D, T, F in U:**
    Die Werte werden in ein alphanumerisches Anzeigeformat konvertiert. Der Ergebniswert wird von alphanumerisch in Unicode umgesetzt und linksbündig ausgerichtet.

Wenn das Ausgangs- und Zielformat identisch sind, kann je nach der Länge und der Anzahl der Bytes (Format A und B) oder Code-Einheiten (Format U) das Ergebnis abgeschnitten oder mit Leerzeichen (Format A und U) oder führenden binären Nullen (Format B) aufgefüllt werden.

Siehe auch *Dynamische Variablen benutzen*.

Abschneiden und Runden von Feldwerten

Die folgenden Regeln gelten für das Abschneiden und Runden von Feldwerten:

- Numerische Felder: vorangestellte Stellen dürfen nur abgeschnitten werden, falls ihr Wert Null ist. Stellen nach einem ausdrücklich angegebenen oder implizierten Komma (Dezimalpunkt) dürfen abgeschnitten werden.
- Alphanumerische Felder: Nachfolgende Stellen dürfen abgeschnitten werden.
- Bei Verwendung der Option `ROUNDED` wird die letzte Stelle im Feld aufgerundet, falls die erste abgeschnittene Stelle größer/gleich 5 ist. Zur Ergebnisgenauigkeit einer Division siehe auch Abschnitt *Genauigkeit von Ergebnissen bei arithmetischen Operationen*.

Format/Länge von Ergebnisfeldern bei arithmetischen Operationen

Die folgende Tabelle zeigt Format/Länge von Ergebnisfeldern bei arithmetischen Operationen:

	I1	I2	I4	N oder P	F4	F8
I1	I1	I2	I4	P*	F4	F8
I2	I2	I2	I4	P*	F4	F8
I4	I4	I4	I4	P*	F4	F8
N oder P	P*	P*	P*	P*	F4	F8
F4	F4	F4	F4	F4	F4	F8
F8	F8	F8	F8	F8	F8	F8

Auf Großrechnern wird Format/Länge F8 anstatt F4 für eine verbesserte Ergebnisgenauigkeit einer arithmetischen Operation benutzt.

P* ergibt sich aus der ganzzahligen Länge und Genauigkeit der einzelnen Operanden je nach Operation (siehe Abschnitt *Genauigkeit von Ergebnissen bei arithmetischen Operationen*.

Für Format I gelten die folgenden dezimalen Ganzzahl-Längen und möglichen Werte:

Format/Länge	Dezimale Ganzzahl-Länge	Mögliche Werte
I1	3	-128 bis 127
I2	5	-32768 bis 32767
I4	10	-2147483648 bis 2147483647

Arithmetische Operationen mit Gleitkomma-Zahlen

Folgende Themen werden behandelt:

- Einige allgemeine Hinweise
- Genauigkeit von Gleitkomma-Zahlen
- Konvertierung in Gleitkomma-Darstellung
- Plattform-abhängige Unterschiede
- Potenzierung

Einige allgemeine Hinweise

Gleitkomma-Zahlen (Format F) werden ebenso wie Ganzzahlen (Format I) als Summe von Zweierpotenzen dargestellt, wohingegen ungepackte und gepackte Zahlen (Formate N und P) als Summe von Zehnerpotenzen dargestellt werden.

Bei ungepackten oder gepackten Zahlen ist die Position des Dezimalkommas fest. Bei Gleitkomma-Zahlen dagegen ist (wie der Name schon andeutet) die Position des Dezimalkommas „gleitend“, d.h. seine Position ist nicht fest, sondern hängt vom tatsächlichen Wert der Zahl ab.

Gleitkomma-Zahlen sind unverzichtbar bei der Berechnung trigonometrischer und mathematischer Funktionen wie etwa Sinus oder Logarithmus.

Genauigkeit von Gleitkomma-Zahlen

Die Genauigkeit von Gleitkomma-Zahlen an sich ist begrenzt:

- Bei einer Variablen mit Format/Länge F4 ist die Genauigkeit auf etwa 7 Stellen begrenzt.
- Bei einer Variablen mit Format/Länge F8 ist die Genauigkeit auf 16 Stellen begrenzt.

Werte mit einer größeren Anzahl signifikanter Stellen lassen sich nicht exakt als Gleitkomma-Zahlen darstellen. Unabhängig von der Zahl zusätzlicher Vor- oder Nachkommastellen kann eine Gleitkomma-Zahl nur die ersten 7 bzw. 16 Stellen abdecken.

Eine Ganzzahl lässt sich nur exakt in einer Variablen mit Format/Länge F4 darstellen, wenn ihr absoluter Wert nicht größer als $2^{24}-1$ ist.

Konvertierung in Gleitkomma-Darstellung

Wenn ein alphanumerischer, ungepackter numerischer oder gepackter numerischer Wert in Gleitkomma-Format umgesetzt wird (zum Beispiel bei einer Zuweisung), muss auch die Darstellungsform geändert werden, d.h. eine Summe von Zehnerpotenzen muss in eine Summe von Zweierpotenzen konvertiert werden.

Folglich lassen sich nur Zahlen, die als endliche Summe von Zweierpotenzen darstellbar sind, exakt darstellen; alle anderen Zahlen lassen sich nur näherungsweise darstellen.

Beispiele:

Diese Zahl hat eine exakte Gleitkomma-Darstellung:

$$1.25 = 2^{0} + 2^{-2}$$

Diese Zahl ist eine periodische Gleitkomma-Zahl ohne exakte Darstellung:

$$1.2 = 2^{0} + 2^{-3} + 2^{-4} + 2^{-7} + 2^{-8} + 2^{-11} + 2^{-12} + \ldots$$

Daher kann die Konvertierung von alphanumerischen, ungepackt numerischen oder gepackt numerischen Werten in Gleitkomma-Werte, und umgekehrt, zu kleineren Fehlern führen.

Anmerkung: Wenn Sie ein ganzzahliges, nicht gepacktes oder gepacktes Ergebnis einer arithmetischen Operation (siehe *Format/Länge von Ergebnisfeldern bei arithmetischen Operationen*) in eine Gleitkommadarstellung konvertiert werden soll, sollten Sie in Betracht ziehen, die arithmetische Operation bereits im Gleitkommaformat durchzuführen, um die Genauigkeit zu verbessern.

Beispiel:

```
#F1 (F8) := 1     / 12  /* Result is +8.333330000000000E-02
#F2 (F8) := 1.0E0 / 12  /* Result is +8.333333333333333E-02
```

Plattform-abhängige Unterschiede

Aufgrund der unterschiedlichen Hardware-Architektur ist die Darstellung von Gleitkomma-Zahlen auf Großrechnern anders als auf anderen Plattformen. Dies erklärt, warum dieselbe Anwendung bei Gleitkomma-Berechnungen auf verschiedenen Plattformen möglicherweise geringfügig andere Ergebnisse liefert. Die entsprechende Darstellung bestimmt auch den möglichen Wertebereich für Gleitkomma-Variablen. Dieser beträgt (ca.):

$\pm 5.4 * 10^{-79}$ bis $\pm 7.2 * 10^{75}$ für F4- und F8-Variablen.

Anmerkung: Die von Ihrem Taschenrechner verwendete Darstellung kann sich ebenfalls von der Ihres Computers unterscheiden und die Ergebnisse für die gleiche Berechnung können daher auch hier unterschiedlich sein.

Potenzierung

Eine Potenzierungsoperation `(operand1 ** operand2)` wird intern berechnet als `EXP(operand2 * LOG(operand1))` unter Verwendung der mathematischen Funktionen `EXP` und `LOG` (siehe *Mathematische Systemfunktionen* in der *Systemfunktionen*-Dokumentation).

Arithmetische Operationen mit Datum und Zeit

Mit Feldern der Formate D (Datum) und T (Time = Zeit) sind nur Addition, Subtraktion, Multiplikation und Division erlaubt. Multiplikation und Division sind nur bei Zwischenergebnissen von Addition und Subtraktion zulässig.

Datums-/Zeitwerte können addiert bzw. voneinander subtrahiert werden oder Ganzzahl-Werte (ohne Nachkommastellen) können zu/von Datums-/Zeitwerten addiert/subtrahiert werden. Solche ganzzahligen Werte können in Feldern der Formate N, P, I, D oder T enthalten sein.

Die Zwischenergebnisse einer solchen Addition oder Subtraktion können als Multiplikand oder Dividend in einer nachfolgenden Operation verwendet werden.

Von ganzzahligen Werten, die zu einem Datumswert addiert oder von einem Datumswert subtrahiert werden, wird angenommen, dass es sich um Tage handelt. Von ganzzahligen Werten, die zu einem Zeitwert addiert oder von einem Zeitwert subtrahiert werden, wird angenommen, dass es sich um Zehntelsekunden handelt.

Bei arithmetischen Operationen mit Datum und Zeit gelten gewisse Einschränkungen, und zwar aufgrund von Naturals interner Behandlung von Datums- und Zeitarithmetik, wie im folgenden erläutert.

Intern behandelt Natural eine arithmetische Operation mit Datums- bzw. Zeitvariablen wie folgt:

```
COMPUTE result-field = operand1 +/- operand2
```

Das obige Statement wird aufgelöst als:

1. `intermediate-result = operand1 +/- operand2`
2. `result-field = intermediate-result`

Das heißt, zunächst berechnet Natural das Ergebnis der Addition/Subtraktion, und erst danach weist es das Ergebnis dem Ergebnisfeld zu.

Komplexere arithmetische Operationen werden nach dem gleichen Muster aufgelöst:

```
COMPUTE result-field = operand1 +/- operand2 +/- operand3 +/- operand4
```

Das obige Statement wird aufgelöst als:

1. `intermediate-result1 = operand1 +/- operand2`
2. `intermediate-result2 = intermediate-result1 +/- operand3`
3. `intermediate-result3 = intermediate-result2 +/- operand4`
4. `result-field = intermediate-result3`

Die Auflösung bei der Multiplikation und Division ist ähnlich wie die Auflösung bei der Addition und Subtraktion.

Das interne Format eines solchen Zwischenergebnisses (`intermediate-result`) hängt vom Format der einzelnen Operanden ab, wie die folgenden Tabellen zeigen.

Addition

Die folgende Tabelle zeigt das Format vom Zwischenergebnis einer Addition (`intermediate-result` = `operand1` + `operand2`):

Format von *operand1*	Format von *operand2*	Format von *intermediate-result*
D	D	Di
D	T	T
D	Di, Ti, N, P, I	D
T	D, T, Di, Ti, N, P, I	T
Di, Ti, N, P, I	D	D
Di, Ti, N, P, I	T	T
Di, N, P, I	Di	Di
Ti, N, P, I	Ti	Ti
Di	Ti, N, P, I	Di
Ti	Di, N, P, I	Ti

Subtraktion

Die folgende Tabelle zeigt das Format des Zwischenergebnisses einer Subtraktion (*intermediate-result* = *operand1* - *operand2*):

Format von *operand1*	Format von *operand2*	Format von *intermediate-result*
D	D	Di
D	T	Ti
D	Di, Ti, N, P, I	D
T	D, T	Ti
T	Di, Ti, N, P, I	T
Di, N, P, I	D	Di
Di, N, P, I	T	Ti
Di	Di, Ti, N, P, I	Di
Ti	D, T, Di, Ti, N, P, I	Ti
N, P, I	Di, Ti	P12

Multiplikation oder Division

Die folgende Tabelle zeigt das Format des Zwischenergebnisses einer Multiplikation: (*intermediate-result* = *operand1* * *operand2*) oder Division (*intermediate-result* = *operand1* / *operand2*):

Format von *operand1*	Format von *operand2*	Format von *intermediate-result*
D	D, Di, Ti, N, P, I	Di
D	T	Ti
T	D, T, Di, Ti, N, P, I	Ti
Di	T	Ti
Di	D, Di, Ti, N, P, I	Di
Ti	D	Di
Ti	Di, T, Ti, N, P, I	Ti
N, P, I	D, Di	Di
N, P, I	T, Ti	Ti

Interne Zuweisungen

Di ist ein Wert im internen Datumsformat; Ti ist ein Wert im internen Zeitformat; solche Werte können zwar in weiteren arithmetischen Datums-/Zeitoperationen verwendet werden, aber sie können keinem Ergebnisfeld vom Format D zugewiesen werden (siehe Zuweisungstabelle unten).

Bei komplexen arithmetischen Operationen, bei denen ein Zwischenergebnis im internen Format Di bzw. Ti als Operand für eine weitere Addition/Subtraktion/Multiplikation/Division verwendet wird, wird davon ausgegangen, dass es Format D bzw. T hat.

Die folgende Tabelle zeigt, welche Zwischenergebnisse intern welchen Ergebnisfeldern zugewiesen werden können (*result-field* = *intermediate-result*).

Format von *result-field*	Format von *intermediate-result*	Zuweisung möglich
D	D, T	ja
D	Di, Ti, N, P, I	nein
T	D, T, Di, Ti, N, P, I	ja
N, P, I	D, T, Di, Ti, N, P, I	ja

Ein Ergebnisfeld vom Format D oder T darf keinen negativen Wert enthalten.

Beispiele 1 und 2 (ungültig):

```
COMPUTE DATE1 (D) = DATE2 (D) + DATE3 (D)
COMPUTE DATE1 (D) = DATE2 (D) - DATE3 (D)
```

Diese Operationen sind nicht möglich, da das Zwischenergebnis der Addition bzw. Subtraktion Format Di hätte, und ein Wert vom Format D*i* keinem Ergebnisfeld vom Format D zugewiesen werden kann.

Beispiele 3 und 4 (ungültig):

```
COMPUTE DATE1 (D) = TIME2 (T) - TIME3 (T)
COMPUTE DATE1 (D) = DATE2 (D) - TIME3 (T)
```

Diese Operationen sind nicht möglich, da das Zwischenergebnis der Addition bzw. Subtraktion Format T *i* hätte, und ein Wert vom Format T *i* keinem Ergebnisfeld vom Format D zugewiesen werden kann.

Beispiel 5 (gültig):

```
COMPUTE DATE1 (D) = DATE2 (D) - DATE3 (D) + TIME3 (T)
```

Diese Operation ist möglich. Zunächst wird `DATE3` von `DATE2` subtrahiert, woraus sich ein Zwischenergebnis vom Format D *i* ergibt; dann wird dieses Zwischenergebnis zu `TIME3` hinzuaddiert, woraus sich ein Zwischenergebnis vom Format T ergibt; und schließlich wird dieses zweite Zwischenergebnis dem Ergebnisfeld `DATE1` zugewiesen.

Beispiele 6 und 7 (ungültig):

```
COMPUTE DATE1 (D) = DATE2 (D) + DATE3 (D) * 2
COMPUTE TIME1 (T) = TIME2 (T) - TIME3 (T) / 3
```

Diese Operationen sind nicht möglich, da die versuchte Multiplikation bzw. Division mit Datums-/Zeitfeldern und nicht mit Zwischenergebnissen durchgeführt wird.

Beispiel 8 (gültig):

```
COMPUTE DATE1 (D) = DATE2 (D) + (DATE3(D) - DATE4 (D)) * 2
```

Diese Operation ist möglich. Zunächst wird `DATE4` von `DATE3` subtrahiert, woraus sich ein Zwischenergebnis vom Format D *i* ergibt; dann wird dieses Zwischenergebnis mit 2 multipliziert, woraus sich ein Zwischenergebnis vom Format D *i* ergibt; dieses Zwischenergebnis wird zu `DATE2` addiert, woraus sich ein Zwischenergebnis vom Format D ergibt; und schließlich wird dieses dritte Zwischenergebnis dem Ergebnisfeld `DATE1` zugewiesen.

Wenn Sie einen Format-T-Wert einem Format-D-Feld zuweisen, müssen Sie dafür sorgen, dass der Zeitwert eine gültige Datumskomponente enthält.

Formatwahl im Hinblick auf die Verarbeitungszeit

Bei arithmetischen Operationen hat die Wahl der richtigen Feldformate starken Einfluss auf die Verarbeitungszeit:

Bei kaufmännischen Berechnungen empfiehlt es sich, nur Felder mit dem Format P (numerisch gepackt) zu verwenden. Die Anzahl der Stellen hinter dem Komma (Dezimalpunkt) sollte möglichst für alle Operanden einheitlich gewählt werden.

Bei wissenschaftlichen Berechnungen empfiehlt es sich, nur Felder mit dem Format F (Gleitkomma-Format) zu verwenden.

Werden die numerischen Formate N und P mit dem Format F vermischt, erfolgt intern eine Umsetzung in Format F; diese Umsetzung führt zu einer nicht unbeträchtlichen CPU-Beanspruchung. Daher sollte es möglichst vermieden werden, bei arithmetischen Operationen unterschiedliche Formate zu verwenden.

Genauigkeit von Ergebnissen bei arithmetischen Operationen

Operation	Stellen vor dem Komma	Stellen nach dem Komma
Addition/Subtraktion	Fi + 1 oder Si + 1 (das jeweils größere)	Fd oder Sd (das jeweils größere)
Multiplikation	Fi + Si	■ wenn Fd + Sd kleiner als MAXPREC sind: Fd + Sd ■ wenn Fd + Sd größer gleich MAXPREC sind: Fd oder Sd oder MAXPREC (das jeweils größere)
Division	Fi + Sd	(siehe unten)
Potenzierung	29 - Fd (Siehe *Ausnahme* unten)	Fd
Quadratwurzel	■ wenn Fi ein geradzahliger Wert ist: Fi/2 ■ wenn Fi ein ungeradzahliger Wert ist: (Fi + 1)/2	■ wenn Fi ein geradzahliger Wert ist: 31 – Fi/2 oder MAXPREC (das jeweils kleinere) ■ wenn Fi ein ungeradzahliger Wert ist: 31 – (Fi + 1)/2 oder MAXPREC (das jeweils kleinere)

Dabei ist:

F	Erster Operand
S	Zweiter Operand
R	Ergebnis
i	Stellen vor dem Komma (Dezimalpunkt)
d	Stellen nach dem Komma (Dezimalpunkt)
MAXPREC	Maximale Anzahl von Nachkommastellen entsprechend der Festlegung mit der Option `MAXPREC` des Systemkommandos `COMPOPT` oder mit dem Subparameter `MAXPREC` des Profilparameters `CMPO`. Der Standrdwertwert ist `7`.

Ausnahme:

Wenn die Hochzahl eine oder mehrere Stellen hinter dem Komma (Dezimalpunkt) aufweist, wird die Potenzierung intern im Gleitkomma-Format ausgeführt und das Ergebnis hat ebenfalls Gleitkomma-Format. Weitere Informationen siehe Abschnitt *Arithmetische Operationen mit Gleitkomma-Zahlen*.

Nachkommastellen bei Divisionsergebnissen

Die Genauigkeit des Ergebnisses einer Division hängt davon ab, ob ein Ergebnisfeld vorhanden ist oder nicht:

- Ist ein Ergebnisfeld vorhanden, ist die Genauigkeit: `Fd` oder `Rd` (das jeweils größere) [*].
- Ist kein Ergebnisfeld vorhanden, ist die Genauigkeit: `Fd` oder `Sd` (das jeweils größere) [*].

[*] Bei Verwendung der `ROUNDED`-Option erhöht sich die Ergebnisgenauigkeit intern um eine Stelle, bevor das Ergebnis tatsächlich gerundet wird, solange der Wert der Option `MAXPREC` dabei nicht überschritten wird.

Ein Ergebnisfeld ist vorhanden (bzw. wird als vorhanden angenommen) in einem `COMPUTE`- und `DIVIDE`-Statement sowie in einer logischen Bedingung, in der die Division hinter dem Vergleichsoperator steht (z.B.: `IF #A = #B / #C THEN ...`).

Ein Ergebnisfeld wird als nicht vorhanden angenommen in einer logischen Bedingung, in der die Division vor dem Vergleichsoperator steht (z.B.: `IF #B / #C = #A THEN ...`).

Ausnahme:

Wenn Dividend und Divisor Ganzzahlen sind und mindestens eine davon eine Variable ist, dann ist auch das Divisionsergebnis eine Ganzzahl (unabhängig von der Genauigkeit des Ergebnisfeldes sowie der Verwendung der `ROUNDED`-Option).

Genauigkeit von Ergebnissen bei arithmetischen Ausdrücken

Die Genauigkeit von arithmetischen Ausdrücken, zum Beispiel `#A / (#B * #C) + #D * (#E - #F + #G)`, wird von der Auswertung der Ergebnisse von arithmetischen Operationen in ihrer Verarbeitungsreihenfolge hergeleitet. Weitere Informationen zu arithmetischen Ausdrücken siehe *arithmetic-expression* in der Beschreibung des `COMPUTE`-Statements.

Fehlerbedingungen bei arithmetischen Operationen

Bei einer Addition, Subtraktion, Multiplikation oder Division kann ein Fehler auftreten, wenn das Ergebnis (insgesamt, d.h. vor und nach dem Komma) mehr als 31 Stellen hat.

Bei einer Potenzierung erhalten Sie unter einer der folgenden Bedingungen einen Fehler:

- wenn die Basis gepacktes Format mit Dezimalstellen (zum Beispiel P3.2) hat und der Exponent größer als 16 ist;
- wenn die Basis Gleitkomma-Format hat und das Ergebnis größer ist als ca. $7 * 10^{75}$.

Verarbeitung von Arrays

Grundsätzlich gelten folgende Regeln:

- Alle Skalar-Operationen können auf Array-Elemente angewandt werden, die aus einer einzigen Ausprägung bestehen.
- Wenn eine Variable mit einem konstanten Wert definiert ist (z.B. `#FIELD (I2) CONSTANT <8>`), dann wird der Wert der Variablen bei der Kompilierung zugewiesen, und die Variable wird als Konstante behandelt. Falls eine solche Variable in einem Array-Index verwendet wird, bedeutet dies, dass die betreffende Dimension eine *bestimmte* Anzahl von Ausprägungen hat.
- Bei einer Zuweisung bzw. einem Vergleich zwischen zwei Arrays mit unterschiedlich vielen Dimensionen wird angenommen, dass die „fehlende“ Dimension in dem Array mit weniger Dimensionen (1:1) ist.

  Beispiel: Wenn das Array `#ARRAY1 (1:2)` dem Array `#ARRAY2 (1:2,1:2)` zugewiesen wird, wird für `#ARRAY1` angenommen, dass es `#ARRAY1 (1:1,1:2)` ist.

Folgende Themen werden behandelt:

- Arithmetische Operationen mit Arrays

Definition von Array-Dimensionen

Die erste, zweite und dritte Dimension eines Arrays werden wie folgt definiert:

Dimensionen	Eigenschaften
3	#a3 (3. Dim., 2. Dim., 1. Dim.)
2	#a2 (2. Dim., 1. Dim.)
1	#a1 (1. Dim.)

Zuweisungen bei Arrays

Wenn Sie einen Array-Bereich einem anderen Array-Bereich zuweisen, erfolgt die Zuweisung Element für Element.

Beispiel:

```
DEFINE DATA LOCAL
1 #ARRAY(I4/1:5) INIT <10,20,30,40,50>
END-DEFINE
*
MOVE #ARRAY(2:4) TO #ARRAY(3:5)
/* is identical to
/* MOVE #ARRAY(2) TO #ARRAY(3)
/* MOVE #ARRAY(3) TO #ARRAY(4)
/* MOVE #ARRAY(4) TO #ARRAY(5)
/*
/* #ARRAY contains  10,20,20,20,20
```

Wenn Sie eine einzelne Ausprägung einem Array-Bereich zuweisen, wird jedes Element des Bereiches mit dem Wert der einzelnen Ausprägung gefüllt. (Bei einer mathematischen Funktion wird jedes Element des Bereiches mit dem Ergebnis der Funktion gefüllt.)

Bevor eine Zuweisung ausgeführt wird, werden die einzelnen Dimensionen der betroffenen Arrays miteinander verglichen, um zu prüfen, ob sie eine der unten aufgeführten Bedingungen erfüllen.

Die Dimensionen werden dabei unabhängig voneinander verglichen; d.h. die 1. Dimension des einen Arrays wird mit der 1. Dimension des anderen Arrays verglichen, die 2. Dimension des einen Arrays wird mit der 2. Dimension des anderen Arrays verglichen, und die 3. Dimension des einen Arrays wird mit der 3. Dimension des anderen Arrays verglichen.

Die Zuweisung von Werten eines Arrays an ein anderes Array ist nur unter einer der folgenden Bedingungen erlaubt:

- Die zwei verglichenen Dimensionen haben die gleiche Anzahl von Ausprägungen.

- Die zwei verglichenen Dimensionen haben beide eine unbestimmte Anzahl von Ausprägungen.
- Die Dimension, die einer anderen Dimension zugewiesen wird, besteht aus einer einzelnen Ausprägung.

Beispiel für Array-Zuweisungen:

Das folgende Programm zeigt, welche Zuweisungen zwischen Arrays möglich sind.

```
 DEFINE DATA LOCAL
 1 A1   (N1/1:8)
 1 B1   (N1/1:8)
 1 A2   (N1/1:8,1:8)
 1 B2   (N1/1:8,1:8)
 1 A3   (N1/1:8,1:8,1:8)
 1 I    (I2)          INIT <4>
 1 J    (I2)          INIT <8>
 1 K    (I2)          CONST <8>
 END-DEFINE
 *
 COMPUTE A1(1:3) = B1(6:8)                                    /* allowed
 COMPUTE A1(1:I) = B1(1:I)                                    /* allowed
 COMPUTE A1(*)   = B1(1:8)                                    /* allowed
 COMPUTE A1(2:3) = B1(I:I+1)                                  /* allowed
 COMPUTE A1(1)   = B1(I)                                      /* allowed
 COMPUTE A1(1:I) = B1(3)                                      /* allowed
 COMPUTE A1(I:J) = B1(I+2)                                    /* allowed
 COMPUTE A1(1:I) = B1(5:J)                                    /* allowed
 COMPUTE A1(1:I) = B1(2)                                      /* allowed
 COMPUTE A1(1:2) = B1(1:J)                                    /* NOT ALLOWED ↩
(NAT0631)
 COMPUTE A1(*)   = B1(1:J)                                    /* NOT ALLOWED ↩
(NAT0631)
 COMPUTE A1(*)   = B1(1:K)                                    /* allowed
 COMPUTE A1(1:J) = B1(1:K)                                    /* NOT ALLOWED ↩
(NAT0631)
 *
 COMPUTE A1(*)       = B2(1,*)                                /* allowed
 COMPUTE A1(1:3)     = B2(1,I:I+2)                            /* allowed
 COMPUTE A1(1:3)     = B2(1:3,1)                              /* NOT ALLOWED ↩
(NAT0631)
 *
 COMPUTE A2(1,1:3)   = B1(6:8)                                /* allowed
 COMPUTE A2(*,1:I)   = B1(5:J)                                /* allowed
 COMPUTE A2(*,1)     = B1(*)                                  /* NOT ALLOWED ↩
(NAT0631)
 COMPUTE A2(1:I,1)   = B1(1:J)                                /* NOT ALLOWED ↩
(NAT0631)
 COMPUTE A2(1:I,1:J) = B1(1:J)                                /* allowed
 *
 COMPUTE A2(1,I)     = B2(1,1)                                /* allowed
```

```
 COMPUTE A2(1:I,1)   = B2(1:I,2)                                  /* allowed
 COMPUTE A2(1:2,1:8) = B2(I:I+1,*)                                /* allowed
 *
 COMPUTE A3(1,1,1:I)   = B1(1)                                    /* allowed
 COMPUTE A3(1,1,1:J)   = B1(*)                                    /* NOT ALLOWED ↩
(NAT0631)
  COMPUTE A3(1,1,1:I)   = B1(1:I)                                 /* allowed
 COMPUTE A3(1,1:2,1:I) = B2(1,1:I)                                /* allowed
 COMPUTE A3(1,1,1:I)   = B2(1:2,1:I)                              /* NOT ALLOWED ↩
(NAT0631)
 END
```

Vergleiche mit Arrays

Grundsätzlich gilt Folgendes: Wenn mehrdimensionale Arrays miteinander verglichen werden, werden die einzelnen Dimensionen unabhängig voneinander behandelt; d.h. die 1. Dimension des einen Arrays wird mit der 1. Dimension des anderen Arrays verglichen, die 2. Dimension des einen Arrays wird mit der 2. Dimension des anderen Arrays verglichen, und die 3. Dimension des einen Arrays wird mit der 3. Dimension des anderen Arrays verglichen.

Der Vergleich zweier Array-Dimensionen ist nur unter einer der folgenden Bedingungen erlaubt:

- Die zwei verglichenen Dimensionen haben die gleiche Anzahl von Ausprägungen.
- Die zwei verglichenen Dimensionen haben beide eine unbestimmte Anzahl von Ausprägungen.
- Alle Dimensionen des einen Arrays bestehen jeweils aus einer einzelnen Ausprägung.

Beispiel für Array-Vergleiche:

Das folgende Programm zeigt, welche Vergleiche zwischen Arrays möglich sind.

```
 DEFINE DATA LOCAL
 1 A3  (N1/1:8,1:8,1:8)
 1 A2  (N1/1:8,1:8)

 1 A1  (N1/1:8)
 1 I   (I2)   INIT <4>
 1 J   (I2)   INIT <8>
 1 K   (I2)   CONST <8>
 END-DEFINE
 *
 IF A2(1,1)     = A1(1)              THEN IGNORE END-IF /* allowed
 IF A2(1,1)     = A1(I)              THEN IGNORE END-IF /* allowed
 IF A2(1,*)     = A1(1)              THEN IGNORE END-IF /* allowed
 IF A2(1,*)     = A1(I)              THEN IGNORE END-IF /* allowed
 IF A2(1,*)     = A1(*)              THEN IGNORE END-IF /* allowed
 IF A2(1,*)     = A1(I -3:I+4)       THEN IGNORE END-IF /* allowed
 IF A2(1,5:J)   = A1(1:I)            THEN IGNORE END-IF /* allowed
 IF A2(1,*)     = A1(1:I)            THEN IGNORE END-IF /* NOT ALLOWED(NAT0629)
```

```
IF A2(1,*)     = A1(1:K)            THEN IGNORE END-IF /* allowed
*
IF A2(1,1)     = A2(1,1)            THEN IGNORE END-IF /* allowed
IF A2(1,1)     = A2(1,I)            THEN IGNORE END-IF /* allowed
IF A2(1,*)     = A2(1,1:8)          THEN IGNORE END-IF /* allowed
IF A2(1,*)     = A2(I,I -3:I+4)     THEN IGNORE END-IF /* allowed
IF A2(1,1:I)   = A2(1,I+1:J)        THEN IGNORE END-IF /* allowed
IF A2(1,1:I)   = A2(1,I:I+1)        THEN IGNORE END-IF /* NOT ALLOWED(NAT0629)
IF A2(*,1)     = A2(1,*)            THEN IGNORE END-IF /* NOT ALLOWED(NAT0629)
IF A2(1,1:I)   = A1(2,1:K)          THEN IGNORE END-IF /* NOT ALLOWED(NAT0629)
*
IF A3(1,1,*)   = A2(1,*)            THEN IGNORE END-IF /* allowed
IF A3(1,1,*)   = A2(1,I -3:I+4)     THEN IGNORE END-IF /* allowed
IF A3(1,*,I:J) = A2(*,1:I+1)        THEN IGNORE END-IF /* allowed
IF A3(1,*,I:J) = A2(*,I:J)          THEN IGNORE END-IF /* allowed
END
```

Wenn Sie zwei Array-Bereiche miteinander vergleichen, beachten Sie bitte, dass die folgenden zwei Ausdrücke zu unterschiedlichen Ergebnissen führen:

```
#ARRAY1(*) NOT EQUAL #ARRAY2(*)
NOT #ARRAY1(*) = #ARRAY2(*)
```

Beispiel:

- **Bedingung A:**

  ```
  IF #ARRAY1(1:2) NOT EQUAL #ARRAY2(1:2)
  ```

  Dies entspricht:

  ```
  IF (#ARRAY1(1) NOT EQUAL #ARRAY2(1)) AND (#ARRAY1(2) NOT EQUAL #ARRAY2(2))
  ```

  Bedingung A ist also erfüllt, wenn die erste Ausprägung von `#ARRAY1` ungleich der ersten Ausprägung von `#ARRAY2` ist und die zweite Ausprägung von `#ARRAY1` ungleich der zweiten Ausprägung von `#ARRAY2` ist.

- **Bedingung B:**

  ```
  IF NOT #ARRAY1(1:2) = #ARRAY2(1:2)
  ```

  Dies entspricht:

```
IF NOT (#ARRAY1(1)= #ARRAY2(1) AND #ARRAY1(2) = #ARRAY2(2))
```

Dies wiederum entspricht:

```
IF (#ARRAY1(1) NOT EQUAL #ARRAY2(1)) OR (#ARRAY1(2) NOT EQUAL #ARRAY2(2))
```

Bedingung B ist also erfüllt, wenn *entweder* die erste Ausprägung von #ARRAY1 ungleich der ersten Ausprägung von #ARRAY2 ist *oder* die zweite Ausprägung von #ARRAY1 ungleich der zweiten Ausprägung von #ARRAY2 ist.

Arithmetische Operationen mit Arrays

Eine allgemeine Regel zu Arrays lautet, dass die Anzahl der Ausprägungen der entsprechenden Dimensionen gleich sein muss.

Das folgende Beispiel veranschaulicht diese Regel:

```
#c(2:3,2:4) := #a(3:4,1:3) + #b(3:5)
```

Mit anderen Worten:

Array	Dimension	Anzahl der Ausprägungen	Bereich
#c	2.	2	2:3
#c	1.	3	2:4
#a	2.	2	3:4
#a	1.	3	1:3
#b	1.	3	3:5

Die Operation wird Element für Element durchgeführt

Anmerkung: Eine arithmetische Operation einer unterschiedlichen Anzahl von Dimensionen ist zulässig.

Für das obige Beispiel werden die folgenden Operationen ausgeführt:

```
#c(2,2) := #a(3,1) + #b(3)

#c(2,3) := #a(3,2) + #b(4)

#c(2,4) := #a(3,3) + #b(5)

#c(3,2) := #a(4,1) + #b(3)
```

```
#c(3,3) := #a(4,2) + #b(4)

#c(3,4) := #a(4,3) + #b(5)
```

In arithmetischen Operationen (in `COMPUTE`-, `ADD`- oder `MULTIPLY`-Statements) können Array-Bereiche auf folgende Arten verwendet werden. In den Beispielen 1 - 4 muss die Anzahl der Ausprägungen der entsprechenden Dimensionen gleich sein.

1. *range + range = range*.

   Die Addition wird Element für Element ausgeführt.

2. *range * range = range*.

   Die Multiplikation wird Element für Element ausgeführt.

3. *scalar + range = range*.

   Der Skalarwert wird zu jedem Element des Bereichs addiert.

4. *range * scalar = range*.

   Jedes Element des Bereichs wird mit dem Skalarwert multipliziert.

5. *range + scalar = scalar*.

   Jedes Element des Bereichs wird zum Skalarwert addiert und das Ergebnis im Skalar ausgegeben.

6. *scalar * range = scalar2*.

   Der Skalarwert wird mit jedem Element des Arrays multipliziert und das Ergebnis in *scalar2* ausgegeben.

Weil, wie aus den Beispielen 1 - 4 hervorgeht, bei arithmetischen Operationen keine Zwischenergebnisse erzeugt werden, muss das errechnete Ergebnis (siehe ***Format/Länge von Ergebnisfeldern bei arithmetischen Operationen***) dasselbe Format haben wie der Ergebnis-Operand (die Formate P und N gelten als gleiche Formate).

Beispiel:

```
DEFINE DATA LOCAL
1 #ARRAYI4(I4/1:5)
1 #ARRAYP5(P5/1:5)
END-DEFINE
*
#ARRAYI4(*) := #ARRAYP5(*) + 1  /* NOT ALLOWED(NAT0294)
```

Weil, wie aus den obigen Beispielen hervorgeht, bei arithmetischen Operationen keine Zwischenergebnisse erzeugt werden, wird das Ergebnis von sich überlappenden Indexbereichen Element für Element berechnet.

Beispiel:

```
DEFINE DATA LOCAL
1 #ARRAY(I4/1:5) INIT <10,20,30,40,50>
END-DEFINE
*
#ARRAY(3:5) := #ARRAY(2:4) + 1
/* is identical to
/* #ARRAY(3) := #ARRAY(2) + 1
/* #ARRAY(4) := #ARRAY(3) + 1
/* #ARRAY(5) := #ARRAY(4) + 1
/*
/* #ARRAY contains  10,20,21,22,23
```

50 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.

Struktur des IF-Statements

Das IF-Statement hat drei Bestandteile: IF, THEN und ELSE.

IF	Mit der IF-Klausel geben Sie eine logische Bedingung an, die erfüllt werden soll.
THEN	Mit der THEN-Klausel geben Sie die Statements an, die ausgeführt werden sollen, wenn diese Bedingung erfüllt wird.
ELSE	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 condition
   THEN execute statement(s)
   ELSE execute other statement(s)
END-IF
```

Anmerkung: 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. das IGNORE-Statement bewirkt, dass die IF-Bedingung ignoriert wird, wenn sie erfüllt wird.

Beispiel 1:

```
** Example 'IFX01': IF
************************************************************************
DEFINE DATA LOCAL
1 MYVIEW VIEW OF EMPLOYEES
  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.

Ausgabe des Programms IFX01:

```
        NAME            DATE      ANNUAL
                         OF       SALARY
                       BIRTH
-------------------- ---------- ----------

***** 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
```

Beispiel 2:

```
** Example 'IFX03': IF
************************************************************************
DEFINE DATA LOCAL
1 EMPLOY-VIEW VIEW OF EMPLOYEES
  2 NAME
  2 CITY
  2 BONUS  (1,1)
  2 SALARY (1)
*
1 #INCOME  (N9)
1 #TEXT    (A26)
END-DEFINE
*
WRITE TITLE '-- DISTRIBUTION OF CATALOGS I AND II --' /
*
READ (3) EMPLOY-VIEW BY CITY = 'SAN FRANSISCO'
  COMPUTE #INCOME = BONUS(1,1) + SALARY(1)
  /*
 IF #INCOME > 40000
    MOVE 'CATALOGS I AND II' TO #TEXT
```

```
  ELSE
    MOVE 'CATALOG I'         TO #TEXT
  END-IF
  /*
  DISPLAY NAME 5X 'SALARY' SALARY(1) / BONUS(1,1)
  WRITE T*SALARY '-'(10) /
        16X 'INCOME:' T*SALARY #INCOME 3X #TEXT /
        16X '='(19)
  SKIP 1
END-READ
END
```

Ausgabe des Programms `IFX03`:

```
                    -- DISTRIBUTION OF CATALOGS I AND II --

        NAME               SALARY
                           BONUS
--------------------     ----------

COLVILLE JR                   56000
                                  0
                           ----------
                INCOME:       56000   CATALOGS I AND II
                ====================

RICHMOND                       9150
                                  0
                           ----------
                INCOME:        9150   CATALOG I
                ====================

MONKTON                       13500
                                600
                           ----------
                INCOME:       14100   CATALOG I
                ====================
```

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:

```
** Example 'IFX02': IF (two IF statements nested)
************************************************************************
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
    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
```

Ausgabe des Programms `IFX02`:

```
        NAME            DATE      ANNUAL     MAKE
                         OF       SALARY
                       BIRTH
-------------------- ---------- ---------- --------

***** COHEN                                          SALARY LT 20000

CREMER               1972-12-14      20000 FORD

***** FLEMING                                        SALARY LT 20000
```

```
PERREAULT            1950-05-12      30500 CHRYSLER

***** SHAW                                              SALARY LT 20000

STANWOOD             1946-09-08      31000 CHRYSLER
                                           FORD
```

51 Logische Bedingungen

Dieses Kapitel beschreibt den Zweck und die Benutzung logischer Bedingungen, die in den folgenden Statements benutzt werden können: FIND, READ, HISTOGRAM, ACCEPT/REJECT, IF, DECIDE FOR, REPEAT

Einleitung

Die Grundform einer logischen Bedingung ist ein relationaler (vergleichender) Ausdruck. Mit den logischen Operatoren AND und OR können mehrere relationale Ausdrücke (AND, OR) zu komplexen logischen Bedingungen verknüpft werden.

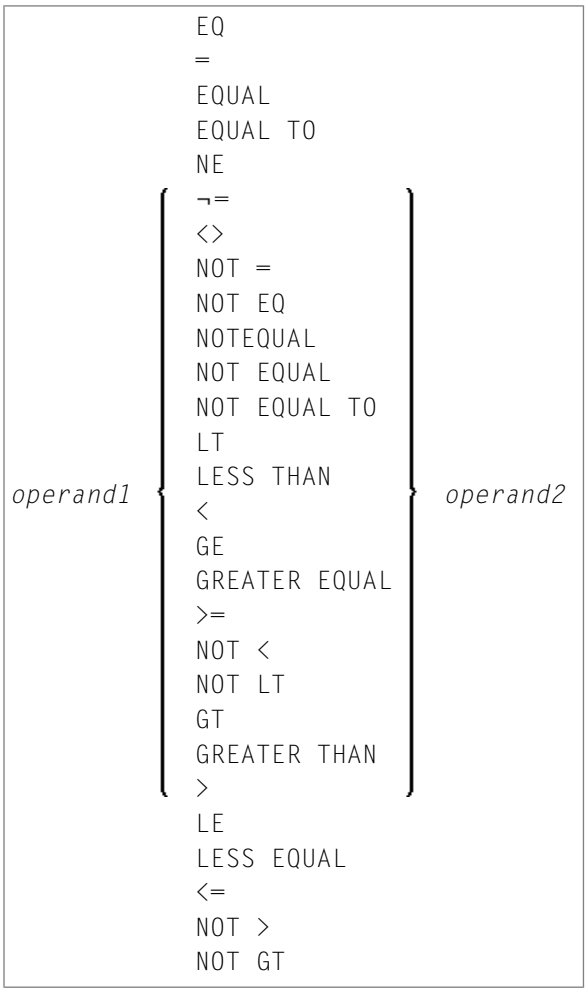
Arithmetische Ausdrücke können ebenfalls in logischen Bedingungen verwendet werden.

Logische Bedingungen können in den folgenden Statements angegeben werden:

Statement	Bedingungen
FIND	Zusätzlich zu dem primären Selektionskriterium, das in der WITH-Klausel angegeben wird, kann in einer WHERE-Klausel als zusätzliches Selektionskriterium eine logische Bedingung angegeben werden. Die in der WHERE-Klausel angegebene Bedingung wird erst ausgewertet, nachdem ein Datensatz aufgrund des WITH-Kriteriums ausgewählt und gelesen worden ist. In der WITH-Klausel werden „Basic Search Criteria“ (Suchkriterien) angegeben (vgl. FIND-Statement), aber keine logische Bedingung.
READ	In einer WHERE-Klausel kann eine logische Bedingung angegeben werden, die darüber entscheidet, ob ein gerade gelesener Datensatz weiterverarbeitet wird oder nicht. Diese Bedingung wird erst ausgewertet, *nachdem* ein Datensatz gelesen wurde.
HISTOGRAM	In einer WHERE-Klausel kann eine logische Bedingung angegeben werden, die darüber entscheidet, ob ein gerade gelesener Datensatz weiterverarbeitet wird oder nicht. Diese Bedingung wird erst ausgewertet, *nachdem* ein Datensatz gelesen wurde.
ACCEPT/REJECT	Zusätzlich zu den Selektionskriterien, aufgrund derer ein Datensatz mit einem FIND-, READ- oder HISTOGRAM-Statement gelesen wurde, kann in der IF-Klausel eines ACCEPT- oder REJECT-Statements eine weitere logische Bedingung angegeben werden, die über die weitere Verarbeitung eines Datensatzes entscheidet. Diese Bedingung wird erst ausgewertet, nachdem ein Datensatz gelesen wurde und die Verarbeitung des Datensatzes begonnen hat.
IF	Die Ausführung des Statements kann von der Erfüllung einer logischen Bedingung abhängig gemacht werden.
DECIDE FOR	Die Ausführung des Statements kann von der Erfüllung einer logischen Bedingung abhängig gemacht werden.
REPEAT	In der UNTIL- oder WHILE-Klausel eines REPEAT-Statements kann eine logische Bedingung angegeben werden, die darüber entscheidet, wann eine Verarbeitungsschleife beendet werden soll.

Relationaler Ausdruck

Syntax:



Operanden-Definitionstabelle:

Operand	Mögliche Struktur						Mögliche Formate												Referenzierung erlaubt	Dynam. Definition
operand1	C	S	A		N	E	A	U	N	P	I	F	B	D	T	L		G	O	ja	ja
operand2	C	S	A		N	E	A	U	N	P	I	F	B	D	T	L		G	O	ja	nein

Die obige Operandentabelle ist in der *Statements*-Dokumentation unter *Syntax-Symbole und Operandentabellen* erklärt.

In der Spalte „Mögliche Struktur“ der Tabelle steht „E“ für arithmetischer Ausdruck, d.h. innerhalb eines relationalen Ausdrucks kann ein beliebiger arithmetischer Ausdruck als Operand verwendet

werden. Weitere Informationen siehe *arithmetic-expression* in der Beschreibung des COMPUTE-Statements.

Erklärung der Vergleichsoperatoren:

Vergleichsoperator	Erklärung
EQ = EQUAL EQUAL TO	gleich
NE ¬= <> NOT = NOT EQ NOTEQUAL NOT EQUAL NOT EQUAL TO	ungleich
LT LESS THAN <	kleiner als
GE GREATER EQUAL >=	größer als oder gleich
NOT < NOT LT	nicht größer als
GT GREATER THAN >	größer als
LE LESS EQUAL <=	kleiner als oder gleich less
NOT > NOT GT	nicht größer als

Beispiele für relationale Ausdrücke:

```
IF NAME = 'SMITH'
IF LEAVE-DUE GT 40
IF NAME = #NAME
```

Weitere Informationen über den Vergleich von Arrays in einem relationalen Ausdruck siehe *Verarbeitung von Arrays*.

Anmerkung: Wird ein Gleitkomma-Operand verwendet, so erfolgt der Vergleich in Gleitkommaform. Da **Gleitkomma-Zahlen** per se nur eine begrenzte Genauigkeit haben, lassen sich Rundungs- bzw. Abschneidefehler bei der Konvertierung von Zahlen in/aus Gleitkommaform nicht ausschließen.

Arithmetische Ausdrücke in logischen Bedingungen

Das folgende Beispiel zeigt, wie arithmetische Ausdrücke in logischen Bedingungen eingesetzt werden können:

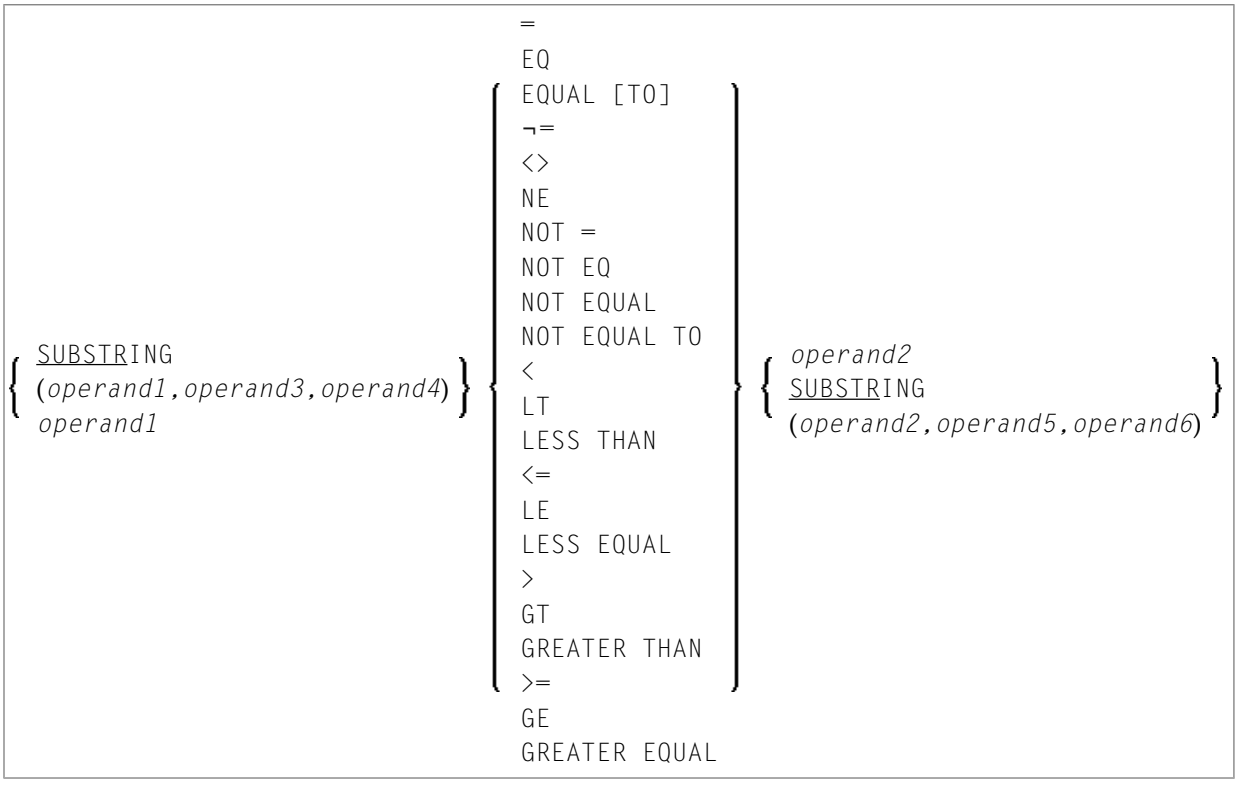
```
IF #A + 3 GT #B - 5 AND #C * 3 LE #A + #B
```

Handles in logischen Bedingungen

Wenn die Operanden in einem relationalen Ausdruck Handles sind, dürfen nur `EQUAL`- und `NOT EQUAL`-Operatoren verwendet werden.

SUBSTRING-Option in relationalem Ausdruck

Syntax:



Operanden-Definitionstabelle:

Operand	Mögliche Struktur						Mögliche Formate												Referenzierung erlaubt	Dynam. Definition
operand1	C	S	A		N		A	U				B							ja	ja
operand2	C	S	A		N		A	U				B							ja	nein
operand3	C	S							N	P	I	B							ja	nein
operand4	C	S							N	P	I								ja	nein
operand5	C	S							N	P	I								ja	nein
operand6	C	S							N	P	I								ja	nein

Mit der SUBSTRING-Option können Sie einen *Teil* eines alphanumerischen, binären oder Unicode-Feldes vergleichen. Nach dem Feldnamen (*operand1*) geben Sie zunächst die erste Stelle (*operand3*) und dann die Länge (*operand4*) des zu vergleichenden Feldteils an.

Sie können auch einen Feldwert mit einem Teil eines anderen Feldwertes vergleichen. Nach dem Feldnamen (*operand2*) geben Sie zunächst die erste Stelle (*operand5*) und dann die Länge (*operand6*) des Feldteils an, mit dem *operand1* verglichen werden soll.

Sie können auch beide Formen miteinander kombinieren, d.h. SUBSTRING gleichzeitig für *operand1* und für *operand2* angeben.

Beispiele:

Dieser Ausdruck vergleicht die 5. bis einschließlich 12. Stelle des Wertes in Feld #A mit dem Wert von Feld #B:

```
SUBSTRING(#A,5,8) = #B
```

wobei 5 die erste Stelle und 8 die Länge ist.

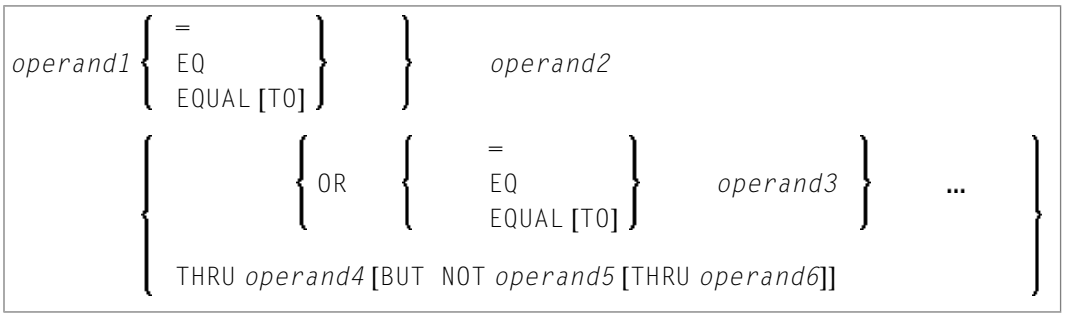
Dieser Ausdruck vergleicht den Wert von Feld #A mit der 3. bis einschließlich 6. Stelle des Wertes in Feld #B:

```
#A = SUBSTRING(#B,3,4)
```

Anmerkung: Wenn Sie *operand3*/*operand5* weglassen, wird ab Anfang des Feldes verglichen. Wenn Sie *operand4*/*operand6* weglassen, wird ab der angegebenen Stelle (*operand3*/*operand5*) bis zum Ende des Feldes verglichen.

Erweiterter Relationaler Ausdruck

Syntax:



Operanden-Definitionstabelle:

Operand	Mögliche Struktur						Mögliche Formate												Referenzierung erlaubt	Dynam. Definition
operand1	C	S	A		N*	E	A	U	N	P	I	F	B	D	T			G	O	ja	nein
operand2	C	S	A		N*	E	A	U	N	P	I	F	B	D	T			G	O	ja	nein
operand3	C	S	A		N*	E	A	U	N	P	I	F	B	D	T			G	O	ja	nein
operand4	C	S	A		N*	E	A	U	N	P	I	F	B	D	T			G	O	ja	nein
operand5	C	S	A		N*	E	A	U	N	P	I	F	B	D	T			G	O	ja	nein
operand6	C	S	A		N*	E	A	U	N	P	I	F	B	D	T			G	O	ja	nein

* Mathematische Funktionen und Systemvariablen sind erlaubt. **Gruppenwechsel**-Funktionen sind nicht erlaubt.

Operand3 kann auch unter Verwendung einer MASK- oder SCAN-Option angegeben werden; d.h. er kann angegeben werden als:

```
MASK (mask-definition) [operand]
MASK operand
SCAN operand
```

Einzelheiten zu diesen Optionen finden Sie unter ***MASK-Option*** und ***SCAN-Option***.

Beispiele:

```
IF #A = 2 OR = 4 OR = 7
IF #A = 5 THRU 11 BUT NOT 7 THRU 8
```

Auswertung einer logischen Variablen

Syntax:

operand1

Diese Option kann in Verbindung mit einer logischen Variablen (Format L) eingesetzt werden. Eine logische Variable kann die Werte `TRUE` (wahr) oder `FALSE` (falsch) haben. Mit *operand1* geben Sie den Namen der Variablen an.

Operanden-Definitionstabelle:

Operand	Mögliche Struktur						Mögliche Formate											Referenzierung erlaubt	Dynam. Definition
operand1	C	S	A											L				nein	nein

Beispiel für eine logische Variable:

```
** Example 'LOGICX05': Logical variable in logical condition
************************************************************************
DEFINE DATA LOCAL
1 #SWITCH (L)  INIT <true>
1 #INDEX  (I1)
END-DEFINE
*
FOR #INDEX 1 5
  WRITE NOTITLE #SWITCH (EM=FALSE/TRUE) 5X 'INDEX =' #INDEX
  WRITE NOTITLE #SWITCH (EM=OFF/ON) 7X 'INDEX =' #INDEX
  IF #SWITCH
    MOVE FALSE TO #SWITCH
  ELSE
    MOVE TRUE TO #SWITCH
  END-IF
  /*
  SKIP 1
END-FOR
END
```

Ausgabe des Programms `LOGICX05`:

```
TRUE      INDEX =    1
ON        INDEX =    1

FALSE     INDEX =    2
OFF       INDEX =    2

TRUE      INDEX =    3
ON        INDEX =    3

FALSE     INDEX =    4
OFF       INDEX =    4

TRUE      INDEX =    5
ON        INDEX =    5
```

Felder in logischen Bedingungen

Bei der Konstruktion logischer Bedingungen dürfen sowohl Datenbankfelder als auch Benutzervariablen verwendet werden. Datenbankfelder, die Teil einer Periodengruppe oder multiple Felder sind, dürfen ebenfalls verwendet werden. Wenn ein Bereich von Werten für multiple Felder oder ein Bereich von Ausprägungen für Periodengruppen angegeben wird, dann ist die Bedingung erfüllt, wenn der Suchwert in einem Wert bzw. einer Ausprägung innerhalb des angegebenen Bereichs gefunden wird.

Jeder verwendete Wert muss mit dem ihm in einem relationalen Ausdruck gegenüberstehenden Feld kompatibel sein. Dezimalstellen können nur bei Werten für numerische Felder angegeben werden, wobei die Anzahl der Dezimalstellen von Wert und Feld kompatibel sein muss.

Haben zwei Operanden unterschiedliches Format, wird das Format des zweiten Operanden in das des ersten umgesetzt.

Die folgende Tabelle zeigt, welche Operandenformate zusammen in einer logischen Bedingung verwendet werden können:

operand1	*operand2*												
	A	**U**	**B***n* **(***n***=<4)**	**B***n* **(***n***>=5)**	**D**	**T**	**I**	**F**	**L**	**N**	**P**	**GH**	**OH**
A	Y	Y	Y	Y									
U	Y	Y	[2]	[2]									
B*n* **(***n***=<4)**	Y	Y	Y	Y	Y	Y	Y	Y		Y	Y		
B*n* **(***n***>=5)**	Y	Y	Y	Y									

operand1	*operand2*												
	A	**U**	**B***n* (*n*=<4)	**B***n* (*n*>=5)	**D**	**T**	**I**	**F**	**L**	**N**	**P**	**GH**	**OH**
D			Y		Y	Y	Y	Y		Y	Y		
T			Y		Y	Y	Y	Y		Y	Y		
I			Y		Y	Y	Y	Y		Y	Y		
F			Y		Y	Y	Y	Y		Y	Y		
L													
N			Y		Y	Y	Y	Y		Y	Y		
P			Y		Y	Y	Y	Y		Y	Y		
GH [1]												Y	
OH [1]													Y

Legende:

Y = ja

[1] GH = GUI Handle, OH = Object Handle.

[2] Es wird davon ausgegangen, dass der Binärwert Unicode-Codepunkte enthält, und der Vergleich wird wie für einen Vergleich zweier Unicode-Werte durchgeführt. Die Länge des binären Feldes muss geradzahlig sein.

Wird ein Array mit einem Skalarwert in Relation gesetzt, so wird jedes Element des Arrays mit dem Skalarwert verglichen; die Bedingung ist erfüllt, wenn mindestens ein Array-Element die Bedingung erfüllt (`ODER`-Verknüpfung).

Wird ein Array mit einem Array in Relation gesetzt, so wird jedes Element des einen Arrays mit dem entsprechenden Element des anderen Arrays verglichen; die Bedingung ist nur dann erfüllt, wenn alle Element-Vergleiche die Bedingung erfüllen (`UND`-Verknüpfung).

Siehe auch *Verarbeitung von Arrays*.

Anmerkung: Phonetische Deskriptoren (Adabas) dürfen in einer logischen Bedingung nicht verwendet werden

Beispiele für logische Bedingungen:

```
FIND EMPLOYEES-VIEW WITH CITY = 'BOSTON' WHERE SEX = 'M'
READ EMPLOYEES-VIEW BY NAME WHERE SEX = 'M'
ACCEPT IF LEAVE-DUE GT 45
IF #A GT #B THEN COMPUTE #C = #A + #B
REPEAT UNTIL #X = 500
```

Logische Operatoren in komplexen logischen Ausdrücken

Mittels der Boole'schen Operatoren `AND`, `OR` und `NOT` ist es möglich, logische Bedingungen miteinander zu verknüpfen. Mit Hilfe von Klammern können logische Bedingungen logisch zusammengefasst werden.

Die Operatoren werden in der folgenden Reihenfolge ausgewertet:

Priorität	Logische Verknüpfung	Bedeutung
1	`( )`	Klammer-Rechnung
2	`NOT`	Negation
3	`AND`	`UND`-Verknüpfung
4	`OR`	`ODER`-Verknüpfung

Die folgenden logischen Bedingungen können miteinander verknüpft werden, um einen komplexen logischen Ausdruck zu bilden:

- **Relationale Ausdrücke**
- **Erweiterte relationale Ausdrücke**
- **`MASK`-Option**
- **`SCAN`-Option**
- **`BREAK`-Option**

Die Syntax für eine logische Bedingung (*logical-expression*) ist wie folgt:

```
[NOT] { logical-condition-criterion } [ { OR  } logical-expression ] ...
      { (logical-expression)        }   { AND }
```

Beispiele für *logical-expressions*:

```
FIND STAFF-VIEW WITH CITY = 'TOKYO'
     WHERE BIRTH GT 19610101 AND SEX = 'F'
IF NOT (#CITY = 'A' THRU 'E')
```

Informationen über den Vergleich von Arrays in einem logischen Ausdruck finden Sie in *Verarbeitung von Arrays*.

Anmerkungen:

1. Wenn mehrere *logical-condition-criteria* mit AND verknüpft werden, wird die Auswertung beendet, sobald das erste dieser Kriterien gefunden wird, das *nicht* erfüllt ist.
2. Wenn mehrere *logical-condition-criteria* mit OR verknüpft werden, wird die Auswertung beendet, sobald das erste dieser Kriterien gefunden wird, das erfüllt ist.

BREAK-Option - Aktuellen Wert mit Wert des vorangegangenen Schleifendurchlaufs vergleichen

Mit der BREAK-Option kann der aktuelle Wert eines Feldes (oder eines Teils eines Feldes) mit dem Wert verglichen werden, den das Feld im vorangegangenen Durchlauf durch die Verarbeitungsschleife hatte.

Syntax:

```
BREAK [OF] operand1 [/n/]
```

Operanden-Definitionstabelle:

Operand	Mögliche Struktur						Mögliche Formate												Referenzierung erlaubt	Dynam. Definition
operand1		S					A	U	N	P	I	F	B	D	T	L			ja	nein

Syntax-Elementbeschreibung:

operand1	Mit *operand1* geben Sie das Feld an, das überprüft werden soll. Eine bestimmte Ausprägung eines Arrays kann auch als ein Kontrollfeld benutzt werden.
lnl	Soll nur ein Teil des Feldes überprüft werden, so geben Sie eine mit Schrägstrichen eingegrenzte Zahl *n* an, die angibt, wieviele Stellen (von links nach rechts gezählt) des Feldes auf einen Wertwechsel überprüft werden sollen. Die Notation */n/* kann nur bei Operanden des Formats A, B, N oder P benutzt werden. Eine BREAK-Bedingung ist erfüllt, wenn der Wert des Kontrollfeldes (bzw. der angegebenen Stellen des Feldes) sich ändert. Eine BREAK-Bedingung ist nicht erfüllt, wenn eine AT END OF DATA-Bedingung auftritt. Beispiel: In diesem Beispiel wird überprüft, ob der Wert der ersten Stelle des Feldes FIRST-NAME sich geändert hat. `BREAK FIRST-NAME /1/` Der Einsatz von Natural-Systemfunktionen (die bei einem AT BREAK-Statement zur Verfügung stehen) ist bei der BREAK-Option nicht erlaubt.

Beispiel für BREAK-Option:

```
** Example 'LOGICX03': BREAK option in logical condition
************************************************************************
DEFINE DATA LOCAL
1 EMPLOY-VIEW VIEW OF EMPLOYEES
  2 NAME
  2 FIRST-NAME
  2 BIRTH
*
1 #BIRTH (A8)
END-DEFINE
*
LIMIT 10
READ EMPLOY-VIEW BY BIRTH
  MOVE EDITED BIRTH (EM=YYYYMMDD) TO #BIRTH
  /*
  IF BREAK OF #BIRTH /6/
    NEWPAGE IF LESS THAN 5 LINES LEFT
    WRITE / '-' (50) /
  END-IF
  /*
  DISPLAY NOTITLE BIRTH (EM=YYYY-MM-DD) NAME FIRST-NAME
END-READ
END
```

Ausgabe des Programms LOGICX03:

```
  DATE            NAME              FIRST-NAME
   OF
 BIRTH
---------- -------------------- --------------------

1940-01-01 GARRET               WILLIAM
1940-01-09 TAILOR               ROBERT
1940-01-09 PIETSCH              VENUS
1940-01-31 LYTTLETON            BETTY

-------------------------------------------------

1940-02-02 WINTRICH             MARIA
1940-02-13 KUNEY                MARY
1940-02-14 KOLENCE              MARSHA
1940-02-24 DILWORTH             TOM

-------------------------------------------------

1940-03-03 DEKKER               SYLVIA
1940-03-06 STEFFERUD            BILL
```

IS-Option - Prüfen ob Inhalt von Alphanumerischem oder Unicode-Feld konvertiert werden kann

Syntax:

operand1 IS (*format*)

Mit dieser Option können Sie prüfen, ob der Inhalt eines alphanumerischen oder Unicode-Feldes (*operand1*) in ein bestimmtes anderes Format übertragbar ist.

Operanden-Definitionstabelle:

Operand	Mögliche Struktur						Mögliche Formate										Referenzierung erlaubt	Dynam. Definition
operand1	C	S	A		N		A	U									ja	nein

Dieses *Format* kann sein:

N*ll.ll*	Numerisch mit Länge *ll.ll*.
F*ll*	Gleitkomma (floating point) mit Länge *ll*.
D	Datum. Folgende Datumsformate sind möglich: *dd-mm-yy*, *dd-mm-yyyy*, *ddmmyyyy* (*dd* = Tag, *mm* = Monat, *yy* oder *yyyy* = Jahr). Die Abfolge der Tages-, Monats- und Jahreskomponenten sowie die Trennzeichen zwischen den Komponenten werden durch den Profilparameter DTFORM (der in der *Parameter-Referenz*-Dokumentation beschrieben ist) bestimmt.
T	Zeit (Time); entsprechend dem Standard-Zeitanzeigeformat.
P*ll.ll*	Gepackt numerisch mit Länge *ll.ll*.
I*ll*	Ganzzahl (integer) mit Länge *ll*.

Bei der Prüfung werden für den Wert in *operand1* vor- oder nachgestellte Leerzeichen ignoriert.

Die Prüfung mit der IS-Option ist sinnvoll, wenn beispielsweise vor Ausführung der mathematischen Funktion VAL (Erhalt des numerischen Wertes eines alphanumerischen Feldes) das Format des Wertes überprüft wird, um zu vermeiden, dass ein falsches Format einen Laufzeitfehler verursacht.

Anmerkung: Mit der IS-Option kann nicht geprüft werden, ob der Wert eines alphanumerischen Feldes in dem angegebenen Format ist, sondern ob er in das Format *übertragbar* ist. Um zu prüfen, ob ein Wert in einem bestimmten Format ist, können Sie die ***MASK-Option*** verwenden.

Beispiel für IS-Option:

```
** Example 'LOGICX04': IS option as format/length check
************************************************************************
DEFINE DATA LOCAL
1 #FIELDA (A10)          /* INPUT FIELD TO BE CHECKED
1 #FIELDB (N5)           /* RECEIVING FIELD OF VAL FUNCTION
1 #DATE   (A10)          /* INPUT FIELD FOR DATE
END-DEFINE
*
INPUT #DATE #FIELDA
IF #DATE IS(D)
  IF #FIELDA IS (N5)
    COMPUTE #FIELDB = VAL(#FIELDA)
    WRITE NOTITLE 'VAL FUNCTION OK' //  '=' #FIELDA '=' #FIELDB
  ELSE
    REINPUT 'FIELD DOES NOT FIT INTO N5 FORMAT'
            MARK *#FIELDA
  END-IF
```

```
ELSE
  REINPUT 'INPUT IS NOT IN DATE FORMAT (YY-MM-DD) '
          MARK *#DATE
END-IF
*
END
```

Ausgabe des Programms `LOGICX04`:

```
#DATE 150487    #FIELDA









INPUT IS NOT IN DATE FORMAT (YY-MM-DD)
```

Weitere Informationen siehe *MASK-Option im Vergleich zur IS-Option*.

MASK-Option - Ausgewählte Stellen eines Feldes auf bestimmten Inhalt prüfen

Mit der `MASK`-Option können Sie bestimmte ausgewählte Stellen eines Feldes nach einem bestimmten Wert absuchen.

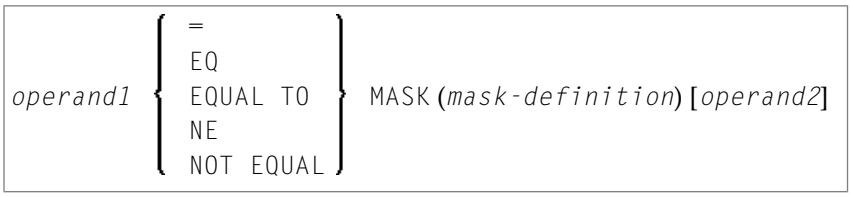
Folgende Themen werden behandelt:

- Konstante Maske
- Variable Maske
- Zeichen in einer Maske
- Maskenlänge
- Datumsprüfungen
- Prüfung unter Verwendung von Konstanten oder Variablen
- Bereichsprüfungen

- Auf gepackte oder ungepackte numerische Daten abprüfen

Konstante Maske

Syntax:



Operanden-Definitionstabelle:

Operand	Mögliche Struktur						Mögliche Formate												Referenzierung erlaubt	Dynam. Definition
operand1	C	S	A		N		A	U	N	P									ja	nein
operand2	C	S					A	U	N	P			B						ja	nein

Operand2 kann nur angegeben werden, wenn die *mask-definition* mindestens ein `X` enthält. *Operand1* und *operand2* müssen formatkompatibel sein:

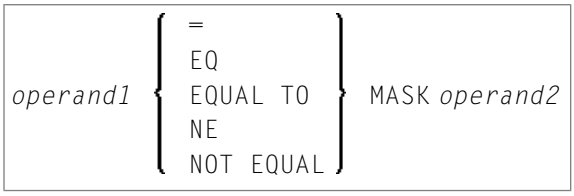
- wenn *operand1* Format A hat, muss *operand2* Format A, B, N oder U haben
- wenn *operand1* Format U hat, muss *operand2* Format A, B, N oder U haben
- wenn *operand1* Format N oder P hat, muss *operand2* Format N oder P haben.

Wird ein `X` in der *mask-definition* angegeben, werden die betreffenden inhaltlichen Stellen von *operand1* und *operand2* zum Wertevergleich ausgewählt.

Variable Maske

Anstatt einer konstanten *mask-definition* (siehe oben) können Sie auch eine variable `MASK`-Definition angeben:

Syntax:



Operanden-Definitionstabelle:

Operand	Mögliche Struktur						Mögliche Formate											Referenzierung erlaubt	Dynam. Definition
operand1	C	S	A		N		A	U	N	P								yes	no
operand2		S					A	U										yes	no

Der Inhalt von *operand2* wird dann als MASK-Definition genommen. Nachgestellte Leerzeichen in *operand2* werden ignoriert.

- Wenn *operand1* Format A, N oder P hat, muss *operand2* Format A haben.
- Wenn *operand1* Format U hat, muss *operand2* Format A haben.

Zeichen in einer Maske

In einer *mask-definition* können Sie folgende Zeichen verwenden (die Masken-Definition ist bei einer konstanten Maske in der *mask-definition* und bei einer variablen Maske in *operand2* enthalten):

Zeichen	Bedeutung
. oder ? oder _	Ein Punkt (.) oder Fragezeichen (?) oder ein Unterstrich (_) markiert eine einzelne Stelle, die nicht überprüft werden soll.
* oder %	Eine beliebige Anzahl von Stellen, die nicht überprüft werden sollen.
/	(Schrägstrich) Prüft, ob der Wert mit einem (oder mehreren) bestimmten Zeichen endet. Beispiel: Die folgende Bedingung ist wahr, wenn entweder an der letzten Stelle des Feldes ein E steht oder nach dem E nur noch Leerzeichen stehen: `IF #FIELD = MASK (*'E'/)`
A	Eine Stelle, die nach Groß- oder Kleinbuchstaben abgesucht werden soll.
'*c*'	Eine oder mehrere Stellen, die nach den in Apostrophen (') stehenden Zeichen abgesucht werden sollen (doppelte Apostrophe bedeuten, dass innerhalb der Zeichenkette nach einem Apostroph gesucht wird). Alphanumerische Zeichen mit hexadezimalen Ziffern kleiner als H'40' (Leerzeichen) sind nicht zulässig.
C	Eine Stelle, die nach alphanumerischem Inhalt (Groß- oder Kleinbuchstabe, Zahl) oder Leerzeichen abgesucht werden soll.
DD	Zwei Stellen, die nach einem gültigen Tagesdatum abgesucht werden sollen (01 - 31; abhängig von den Werten für MM und YY/YYYY, falls angegeben; siehe auch *Datumsprüfungen*).
H	Eine Stelle, die nach einem Hexadezimalzeichen (A - F, 0 - 9) abgesucht werden soll.
JJJ	Die Stellen sollen nach einem gültigen Julianischen Tag abgesucht werden, d.h. die Tageszahl im Jahr: 001 - 366, abhängig vom Wert von YY/YYYY, wenn angegeben; siehe auch *Datumsprüfungen*.
L	Eine Stelle, die nach Kleinbuchstaben (a - z) abgesucht werden soll.

Zeichen	Bedeutung
MM	Zwei Stellen, die nach einer gültigen Monatsangabe (01 – 12) abgesucht werden sollen; siehe auch *Datumsprüfungen*.
N	Eine Stelle, die nach einer Ziffer abgesucht werden soll.
n...	Eine oder mehrere Stellen, die nach einem numerischen Wert im Bereich von 0 bis *n* abgesucht werden sollen.
n1-*n2* oder *n1*:*n2*	Stellen, die nach einem numerischen Wert im Bereich von *n1*-*n2* abgesucht werden sollen. *n1* und *n2* müssen gleich lang sein.
P	Eine Stelle, die nach einem druckbaren Zeichen (U, L, N oder S – Buchstabe, Zahl oder Sonderzeichen) abgesucht werden soll.
S	Eine Stelle, die nach Sonderzeichen abgesucht werden soll.
U	Eine Stelle, die nach Großbuchstaben (A – Z) abgesucht werden soll.
X	Eine Stelle, die mit der entsprechenden Stelle des auf die *mask-definition* folgenden Wertes (*operand2*) verglichen werden soll. In einer variablen Maske ist X nicht erlaubt, da sinnlos.
YY	Zwei Stellen, die nach einer gültigen Jahreszahl (00 - 99) abgesucht werden sollen; siehe auch *Datumsprüfungen*.
YYYY	Vier Stellen, die nach einer gültigen Jahreszahl (0000-2699) abgesucht werden sollen. Benutzen Sie die COMPOPT-Option MASKCME=ON, um den Bereich der gültigen Jahre von 1582–2699 zu begrenzen; siehe auch *Datumsprüfungen*. Wenn der Profilparameter MAXYEAR auf 9999 gesetzt ist, betrtägt das obere Limit 9999.
Z	Eine Stelle, die nach einem Zeichen abgesucht werden soll, dessen linkes Halbbyte hexadezimal A – F und dessen rechtes Halbbyte hexadezimal 0 – 9 ist. Damit können Sie korrekt nach Ziffern in negativen Zahlen suchen. Mit N (womit Sie eine Stelle nach einer Ziffer absuchen können) erhalten Sie bei Suche von Ziffern in negativen Zahlen falsche Ergebnisse, da das Vorzeichen in der letzten Stelle der Zahl gespeichert ist, wodurch diese Stelle hexadezimal als Nicht-Ziffer dargestellt wird. Geben Sie innerhalb einer Maske für jede Reihe numerischer Stellen, die geprüft werden sollen, nur jeweils ein Z an.

Die Definition der Zeicheneigenschaften (z.B. Sonderzeichen) kann mit dem Profilparameter SCTAB geändert werden.

Wenn der Profilparameter CP nicht auf OFF, sondern auf einen anderen Wert gesetzt ist, leiten sich die Eigenschaften der Zeichen (zum Beispiel: alphabetisches Zeichen in Groß- oder Kleinschreibung) von den Zeicheneigenschaften der Codepage ab, die für die Natural-Session verwendet wird. Das hat zur Folge, dass beispielsweise die als alphabetische Zeichen erkannten Zeichen von den Standardzeichen a - z und A - Z abweichen können.

Maskenlänge

Welche Stellen abgesucht werden sollen, ergibt sich aus der Definition der Maske.

Beispiel:

```
DEFINE DATA LOCAL
1 #CODE (A15)
END-DEFINE
...
IF #CODE = MASK (NN'ABC'....NN)
...
```

Die ersten beiden Stellen von `#CODE` werden nach einem numerischen Wert abgesucht, die 3. bis 5. Stelle nach dem Wert `ABC`, die 10. und 11. Stelle nach einem numerischen Wert; die 6. bis 9. und 12. bis 15. Stelle werden nicht überprüft.

Datumsprüfungen

Pro Maske darf nur ein Datum geprüft werden. Wenn in der Maske derselbe Datumsbestandteil (`JJJ`, `DD`, `MM`, `YY` or `YYYY`) mehr als einmal angegeben wird, dann wird nur der Wert der letzten Ausprägung auf Konsistenz mit anderen Datumsbestandteilen geprüft.

Wird bei der Prüfung eines Tagesdatums (`DD`) keine Monatsangabe (`MM`) in der Maske gemacht, wird der aktuelle Monat angenommen.

Wird bei der Prüfung eines Tagesdatums (`DD`) oder Julianischen Tagesdatums (`JJJ`) keine Jahresangabe (`YY` bzw. `YYYY`) in der Maske gemacht, wird das aktuelle Jahr angenommen.

Bei der Prüfung einer zweistelligen Jahreszahl (`YY`) wird das aktuelle Jahrhundert angenommen, sofern kein Sliding Window oder Fixed Window gesetzt ist. Weitere Einzelheiten über Sliding oder Fixed Windows, entnehmen Sie dem Profilparameter `YSLW` in der *Parameter Reference*-Dokumentation.

Beispiel 1:

```
MOVE 1131 TO #DATE (N4)
IF #DATE = MASK (MMDD)
```

In diesem Beispiel werden Monat und Tag auf ihre Gültigkeit überprüft. Der Monatswert `11` ist gültig, wohingegen der Tageswert `31` ungültig ist, da der 11. Monat nur 30 Tage hat.

Beispiel 2:

```
IF #DATE(A8) = MASK (MM'/'DD'/'YY)
```

In diesem Beispiel wird überprüft, ob das Feld #DATE ein gültiges Datum im Format MM/DD/YY (Monat/Tag/Jahr) enthält.

Beispiel 3:

```
IF #DATE (A8) = MASK (1950-2020MMDD)
```

In diesem Beispiel wird der Inhalt des Feldes #DATE auf eine vierstellige Zahl im Bereich 1950 bis 2020 geprüft, auf die ein gültiger Monat und Tag im aktuellen Jahr folgen:

Anmerkung: Obwohl es so aussieht, ermöglicht die oben angegebene Maske nicht das Abprüfen auf ein gültiges Datum in den Jahren 1950 - 2020, weil der numerische Wertebereich 1950-2020 unabhängig von der Gültigkeitsprüfung für Monat und Tag abgeprüft wird. Die Prüfung liefert die beabsichtigten Ergebnisse mit Ausnahme des 29. Februars, denn an diesem Tag ist das Ergebnis davon abhängig, ob das aktuelle Jahr ein Schaltjahr ist oder nicht. Um zusätzlich zur Datumsgültigkeitsprüfung auf einen bestimmten Bereich von Jahren zu prüfen, bauen Sie eine Gültigkeitsprüfung für das Datum und eine andere für den Bereich in Ihrem Programm ein.

```
IF #DATE (A8) = MASK (YYYYMMDD) AND #DATE = MASK (1950-2020)
```

Beispiel 4:

```
IF #DATE (A4) = MASK (19-20YY)
```

In diesem Beispiel wird überprüft, ob das Feld #DATE eine zweistellige Zahl im Bereich von 19 bis 20, gefolgt von einem gültigen zweistelligen Jahr (00 bis 99) enthält. Das Jahrhundert wird wie oben beschrieben von Natural angegeben.

Anmerkung: Obwohl es so aussieht, ermöglicht die oben angegebene Maske nicht das Abprüfen auf ein gültiges Jahr im Bereich von 1900 bis 2099, weil der numerische Wertebereich 19 - 20 unabhängig von der Gültigkeitsprüfung für das Jahr abgeprüft wird. Um auf Bereiche von Jahren abzuprüfen, bauen Sie eine Gültigkeitsprüfung für das Datum und eine andere für den Bereich in Ihrem Programm ein:

```
IF #DATE (A10) = MASK (YYYY'-'MM'-'DD) AND #DATE = MASK (19-20)
```

Prüfung unter Verwendung von Konstanten oder Variablen

Ist der für die Maskenprüfung verwendete Wert eine Konstante oder der Inhalt einer Variablen, dann muss dieser Wert (*operand2*) unmittelbar nach der *mask-definition* angegeben werden.

operand2 muss mindestens so lang sein wie die Maske.

In der Maske geben Sie für jede zu überprüfende Stelle ein `X` und für jede nicht zu überprüfende Stelle einen Punkt (.) (oder `?` oder _) an.

Beispiel:

```
DEFINE DATA LOCAL
1 #NAME (A15)
END-DEFINE
...
IF #NAME = MASK (..XX) 'ABCD'
...
```

Hier wird geprüft, ob die 3. bis 4. Stelle des Feldes `#NAME` den Wert `CD` enthält. Die ersten beiden Stellen werden nicht überprüft.

Wieviele Stellen geprüft werden, hängt von der Länge der definierten Maske ab. Die Maske wird immer linksbündig auf das zu überprüfende Feld bzw. die zu prüfende Konstante ausgerichtet. *Operand1* muss dasselbe Format haben wie *operand2*.

Hat das zu überprüfende Feld (*operand1*) Format A, muss ein konstanter Wert (*operand2*) in Apostrophen stehen. Ist das Feld numerisch, muss der Wert eine Zahl oder der Inhalt eines numerischen Datenbankfeldes bzw. einer numerischen Benutzervariablen sein.

In jedem Fall werden Zeichen/Stellen, die nicht an einer in der Maskendefinition mit `X` markierten Stelle stehen, ignoriert.

Die `MASK`-Bedingung ist erfüllt, wenn alle in der Maske angegebenen Stellen dem geforderten Wert entsprechen.

Beispiel:

```
** Example 'LOGICX01': MASK option in logical condition
************************************************************************
DEFINE DATA LOCAL
1 EMPLOY-VIEW VIEW OF EMPLOYEES
  2 CITY
END-DEFINE
*
HISTOGRAM EMPLOY-VIEW CITY
  IF CITY =
MASK (....XX) '....NN'


    DISPLAY NOTITLE CITY *NUMBER
  END-IF
END-HISTOGRAM
*
END
```

In diesem Beispielprogramm werden nur Datensätze akzeptiert, bei denen das Feld `CITY` einen Wert enthält, der an der 5. und 6. Stelle jeweils ein `N` hat.

Bereichsprüfungen

Bei Bereichsprüfungen wird die Anzahl der verifizierten Stellen durch die Genauigkeit des in der Maske angegebenen Wertes definiert. Zum Beispiel würde die Maske `(...193...)` die Stellen 4 bis 6 nach einer dreistelligen Zahl im Bereich von `000` bis `193` überprüfen.

Weitere Beispiele für Masken-Definitionen:

- In diesem Beispiel wird überprüft, ob alle Stellen des Feldes `#NAME` einen Buchstaben enthalten:

  ```
  IF #NAME (A10) = MASK (AAAAAAAAAA)
  ```

- In diesem Beispiel wird überprüft, ob die 4. bis 6. Stelle von `#NUMBER` eine Zahl enthält:

  ```
  IF #NUMBER (A6) = MASK (...NNN)
  ```

- In diesem Beispiel wird überprüft, ob die 4. - 6. Stelle von `#VALUE` den Wert `123` enthält:

```
IF #VALUE(A10) = MASK (...'123')
```

- In diesem Beispiel wird überprüft, ob das Nummernschild-Feld #LICENSE ein KFZ-Kennzeichen enthält, das mit NY- beginnt, gefolgt vom Wert der letzten fünf Stellen des Feldes #VALUE:

```
DEFINE DATA LOCAL
1 #VALUE(A8)
1 #LICENSE(A8)
END-DEFINE
INPUT 'ENTER KNOWN POSITIONS OF LICENSE PLATE:' #VALUE
IF #LICENSE = MASK ('NY-'XXXXX) #VALUE
```

- Die folgende Bedingung wird von jedem Wert erfüllt, der NAT und AL enthält, ganz gleich wieviele andere Zeichen zwischen NAT und AL stehen (dies würde z.B. auf die Werte NATURAL und NATIONALITAET genauso zutreffen wie auf den Wert NATAL):

```
MASK('NAT'*'AL')
```

Auf gepackte oder ungepackte numerische Daten abprüfen

In Altanwendungen sind gepackte oder ungepackte numerische Variablen häufig mit alphanumerischen oder binären Feldern neu definiert. Solche Neudefinitionen sind nicht empfehlenswert, weil die Verwendung einer gepackten oder ungepackten Variablen in einer Zuweisung oder Berechnung zu Fehlern oder unvorhersagbaren Ergebnissen führen kann.

Um den Inhalt einer solchen neu definierten Variablen auf Gültigkeit hin abzuprüfen, bevor die Variable verwendet wird, benutzen Sie die Option N (siehe *Zeichen in einer Maske*) so oft wie die Anzahl der Stellen minus 1 mal, gefolgt von einer einzelnen Option Z.

Beispiele :

```
IF #P1 (P1) = MASK (Z)
IF #N4 (N4) = MASK (NNNZ)
IF #P5 (P5) = MASK (NNNNZ)
```

Weitere Informationen zum Abprüfen von Feldinhalten siehe *MASK-Option im Vergleich zur IS-Option*.

MASK-Option im Vergleich zur IS Option

Dieser Abschnitt beschreibt den Unterschied zwischen der MASK-Option und der IS-Option und enthält ein Beispielprogramm, das den Unterschied zwischen den beiden Optionen veranschaulicht.

Mit der IS-Option können Sie prüfen, ob der Inhalt eines alphanumerischen oder Unicode-Felds in ein bestimmtes anderes Format umgesetzt werden kann. Sie können mit dieser Option jedoch nicht abprüfen, ob der Wert eines alphanumerischen Feldes im angegeben Format vorliegt.

Mit der MASK-Option können Sie den Inhalt einer neu definierten gepackten oder ungepackten numerischen Variablen auf Gültigkeit abprüfen.

Beispiel zur Erläuterung des Unterschieds:

```
** Example 'LOGICX09': MASK versus IS option in logical condition
************************************************************************
DEFINE DATA LOCAL
1 #A2   (A2)
1 REDEFINE #A2
  2 #N2 (N2)
1 REDEFINE #A2
  2 #P3 (P3)
1 #CONV-N2 (N2)
1 #CONV-P3 (P3)
END-DEFINE
*
#A2 := '12'
WRITE NOTITLE   'Assignment #A2 := "12" results in:'
PERFORM SUBTEST
#A2 := '-1'
WRITE NOTITLE / 'Assignment #A2 := "-1" results in:'
PERFORM SUBTEST
#N2 :=  12
WRITE NOTITLE / 'Assignment #N2 :=  12  results in:'
PERFORM SUBTEST
#N2 :=  -1
WRITE NOTITLE / 'Assignment #N2 :=  -1  results in:'
PERFORM SUBTEST
#P3 :=  12
WRITE NOTITLE / 'Assignment #P3 :=  12  results in:'
PERFORM SUBTEST
#P3 :=  -1
WRITE NOTITLE / 'Assignment #P3 :=  -1  results in:'
PERFORM SUBTEST
*                                                                      ↩

DEFINE SUBROUTINE SUBTEST
IF #A2 IS (N2) THEN
```

```
  #CONV-N2 := VAL(#A2)
  WRITE NOTITLE 12T '#A2 can be converted to' #CONV-N2 '(N2)'
END-IF
IF #A2 IS (P3) THEN
  #CONV-P3 := VAL(#A2)
  WRITE NOTITLE 12T '#A2 can be converted to' #CONV-P3 '(P3)'
END-IF
IF #N2 = MASK(NZ) THEN
  WRITE NOTITLE 12T '#N2 contains the valid unpacked number' #N2
END-IF
IF #P3 = MASK(NNZ) THEN
  WRITE NOTITLE 12T '#P3 contains the valid packed number' #P3
END-IF
END-SUBROUTINE
*
END                                                                    ↩
```

Ausgabe des Programms `LOGICX09`:

```
Assignment #A2 := '12' results in:
           #A2 can be converted to  12 (N2)
           #A2 can be converted to   12 (P3)
           #N2 contains the valid unpacked number  12

Assignment #A2 := '-1' results in:
           #A2 can be converted to  -1 (N2)
           #A2 can be converted to   -1 (P3)

Assignment #N2 :=  12  results in:
           #A2 can be converted to  12 (N2)
           #A2 can be converted to   12 (P3)
           #N2 contains the valid unpacked number  12

Assignment #N2 :=  -1  results in:
           #N2 contains the valid unpacked number  -1

Assignment #P3 :=  12  results in:
           #P3 contains the valid packed number   12

Assignment #P3 :=  -1  results in:
           #P3 contains the valid packed number   -1
```

MODIFIED-Option - Prüfen ob Feldinhalt verändert worden ist

Syntax:

```
operand1 [NOT] MODIFIED
```

Mit dieser Option wird überprüft, ob der Inhalt eines Feldes während der Ausführung eines INPUT- oder PROCESS PAGE-Statements verändert worden ist. Als Voraussetzung muss dem Feld eine Kontrollvariable mit dem Parameter CV zugewiesen worden sein.

Operanden-Definitionstabelle:

Operand	Mögliche Struktur						Mögliche Formate										Referenzierung erlaubt	Dynam. Definition
operand1		S	A												C		nein	nein

Von einem INPUT- oder PROCESS PAGE-Statement referenzierte Kontrollvariablen erhalten immer den Status „not modified“ (nicht verändert), wenn die Map zur Ausgabe an das Terminal übertragen wird.

Wird der Inhalt eines Feldes, das eine Kontrollvariable (*operand1*) referenziert, verändert, erhält die Kontrollvariable den Status „modified“ (verändert). Referenzieren mehrere multiple Felder dieselbe Kontrollvariable, so erhält die Variable den Status MODIFIED, wenn mindestens eines dieser Felder verändert wurde.

Ist die Kontrollvariable *operand1* ein Array, so erhält die Variable den Status „modified“, wenn mindestens eins der Elemente des Arrays verändert wurde (ODER-Verknüpfung).

Mit dem Profilparameter CVMIN (siehe *Parameter-Referenz*-Dokumentation) kann festgelegt werden, ob eine Kontrollvariable auch dann auf „modified“ gesetzt werden soll, wenn der Wert des betreffenden Feldes durch einen *identischen* Wert überschrieben wurde.

Beispiel für MODIFIED-Option:

```
** Example 'LOGICX06': MODIFIED option in logical condition
************************************************************************
DEFINE DATA LOCAL
1 #ATTR (C)
1 #A    (A1)
1 #B    (A1)
END-DEFINE
*
MOVE (AD=I) TO #ATTR
*
```

```
INPUT (CV=#ATTR) #A #B
IF #ATTR NOT MODIFIED
  WRITE NOTITLE 'FIELD #A OR #B HAS NOT BEEN MODIFIED'
END-IF
*
IF #ATTR MODIFIED
  WRITE NOTITLE 'FIELD #A OR #B HAS BEEN MODIFIED'
END-IF
*
END
```

Ausgabe des Programms `LOGICX06`:

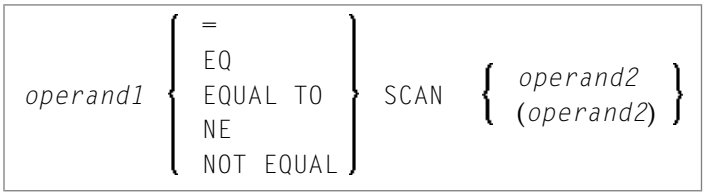
```
#A   #B
```

Nach Eingabe eines Wertes und Drücken der Freigabetaste, wird die folgende Ausgabe angezeigt:

```
FIELD #A OR #B HAS BEEN MODIFIED
```

SCAN-Option - Nach einem bestimmten Wert in einem Feld suchen

Syntax:



Operanden-Definitionstabelle:

Operand	Mögliche Struktur						Mögliche Formate											Referenzierung erlaubt	Dynam. Definition
operand1	C	S	A		N		A	U	N	P								ja	nein
operand2	C	S					A	U				B*						ja	nein

* *Operand2* darf nur binär sein, wenn *operand1* alphanumerisch oder Unicode ist. Wenn *operand1* Format U hat und *operand2* Format B, dann muss die Länge von *operand2* geradzahlig sein.

Mit der `SCAN`-Option können Sie nach einem bestimmten Wert in einem Feld suchen.

Der zu suchende Wert (*operand2*) kann entweder als alphanumerische oder Unicode-Konstante (eine in Apostrophen stehende Zeichenkette) oder als Inhalt einer alphanumerischen oder Unicode-Variablen (Datenbankfeld oder Benutzervariable) angegeben werden.

Vorsicht: Dem Wert nachgestellte Leerzeichen in *operand1* werden automatisch eliminiert. Deshalb kann die SCAN-Option nicht zum Suchen nach Leerzeichen verwendet werden. *operand1* und *operand2* dürfen vorangestellte oder eingebettete Leerzeichen enthalten. Falls *operand2* nur aus Leerzeichen besteht, dann gilt, unabhängig vom Wert in *operand1*, die Suche immer als erfolgreich; vergleiche EXAMINE FULL-Statement, wenn vorangestellte Leerzeichen bei der Suche nicht ignoriert werden sollen.

Das Feld, das abgesucht werden soll (*operand1*), kann das Format A, N, P oder U haben. Die SCAN-Operation kann mit den Operatoren EQ (gleich) oder NE (ungleich) angegeben werden.

Die Länge der gesuchten Zeichenkette sollte kürzer als die Länge des abgesuchten Feldes sein. Ist die Länge des Wertes gleich der des Feldes, sollte statt der SCAN-Option ein relationaler Ausdruck mit dem Operator EQUAL TO verwendet werden.

Beispiel für SCAN-Option:

```
** Example 'LOGICX02': SCAN option in logical condition
************************************************************************
DEFINE DATA
LOCAL
1 EMPLOY-VIEW VIEW OF EMPLOYEES
  2 NAME
*
1 #VALUE   (A4)
1 #COMMENT (A10)
END-DEFINE
*
INPUT 'ENTER SCAN VALUE:' #VALUE
LIMIT 15
*
HISTOGRAM EMPLOY-VIEW FOR NAME
  RESET #COMMENT
  IF NAME = SCAN #VALUE
    MOVE 'MATCH' TO #COMMENT
  END-IF
  DISPLAY NOTITLE NAME *NUMBER #COMMENT
END-HISTOGRAM
*
END
```

Ausgabe des Programms LOGICX02:

```
ENTER SCAN VALUE:
```

Eine Suche nach `LL` führt zu drei Übereinstimmungen bei 15 Namen:

```
        NAME           NMBR      #COMMENT
-------------------- --------- ----------

ABELLAN                      1 MATCH
ACHIESON                     1
ADAM                         1
ADKINSON                     8
AECKERLE                     1
AFANASSIEV                   2
AHL                          1
AKROYD                       1
ALEMAN                       1
ALESTIA                      1
ALEXANDER                    5
ALLEGRE                      1 MATCH
ALLSOP                       1 MATCH
ALTINOK                      1
ALVAREZ                      1
```

SPECIFIED-Option - Prüfen ob ein Wert für einen optionalen Parameter übergeben wird

Syntax:

parameter-name [NOT] SPECIFIED

Mit dieser Option wird überprüft, ob ein optionaler Parameter in einem aufgerufenen Objekt (Subprogramm oder externe Subroutine) vom aufrufenden Objekt einen Wert aufgenommen hat oder nicht.

Ein optionaler Parameter ist ein Feld, das mit dem Schlüsselwort `OPTIONAL` im `DEFINE DATA PARAMETER`-Statement des aufgerufenen Objekts definiert worden ist. Wenn ein Feld als `OPTIONAL` definiert wird, kann ein Wert von einem aufrufenden Objekt an dieses Feld übergeben werden.

Im aufrufenden Statement wird die Notation *n*`X` benutzt, um Parameter anzugeben, für die keine Werte übergeben werden.

Wenn Sie einen optionalen Parameter verarbeiten, der keinen Wert empfangen hat, so führt dies zu einem Laufzeit-Fehler. Um solch einen Fehler zu vermeiden, benutzen Sie die

SPECIFIED-Option im aufgerufenen Objekt, um zu überprüfen, ob ein optionaler Parameter einen Wert aufgenommen hat oder nicht, und ihn erst dann zu verarbeiten, wenn dies der Fall ist.

parameter-name ist der Name des im DEFINE DATA PARAMETER-Statement des aufgerufenen Objekts angegebenen Parameters.

Bei einem nicht als OPTIONAL definierten Feld ist die SPECIFIED-Bedingung immer TRUE.

Beispiel für die SPECIFIED-Option:

Aufrufendes Programm:

```
** Example 'LOGICX07': SPECIFIED option in logical condition
************************************************************************
DEFINE DATA LOCAL
1 #PARM1 (A3)
1 #PARM3 (N2)
END-DEFINE
*
#PARM1 := 'ABC'
#PARM3 := 20
*
CALLNAT 'LOGICX08' #PARM1 1X #PARM3
*
END
```

Aufgerufenes Subprogramm:

```
** Example 'LOGICX08': SPECIFIED option in logical condition
************************************************************************
DEFINE DATA PARAMETER
1 #PARM1 (A3)
1 #PARM2 (N2) OPTIONAL
1 #PARM3 (N2) OPTIONAL
END-DEFINE
*
WRITE '=' #PARM1
*
IF #PARM2 SPECIFIED
  WRITE '#PARM2 is specified'
  WRITE '=' #PARM2
ELSE
  WRITE '#PARM2 is not specified'
* WRITE '=' #PARM2                /* would cause runtime error NAT1322
END-IF
*
IF #PARM3 NOT SPECIFIED
  WRITE '#PARM3 is not specified'
ELSE
```

```
  WRITE '#PARM3 is specified'
  WRITE '=' #PARM3
END-IF
END
```

Ausgabe des Programms `LOGICX07`:

```
Page      1                                                  04-12-15  11:25:41

#PARM1: ABC
#PARM2 is not specified
#PARM3 is specified
#PARM3:  20
```

52 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.

Verwendung von Verarbeitungsschleifen

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* 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.

Schleifendurchläufe bei Datenbankzugriffen begrenzen

Die folgenden Themen werden behandelt:

- Möglichkeiten der Begrenzung von Datenbankschleifen
- LT-Session-Parameter
- LIMIT-Statement
- Limit-Notation

Möglichkeiten der Begrenzung von Datenbankschleifen

Bei den Statements READ, FIND oder 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- oder 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.

Priorität der Limit-Einstellungen

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. Sie werden mit einem der folgenden Statements erzeugt: `REPEAT`, `FOR`, `CALL FILE`, `CALL LOOP`, `SORT` und `READ WORK FILE`.

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 `REPEAT`-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 der folgenden Statements verlassen werden:

- `ESCAPE` (siehe **nächsten Abschnitt**) beendet die Verarbeitung der Schleife und setzt die Verarbeitung außerhalb der Schleife fort.
- `STOP` bricht die Ausführung der gesamten Natural-Anwendung ab.
- `TERMINATE` bricht die Ausführung der Natural-Anwendung ab und beendet die Natural-Session.

Beispiel für Verarbeitungsschleife mit REPEAT-Statement

```
** Example 'REPEAX01': REPEAT
************************************************************************
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
```

Ausgabe des Programms `REPEAX01`:

```
Page      1                                                 04-11-11  14:15:54

        NAME           ANNUAL     #PAY1
                       SALARY
-------------------- ---------- ---------

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 in der *Statements*-Dokumentation.

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.

Beispiel für geschachtelte FIND-Statements

Das folgende Programm zeigt eine Hierarchie zweier Verarbeitungsschleifen, wobei sich eine FIND-Schleife innerhalb einer anderen FIND-Schleife befindet.

```
** Example 'FINDX06': FIND (two FIND statements nested)
************************************************************************
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.

Ausgabe des Programms FINDX06:

```
        NAME                 CITY                 MAKE
-------------------- -------------------- --------------------

RUBIN                NEW YORK             FORD
OLLE                 BEVERLEY HILLS       GENERAL MOTORS
WALLACE              NEW YORK             MAZDA
JONES                BEVERLEY HILLS       FORD
SPEISER              BEVERLEY HILLS       GENERAL MOTORS
```

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,
- Naturals **Standard-Referenzierung** (die bei jedem betroffenen Statement in der Dokumentation beschrieben ist) aufzuheben
- oder zu Programmdokumentationszwecken.

Sie können jedes Natural-Statement referenzieren, das eine Verarbeitungsschleife initiiert und/oder auf Datenelemente in einer Datenbank zugreift:

- `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 in der Statements-Dokumentation in den *Operandentabellen* der einzelnen Statements aus der Spalte *Referenzierung erlaubt*. Siehe auch ***Benutzervariablen***, *Datenbankfelder mit der `(r)`-Notation referenzieren*.

Außerdem kann eine Referenzierungsnotation in einigen Statements angegeben werden, z.B. bei:

- `AT START OF DATA`
- `AT END OF DATA`
- `AT BREAK`
- `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 *Quellcode-Zeilennummer* verwenden.

- **Statement-Label**
  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 in der *Statements*-Dokumentation) eingefügt, zum Beispiel:

```
AT BREAK (READ1.) OF NAME
```

- **Quellcode-Zeilennummern**
  Wenn Sie Quellcode-Zeilennummern 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 Statement ein bestimmtes Feld auf ein vorhergegangenes Statement, so wird das Label bzw. die Zeilennummer in Klammern hinter dem jeweiligen Feldnamen angegeben, zum Beispiel:

```
DISPLAY NAME (READ1.) JOB-TITLE (READ1.) MAKE MODEL
```

Quellcode-Zeilennummern und Statement-Labels können wahlweise verwendet werden.

Siehe auch *Benutzervariablen*, *Datenbankfelder mit der (r)- Notation referenzieren*.

Beispiel für das Referenzieren mit Zeilennummern

Das folgende Programm verwendet Quellcode-Zeilennummern (vierstellige Ziffern in Klammern) zur Referenzierung.

In diesem Beispiel beziehen sich die Zeilennummern auf Statements, die aufgrund der Programmstruktur ohnehin, auch ohne explizite Referenzierung, referenziert worden wären.

```
0010 ** Example 'LABELX01': Labels for READ and FIND loops (line numbers)
0020 ************************************************************************
0030 DEFINE DATA LOCAL
0040 1 MYVIEW1 VIEW OF EMPLOYEES
0050   2 NAME
0060  2 FIRST-NAME
0070  2 PERSONNEL-ID
0080 1 MYVIEW2 VIEW OF VEHICLES
0090  2 PERSONNEL-ID
0100  2 MAKE
0110 END-DEFINE
0120 *
0130 LIMIT 15
0140 READ MYVIEW1 BY NAME STARTING FROM 'JONES'
0150  FIND MYVIEW2 WITH PERSONNEL-ID = PERSONNEL-ID (0140)
0160    IF NO RECORDS FOUND
0170      MOVE '***NO CAR***' TO MAKE
0180    END-NOREC
0190    DISPLAY NOTITLE NAME       (0140) (IS=ON)
0200                    FIRST-NAME (0140) (IS=ON)
0210                    MAKE       (0150)
0220  END-FIND /* (0150)
0230 END-READ   /* (0140)
0240 END
```

Beispiel mit Statement-Labels

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.

```
** Example 'LABELX02': Labels for READ and FIND loops (user labels)
************************************************************************
DEFINE DATA LOCAL
1 MYVIEW1 VIEW OF EMPLOYEES
  2 NAME
  2 FIRST-NAME
  2 PERSONNEL-ID
1 MYVIEW2 VIEW OF VEHICLES
  2 PERSONNEL-ID
  2 MAKE
END-DEFINE
*
LIMIT 15
RD. READ MYVIEW1 BY NAME STARTING FROM 'JONES'
  FD. FIND MYVIEW2 WITH PERSONNEL-ID = PERSONNEL-ID (RD.)
    IF NO RECORDS FOUND
      MOVE '***NO CAR***' TO MAKE
    END-NOREC
    DISPLAY NOTITLE NAME       (RD.) (IS=ON)
                    FIRST-NAME (RD.) (IS=ON)
                    MAKE       (FD.)
  END-FIND /* (FD.)
END-READ   /* (RD.)
END ↩
```

Beide Programme erzeugen folgende Ausgabe:

```
        NAME              FIRST-NAME              MAKE
-------------------- -------------------- --------------------

JONES                VIRGINIA             CHRYSLER
                     MARSHA               CHRYSLER
                                          CHRYSLER
                     ROBERT               GENERAL MOTORS
                     LILLY                FORD
                                          MG
                     EDWARD               GENERAL MOTORS
                     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***
KANT                 HEIKE                ***NO CAR***
```

53 Gruppenwechsel

Dieses Kapitel beschreibt, wie die Ausführung eines Statements von einem Gruppenwechsel abhängig gemacht werden kann, und wie Gruppenwechsel für die Auswertung von Natural-Systemfunktionen benutzt werden können.

Verwendung von Gruppenwechseln

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 werden im Abschnitt ***Systemvariablen und Systemfunktionen*** behandelt. Genauere Beschreibungen der verfügbaren Systemfunktionen finden Sie in der *Systemfunktionen*-Dokumentation.

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.

In diesem Abschnitt werden folgende Themen behandelt:

- Gruppenwechsel basierend auf einem Datenbankfeld
- Gruppenwechsel basierend auf einer Benutzervariablen
- Gruppenwechsel auf mehreren Ebenen

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:

```
** Example 'ATBREX01': AT BREAK OF (with database field)
************************************************************************
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^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 `ATBREX01` erzeugt folgende Ausgabe:

```
Page      1                                              04-12-14  14:07:26

  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.

```
** Example 'ATBREX02': AT BREAK OF  (with user-defined variable and
**                     in conjunction with BEFORE BREAK PROCESSING)
************************************************************************
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
```

Ausgabe des Programms `ATBREX02`:

```
Page      1                                                 04-12-14  14:08:36

     #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 **oben** 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/).

```
** Example 'ATBREX03': AT BREAK OF (two statements in combination)
************************************************************************
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
```

Ausgabe des Programms ATBREX03:

```
Page      1                                                  04-12-14  14:09:20

DEPARTMENT         NAME                 POSITION
   CODE
---------- -------------------- -------------------------

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.

```
** Example 'ATBREX04': AT BREAK OF (two statements in combination)
************************************************************************
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
```

Ausgabe des Programms `ATBREX04`:

```
           ** 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

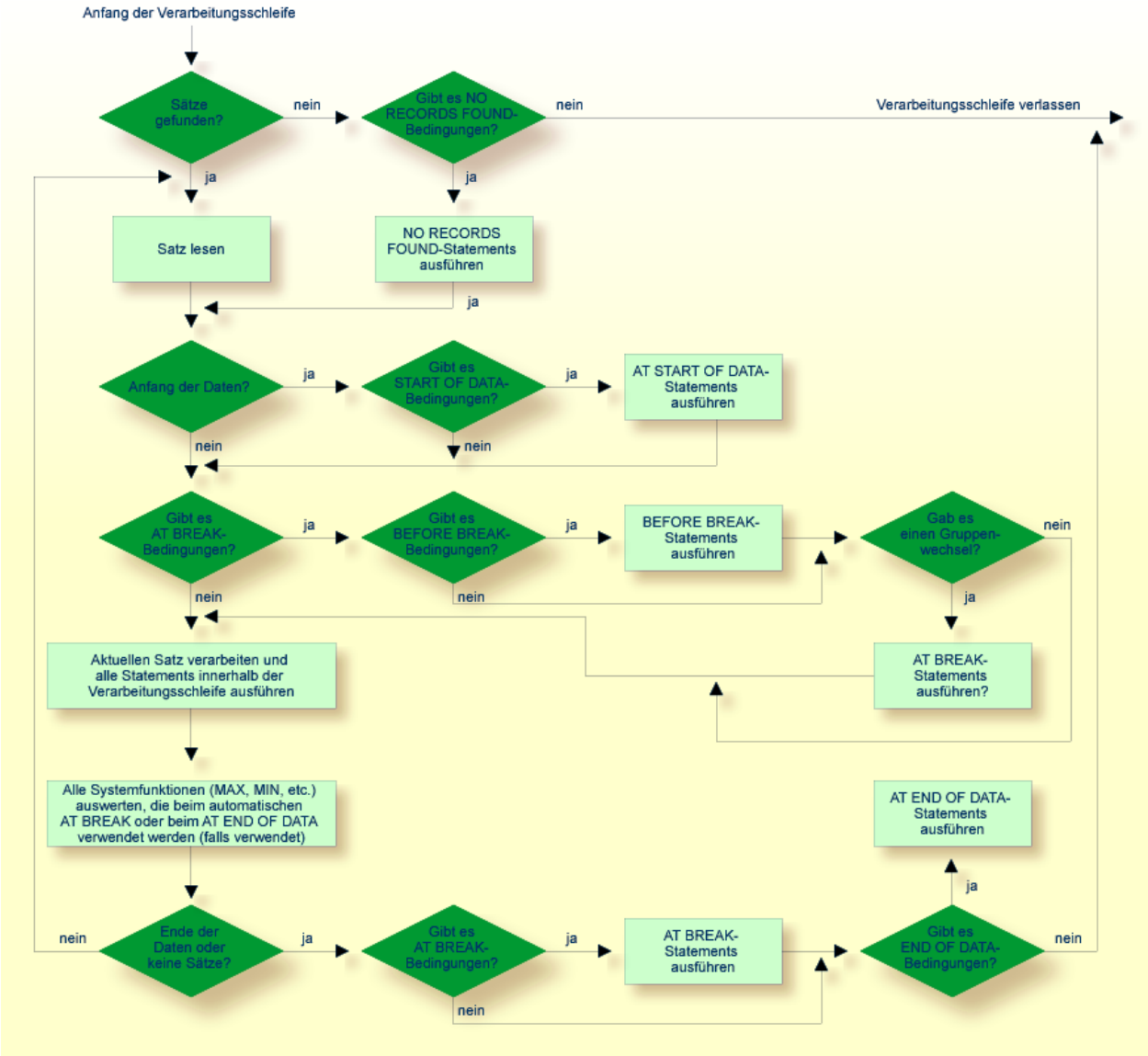
Automatische Gruppenwechsel-Verarbeitung ist für eine Verarbeitungsschleife aktiv, die ein `AT BREAK`-Statement enthält. Dies gilt für die folgenden Statements:

- `FIND`
- `READ`
- `HISTOGRAM`
- `SORT`
- `READ WORK FILE`

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.



Beispiel für Systemfunktionen in einem AT BREAK-Statement

Das folgende Beispiel zeigt die Verwendung der Natural-Systemfunktionen `OLD`, `MIN`, `AVER`, `MAX`, `SUM` und `COUNT` in einem `AT BREAK`-Statement (und der Systemfunktion `TOTAL` in einem `AT END OF DATA`-Statement).

```
** Example 'ATBREX05': AT BREAK OF (with system functions)
************************************************************************
DEFINE DATA LOCAL
1 MYVIEW VIEW OF EMPLOYEES
  2 NAME
  2 CITY
  2 SALARY    (1:1)
  2 CURR-CODE (1:1)
END-DEFINE
*
LIMIT 3
READ MYVIEW BY CITY = 'SALT LAKE CITY'
  DISPLAY NOTITLE CITY NAME 'SALARY' SALARY(1) 'CURRENCY' CURR-CODE(1)
  /*
  AT BREAK OF CITY
    WRITE /  OLD(CITY) (EM=X^X^X^X^X^X^X^X^X^X^X^X^X^X^X^X^X^X)
      31T ' - MINIMUM:' MIN(SALARY(1))  CURR-CODE(1) /
      31T ' - AVERAGE:' AVER(SALARY(1)) CURR-CODE(1) /
      31T ' - MAXIMUM:' MAX(SALARY(1))  CURR-CODE(1) /
      31T ' -     SUM:' SUM(SALARY(1))  CURR-CODE(1) /
      33T COUNT(SALARY(1)) 'RECORDS FOUND' /
  END-BREAK
  /*
  AT END OF DATA
    WRITE 22T 'TOTAL (ALL RECORDS):'
              T*SALARY TOTAL(SALARY(1)) CURR-CODE(1)
  END-ENDDATA
END-READ
END
```

Ausgabe des Programms `ATBREX05`:

```
        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
```

Weiteres Beispiel für AT BREAK-Statement

Siehe folgendes Beispielprogramm:

- *ATBREX06 - AT BREAK OF-Statement (zum Vergleichen von NMIN, NAVER, NCOUNT mit MIN, AVER, COUNT)*

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 Natural-Systemfunktionen ausgewertet werden.

Beispiel für BEFORE BREAK PROCESSING-Statement

```
** Example 'BEFORX01': BEFORE BREAK PROCESSING
************************************************************************
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
```

Ausgabe des Programms BEFORX01:

```
        NAME          FIRST-NAME    ANNUAL      SALARY
                                    INCOME      + BONUS
-------------------- ---------- ------------ -----------

BACHMANN             HANS              56800 =     52800
                                                   +4000
                                 ------------------------
BAECKER              JOHANNES          81000 =     74400
                                                   +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, und zwar 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.

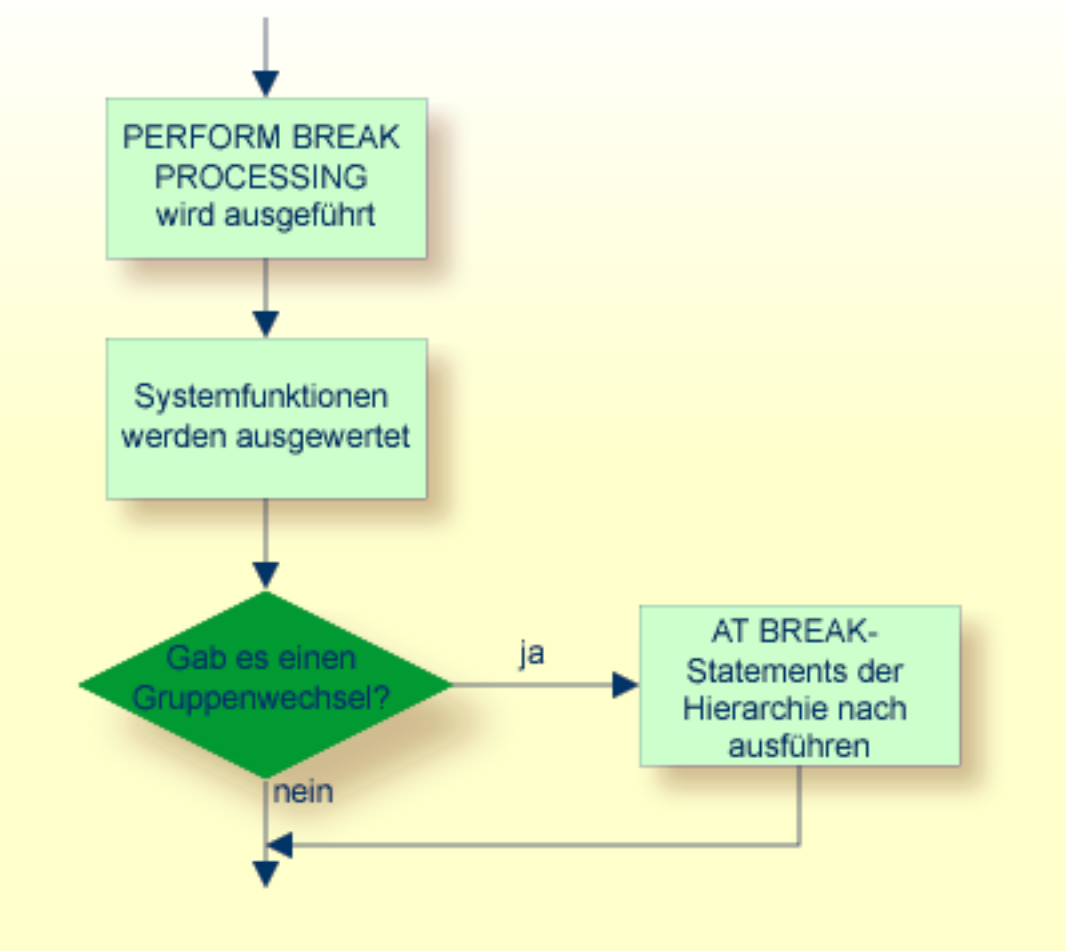
Unmittelbar nach dem PERFORM BREAK PROCESSING-Statement geben Sie einen oder mehrere AT BREAK-Statement-Blöcke an:

```
...
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.

Die folgende Abbildung zeigt den logischen Ablauf einer programmabhängigen Gruppenwechsel-Verarbeitung:



Beispiel für PERFORM BREAK PROCESSING-Statement

```
** Example 'PERFBX01': PERFORM BREAK PROCESSING (with BREAK option
**                      in IF statement)
************************************************************************
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              /* <- automatic break processing
    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    /* <- user-initiated break processing
    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    /* <- user-initiated break processing
    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
```

Ausgabe des Programms `PERFBX01`:

```
Page      1                                              04-12-14  14:13:35

        NAME         DEPARTMENT   ANNUAL
                        CODE      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
```

54 Natural-Stack

Der Natural-Stack ist eine Art „Zwischenablage“, in der Sie Natural-Kommandos, Benutzerkommandos und Daten für ein INPUT-Statement speichern können.

Verwendung des Natural-Stack

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 als auch unten im 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 *Systemvariablen*-Dokumentation).

Reihenfolge bei Verarbeitung von im Stack gespeicherten Kommandos und Daten

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. Siehe auch *INPUT-Daten aus dem Natural-Stack* in der Beschreibung des INPUT-Statements.

Falls ein INPUT-Statement mit Stack-Daten ausgeführt wird und dieses INPUT-Statement durch ein REINPUT-Statement nochmals ausgeführt wird, wird der INPUT-Schirm angezeigt, und zwar mit den gleichen Stack-Daten wie beim ersten Mal. 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
- STACK-Statement
- FETCH- und RUN-Statements

STACK-Parameter

Sie können den Natural-Profilparameter `STACK` benutzen, um Daten oder Kommandos im Stack abzulegen. Der `STACK`-Parameter, der in der Natural *Parameter-Referenz*-Dokumentation beschrieben ist, kann bei der Installation von Natural vom Natural-Administrator im Natural-Parametermodul 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. Falls die im Stack abzulegenden Daten Begrenzungs-, Steuer- oder DBCS-Zeichen enthalten, sollten sie, um eine unbeabsichtigte Interpretation dieser Zeichen zu vermeiden, formatiert im Stack abgelegt werden.

Eine ausführliche Beschreibung des Statements `STACK` finden Sie in der *Statements*-Dokumentation.

FETCH- und RUN-Statements

Werden bei der Ausführung eines `FETCH`- oder `RUN`-Statements Parameter an das aufgerufene Programm übergeben, so werden diese Parameter oben auf dem Natural-Stack abgelegt.

Ersten Eintrag vom Natural-Stack löschen

Mit dem Natural-Terminalkommando `%.P` löschen Sie den obersten Eintrag vom Natural-Stack.

Stack-Inhalt löschen

Der Inhalt des Natural-Stack kann mit dem Statement `RELEASE` gelöscht werden. Eine ausführliche Beschreibung des Statements finden Sie in der *Statements*-Dokumentation.

Anmerkung: Wenn ein Natural-Programm mittels des Terminalkommandos `%%` oder mit einem Fehler beendet wird, wird der Stack vollständig gelöscht.

55 Systemvariablen und Systemfunktionen

Dieses Kapitel beschreibt den Sinn und Zweck von Natural-Systemvariablen und Natural-Systemfunktionen, und wie sie in Natural-Programmen benutzt werden.

Systemvariablen

Folgende Themen werden behandelt:

Sinn und Zweck von Systemvariablen

Systemvariablen werden zur Anzeige von System-Informationen benutzt. Sie können an einem beliebigen Punkt in einem Natural-Programm referenziert werden.

Natural-Systemvariablen liefern Variablen-Informationen, z.B. zur aktuellen Natural-Session:

- die aktuelle Library
- die Benutzer-ID und die Terminal-ID
- den aktuellen Status eines Schleifendurchlaufs
- den aktuellen Verarbeitungs-Status von Reports
- das aktuelle Datum und die aktuelle Uhrzeit

Die typische Benutzung von Systemvariablen wird im *Beispiel für Systemvariablen und Systemfunktionen* weiter unten und in den in der Library `SYSEXPG` enthaltenen Beispielen veranschaulicht.

Die in einer Systemvariable enthaltenen Informationen können in Natural-Programmen unter Angabe der betreffenden Systemvariablen benutzt werden. Beispielsweise können Datums- und Zeit-Systemvariablen in einem `DISPLAY`-, `WRITE`-, `PRINT`-, `MOVE`- oder `COMPUTE`-Statement angegeben werden.

Charakteristika von Systemvariablen

Die Namen der Systemvariablen beginnen mit einem Stern (*).

Format/Länge

Folgende Abkürzungen werden verwendet:

Format	
A	Alphanumerisch
B	Binär
D	Datum
I	Integer (Ganzzahl)
L	Logisch
N	Numerisch (ungepackt)
P	Numerisch (gepackt)
T	Zeit

Inhalt änderbar

In den einzelnen Beschreibungen verweist dies darauf, ob Sie in einem Natural-Programm der Systemvariablen einen anderen Wert zuweisen können, d.h. ihren von Natural generierten Inhalt überschreiben können.

Aufteilung von Systemvariablen nach Funktionen

Die Natural-Systemvariablen sind wie folgt unterteilt:

- Anwendungsbezogene Systemvariablen
- Datums- und Zeit-Systemvariablen
- Eingabe/Ausgabe-bezogene Systemvariablen
- Natural-Umgebungsbezogene Systemvariablen
- System-Umgebungsbezogene Systemvariablen
- XML-bezogene Systemvariablen

Ausführliche Beschreibungen aller Systemvariablen finden Sie in der *Systemvariablen* -Dokumentation.

Systemfunktionen

Natural-Systemfunktionen sind in Natural eingebaute 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** gezeigt).

Es gibt folgende nach Funktionen aufgeteilten Systemfunktionen:

- Systemfunktionen zur Verwendung in Verarbeitungsschleifen
- Mathematische Funktionen
- Verschiedene Funktionen

Weitere Informationen zu Systemfunktionen finden Sie in der *Systemfunktionen*-Dokumentation.

Siehe dort auch *Natural-Systemfunktionen für Verarbeitungsschleifen* in der *Systemfunktionen*-Dokumentation.

Die typische Benutzung von Systemfunktionen ist in den folgenden Beispielprogrammen und in den in der Library SYSEXPG enthaltenen Beispielen erläutert.

Beispiel für Systemvariablen und Systemfunktionen

Das folgende Beispielprogramm veranschaulicht die Verwendung von Systemvariablen und Systemfunktionen:

```
** Example 'SYSVAX01': System variables and system functions
************************************************************************
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 LEFT JUSTIFIED '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
```

Erläuterung:

- Die Systemvariable `*DATE` wird mit dem `WRITE TITLE`-Statement ausgegeben.
- Die Systemvariable `*TIME` wird mit dem `AT START OF DATA`-Statement ausgegeben.
- Die Systemfunktion `OLD` wird im `AT END OF DATA`-Statement benutzt.
- Die Systemfunktion `AVER` wird im `AT END OF PAGE`-Statement benutzt.

Ausgabe des Programms `SYSVAX01`:

Beachten Sie, wie die Systemvariablen und Systemfunktionen angezeigt werden:

```
EMPLOYEE SALARY REPORT AS OF 11/11/2004

     NAME           CURRENT                 INCOME
                   POSITION
                                  CURRENCY   ANNUAL     BONUS
                                    CODE     SALARY
--------------- --------------- -------- ---------- ----------

REPORT CREATED AT: 14:15:55.0 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 folgende Beispielprogramme:

- ***EDITMX05 - Editiermaske (EM-Angaben für Datums- und Uhrzeit-Systemvariablen)***
- ***READX04 – READ-Statement (in Kombination mit FIND und den Systemvariablen *NUMBER und *COUNTER)***
- ***WTITLX01 – WRITE TITLE-Statement (mit *PAGE–NUMBER)***

Weitere Beispiele für Systemfunktionen

Siehe folgende Beispielprogramme:

- *ATBREX06 - AT BREAK OF-Statement (zum Vergleichen von NMIN, NAVER, NCOUNT mit MIN, AVER, COUNT)*
- *ATENPX01 - AT END OF PAGE-Statement (mit der durch GIVE SYSTEM FUNCTIONS in DISPLAY verfügbaren Systemfunktion)*

56 Verarbeitung von Datumsinformationen

Dieses Kapitel behandelt verschiedene Aspekte der Behandlung von Datumsinformationen in Ihren Natural-Anwendungen.

Editiermasken für Datumsfelder und 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.

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 zweistellig, bei manchen vierstellig.

Weitere Informationen sowie eine Liste aller Datumssystemvariablen finden Sie in der *Systemvariablen*-Dokumentation.

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:

Setting	Datumsformat[1]	Beispiel[2]
`DTFORM=I`	*yyyy-mm-dd*	2014-01-31
`DTFORM=G`	*dd.mm.yyyy*	31.01.2014
`DTFORM=E`	*dd/mm/yyyy*	31/01/2014
`DTFORM=U`	*mm/dd/yyyy*	01/31/2014

[1] *dd* = day/Tag, *mm* = month/Monat, *yyyy* = year/Jahr
[2] Es wird angenommen, dass `DF` oder `DFTITLE` auf `L` gesetzt ist.

Der `DTFORM`-Parameter kann im Natural-Parametermodul oder dynamisch beim Aufruf von Natural gesetzt werden. Standardmäßig gilt `DTFORM=I`.

Datumsformat für alphanumerische Darstellung — der DF-Parameter

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 Jahreszahl und Begrenzungszeichen. Beispiel: *yy-mm-dd*
DF=I	8-Byte-Darstellung mit 4-stelliger Jahreszahl ohne Begrenzungszeichen. Beispiel: *yyyymmdd*
DF=L	10-Byte-Darstellung mit 4-stelliger Jahreszahl und Begrenzungszeichen. Beispiel: *yyyy-mm-dd*).

Bei jeder Darstellung wird die Reihenfolge von Tag, Monat und Jahr sowie die zu verwendenden Begrenzungszeichen durch den DTFORM-Parameter bestimmt.

Standardmäßig gilt DF=S (außer bei INPUT-Statements; siehe unten).

Der DF-Parameter wird bei der Kompilierung ausgewertet.

Der DF-Parameter gilt bei folgenden Statements und in folgenden Situationen:

- FORMAT
- INPUT, DISPLAY, WRITE und PRINT auf Statement- und Elementebene (Feldebene)
- MOVE, COMPRESS, STACK, RUN und FETCH auf Elementebene (Feldebene)

Bei Angabe in einem dieser Statements hat der DF-Parameter folgende Auswirkung:

Statement	Auswirkung des DF-Parameters
DISPLAY, WRITE, 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, 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.

Statement	Auswirkung des DF-Parameters
STACK, RUN, FETCH	Wenn der Wert einer Datumsvariablen auf dem Natural-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 Begrenzungszeichen oder mit 4-stelliger Jahresangabe ohne Begrenzungszeichen eingegeben werden. Auch in diesem Fall wird die Reihenfolge von Tag, Monat und Jahr sowie die zu verwendenden Begrenzungszeichen durch den DTFORM-Parameter bestimmt.

Anmerkung: Bei DF=S stehen nur zwei Stellen für die Jahresangabe zur Verfügung; d.h. selbst wenn der Datumswert das Jahrhundert enthielte, ginge diese Information bei der Umsetzung verloren. Um die Jahrhundertinformation zu behalten, müssen Sie DF=I oder DF=L setzen.

Beispiele für DF-Parameter beim WRITE-Statements

Diese Beispiele basieren auf der Annahme, dass DTFORM=G.

```
/* DF=S (default)
WRITE *DATX   /* Output has this format: dd.mm.yy
END
```

```
FORMAT DF=I
WRITE *DATX   /* Output has this format: ddmmyyyy
END
```

```
FORMAT DF=L
WRITE *DATX   /* Output has this format: dd.mm.yyyy
END
```

Beispiel für DF-Parameter beim MOVE-Statement

Dieses Beispiel basiert auf der Annahme, dass DTFORM=E.

```
DEFINE DATA LOCAL
  1 #DATE  (D) INIT <D'31/01/2014'>
  1 #ALPHA (A10)
END-DEFINE
...
MOVE #DATE        TO #ALPHA /* Result: #ALPHA contains 31/01/14
MOVE #DATE (DF=I) TO #ALPHA /* Result: #ALPHA contains 31012014
MOVE #DATE (DF=L) TO #ALPHA /* Result: #ALPHA contains 31/01/2014
...
```

Beispiel für DF-Parameter beim STACK-Statement

Dieses Beispiel basiert auf der Annahme, dass DTFORM=I.

```
DEFINE DATA LOCAL
  1 #DATE  (D) INIT <D'2014-01-31'>
  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
...
/* Result: #ALPHA1 contains 14-01-31
/*         #ALPHA2 contains 20140131
/*         #ALPHA3 contains 2014-01-31
...
```

Beispiel für DF-Parameter beim INPUT-Statement

Dieses Beispiel basiert auf der Annahme, dass DTFORM=I.

```
DEFINE DATA LOCAL
  1 #DATE1 (D)
  1 #DATE2 (D)
  1 #DATE3 (D)
  1 #DATE4 (D)
END-DEFINE
...
INPUT #DATE1 (DF=S) /* Input must have this format: yy-mm-dd
      #DATE2 (DF=I) /* Input must have this format: yyyymmdd
      #DATE3 (DF=L) /* Input must have this format: yyyy-mm-dd
```

```
      #DATE4        /* Input must have this format: yy-mm-dd or yyyymmdd
...
```

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.

Mit dem DFOUT-Parameter können Sie eine der folgenden Datumsdarstellungsarten wählen:

DFOUT=S	Datumsvariablen werden mit zweistelligem Jahr und mit durch den DTFORM-Parameter bestimmten Begrenzungszeichen angezeigt: Beispiel: *yy-mm-dd*
DFOUT=I	Datumsvariablen werden mit mit vierstelligem Jahr und ohne Begrenzungszeichen angezeigt: Beispiel: *yyyymmdd*

Standardmäßig gilt DFOUT=S.

Bei beiden DFOUT-Einstellungen wird die Reihenfolge von Tag, Monat und Jahr in den Datumswerten durch den DTFORM-Parameter bestimmt.

Die Länge eines Datumsfeldes wird durch die DFOUT-Einstellung nicht beeinflusst, da jede der beiden Datumswertdarstellungen in ein 8 Byte langes Feld passt.

Der DFOUT-Parameter kann im Natural-Parametermodul dynamisch beim Aufrufen von Natural oder mit dem Systemkommando GLOBALS gesetzt werden. Er wird zur Laufzeit ausgewertet.

Beispiel:

Dieses Beispiel basiert auf der Annahme, dass DTFORM=I.

```
DEFINE DATA LOCAL
1 #DATE  (D) INIT <D'2014-01-31'>
END-DEFINE
...
WRITE #DATE        /* Output if DFOUT=S is set ...: 14-01-31
                   /* Output if DFOUT=I is set ...: 20140131
WRITE #DATE (DF=L) /* Output (regardless of DFOUT): 2014-01-31
...
```

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 RUN-Statement auf dem **Natural-Stack** abgelegt werden.

Mögliche DFSTACK-Einstellungen sind:

DFSTACK=S	8-Byte-Datumsvariablen werden mit zweistelligem Jahr und mit durch den DTFORM-Parameter bestimmten Begrenzungszeichen auf dem **Natural-Stack** abgelegt: Beispiel: *yy-mm-dd* Diese Einstellung bewirkt, dass die Jahrhundert-Informationen nicht mit abgelegt werden (und verlorengehen), wenn ein Datumswert auf dem Stack abgelegt wird. Wenn dann der Wert vom **Natural-Stack** gelesen und in einer anderen Datumsvariable abgelegt wird, wird entweder als Jahrhundert das aktuelle Jahrhundert genommen oder das Jahrhundert durch die Einstellung des YSLW-Parameters (siehe *Gleitendes Jahr-Fenster — der YSLW-Parameter*) 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	Wie bei DFSTACK=S, aber mit folgendem Unterschied: Natural gibt einen Laufzeitfehler aus, wenn der auf dem **Natural-Stack** abzulegende Datumswert nicht mit dem des ursprünglichen Datumswerts identisch ist (entweder aufgrund des YSLW-Parameters oder weil das ursprüngliche Jahrhundert nicht mit dem aktuellen Jahrhundert übereinstimmt).
DFSTACK=I	8-Byte-Datumsvariablen werden mit vierstelligem Jahr und ohne Begrenzungszeichen auf dem **Natural-Stack** abgelegt: Beispiel: *yyyymmdd*

Standardmäßig gilt DFSTACK=S.

Bei jeder Datumsdarstellung werden die Reihenfolge von Tag, Monat und Jahr und die Begrenzungszeichen (falls vorhanden) in den Datumswerten durch den DTFORM-Parameter bestimmt.

Beispiel:

Dieses Beispiel basiert auf der Annahme, dass DTFORM=I und YSLW=0.

```
DEFINE DATA LOCAL
  1 #DATE  (D) INIT <D'2014-01-31'>
  1 #ALPHA1(A8)
  1 #ALPHA2(A10)
END-DEFINE
...
STACK TOP DATA #DATE #DATE (DF=L)
...
INPUT #ALPHA1 #ALPHA2
...
/* Result if DFSTACK=S or =C is set: #ALPHA1 contains 14-01-31
/* Result if DFSTACK=I is set .....: #ALPHA1 contains 20140131
/* Result (regardless of DFSTACK) .: #ALPHA2 contains 2014-01-31
...
```

Gleitendes Jahr-Fenster — der YSLW-Parameter

Mit dem Profilparameter `YSLW` können Sie das Jahrhundert eines zweistelligen Jahr-Wertes bestimmen.

Der YSLW-Parameter kann im Natural-Parametermodul oder dynamisch beim Aufrufen von Natural gesetzt werden. Er wird zur Laufzeit ausgewertet, wenn ein alphanumerischer Datumswert mit einem zweistelligen Jahr in eine Datumsvariable übertragen wird. Dies betrifft Datumswerte, die:

- mit der **mathematischen Funktion** `VAL(field)` verwendet werden,
- mit der Option `IS(D)` in einer logischen Bedingung verwendet werden,
- vom **Natural-Stack** als Eingabedaten gelesen werden,
- in ein Feld als Eingabedaten eingegeben werden.

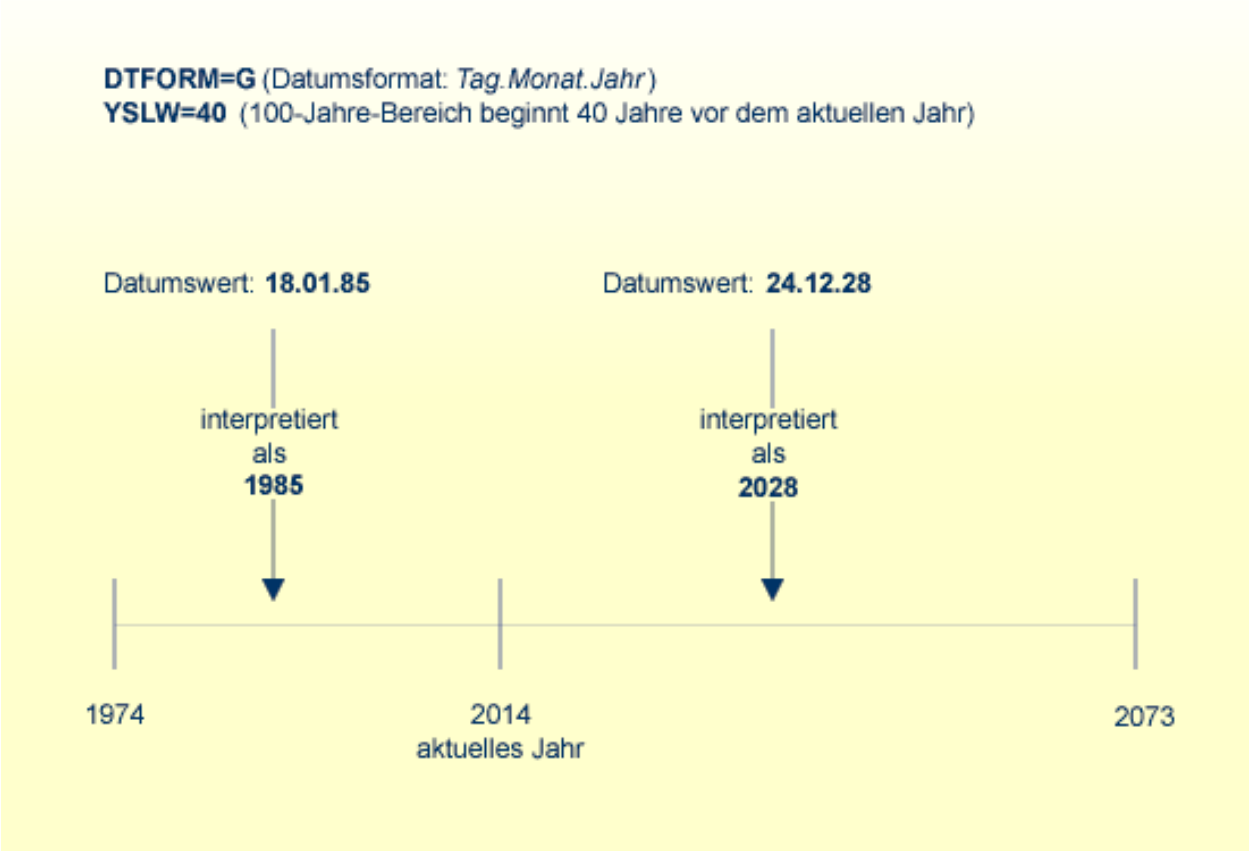
Der `YSLW`-Parameter bestimmt den Bereich von Jahren, der von einem sogenannten „gleitenden Jahr-Fenster“ abgedeckt wird. Dieser Mechanismus geht davon aus, dass ein Datum mit einem zweistelligen Jahr innerhalb eines „Fensters“ von 100 Jahren liegt. Innerhalb dieser 100 Jahre kann jeder zweistellige Jahr-Wert eindeutig einem bestimmten Jahrhundert zugeordnet werden.

Mit dem `YSLW`-Parameter legen Sie fest, mit wie vielen Jahren 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 „gleitende Jahr-Fenster“-Mechanismus ist nicht aktiv; bei einem zweistelligen Jahr wird dann angenommen, dass es im aktuellen Jahrhundert liegt.

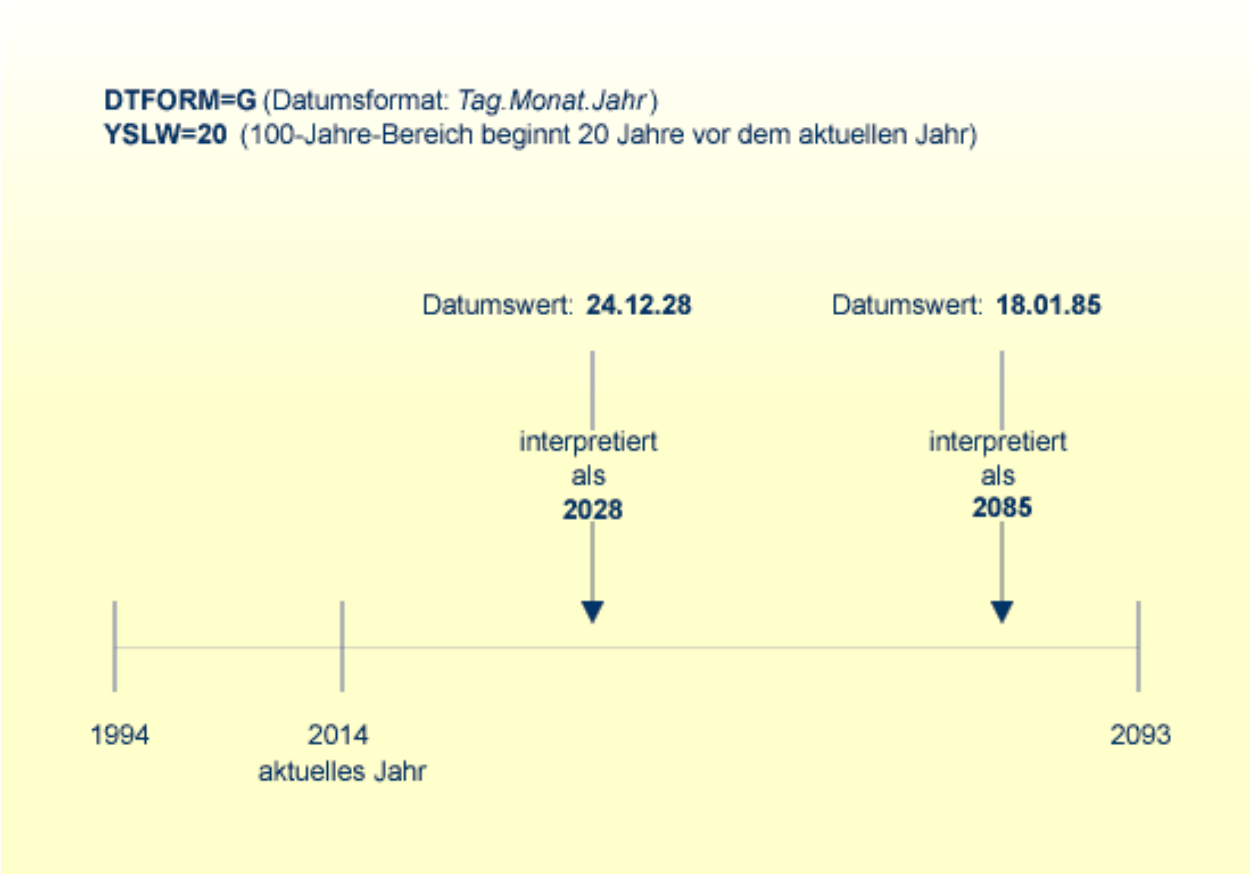
Beispiel 1:

Wenn das aktuelle Jahr 2014 ist und Sie `YSLW=40` angeben, deckt das „gleitende Jahr-Fenster“ die Jahre 1974 bis 2072 ab. Ein zweistelliger Jahr-Wert *nn* von `74` bis `99` wird dementsprechend als `19`*nn* interpretiert, während ein zweistelliger Jahr-Wert *nn* von `00` bis `73` als `20`*nn* interpretiert wird.



Beispiel 2:

Wenn das aktuelle Jahr 2014 ist und Sie `YSLW=20` angeben, deckt das „gleitende Jahr-Fenster“ die Jahre 1994 bis 2093 ab. Ein zweistelliger Jahr-Wert *nn* von `94` bis `99` wird dementsprechend als 19*nn* interpretiert, während ein zweistelliger Jahr-Wert *nn* von `00` bis `93` als 20*nn* interpretiert wird.



Kombinationen von DFSTACK und YSLW

Die folgenden Beispiele veranschaulichen die Auswirkung verschiedener Kombinationen der Parameter `DFSTACK` und `YSLW`.

Anmerkung: Alle diese Beispiele basieren auf der Annahme, dass `DTFORM=I`.

Beispiel 1:

Dieses Beispiel geht vom aktuellen Jahr 2014 und folgenden Parametereinstellungen aus:

`DFSTACK=S` (Standardeinstellung) und `YSLW=20`

```
DEFINE DATA LOCAL
  1 #DATE1 (D) INIT <D'1956-12-31'>
  1 #DATE2 (D)
END-DEFINE
...
STACK TOP DATA #DATE1 /* century information is lost (year 56 is stacked)
...
INPUT #DATE2          /* year sliding window determines 56 to be 1956
...
/* Result: #DATE2 contains 1956-12-31
```

In diesem Fall ist das „gleitende Jahr-Fenster“ unpassend gesetzt, so dass die Jahrhundert-Informationen sich (unbeabsichtigt) ändern.

Beispiel 2:

Dieses Beispiel geht vom aktuellen Jahr 2014 und folgenden Parametereinstellungen aus:

`DFSTACK=S` (Standardeinstellung) und `YSLW=60`

```
DEFINE DATA LOCAL
  1 #DATE1 (D) INIT <D'1956-12-31'>
  1 #DATE2 (D)
END-DEFINE
...
STACK TOP DATA #DATE1 /* century information is lost (year 56 is stacked)
...
INPUT #DATE2          /* year sliding window determines 56 to be 1956
...
/* Result: #DATE2 contains 1956-12-31
```

In diesem Fall ist das „gleitende Jahr-Fenster“ passend gesetzt, so dass die ursprünglichen Jahrhundert-Informationen korrekt wiederhergestellt werden.

Beispiel 3:

Dieses Beispiel geht vom aktuellen Jahr 2014 und folgenden Parametereinstellungen aus:

`DFSTACK=C` und `YSLW=0` (Standardeinstellung)

```
DEFINE DATA LOCAL
  1 #DATE1 (D) INIT <D'1956-12-31'>
  1 #DATE2 (D)
END-DEFINE
...
STACK TOP DATA #DATE1 /* century information is lost (year 56 is stacked)
...
INPUT #DATE2          /* 56 is assumed to be in current century -> 2056
...
/* Result: NAT1130 runtime error (Unintended century switch...)
```

In diesem Fall ändern sich (unbeabsichtigt) die Jahrhundert-Informationen. Allerdings wird diese Änderung durch die Parametereinstellung DFSTACK=C abgefangen.

Beispiel 4:

Dieses Beispiel geht vom aktuellen Jahr 2014 und folgenden Parametereinstellungen aus:

DFSTACK=C und YSLW=60

```
DEFINE DATA LOCAL
  1 #DATE1 (D) INIT <D'2056-12-31'>
  1 #DATE2 (D)
END-DEFINE
...
STACK TOP DATA #DATE1 /* century information is lost (year 56 is stacked)
...
INPUT #DATE2          /* year sliding window determines 56 to be 1956
...
/* Result: NAT1130 runtime error (Unintended century switch...)
```

In diesem Fall ändern sich die Jahrhundert-Informationen aufgrund des „gleitenden Jahr-Fensters“. Allerdings wird diese Änderung durch die Parametereinstellung DFSTACK=C abgefangen.

Festes Jahr-Fenster

Informationen zu diesem Thema entnehmen Sie der Beschreibung des Profilparameters YSLW.

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).

Mit dem DFTITLE-Parameter können Sie eine der folgenden Datumsdarstellungsarten wählen:

DFTITLE=S	8-Byte-Datumsdarstellung mit zweistelligem Jahr und Begrenzungszeichen. Beispiel: *yy-mm-dd*.
DFTITLE=L	10-Byte-Datumsdarstellung mit vierstelligem Jahr und Begrenzungszeichen. Beispiel: *yyyy-mm-dd*.
DFTITLE=I	8-Byte-Datumsdarstellung mit vierstelligem Jahr ohne Begrenzungszeichen. Beispiel: *yyyymmdd*.

Bei jeder dieser Ausgabeformen werden die Reihenfolge der Tages-, Monats- und Jahreskomponenten sowie die verwendeten Begrenzungszeichen durch den DTFORM-Parameter bestimmt.

Der DFTITLE-Parameter kann im Natural-Parametermodul dynamisch beim Aufrufen von Natural oder mit dem Systemkommando GLOBALS gesetzt werden. Er wird zur Laufzeit ausgewertet.

Beispiel:

Dieses Beispiel geht von DTFORM=I aus.

```
WRITE 'HELLO'
END
/*
/* Date in page title if DFTITLE=S is set ...: 14-01-31
/* Date in page title if DFTITLE=L is set ...: 2014-01-31
/* Date in page title if DFTITLE=I is set ...: 20140131
```

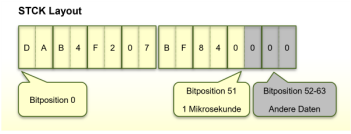
Anmerkung: Der DFTITLE-Parameter hat keine Auswirkungen auf benutzerdefinierte Seitenüberschriften, wie sie mit einem WRITE TITLE-Statement angegeben werden.

57 Verarbeitung von Store Clock-Werten

Dieses Dokument behandelt verschiedene Aspekte der Verarbeitung von Store Clock-Werten in Natural-Anwendungen.

Der originale Store Clock und das Jahr-2042-Problem

Die Natural-Systemvariable `*TIMESTMP` liefert den maschineninternen Store Clock-Wert, wie er durch die Großrechner-Maschineninstruktion `STCK` (Store Clock Format) bereitgestellt wird. Der Store Clock ist ein 8 Byte langes binäres Feld. Er zählt die Mikrosekunden seit 01.01.1900 00:00:00 (UTC), wobei Bitposition 51 exakt eine Mikrosekunde darstellt (Bitposition `0` ist das ganz linke Bit). Die übrigen Bits können je nach Maschinenimplementierung höhere Genauigkeitswerte wie Nanosekunden oder eine Prozessorkennung (ID) enthalten.



Der Store Clock wird seinen höchstmöglichen Wert am 17. September 2042 um 23:53:47.370495 (UTC) erreichen, wobei dann alle 52 Bits auf `1` gesetzt sein werden. Danach wird der Store Clock-Zähler zurückgesetzt und es wird wieder ab `0` hochgezählt. Das Datum, an dem die Uhr zurückgestellt wird, wird im Folgenden als Rücksprungdatum (*Wrap Date*) bezeichnet.

Nach dem Rücksprungdatum wird der Store Clock dieselben Werte wie im Bereich von 1900 bis 2042 verwenden. Deshalb ist es nicht möglich zu entscheiden, ob ein Store Clock vor oder nach dem Rücksprungdatum genommen wurde.

Wenn der Store Clock die Jahre von 1900 bis 2042 abdeckt, nennen wir ihn den „originalen Store Clock“.

Für das Jahr 2042 ist Folgendes zu beachten („Jahr-2042-Problem“):

- **Vergleich und Sortierung**
  Ein Vergleich von zeitbezogenen Werten sollte die chronologische Reihenfolge so widerspiegeln, dass frühere Werte kleiner als spätere Werte sind. Leider ist ein Store Clock-Wert, der kurz vor dem Rücksprungdatum genommen wurde, größer als ein Store Clock-Wert, der nach dem Datum genommen wurde, an dem der Store Clock auf `0` gestellt wurde. Somit führt ein Vergleich solcher Store Clock-Werte zu unrichtigen Ergebnissen. Beispiel: Der Store Clock-Wert vom 01.01.2040 (vor dem Rücksprungdatum) ist größer als der Store Clock-Wert vom 01.01.2043 (nach dem Rücksprungdatum). Sortiert man Store Clock-Werte, die vor und nach dem Rücksprungdatum genommen wurden, werden diese nicht die chronologische Reihenfolge widerspiegeln. Ein `READ`-Statement, das die genannten Werte als Start- und Endwerte verwendet, wird keinen Wert zurückgeben, da der Endwert kleiner ist als der Startwert.

- **Konvertierung**
  Wenn man einen Store Clock-Wert in ein Datum konvertiert, wird davon ausgegangen, dass er in den Jahren von 1900 bis 2042 genommen wurde. Wurde der Store Clock-Wert aber tatsächlich nach dem Rücksprungdatum genommen, wird die Konvertierung deshalb zu einem falschen Ergebnis führen. Beispiel: Ein Store Clock-Wert wurde am 07.12.2043 (d.h. nach dem Rücksprungdatum) genommen. Wenn man diesen Store Clock-Wert in ein Datum konvertiert, wird das Ergebnis der 21.03.1901 sein.
- **Berechnung einer Zeitdifferenz**
  Wenn man Store Clock-Werte in Mikrosekunden konvertiert, kann man sie zur Berechnung einer Zeitdifferenz benutzen. Das funktioniert problemlos, falls das Rücksprungdatum nicht beteiligt ist. Wenn man einen (großen) Store Clock-Wert, der vor dem Rücksprungdatum genommen wurde, von einem (kleinen) Store Clock-Wert, der nach dem Rücksprungdatum genommen wurde, subtrahiert, erhält man ein falsches (und zwar negatives) Ergebnis. Beispiel: Wenn man die Store Clock-Zeitdifferenz zwischen den Jahren 2039 und 2043 berechnet, erhält man `-138` Jahre als Ergebnis.

Die Zeit, in der wir nur Store Clock-Werte kleiner als das Rücksprungdatum (17.09.2042) verarbeiten, bezeichnen wir als Übergangszeit. Während dieser Zeit sollten die ursprünglichen Store Clock-Werte durch einen der in den folgenden Abschnitten beschriebenen Lösungsansätze ersetzt werden.

Wie das Jahr-2042-Problem gelöst werden kann

Um das Jahr-2042-Problem zu lösen, gibt es folgende Ansätze:

1. **Temporärer Ansatz: Store Clock mit gleitendem Jahr-Fenster.**
   Der **Store Clock mit gleitendem Jahr-Fenster** interpretiert den Store Clock-Wert in einem gleitendem Jahr-Fenster. Er unterstützt den Zeitbereich von 1971 bis 2114. Die binären Werte können nicht zum Sortieren oder Vergleichen verwendet werden.
2. **Genereller Ansatz: Erweiterter Store Clock.**
   Der **erweiterte Store Clock** ist 16 Bytes lang und unterstützt den Zeitbereich von 1900 bis 38434. Die binären Werte können zum Sortieren und Vergleichen verwendet werden. Erweiterte Store Clock-Werte sind nicht kompatibel zu Store Clock-Werten (weder original noch mit gleitendem Jahr-Fenster). Daten und Programme, die sich auf Store Clock-Werte beziehen, müssen in einem einzigen, großen Schritt auf erweiterte Store Clock-Werte umgestellt werden.
3. **Smarter Ansatz: Smarter Store Clock**
   Der **smarte Store Clock** ist 9 Bytes lang und unterstützt den Zeitbereich von 1900 bis 38434. Die binären Werte können zum Sortieren und Vergleichen verwendet werden. Während der Übergangszeit sind die smarten Store Clock-Werte mit den originalen Store Clock-Werten kompatibel. Daten und Programme, die sich auf Store Clock-Werte beziehen, können schrittweise in Smart Store Clock-Werte umgewandelt werden. Dies ist der empfohlene Ansatz, um das Jahr-2042-Problem zu lösen.

Für diese Ansätze stehen in der Natural System Library SYSEXT (siehe auch SYSEXT Utility) Anwendungsprogrammierschnittstellen (APIs) zur Verfügung, siehe *Natural-APIs zur Store Clock-Verarbeitung*.

Store Clock-Wert mit gleitendem Jahr-Fenster

Wenn ein Store Clock-Wert in ein Datum konvertiert wird, wird bei der Konvertierung des originalen Store Clock-Wertes davon ausgegangen, dass er aus dem Bereich von 1900 bis 2042 genommen wurde. Um den Bereich zu erweitern, verwenden wir ein „gleitendes Jahr-Fenster“ (*Year Sliding Window*), das den Bereich von 1971 bis 2114 unterstützt. Das impliziert, dass die Jahre von 1900 bis 1971 nicht mehr unterstützt werden.

Im Jahr 1971 schaltet das erste Bit im Store Clock-Wert auf `1`. Deshalb wird dieses Bit zur Realisierung des gleitenden Jahr-Fensters genutzt:

- Von 1971 bis 2042 ist das erste Bit `1`. Store Clock-Werte aus diesem Bereich werden wie zuvor interpretiert.
- Von 1900 bis 1971 und von 2042 bis 2114 ist das erste Bit `0`. Mit dem gleitendem Jahr-Fenster werden diese Store Clock-Werte als 2042 bis 2114 interpretiert.

Wenn ein Store Clock-Wert mit gleitendem Jahr-Fenster in Mikrosekunden konvertiert wird und das erste Bit `0` ist, wird vor der Konvertierung der binäre Store Clock-Wert auf der linken Seite mit `x'01'` erweitert, damit das Ergebnis immer die Anzahl an Mikrosekunden seit dem 01.01.1900 widerspiegelt.

Der Bereich des Store Clock mit gleitendem Jahr-Fenster

- beginnt am 11.05.1971 um 11:56:53,685248 Uhr (UTC) und
- endet am 26.01.2114 um 11:50:41,055743 Uhr (UTC).

Anmerkungen:

1. Wenn Sie 8 Byte lange binäre Store Clock-Werte sortieren, werden diese nicht in chronologischer Reihenfolge sein, und zwar selbst dann nicht, wenn Sie das gleitende Jahr-Fenster benutzen. Für diese Aufgabe müssen Sie smarte oder erweiterte Store Clock-Werte benutzen.
2. Wenn Sie 8 Byte lange binäre Store Clock-Werte vergleichen, kann es vorkommen, dass das Ergebnis nicht die chronologische Reihenfolge widerspiegelt, und zwar selbst dann nicht, wenn Sie das gleitende Jahr-Fenster benutzen. Für diese Aufgabe können Sie den Copycode **USR9201D** benutzen.
3. Wenn der Store Clock-Wert nach dem Ende des gleitenden Jahr-Fensters genommen wird, wird er falsch interpretiert. In diesem Fall müssen Sie ebenfalls smarte oder erweiterte Store Clock-Werte benutzen.

4. Ein Store Clock-Wert wird in einem 8 Byte langen Feld gespeichert, in dem die letzten 12 Bits niemals Null sind. Diese Tatsache kann genutzt werden, um zu bestimmen, ob das Feld für den Store Clock noch unbenutzt ist. Wenn alle 8 Bytes in dem Feld Null sind (`NULL`-Wert), dann ist das Feld noch unbenutzt und es wurde noch kein Store Clock-Wert gespeichert. Sie müssen sicherstellen, dass das Store Clock-Feld keinen `NULL`-Wert enthält, bevor Sie es in einer Konvertierungsfunktion benutzen.

Im Vergleich zum originalen Store Clock bietet der Store Clock mit gleitendem Jahr-Fenster folgende Verbesserungen:

- **Konvertierung**
  Natural APIs können benutzt werden, um Store Clock-Werte mit gleitendem Jahr-Fenster richtig zu konvertieren, wenn die Store Clock-Werte im Zeitraum von 1971 bis 2114 genommen worden sind.
- **Berechnung einer Zeitdifferenz**
  Zum Konvertieren von Store Clock-Werten mit gleitendem Jahr-Fenster in Mikrosekunden (seit 01.01.1900) können Natural-APIs benutzt werden. Eine Zeitdifferenzberechnung mit diesen Werten liefert die richtige Zeitdauer (wenn die Store Clock-Werte von 1971 bis 2114 sind).
- **Ausblick**
  Das Jahr-2042-Problem wird bis 2114 hinausgeschoben.

Im Vergleich zum smarten oder zum erweiterten Store Clock bietet der Store Clock mit gleitendem Jahr-Fenster die folgenden temporären Vorteile:

- **Größe**
  Die Größe des Store Clock-Werts bleibt bei 8 Bytes.
- **Datenkonvertierung (im Vergleich zum erweiterten Store Clock**
  Es besteht keine Notwendigkeit, die Store Clock-Daten auf 16 Byte lange Werte zu konvertieren. Gespeicherte 8 Byte lange Store Clock-Daten werden richtig interpretiert, wenn sie im Bereich 1971 bis 2114 sind.
- **Änderungen an Programmen**
  Die Konvertierung erfordert nur einen geringen Änderungsaufwand. Die Systemvariable `*TIMESTMP` kann weiterhin benutzt werden.

Migration vom originalen Store Clock zum Store Clock mit gleitendem Jahr-Fenster

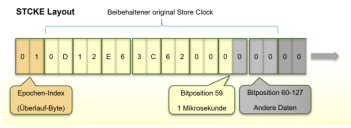
Vor einer Migration vom originalen Store Clock zum Store Clock mit gleitendem Jahr-Fenster ist Folgendes zu berücksichtigen:

- Vor Ende des gleitenden Jahr-Fensters im Jahre 2114 werden weitere Codeänderungen erforderlich.
- Falls Ihre Anwendung eine Natural API aufruft, die den originalen Store Clock-Wert unterstützt, müssen Sie diese API durch eine API ersetzen, die den Store Clock mit gleitendem Jahr-Fenster unterstützt.

- Wenn Sie binäre Store Clock-Werte vergleichen oder sortieren, werden diese nicht in chronologischer Reihenfolge sein. Siehe **Anmerkung** weiter oben.

Erweiterter Store Clock

Die Natural-Systemvariable `*TIMESTMPX` liefert den maschineninternen erweiterten Store Clock-Wert, so wie er durch die Großrechner-Instruktion `STCKE` (Store Clock Extended Format) zur Verfügung gestellt wird. Der erweiterte Store Clock ist ein 16 Byte langes binäres Feld. Er zählt die Mikrosekunden seit dem 01.01.1900 00:00:00 (UTC), wobei Bitposition 59 exakt eine Mikrosekunde darstellt (Bitposition `0` ist das ganz linke Bit). Die originale Store Clock-Zeitdarstellung wird in Byte 2 bis 9 beibehalten. Links wird ein zusätzliches Überlauf-Byte hinzugefügt, der so genannte Epochen-Index.



Der erweiterter Store Clock deckt die Jahre von 1900 bis 38434 ab.

Im Vergleich zum originalen Store Clock bietet der erweiterte Store Clock folgende Verbesserungen:

- **Vergleich und Sortierung**
  Wenn man Werte vergleicht oder sortiert, liefert der erweiterte Store Clock immer die erwartete chronologische Reihenfolge.
- **Konvertierung**
  Zum Konvertieren von erweiterten Store Clock-Werten können Natural-APIs benutzt werden. Weil jeder erweiterte Store Clock-Wert im erweiterten Store Clock-Wertebereich eindeutig ist, ist die Berechnung immer richtig.
- **Berechnung einer Zeitdifferenz**
  Zum Konvertieren von erweiterten Store Clock-Werten in Mikrosekunden (seit 01.01.1900) können Natural-APIs benutzt werden. Eine Zeitdifferenzberechnung mit diesen Werten liefert die richtige Zeitdauer.
- **Ausblick**
  Für die Zukunft ist keine weitere Codeänderung erforderlich.

Migration vom originalen Store Clock zum erweiterten Store Clock

Vor einer Migration vom originalen Store Clock zum erweiterten Store Clock ist Folgendes zu berücksichtigen:

- Die Größe des binären Feldes erhöht sich von 8 Bytes auf 16 Bytes.

- Erweiterte Store Clock-Werte sind nicht kompatibel mit Store Clock-Werten im Format B8. Es ist nicht möglich, einen Store Clock-Wert in einen erweiterten Store Clock-Wert zu verschieben oder umgekehrt.

  Beispiel:

```
#STORECLOCK: DD943485BC302002

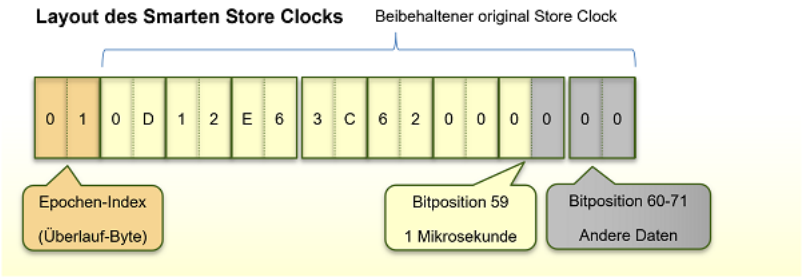
MOVE #STORECLOCK TO #EXTENDED

Erhalten: 00000000000000DD943485BC302002
Erwartet: 00DD943485BC30200200000000000000 ↩
```

- Ein erweiterter Store Clock-Wert kann nicht direkt mit einem originalen Store Clock-Wert verglichen werden (und auch nicht mit einem Store Clock-Wert mit gleitendem Jahr-Fenster). Der erweiterte Store Clock-Wert enthält sieben auf der rechten Seite hinzugefügte zusätzliche Bytes. Daher wird ein Vergleich kein richtiges Ergebnis liefern.
- Gespeicherte Store Clock-Werte müssen in erweiterte Store Clock-Werte konvertiert werden. Für diese Aufgabe können Sie den **Copycode USR9201R** benutzen.
- Wenn Store Clock-Werte im Format B8 in Adabas gespeichert werden, müssen Sie die Daten so editieren, dass das neue Feld vom Format B16 richtige erweiterte Store Clock-Werte enthält.
- Jedes Programm, das Store Clock-Werte verarbeitet, muss an erweiterte Store Clock-Werte angepasst werden. Weil aber Store Clock-Werte und erweiterte Store Clock-Werte nicht kompatibel sind, müssen die Änderungen in einem einzigen Schritt erledigt werden.
- Falls Ihre Anwendung eine Natural API aufruft, die den originalen Store Clock-Wert unterstützt, müssen Sie sie durch eine API ersetzen, die den erweiterten Store Clock-Wert unterstützt.
- Diese komplette Konvertierung erfordert größere Änderungen.
- Es wird empfohlen, statt des erweiterten Store Clock den smarten Store Clock zu verwenden. Dies wird im folgenden Abschnitt beschrieben.

Smarter Store Clock

Der smarte Store Clock ist ein 9 Byte langes binäres Feld. Es zählt die Mikrosekunden seit dem 01.01.1900 00:00:00 (UTC), wobei Bitposition 59 exakt eine Mikrosekunde darstellt (Bitposition 0 ist das ganz linke Bit). Die Zeitdarstellung des originalen Store Clock wird in Byte 2 bis 9 beibehalten. Auf der linken Seite wird ein zusätzliches Überlaufbyte hinzugefügt, der so genannte Epochen-Index. Mit anderen Worten: Der smarte Store Clock umfasst die ersten 9 Bytes des erweiterten Store Clock.



Der smarte Store Clock deckt die Jahre von 1900 bis 38434 ab.

Im Vergleich zum originalen Store Clock bietet der smarte Store Clock folgende Verbesserungen:

- **Vergleich und Sortierung**
  Wenn man Werte vergleicht oder sortiert, liefert der smarte Store Clock immer die erwartete chronologische Reihenfolge.
- **Konvertierung**
  Zum Konvertieren von smarten Store Clock-Werten können Natural-APIs benutzt werden. Weil jeder smarte Store Clock-Wert im smarten Store Clock-Wertebereich eindeutig ist, ist die Interpretation immer richtig.
- **Berechnung einer Zeitdifferenz**
  Zum Konvertieren von smarten Store Clock-Werten in Mikrosekunden (seit 01.01.1900) können Natural-APIs benutzt werden. Eine Zeitdifferenzberechnung mit diesen Werten liefert die richtige Zeitdauer.
- **Ausblick**
  Für die Zukunft ist keine weitere Codeänderung erforderlich.

Migration vom originalen Store Clock zum smarten Store Clock

Vor einer Migration vom originalen Store Clock zum erweiterten Store Clock ist Folgendes zu berücksichtigen:

- Die Größe des binären Feldes erhöht sich von 8 Bytes auf 9 Bytes.
- Während des Übergangszeitraums:
  - Erhöht man die Länge des originalen Store Clock-Wertes von B8 nach B9, erhält man den richtigen smarten Store Code-Wert. Der Grund dafür ist, dass in Natural und Adabas binäre Felder rechtsbündig sind. Durch Vergrößerung der Länge wird das Überlaufbyte mit `H'00'` gefüllt.
  - Smarte Store Clock-Werte sind kompatibel mit originalen Store Clock-Werten. Man kann einen originalen Store Clock-Wert in einen smarten Store Clock-Wert verschieben und umgekehrt.

    Beispiel:

```
#STORECLOCK: DD943485BC302002

MOVE #STORECLOCK TO #SMART

Erhalten: 00DD943485BC302002
Erwartet: 00DD943485BC302002
```

  - Smarte Store Clock-Werte können mit originalen Store Clock-Werten verglichen werden.
- Jedes Programm, das Store Clock-Werte verarbeitet, muss an smarte Store Clock-Werte angepasst werden. Weil Store Clock-Werte und smarte Store Clock-Werte kompatibel sind, können die Änderungen schrittweise erledigt werden.
- Falls Ihre Anwendung eine Natural API aufruft, die den originalen Store Clock-Wert unterstützt, müssen Sie sie durch eine API ersetzen, die den smarten Store Clock-Wert unterstützt.
- Diese komplette Konvertierung erfordert geringfügige Änderungen.

Übergang zum smartem Store Clock

Datendefinition in Natural ersetzen

Ein Feld, das Store Clock-Werte enthält, wird wie folgt definiert:

```
1 #TIMESTMP (B8)   /* Originaler Store Clock-Wert
```

Ersetzen Sie die Definition durch:

```
1 #TIMESTMP (B9)   /* Smarter Store Clock-Wert
```

Aktuellen Store Clock-Wert erhalten

Um den aktuellen Store Clock-Wert zu erhalten, wird das folgende Statement ausgeführt:

```
MOVE *TIMESTMP TO #TIMESTMP   /* Aktuellen Store Clock-Wert ermitteln
```

Während des Übergangszeitraums kann das Statement noch verwendet werden, auch wenn die Länge von `#TIMESTMP` auf B9 erhöht worden ist. Langfristig sollte das Statement aber ersetzt werden durch:

```
INCLUDE USR9201F '*TIMESTMPX' '#TIMESTMP' /* Aktuellen Store Clock-Wert ermitteln
```

Alternativ können Sie aber auch eine Redefinition durchführen:

```
1 #TIMESTMPX (B16)   /* Erweiterter Store Clock-Wert
1 REDEFINE #TIMESTMPX
  2 #TIMESTMP (B9)   /* Smarter Store Clock-Wert
```

Und das folgende Statement verwenden:

```
MOVE *TIMESTMPX TO #TIMESTMPX /* Aktuellen Store Clock-Wert ermitteln
```

Store Clock-Wert verschieben

Während des Übergangszeitraums können Store Clock-Werte von einem zum anderen verschoben werden, auch wenn die Länge des einen B9 und die des anderen noch B8 ist.

```
MOVE #TIMESTMP-A TO #TIMESTMP-B   /* Store Clock-Wert verschieben
```

Aufruf eines Subprogramms durchführen

Bei einem Subprogramm wird ein Store Clock-Wert als Parameter verwendet:

```
1 #TIMESTMP (B8)   /* Store Clock-Wert
```

Wenn Sie die Länge des Feldes auf B9 erhöhen und es Programme gibt, die es mit B8 aufrufen, wird das Programm fehlschlagen. Verwenden Sie in diesem Fall die folgende Definition:

```
1 #TIMESTMP (B9) BY VALUE RESULT  /* Store Clock-Wert
```

Sobald alle Aufrufer Store Clock-Werte im Format B9 verwenden, kann das `BY VALUE RESULT` entfernt werden.

Store Clock-Werte vergleichen

Während des Übergangszeitraums können Store Clock-Werte miteinander verglichen werden, auch wenn die Länge des einen Werts B9 ist und die des anderen noch B8 ist.

```
IF #TIMESTMP-A > #TIMESTMP-B   /* Store Clock-Werte vergleichen
```

Felddefinition in Adabas ändern

Ein Feld, das Store Clock-Werte enthält, wird wie folgt definiert:

```
1,AA,8,B,...
```

Ersetzen Sie die Definition durch:

```
1,AA,9,B,...
```

Auf dem Großrechner können Sie dies mit dem Adabas Online Service oder dem Dienstprogramm ADADBS (Database Services) erreichen, unter Linux und Windows mit dem Dienstprogramm ADAFDU (File Definition). Beachten Sie, dass unter Linux und Windows ein Feld, das Store Clock-Werte enthält, mit der Option HF (high order first) definiert werden muss.

Aufgrund der Änderung der Länge enthält die Datenbank gültige smarte Store Clock-Werte. Es ist nicht notwendig, die Daten zu bearbeiten. Handelt es sich bei dem Feld um einen Deskriptor, liefert es die richtige chronologische Reihenfolge.

Auf der Natural-Seite müssen Sie das Datendefinitionsmodul (DDM) entsprechend ändern. Falls eine Länge angegeben ist, müssen Sie außerdem die zugehörigen Datensichten (Views) ändern.

Superdeskriptor-Definition in Adabas anpassen

Ein Feld `AA`, das Store Clock-Werte enthält, wird als übergeordnetes Feld eines Superdeskriptors verwendet:

```
SA=AC(1,20),AA(1,8)
```

Während des Übergangszeitraums kann der Superdeskriptor auch dann noch weiterverwendet werden, wenn die Feldlänge des übergeordneten Feldes auf 9 Bytes erhöht worden ist. Das liegt daran, dass die Bytes in einem binären Feld von rechts nach links gezählt werden.

Langfristig sollte das Statement aber ersetzt werden durch:

```
SA=AC(1,20),AA(1,9)
```

Alle Variablen in Natural-Programmen (z.B. Start- und Endwerte), die sich auf den Superdeskriptor beziehen, müssen zusammen mit den Datenbankänderungen angepasst werden. Alternativ können Sie auch den ursprünglichen Superdeskriptor beibehalten und zusätzlich einen neuen Superdeskriptor mit der neuen Länge anlegen. Dann können Sie ein Programm nach dem anderen auf den neuen Superdeskriptor umstellen. Sobald alle Programme auf den neuen Superdeskriptor zugreifen, kann der alte Superdeskriptor freigegeben (gelöscht) werden.

Lokaler Store Clock

Die Natural-Systemvariable `*TIMX` stellt die lokale Uhrzeit (Ortszeit) in Zehntelsekunden zur Verfügung. Wird die lokale Uhrzeit mit höherer Genauigkeit benötigt, kann die Natural API `USR9178N` verwendet werden. Sie liefert die lokale Uhrzeit und die entsprechende UTC-Zeit im Store Clock-Format (B8), in Mikrosekunden seit 01.01.1900 und im Natural-Zeitformat (T). Zusätzlich liefert sie die Zeitdifferenz, einen Standard-`DATETIME`-Wert und die Zeitzone. Es sind Funktionen vorhanden, mit denen die Werte in andere Formate konvertiert werden können. Die Copycodes `USR9178X`, `USR9178Y` und `USR9178Z` bieten im Prinzip die gleiche Funktionalität, jedoch bei wesentlich besserer Performance.

Layout eines lokalen Store Clock-Werts

Die ersten 7 Bytes des lokalen Store Clock-Werts haben den gleichen Inhalt wie der originale Store Clock, wobei die Bitpositionen 0 – 51 die Mikrosekunden der lokalen Uhrzeit enthalten. Deshalb können zur Interpretation eines lokalen Store Clock-Werts Standard-APIs (z.B. `USR1023*` oder `USR9201*`) verwendet werden.

Das letzte Byte eines mit `USR9178*` generierten lokalen Store Clock-Werts enthält die Zeitdifferenz. Mit dieser Information kann später festgestellt werden, in welcher Zeitzone der lokale Store Clock-Wert genommen wurde, insbesondere, ob in der Sommer- oder der Winterzeit.

Zeitdifferenz und Zeitzone

Die Zeitdifferenz drückt den Unterschied zwischen Local Time und UTC Time aus:

```
Zeitdifferenz = Local time - UTC time
```

Die API `USR9178N` und die entsprechenden Copycodes liefern die Zeitdifferenz in Einheiten von 1/10 Sekunden.

Das letzte Byte eines lokalen Store Clock-Werts enthält die Zeitdifferenz in Einheiten von 15 Minuten als Ganzzahl mit Vorzeichen (Format I1).

Die API `USR9178N` gibt außerdem die der Zeitdifferenz zugeordnete Zeitzone zurück.

- *-hh:ii* bei einer negativen Zeitdifferenz oder
- *+hh:ii* andernfalls.

Standard-DATETIME

Die API USR9178N gibt den Standard-DATETIME-Wert zurück:

*yyyy-mm-dd*T*hh:ii:ss.fff*

dabei ist *fff* der Teilbetrag in Millisekunden.

Natural-APIs zur Store Clock-Verarbeitung

In der Natural Library SYSEXT (siehe SYSEXT Utility) stehen folgende Natural Anwendungsprogrammierschnittstellen (APIs) zur Verfügung:

Natural API	Funktion	Store Clock			Vorgänger-Version
		Original	mit gleitendem Jahr-Fenster	Smart / Erweitert	
USR1009N	Store Clock mit gleitendem Jahr-Fenster in Mikrosekunden konvertieren	ja	ja	nein	
USR1023N	Zeitbezogene Systemvariablen konvertieren	ja	nein	nein	
USR9178N	Pflege von lokalen Store Clock-Werten	nein	ja	nein	
USR9201N	Zeitbezogene Variablen konvertieren	nein	ja	ja	USR1023N

USR1009N

Die API USR1009N konvertiert einen Store Clock-Wert in Mikrosekunden seit 01.01.1900. Standardmäßig wird der Store Clock-Wert *ohne* gleitendes Jahr-Fenster (Bereich 1900-2042) interpretiert. Ein optionaler Parameter steht zur Verfügung, um den Store Clock-Wert *mit* dem gleitendem Jahr-Fenster (Bereich 1971-2114) zu interpretieren.

Die Copycodes **USR9201V** (mit gleitendem Jahr-Fenster) und **USR1023Y** (ohne gleitendes Jahr-Fenster) bieten bei besserer Performance die gleiche Funktionalität an wie die API USR1009N. Es wird daher empfohlen, diese Copycodes zu benutzen.

USR1023N

Die API USR1023N konvertiert eine zeitbezogene Variable in andere Formate. Sie verwendet dazu den originalen Store Clock (1900 – 2042).

Es wird dringend empfohlen, anstelle der API USR1023N die API **USR9201N** zu verwenden. Andernfalls werden Store Clock-Werte nach dem Jahr 2042 falsch interpretiert. Beispiel: Wenn Sie einen Store Clock-Wert am 07.12.2043 nehmen, wird ihn die API USR1023N als 21.03.1901 interpretieren.

Als Eingabe erwartet die API USR1023N eines der folgenden Formate:

- Natural-Datum und Uhrzeit,
- Natural-Datum,
- Mikrosekunden (seit 01.01.1900) oder
- originaler Store Clock.

Die API USR1023N konvertiert die Eingabe in alle anderen Formate.

Zusätzlich stehen zur API USR1023N die folgenden Copycodes zur Verfügung:

Copycode	Funktion
USR1023W	Mikrosekunden in originalen Store Clock konvertieren.
USR1023X	Mikrosekunden in Natural-Uhrzeit konvertieren.
USR1023Y	Originalen Store Clock in Mikrosekunden konvertieren.
USR1023Z	Natural-Uhrzeit in Mikrosekunden konvertieren.

Die Copycodes sind wesentlich schneller als das Subprogramm USR1023N.

USR9178N

Die API USR9178N dient zur Pflege von lokalen Store Clock-Werten. Sie unterstützt Store Clock mit gleitendem Jahr-Fenster (1971 - 2114).

Bei der Funktion **C** gibt die API die folgenden aktuellen Werte zurück:

- Lokaler Store Clock
- UTC Store Clock
- Zeitdifferenz in 1/10 Sekunden
- Lokale Uhrzeit (Ortszeit) in Mikrosekunden
- UTC-Uhrzeit in Mikrosekunden
- Lokale Uhrzeit (Ortszeit) im Natural-Format (T)

- UTC-Uhrzeit im Natural-Format (T)
- Standard-`DATETIME`
- Zeitzone

Bei der Funktion **L** konvertiert die API einen lokalen Store Clock-Wert in die anderen Formate.

Bei der Funktion **U** konvertiert die API einen UTC Store Clock-Wert und eine Zeitdifferenz in die anderen Formate.

Zusätzlich zu der API stehen die folgenden Copycodes zur Verfügung:

- USR9178X - Aktuellen lokalen Store Clock, UTC Store Clock und Zeitdifferenz abfragen.
- USR9178Y - Lokalen Store Clock in UTC Store Clock und Zeitdifferenz konvertieren.
- USR9178Z - UTC Store Clock und Zeitdifferenz in lokalen Store Clock konvertieren.

USR9201N

Die API USR9201N ist der Nachfolger der API **USR1023N**. Sie unterstützt Store Clock mit gleitendem Jahr-Fenster (1971 - 2114), smarten und erweiterten Store Clock (1900 - 38434).

Als Eingabe erwartet die API USR9201N eines der folgenden Formate:

- Natural-Datum und Uhrzeit,
- Natural-Uhrzeit,
- Natural-Datum,
- Mikrosekunden (seit 01.01.1900),
- Store Clock mit gleitendem Jahr-Fenster,
- erweiterten Store Clock oder
- smarten Store Clock.

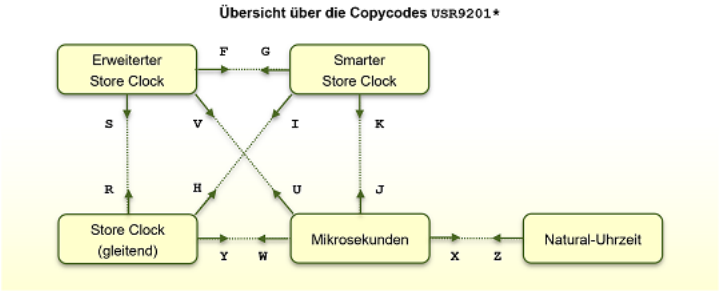
Die API USR9201N konvertiert die Eingabe in alle anderen Formate.

Zusätzlich stehen zur API USR9201N die folgenden Copycodes zur Verfügung:

Copycode	Funktion
USR9201D	Zeitdifferenz von zwei Store Clock Werten mit gleitendem Jahr-Fenster berechnen.
USR9201F	Erweiterten Store Clock in smarten Store Clock konvertieren.
USR9201G	Smarten Store Clock in erweiterten Store Clock konvertieren.
USR9201H	Store Clock mit gleitendem Jahr-Fenster in smarten Store Clock konvertieren.
USR9201I	Smarten Store Clock in Store Clock mit gleitendem Jahr-Fenster konvertieren.
USR9201J	Mikrosekunden in smarten Store Clock konvertieren.

Copycode	Funktion
USR9201K	Smarten Store Clock in Mikrosekunden konvertieren.
USR9201R	Store Clock mit gleitendem Jahr-Fenster in erweiterten Store Clock konvertieren.
USR9201S	Erweiterten Store Clock in Store Clock mit gleitendem Jahr-Fenster konvertieren.
USR9201U	Mikrosekunden in erweiterten Store Clock konvertieren.
USR9201V	Erweiterten Store Clock in Mikrosekunden konvertieren.
USR9201W	Mikrosekunden in Store Clock mit gleitendem Jahr-Fenster konvertieren.
USR9201X	Mikrosekunden in Natural-Uhrzeit konvertieren.
USR9201Y	Store Clock mit gleitendem Jahr-Fenster in Mikrosekunden konvertieren.
USR9201Z	Natural-Uhrzeit in Mikrosekunden konvertieren.

Anmerkung: Der Copycode USR9201D kann auch benutzt werden, um zwei Store Clock-Werte miteinander zu vergleichen, wenn diese innerhalb des gleitenden Jahr-Fensters (1971 - 2114) erfasst worden sind.



Die Copycodes sind wesentlich schneller als das API Subprogramm USR9201N.

Beispiel - Store Clock mit gleitendem Jahr-Fenster in Mikrosekunden konvertieren

Für einen Test mit 10.000 Konvertierungen benötigt:

- das Subprogramm USR9201N etwa 35 ms und
- der Copycode USR9201Y etwa 5 ms.

58 Ende eines Statements, Programms oder einer Anwendung

Ende eines Statements

Um das Ende eines Statements explizit zu markieren, fügen Sie ein Semikolon (;) zwischen diesem Statement und dem nächsten Statement ein. Dies dient dazu, die Programm-Struktur klarer zu gestalten, ist aber nicht erforderlich.

Ende eines Programms

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

Ausführung einer Anwendung mit einem STOP-Statement beenden

Das `STOP`-Statement dient dazu, die Ausführung einer Natural-Anwendung abzubrechen. Ganz gleich, wo ein `STOP`-Statement in einer Anwendung ausgeführt wird, beendet es sofort die Ausführung der gesamten Anwendung.

Ausführung einer Anwendung mit einem TERMINATE-Statement beenden

Das `TERMINATE`-Statement bricht die Ausführung der Natural-Anwendung ab und beendet die Natural-Session.

59 Verarbeitung von Anwendungsfehlern

Dieser Abschnitt beschreibt die zwei grundlegenden Verfahren, die Natural für die Behandlung von Anwendungsfehlern bietet: Standardverarbeitung und anwendungsspezifische Verarbeitung. Darüber hinaus beschreibt es, auf welche Weise die anwendungsspezifische Verarbeitung von Fehlern ermöglicht werden kann: durch das Kodieren eines `ON ERROR`-Statement-Blocks innerhalb eines Programmierobjekts oder durch den Einsatz eines separaten Fehlertransaktionsprogramms.

Schließlich enthält dieser Abschnitt noch eine Übersicht über die Natural-Funktionalität, mit der Sie die Fehlerverarbeitung durch Natural konfigurieren, Informationen über einen Fehler abrufen oder einen Anwendungsfehler verarbeiten oder bereinigen können.

Informationen zur Behandlung von Fehlern in einer Natural RPC-Umgebung siehe *Handling Errors* in the *Natural Remote Procedure Call*-Dokumentation.

Naturals Standard-Fehlerverarbeitung

Wenn in einer Natural-Anwendung ein Fehler auftritt, geht Natural standardmäßig folgendermaßen vor:

1. Natural beendet die Ausführung des zurzeit laufenden Anwendungsobjekts;
2. Natural gibt eine Fehlermeldung aus;
3. Natural kehrt zum Kommandoeingabe-Modus zurück.

„Kommandoeingabe-Modus“ bedeutet, dass in Abhängigkeit von Ihrer jeweiligen Natural-Konfiguration das Natural-Hauptmenü, die `NEXT`-Zeile oder ein benutzerdefiniertes Einstiegsmenü erscheinen kann.

Die angezeigte Fehlermeldung enthält die Natural-Fehlernummer, den zugehörigen Meldungstext sowie das betroffene Natural-Objekt und die Nummer der Zeile, in der der Fehler aufgetreten ist.

Da die Ausführung des Anwendungsobjekts beendet wird, kann der Status von anhängigen Datenbanktransaktionen von Maßnahmen betroffen sein, die durch die Einstellungen der Profilparameter `ETEOP` und `ETIO` bedingt sind. Falls Natural (infolge der Einstellungen dieser Parameter) kein `END TRANSACTION`-Statement ausgegeben hat, dann wird bei der Rückkehr in den Kommandoeingabe-Modus ein `BACKOUT TRANSACTION`-Statement ausgegeben.

Anwendungsspezifische Fehlerverarbeitung

Wenn die Standard-Fehlerverarbeitung nicht den Erfordernissen Ihrer Anwendung entspricht, können Sie die Verarbeitung anwendungsspezifisch anpassen. Mögliche Gründe hierfür können beispielsweise sein:

- Informationen zum Fehler sollen zwecks weiterer Untersuchung durch den Anwendungsentwickler gespeichert werden.
- Die Ausführung der Anwendung soll, falls möglich, nach der Fehlerbehebung fortgesetzt werden.
- Es ist eine spezifische Transaktionsbehandlung nötigt.

Da nach Auftreten eines Fehlers die Ausführung des betroffenen Anwendungsobjekts beendet wird, kann der Status von anhängigen Datenbanktransaktionen von Maßnahmen betroffen sein, die durch die Einstellungen der Profilparameter `ETEOP` und `ETIO` ausgelöst werden. Deshalb muss die weitere Transaktionsbehandlung (mittels `END TRANSACTION`- oder `BACKOUT TRANSACTION`-Statement) über die Fehlerverarbeitung der Anwendung erfolgen.

Um eine anwendungsspezifische Fehlerverarbeitung zu ermöglichen, haben Sie folgende Möglichkeiten:

- Sie können innerhalb eines Programmierobjekts einen `ON ERROR`-Statement-Block kodieren.
- Sie können ein separates Fehlertransaktionsprogramm verwenden.

Diese Möglichkeiten werden in den folgenden Abschnitten behandelt.

Verwendung eines ON ERROR-Statement-Blocks

Sie können das `ON ERROR`-Statement verwenden, um zur Ausführungszeit auftretende Fehler in einer Anwendung an der Stelle abzufangen, an der ein Fehler auftritt.

Aus einem solchen `ON ERROR`-Statement-Block heraus kann die Ausführung der Anwendung auf der aktuellen Ebene oder auf einer übergeordneten Ebene wieder aufgenommen werden.

Außerdem können Sie ein `ON ERROR`-Statement in mehreren Objekten einer Anwendung angeben, um Fehler, die auf untergeordneten Ebenen aufgetreten sind, zu verarbeiten. Auf diese Weise können Sie die Fehlerverarbeitung exakt auf die Erfordernisse Ihrer Anwendung zuschneiden.

Verlassen eines `ON ERROR`-Statement-Blocks

Um einen `ON ERROR`-Statement-Block zu verlassen, können Sie eines der folgenden Statements angeben:

- `RETRY`

  Die Ausführung der Anwendung wird auf der aktuellen Ebene wieder aufgenommen.

- `ESCAPE ROUTINE`

  Es wird davon ausgegangen, dass die Fehlerverarbeitung abgeschlossen ist, und die Ausführung der Anwendung wird auf der übergeordneten Ebene wieder aufgenommen.

- `FETCH`

  Es wird davon ausgegangen, dass die Fehlerverarbeitung abgeschlossen ist, und das beim `FETCH`-Statement angegebene Programm wird ausgeführt.

  `STOP`

  Natural stoppt die Ausführung des betroffenen Programms, beendet die Anwendung und kehrt zum Kommandoeingabe-Modus zurück.

- `TERMINATE`

  Die Ausführung der Natural-Anwendung wird gestoppt, und die Natural-Session wird beendet.

Fehlerverarbeitungsregeln

- Wird die Ausführung des `ON ERROR`-Statement-Blocks nicht durch eines der oben genannten Statements beendet, dann wird der Fehler an das Natural-Objekt auf der übergeordneten Ebene durchgereicht, damit er durch einen dort vorhandenen `ON ERROR`-Statement-Block verarbeitet wird.
- Falls keines der Objekte auf einer der übergeordneten Ebenen einen `ON ERROR`-Statement-Block enthält, falls aber ein Fehlertransaktionsprogramm (wie im folgenden **Abschnitt** beschrieben) angegeben ist, erhält dieses Fehlertransaktionsprogramm die Kontrolle.
- Falls keines der Objekte auf einer der übergeordneten Ebenen einen `ON ERROR`-Statement-Block enthält und falls dort kein Fehlertransaktionsprogramm angegeben ist, dann greift die Standard-Fehlerverarbeitung von Natural wie **oben** beschrieben.

Verwendung eines Fehlertransaktionsprogramms

Sie können an den folgenden Stellen ein Fehlertransaktionsprogramm angeben:

- Im Profilparameter `ETA`.
- Im Natural Security Library Profile, falls Natural Security installiert ist; siehe *Components of a Library Profile* in der *Natural Security*-Dokumentation.

- Innerhalb eines Natural-Objekts, indem Sie dort mittels eines ASSIGN-, COMPUTE- oder MOVE-Statements den Namen des Fehlertransaktionsprogramms der Systemvariablen *ERROR-TA als Wert zuordnen.

Wenn Sie während der Natural-Session den Namen eines Fehlertransaktionsprogramms der Systemvariablen *ERROR-TA zuweisen, dann wird durch diese Zuweisung ein mittels Profilparameter ETA angegebenes Fehlertransaktionsprogramm ersetzt. Aber ganz gleich, ob Sie den Profilparameter ETA verwenden oder der Systemvariablen *ERROR-TA einen Wert zuweisen, die Namen von Fehlertransaktionsprogramm werden nicht gespeichert und werden von Natural nicht für die verschiedenen Ebenen der Aufruf-Hierarchie wieder hergestellt. Darum wird, wenn Sie den Namen des Programms in einem Natural-Objekt der Systemvariablen *ERROR-TA zuweisen, dieses Programm aufgerufen, um jeden Fehler zu verarbeiten, der nach dieser Zuweisung in der aktuellen Natural-Session auftritt.

Einerseits wird also, wenn Sie ein Fehlertransaktionsprogramm mit dem Profilparameter ETA angeben, eine Fehlertransaktion für die ganze Natural-Session definiert, ohne dass die Notwendigkeit besteht, innerhalb von Natural-Objekten Einzelzuweisungen vorzunehmen. Anderseits bietet das Verfahren, ein Programm der Systemvariablen *ERROR-TA zuzuweisen, mehr Flexibilität und gestattet es Ihnen zum Beispiel, in verschiedenen Zweigen der Anwendung verschiedene Fehlertransaktionsprogramme zu benutzen.

Wenn die Systemvariable *ERROR-TA zurückgesetzt (leer) wird, dann wird wie **oben** beschrieben Naturals Standard-Fehlerverarbeitung durchgeführt.

Wenn ein Fehlertransaktionsprogramm angegeben ist und ein Anwendungsfehler auftritt, wird die Ausführung der Anwendung beendet. Das angegebene Fehlertransaktionsprogramm erhält dann die Kontrolle, um eine der folgenden Maßnahmen auszuführen:

- Analyse des Fehlers
- Protokollieren der Fehlerinformationen
- Beenden der Natural-Session
- Fortsetzen der Anwendungsausführung durch Aufrufen eines Programms mittels FETCH-Statement.

Da das Fehlertransaktionsprogramm die Kontrolle so erhält, als wenn es im Kommandoeingabe-Modus eingegeben worden wäre, ist es nicht möglich, die Ausführung der Anwendung in einem der Natural-Objekte, die zum Zeitpunkt des Auftretens des Fehlers aktiv waren, wieder aufzunehmen.

Wenn ein Syntaxfehler auftritt und der Natural-Profilparameter SYNERR auf ON gesetzt ist, erhält das Fehlertransaktionsprogramm auch die Kontrolle.

Fehlertransaktionsprogramme müssen sich in einer Library befinden, in der Sie zurzeit angemeldet sind, oder in einer aktuellen Steplib Library.

Wenn ein Fehler auftritt, führt Natural ein STACK TOP DATA-Statement aus und legt folgende Informationen oben auf dem Natural-Stack ab:

Stack-Daten	Format/Länge	Beschreibung	
Fehlernummer	N4	Natural-Fehlernummer. **Anmerkung:** Wenn der Session-Parameter SG auf ON gesetzt ist, dann ist Format/Länge N5.	
Zeilennummer	N4	Nummer der Zeile, in welcher der Fehler aufgetreten ist . Wenn der Status C oder L ist, dann ist die Zeilennummer 0.	
Status	A1	Status-Code:	
		C	Kommandoverarbeitungsfehler
		L	Logon-Verarbeitungsfehler
		O	Objekt-(Ausführungs-)Zeitfehler
		R	Fehler auf einem Remote Server (in Verbindung mit dem Natural RPC)
		S	Syntaxfehler
Objektname	A8	Name des Natural-Objekts, in dem der Fehler aufgetreten ist.	
Programmebene	N2	Nummer der Programmebene des Natural-Objekts, auf der der Fehler aufgetreten ist. Wenn der Natural-Profilparameter SYNERR auf ON gesetzt ist und zur Laufzeit ein Syntaxfehler auftritt, dann ist die Nummer der Programmebene 0. Wenn ein Natural-Laufzeitfehler auftritt und die Nummer der Programmebene ist größer als 99, dann wird der Wert 99 auf den Natural-Stack gelegt und der aktuelle Wert wird in den zusätzlichen Stack-Daten als „Programmebene, verbessert“ abgelegt.	
Wenn ein Natural-Laufzeitfehler auftritt und die Nummer der Programmebene des Natural-Objekts größer als 99 ist:			
Programmebene, verbessert	I4	Aktuelle Nummer der Programmeben (512 maximal).	
Wenn ein Natural-Syntaxfehler zur Kompilierungszeit auftritt und der Profilparameter SYNERR auf ON gesetzt ist:			
Fehlerposition	N3	Position des den Fehler verursachenden Bestandteils in der Source-Zeile	
Bestandteillänge	N3	Länge des den Fehler verursachenden Bestandteils.	

Diese Informationen können in einem Fehlertransaktionsprogramm mittels eines INPUT-Statements abgefragt werden.

Beispiel:

```
DEFINE DATA LOCAL
1 #ERROR-NR           (N5)
1 #LINE               (N4)
1 #STATUS-CODE        (A1)
1 #PROGRAM            (A8)
1 #LEVEL              (N2)
1 #LEVELI4            (I4)
1 #POSITION-IN-LINE   (N3)
1 #LENGTH-OF-ERRTOKEN (N3)
END-DEFINE
IF *DATA > 6 THEN       /* SYNERR = ON and a syntax error occurred
  INPUT
    #ERROR-NR
    #LINE
    #STATUS-CODE
    #PROGRAM
    #LEVEL
    #POSITION-IN-LINE
    #LENGTH-OF-ERRTOKEN
ELSE
  INPUT                 /* other error
    #ERROR-NR
    #LINE
    #STATUS-CODE
    #PROGRAM
    #LEVEL
    #LEVELI4
END-IF
WRITE #STATUS-CODE
* DECIDE ON FIRST VALUE OF STATUS-CODE
*  ... /* process error
* END-DECIDE
END
```

Einige der oben auf dem Natural-Stack abgelegten Informationen entsprechen den Inhalten von Systemvariablen, welche in einem `ON ERROR`-Statement-Block zur Verfügung stehen.

Stack-Daten	Entsprechende Systemvariable im ON ERROR-Statement-Block
Natural-Fehlernummer	`*ERROR-NR`
Nummer der Zeile, in welcher der Fehler aufgetreten ist	`*ERROR-LINE`
Name des Natural-Objekts, in dem der Fehler aufgetreten ist	`*PROGRAM`
Nummer der Programmebene, auf der der Fehler aufgetreten ist	`*LEVEL`

Regeln unter Natural Security

Wenn Natural Security installiert ist, gelten zusätzliche Regeln für die Verarbeitung von Fehlern, die beim Anmelden auftreten. Weitere Informationen siehe *Transactions* in der *Natural Security*-Dokumentation.

Funktionalität für die Fehlerverarbeitung

Natural bietet Ihnen umfangreiche Funktionalität, die Sie im Zusammenhang mit der Fehlerverarbeitung verwenden können. Sie können damit

- das Verhalten von Natural bei der Fehlerverarbeitung konfigurieren,
- Informationen über aufgetretene Fehler abrufen,
- Unterstützung bei der Verarbeitung dieser Fehler anfordern,
- Unterstützung bei der Bereinigung von Anwendungsfehlern erhalten.

Diese Funktionalität umfasst spezielle:

- **Profilparameter**
- **Systemvariablen**
- **Terminalkommandos**
- **Systemkommandos**
- **Anwendungsprogrammierschnittstellen (APIs)**

Profilparameter

Folgende Profilparameter beeinflussen das Verhalten von Natural im Fehlerfall:

Profilparameter	Zweck
`CC`	Fehlerverarbeitung im Batch-Modus
`CPCVERR`	Codepage-Umsetzungsfehler
`DBGERR`	Automatischer Debugger-Start bei Laufzeitfehler
`DU`	Dump-Erstellung
`DUE`	Dump-Erstellung, fehlerspezifisch
`ETA`	Fehlertransaktionsprogramm
`ETEOP`	Ausgabe eines `END TRANSACTION`-Statements bei einem End of Program
`ETIO`	Ausgabe eines `END TRANSACTION`-Statements bei Terminal-Eingabe/Ausgabe
`MADIO`	Maximale Anzahl der DBMS-Aufrufe zwischen Bildschirm-Ein-/Ausgaben
`MAXCL`	Maximale Anzahl an Programmaufrufen

Profilparameter	Zweck
RCFIND	Maßnahme bei Adabas Response Code 113 beim FIND-Statement
RCGET	Maßnahme bei Adabas Response Code 113 beim GET-Statement
SYNERR	Übergabe von Syntaxfehlern an das Fehlertransaktionsprogramm
ZD	Division durch Null

Systemvariablen

Die folgenden anwendungsbezogenen Systemvariablen können verwendet werden, um einen Fehler zu lokalisieren oder um den Namen des Programms zu erhalten bzw. anzugeben, das die Kontrolle im Fehlerfall erhalten soll:

Systemvariable	Inhalt
*ERROR-LINE	Source-Zeilennummer des Statements, das den Fehler verursacht hat. Siehe **Beispiel 1**.
*ERROR-NR	Fehlernummer des Fehlers, der eine abzuarbeitende ON ERROR-Bedingung verursacht hat.
*ERROR-TA	Name des Programms, das die Kontrolle im Fehlerfall erhalten soll. Siehe **Beispiel 2**.
*LEVEL	Nummer der Ebene des Natural-Objekts, auf der der Fehler aufgetreten ist.
*LIBRARY-ID	Name der Library, in welcher der Benutzer zurzeit angemeldet ist.
*PROGRAM	Name des Natural-Objekts, das zurzeit ausgeführt wird. Siehe **Beispiel 1**.

Beispiel 1:

```
...
    /*
    ON ERROR
      IF *ERROR-NR = 3009 THEN
        WRITE 'LAST TRANSACTION NOT SUCCESSFUL'
            / 'HIT ENTER TO RESTART PROGRAM'
        FETCH 'ONEEX1'
      END-IF
      WRITE 'ERROR' *ERROR-NR 'OCCURRED IN PROGRAM' *PROGRAM
            'AT LINE' *ERROR-LINE
      FETCH 'MENU'
    END-ERROR
    /*
...
```

Beispiel 2:

```
...
    *ERROR-TA := 'ERRORTA1'
    /* from now on, program ERRORTA1 will be invoked
    /* to process application errors
...
    MOVE 'ERRORTA2' TO *ERROR-TA
    /* change error transaction program to ERRORTA2
...
```

Weitere Informationen zu diesen Systemvariablen finden Sie in den entsprechenden Abschnitten der *Systemvariablen*-Dokumentation.

Terminalkommandos

Das folgende Terminalkommando beeinflusst das Verhalten von Natural im Fehlerfall:

Terminalkommando	Zweck
%E=	Fehlerbehandlung ein-/ausschalten

Systemkommandos

Die folgenden Systemkommandos liefern zusätzliche Informationen über eine Fehlersituation bzw. dienen zum Aufrufen von Utilities zur Fehlerbereinigung bei oder zum Protokollieren von Datenbankaufrufen:

Systemkommando	Zweck
LASTMSG	Anzeige von zusätzlichen Informationen zu der zuletzt aufgetretenen Fehlersituation.
RPCERR	Gilt nur in einer Natural RPC-Umgebung (Remote Procedure Call). Anzeige von Natural-, EntireX Broker- und EntireX RPC-Server-Fehlern, die zuletzt während einer RPC-Sitzung aufgetreten sind. Weitere Informationen siehe *Programm RPCERR verwenden* im Abschnitt *Betrieb einer Natural RPC-Umgebung* in der *Natural RPC (Remote Procedure Call)*-Dokumentation.
TECH	Anzeige von technischen und sonstigen Informationen über Ihre Natural-Session, zum Beispiel Informationen über den zuletzt aufgetretenen Fehler.
TEST	Ruft den Debugger auf.
TEST DBLOG	Ruft die DBLOG-Utility auf, die zum Protokollieren von Datenbankaufrufen verwendet wird.

Anwendungsprogrammierschnittstellen

Die folgenden Anwendungsprogrammierschnittstellen (APIs) stehen generell zur Verfügung, um zusätzliche Informationen über eine Fehlersituation abzurufen oder um eine Fehlertransaktion einzurichten.

API	Zweck
RPCINFO	Gilt nur in einer Natural RPC-Umgebung (Remote Procedure Call). Dieses Subprogramm ruft Natural-, EntireX Broker- und EntireX RPC-Server-Fehler ab, die zuletzt während einer RPC-Sitzung aufgetreten sind. Weitere Informationen finden Sie unter *Subprogramm RPCINFO verwenden* in der *Natural RPC* (Remote Procedure Call)-Dokumentation.
USR0040N	Art des letzten Fehlers abfragen
USR0622N	Zähler in ON ERROR-Statement-Block zurücksetzen.
USR1016N	Fehlerebene bei Copycode zeigen
USR1037N	Informationen über Natural ABEND-Daten
USR1041N	Beispielprogramm für eine Fehlertransaktion (*ERROR-TA)
USR2001N	Informationen über letzten Fehler auslesen
USR2006N	Ausführliche Informationen zur Meldung holen
USR2007N	Informationen zum Standard-RPC-Server setzen/abrufen
USR2010N	Informationen über Datenbankfehler zeigen
USR2026N	Technische Informationen holen (TECH)
USR2030N	Dynamische Fehler-Teile :1:,... lesen
USR2034N	System- bzw. Benutzer-Fehlermeldungstext von FNAT bzw. FUSER auslesen
USR3320N	Fehler-Kurztext auf FNAT bzw. FUSER suchen
USR4012N	Anwendungsfehler beim RPC-Server setzen
USR4214N	Information über Programmebene zeigen
USR8202N	Erweiterte Fehlerinformationen zu Fehler NAT3145 holen

Weitere Informationen siehe *SYSEXT Utility* in der *Debugger und Dienstprogramme*-Dokumentation.

Für SQL-Aufrufe stehen die folgenden Anwendungsprogrammierschnittstellen zur Verfügung:

API	Zweck
NDBERR	Liefert Diagnose-Informationen über den zuletzt ausgeführten SQL-Aufruf.
NDBNOERR	Unterbindet die Fehlerbehandlung durch Natural im Falle von Fehlern, die durch den nächsten SQL-Aufruf verursacht werden.

Weitere Informationen siehe *Interface Subprograms* in der *Natural for Db2*-Dokumentation.

60 Kompilierungsaspekte

Der folgende Abschnitt enthält eine Übersicht über die Kompilierungsoptionen und -parameter, die einen Einfluss darauf haben, wie der fertige Quellcode vom Compiler geprüft und wie es zu Natural-internem Objektcode generiert wird, der von Natural-Laufzeitsystem interpretiert und ausgeführt werden kann.

Weitere Informationen zum Natural-Compiler und zu den verschiedenen Systemkommandos zum Kompilieren des Quellcodes siehe Abschnitt *Natural-Compiler* in der *System-Architektur*-Dokumentation.

Compiler-Optionen und Parameter

Kompilierungsoptionen/-parameter können auf folgenden Ebenen angegeben werden:

1. **Statisch**

   im Natural-Parametermodul durch Angabe von

   - Parametermakro `NTCMPO`
   - Profilparameter `FS`, `XREF`

2. **Dynamisch**

   durch Angabe der Profilparameter

   - `CMPO`
   - `FS`, `XREF`

3. **In einer Natural-Session**

   durch Angabe von

   - `COMPOPT`-Systemkommando-Optionen
   - `GLOBALS`-Systemkommando (Session-Parameter `FS`)

4. **Für das aktuelle Natural-Programmierobjekt**

   durch Angabe von

   - einem oder mehreren `OPTIONS`-Statements

     (bietet dieselben Optionen wie beim Systemkommando `COMPOPT` und darüber hinaus die Natural Optimizer Compiler-Optionen);

   - `SET GLOBALS`-Statement (nur die Session-Parameter `LS`, `PS`, `ZP`)

Weitere, den Compiler beeinfussende Parameter

Die Einstellungen der folgenden Profil- bzw. Session-Parameter werden bei der Kompilierung berücksichtigt und können eine Fehlermeldung verursachen, falls der Quellcode davon abweichende Einstellungen enthält:

Parameter-Name	Kurztext	Bemerkung
CFICU	Unicode- und Codepage-Unterstützung	Ausgabe von Unicode-Konstanten in I/O-Statements (zum Beispiel WRITE U'ABC') ist nur erlaubt, wenn CFICU=ON ist.
CP	Name der Standard-Codepage	Gibt die Codepage der Source an.
DC	Dezimalstellenzeichen	Zeichen zur Trennung von Integer/Precision Digits bei der Definition von Feldern des Formats N oder P und Konstanten.
FS	Format-Spezifikation für Benutzervariablen	Erlaubt oder verbietet die Verwendung von Variablen ohne vorherige Definition.
ID	Input-Begrenzungszeichen	Input-Begrenzungszeichen, um zum Beispiel Werte in einer INIT-Klausel voneinander zu trennen.
LS	Zeilenlänge	Angabe der Standard-Zeilenlänge für eine Druckzeile in einem Report (0 - 31). Die Einstellung kann durch ein explizites FORMAT-Statement im Quellcode überschrieben werden.
PS	Länge einer Reportseite	Maximale Anzahl Zeilen, die ein einzelnes I/O-Statement (zum Beispiel WRITE) erzeugen darf. Nichtbeachtung des Grundwertes führt zu einem Fehler. Die Einstellung kann durch ein explizites FORMAT-Statement im Quellcode überschrieben werden.
SF	Spaltenabstand	Abstand zwischen Spalten, die mit einem DISPLAY-Statement ausgegeben werden. Die Einstellung kann durch ein explizites FORMAT-Statement im Quellcode überschrieben werden.
SOSI	Shift-Out/Shift-In-Codes für Doppel-Byte-Zeichensätze	Shift-Zeichen, um einen DBCS-String zu erkennen.
ZP	Anzeige von Nullwerten	Ausgabe von Leerstellen oder 0 für eine numerische Konstante mit Wert Null. Die Einstellung kann durch ein explizites FORMAT-Statement im Quellcode überschrieben werden.

VIII Statements für Internet-Zugang und Parsing

61 Statements für Internet-Zugang und Parsing

Dieses Kapitel gibt einen Überblick über die Natural-Statements `PARSE JSON` und `PARSE XML` für den Zugriff auf das Internet, behandelt grundsätzliche Voraussetzungen für die Benutzung dieser Statements in einer Großrechner-Umgebung, informiert über geltende Einschränkungen und enthält ein Verzeichnis sonstiger Informationsquellen. Um diese Statements vollständig nutzen zu können, ist eine gründliche Kenntnis der zugrunde liegenden Kommunikationsstandards erforderlich.

Verfügbare Statements

Die folgenden Natural-Statements stehen zum Zugriff auf das Internet und auf JSON-/XML-Dokumente zur Verfügung:

- REQUEST DOCUMENT
- PARSE XML
- PARSE JSON

REQUEST DOCUMENT

- Leistungsumfang
- Technische Umsetzung
- Syntax
- Plattform-Unterstützung für REQUEST DOCUMENT
- IPv6-Unterstützung bei REQUEST DOCUMENT

Leistungsumfang

Dieses Statement ermöglicht die Verwendung des Hypertext-Übertragungsprotokolls (Hypertext Transfer Protocol, HTTP) sowie - nur unter z/OS - des sicheren Hypertext-Übertragungsprotokolls (Hypertext Transfer Protocol Secure, HTTPS), um mit einem gegebenen Uniform Resource Identifier (URI) oder Uniform Resource Locator (URL), oder anders ausgedrückt, mittels einer Internet- oder Intranet-Adresse einer Web-Seite, auf Dokumente im Web zuzugreifen.

Das Statement `REQUEST DOCUMENT` implementiert einen HTTP-Client auf Natural-Statement-Ebene. Damit können Anwendungen auf jeden beliebigen HTTP-Server entweder im Intranet oder im Internet zugreifen. Das Statement bietet verschiedene Operanden, mit denen HTTP-Anfragen entsprechend den Erfordernissen der Benutzeranwendung formuliert werden können. Zum Beispiel kann man mit den nach Außen weisenden Operanden benutzerdefinierte HTTP-Nachrichtenköpfe (Header), Form-Daten oder ganze Dokumente an einen HTTP-Server senden. Die nach Innen weisenden Operanden können verwendet werden, um ein Dokument vom Server abzurufen, den gesamten vom Server zurückgesandten HTTP Nachrichtenkopfblock (Header Block) zu sichten oder um die Werte spezieller Nachrichtenköpfe zurückzusenden usw. Über Binär-Format-Operanden können auch binäre Objekte wie zum Beispiel GIF-Dateien mit dem HTTP-Server ausgetauscht werden. Außerdem können Operanden zur Basis-Authentifizierung mit Benutzerkennung und

Passwort angeben werden, deren Inhalt gemäß den HTTP-Standards mittels Base64-Kodierung verschlüsselt übertragen wird.

Natural unterstützt die folgenden REQUEST-METHODs:

- GET - Dokumente und HTTP-Header abrufen.
- HEAD - Nur HTTP-Header abrufen.
- POST - Form-Daten zu einem HTTP-Server übertragen.
- PUT - Eine Datei auf einen HTTP-Server hochladen.

Die Auswertung der REQUEST-METHODs erfolgt normalerweise automatisch anhand der beim REQUEST DOCUMENT-Statement kodierten Operanden. Die dadurch vorgegebene REQUEST-METHOD kann jedoch durch eine explizite Benutzerangabe in einem REQUEST-METHOD Header überschrieben werden.

Zusätzlich zu den zuvor erwähnten Standard-REQUEST-METHODs können in einem REQUEST-METHOD Header folgende Methoden angegeben werden:

- DELETE - Ein Dokument von einem HTTP-Server löschen.
- PATCH - Ein Dokument auf einem HTTP-Server ändern.
- OPTIONS - Die von einem HTTP-Server unterstützten REQUEST-METHODs abrufen.
- TRACE - Die Nachricht abrufen, die durch einen HTTP-Server empfangen wurde.

Bei der Übertragung von Daten mit Hilfe des REQUEST DOCUMENT-Statements erfolgt normalerweise keine Codepage-Umsetzung. Wenn Sie möchten, dass die ausgehenden und/oder eingehenden Daten in einer spezifischen Codepage kodiert werden, können Sie die DATA ALL-Klausel und/oder die RETURN PAGE-Klausel des REQUEST DOCUMENT-Statements benutzen, um dies anzugeben.

Um den Datenaustausch von einem EBCDIC-basierten Großrechner mit HTTP-Servern, die meistens mit UTF-8- or ISO-8859-1-kodierten Daten arbeiten, zu erleichtern, bietet das Statements ENCODED-Klauseln, die eine implizite oder automatische Konvertierung von ausgehenden und eingehenden Dokumentendaten gestatten.

Beispiel

Das folgende Beispiel zeigt, wie dieses Statement benutzt werden kann, um auf ein extern vorliegendes Dokument zuzugreifen:

```
REQUEST DOCUMENT FROM
"http://bolsap1:5555/invoke/sap.demo/handle_RFC_XML_POST"
WITH
USER #User PASSWORD #Password
DATA
NAME 'XMLData' VALUE #Queryxml
NAME 'repServerName' VALUE 'NT2'
RETURN
PAGE #Resultxml
RESPONSE #rc ↩
```

Technische Umsetzung

Die Implementierung des `REQUEST DOCUMENT`-Statement erfolgt im Wesentlichen in zwei Schichten:

- eine unabhängige Laufzeitschicht, in der die gesamte HTTP-Verarbeitung, URL-Analyse, Datenkonvertierung usw. stattfindet, und
- eine Schicht, in der eine umgebungsabhängige Routine die TCP/IP-Kommunikation zwischen Natural und dem HTTP-Server verarbeitet. Die Implementierung dieser Schicht erfolgt auf der Basis von LE (Language Environment) Sockets für z/OS, SMARTS Sockets für Com-plete und den Natural Development Server. Für CICS ist die entsprechende Socket Library im Build-Prozess enthalten.

Natural for z/OS unterstützt nur die HTTP-Protokoll-Version 1.0. Das heißt, dass keine persistente Verbindung zum Server aufrechterhalten wird. Da in fast allen Firmennetzen der Zugang zum Internet vom Client aus über einen Proxy-Server abgewickelt wird, ist es möglich, Natural mit entsprechenden Einstellungen für den Proxy-Server und für den Port zu konfigurieren, auf dem der Proxy-Server läuft. Darüber hinaus besteht die Möglichkeit, Namenssuffixe lokaler Domains (Intranet Sites) anzugeben, auf die direkt statt über den Proxy-Server zugegriffen werden soll. Siehe auch ***Übersicht über Relevante Natural-Parameter*** weiter unten.

Der Proxy-Server, der zwischen dem Client (Benutzer) und dem Internet angeordnet ist, hat folgende Aufgaben: Er empfängt die Anfrage vom Client, leitet sie an den Ziel-Server weiter, speichert das gelieferte Dokument zwischen und leitet es dann an den Client weiter. Die Nutzung eines Proxy-Servers ist von Vorteil, weil er dank Zwischenspeicherung eine verbesserte Performance bewirkt und weil er Sicherheitsprobleme vermeiden hilft (die meisten Proxy-Server fungieren auch als Firewall).

Syntax

Die Syntax des REQUEST DOCUMENT-Statements sowie konkrete Anwendungshinweise finden Sie in der *Statements*-Dokumentation.

Plattform-Unterstützung für REQUEST DOCUMENT

Das REQUEST DOCUMENT-Statement wird auf folgenden Großrechner-Plattformen unterstützt:

- **z/OS:** Batch, TSO, CICS, Com-plete und IMS TM

Darüber hinaus steht dieses Statement auch auf allen von Natural unterstützten Open-Systems-Plattformen zur Verfügung.

IPv6-Unterstützung bei REQUEST DOCUMENT

Das Kontingent der mit Internet-Protokoll-Version 4 (IPv4) verfügbaren Adressen ist fast aufgebraucht. Neue Internet-Adressen werden mit der Internet-Protokoll-Version 6 (IPv6) zur Verfügung gestellt, deren Adressraum Platz für 3,4 x 10^{38} IPv6-Adressen bietet (eine einzelne IPv6-Adresse besteht aus 128 Bits).

Natural unterstützt beide Internet-Protokolle, um die Funktionalität des REQUEST DOCUMENT-Statements auch in einem Internet mit wachsender Anzahl an IPv6-basierten HTTP-Servern zu gewährleisten. Da IPv6 nicht kompatibel mit IPv4 ist, bietet Natural eine zusätzliche Logik und Konfigurationsparameter, um die neue Protokollversion zu unterstützen.

Voraussetzungen für die IPv6-Unterstützung

Bezüglich der zur Unterstützung von REQUEST DOCUMENT-Statements mit IPv6 zu erfüllenden Voraussetzungen siehe *Prerequisites* in *Installation for REQUEST DOCUMENT and PARSE XML Statements* in der Natural *Installation*-Dokumentation für z/OS.

IPv6-Verifizierung beim Start der Natural-Session

Die IPv6-Unterstützung wird mit dem Schlüsselwort-Subparameter RDIPV6 des Profilparameters XML eingeschaltet (siehe *Parameter-Referenz*-Dokumentation). Natural prüft, ob beim Session-Start ein IPv6 TCP/IP-Stack auf dem lokalen Host vorhanden ist. Falls kein IPv6 TCP/IP-Stack auf dem lokalen Host vorhanden ist, gibt Natural eine entsprechende Warnmitteilung aus und schaltet die IPv6-Unterstützung bei der Ausführung des REQUEST DOCUMENT-Statements aus.

IPv4 und IPv6 Adress-Notationen

Die häufigste Art, einen HTTP-Server zu adressieren, besteht in der Angabe des symbolischen Namens, der die IP-Adresse ersetzt, zum Beispiel: *www.mycompany.com*. Sie können aber einen HTTP-Server auch direkt adressieren, indem Sie seine IP-Adresse benutzen.

Die übliche Notation für eine IPv4-Adresse besteht aus einer Gruppe von vier dezimal kodierten Achtbitzeichen, die durch Punkte voneinander abgegrenzt sind, zum Beispiel: 192.168.0.1.

Für die Notation bei IPv6-Adressen gelten (gemäß RFC 4291, IP Version 6 Addressing Architecture) folgende Regeln:

1. Schreiben Sie die IPv6-Adresse als acht Gruppen hexadezimaler Ziffern, und trennen Sie diese durch Doppelpunkte voneinander ab. Zum Beispiel:

```
2031:AF04:87C2:0000:0412:988:7F2C:1C1B
```

2. Führende Nullen innerhalb einer Gruppe können Sie weglassen.
3. Eine einzelne Gruppe oder mehrere (aufeinanderfolgende) Gruppen, die nur aus Nullen bestehen (`0000`), können Sie weglassen, indem Sie stattdessen zwei Doppelpunkte (::) angeben.
4. Regel 3 dürfen Sie nur einmal anwenden.
5. Bei eingebetteten IPv4-Adressen können Sie bei den letzten 4 Bytes die alte Dezimalnotation benutzen, zum Beispiel:

```
0000::FFFF:192.203.55.07
```

Anmerkung: Eingebettete IPv4-Adressen werden von Natural *nicht* unterstützt.

Wenn Sie eine IPv6-Adresse in einem `REQUEST DOCUMENT`-Statement angeben möchten, müssen Sie diese Adresse in eckigen Klammern ([ ]) einschließen, zum Beispiel:

```
http://[2BFC:5022:4081:0000::C:41]:8080
```

Damit eine IPv6-Adresse als lokale Adresse erkannt wird, muss die Präfix-Notation exakt auf die IPv6 No-Proxy-Spezifikation abgebildet werden, d.h., im obigen Beispiel ließe sich die Adresse nicht auf die folgende No-Proxy-Spezifikation abbilden: `2BFC:5022:4081:0::C:41`. Sie können aber bei einer lokalen Site die folgende Präfix-Notation benutzen: `2BFC:5022:4081:0000::`

PARSE XML

- Leistungsumfang PARSE XML
- Technische Umsetzung PARSE XML
- Syntax PARSE XML

Leistungsumfang PARSE XML

Das `PARSE XML`-Statement ermöglicht es, XML-Dokumente von einem Natural-Programm aus zu parsen.

Mit dem `PARSE XML`-Statement wird ein vollständiger XML-Parser in Natural integriert. Damit können Natural-Anwendungen XML-Dokumente parsen und ihren Inhalt auf einfache Weise verarbeiten. Das Statement öffnet eine Verarbeitungsschleife und liefert, wenn eines der Ereignisse aus einer Ereignisliste während des Parse-Prozesses auftritt, den entsprechenden Pfad durch das Dokument, Name und Wert von geparsten Elementen sowie einige Parser-Status-Systemvariablen.

Technische Umsetzung PARSE XML

Für das Parsen von XML-Dokumenten sind die folgenden Parse-Strategien am geläufigsten:

- DOM (Document Object Model), ein objektorientierter Ansatz
- SAX (Simple Access to XML), ein datenstromorientiertes Parse-Verfahren

Die Implementierung des `PARSE XML`-Statements in Natural for z/OS basiert auf der SAX-Methode. Verwendet wird ein Großrechner-Port des SAX-Parsers CentiJSON.

Das Parsen erfolgt intern mit einem UTF-16-kodierten Image des zu parsenden Dokuments. Wird das Dokument nicht in dieser Kodierung angeliefert, erfolgt eine interne Konvertierung nach UTF-16, bevor der Parse-Vorgang beginnt. Dies muss bei der Installation von Natural berücksichtigt werden, zum Beispiel, wenn die Thread-Größe für die TP-Umgebung bestimmt wird.

Die Kodierung des zu parsenden Dokuments wird automatisch geprüft:

1. Zunächst wird auf das Vorhandensein einer BOM (Byte Order Mark), die die Kodierung des Dokuments kennzeichnet, geprüft.
2. Wird keine BOM gefunden, erfolgt eine Prüfung auf ASCII, EBCDIC oder UTF-16 (BE oder LE: Big Endian oder Little Endian).
3. Wird eine EBCDIC- oder ASCII-Kodierung festgestellt, dann wird nach einer Kodierungsverarbeitungsanweisung (PI, Processing Instruction) gesucht.

   Kann keine Kodierung festgestellt werden, dann wird eine entsprechende Fehlermeldung ausgegeben und der Parse-Vorgang beendet. Intern arbeitet der Parser mit UTF-16BE, deshalb wird das zu parsende Dokument immer in diese Kodierung konvertiert, bevor es an den Expat-Parser weitergeleitet wird.
4. Falls eine Kodier-PI gefunden wird, gelten die folgenden Standardvorgaben:
   - Für ASCII wird UTF-8 als Kodierung angenommen.

- Für EBCDIC wird die Default-Codepage (siehe Systemvariable `*CODEPAGE`) als Kodierung angenommen.

Der Parse-Vorgang selbst läuft in zwei Phasen ab:

- In der ersten Phase wird der Parser wiederholt gerufen, um einen genau festgelegten Satz an Callback-Einträgen anzukündigen. Diese Einträge werden vom Parser jedes Mal vorgenommen, wenn in dem zurzeit geparsten Dokument ein entsprechendes Element vorgefunden wird. Ein solches Ereignis, das einen Callback auf den entsprechenden Eintrag auslöst, ist zum Beispiel das Auftreten einer Start-Markierung (Tag). Die Callback-Einträge bilden die Natural-Laufzeitlogik für die Ausführung des Parse-Vorgangs.
- In der zweiten Phase erfolgt der eigentliche Parse-Vorgang. Der Parser wird mit dem zu parsenden Dokument als Eingabe-Operand aufgerufen. Jetzt wird jedes Element geparst, und für jeden Elementtyp wird die entsprechende Callback-Routine aufgerufen. Dann verarbeitet die Natural-Laufzeitumgebung das zurückgegebene Element, aktualisiert die Rückgabe-Operanden und beginnt mit der Ausführung der Parse-Schleife zum Verarbeiten dieser Operanden. Danach wird der Parser erneut gestartet, um den Parse-Vorgang fortzusetzen. Der Parse-Vorgang wird beendet, wenn das Dokument vollständig geparst ist oder wenn im aktuellen Dokument ein XML-Syntaxfehler auftritt, was bedeutet, dass das Dokument formell nicht in Ordnung ist.

Anmerkung: Aus technischen Gründen unterstützt Natural for z/OS keine verschachtelten Parse-Schleifen.

Syntax PARSE XML

Die Syntax des `PARSE XML`-Statements und konkrete Anwendungshinweise finden Sie in der *Statements*-Dokumentation.

Plattform-Unterstützung für PARSE XML

Das `PARSE XML`-Statement wird auf folgenden Großrechner-Plattformen unterstützt:

- **z/OS:** Batch, TSO, CICS, Com-plete, IMS TM *

* Siehe ***Einschränkung bezüglich IMS TM*** weiter unten.

Darüber hinaus steht Ihnen dieses Statement auch auf allen von Natural unterstützten Linux-, UNIX- und Windows-Plattformen zur Verfügung.

PARSE JSON

Leistungsumfang PARSE JSON

Das `PARSE JSON`-Statement ermöglicht es, JSON-Dokumente innerhalb eines Natural-Programm zu parsen.

Das `PARSE JSON`-Statement integriert in Natural einen vollständigen JSON-Parser, der es Natural-Anwendungen ermöglicht, JSON-Dokumente zu parsen. Bei der Ausführung des PARSE JSON-Statements wird eine Verarbeitungsschleife ausgelöst, die den Pfad, den Namen und den Wert der geparsten Elemente sowie spezifische Systemvariablen für den Parser-Status liefert.

Technische Umsetzung PARSE JSON

Für das Parsen von JSON-Dokumenten sind die folgenden Parse-Strategien oder Modelle am geläufigsten:

- DOM (Document Object Model), ein objektorientierter Ansatz
- SAX (Simple Access to XML), ein datenstromorientiertes Parse-Verfahren

Die Implementierung des `PARSE JSON`-Statements in Natural for z/OS basiert auf der SAX-Methode, unter Verwendung eines Großrechner-Ports des SAX Parsers CentiJSON..

Das Parsen erfolgt intern mit einem UTF-8-kodierten Image des zu parsenden Dokuments. Wird das Dokument nicht in dieser Kodierung zur Verfügung gestellt, erfolgt eine interne Konvertierung nach UTF-8-kodierten, bevor der Parse-Vorgang beginnt. Das System prüft automatisch die Kodierung des zu parsenden Dokuments.Die Kriterien für diese Validierung und die Wandlung in UTF-8 basieren auf den folgenden Richtlinien:

1. Zunächst prüft das System auf das Vorhandensein einer BOM (Byte Order Mark), die die Kodierung des Dokuments kennzeichnet.
2. Wird keine BOM gefunden, so wird `ENCODED [IN] CODEPAGE` (*operand2*) berücksichtigt.
3. Ist kein `ENCODED [IN] CODEPAGE` (*operand2*) angegeben, so gelten die folgenden Richtlinien:
   - Wenn der Datentyp des JSON-Dokuments (*operand1*) mit „B“ (Binary) angegeben ist, wird es als UTF-8 behandelt und es wird keine Codepage-Wandlung durchgeführt.
   - Wenn der Datentyp des JSON-Dokuments (*operand1*) als „U“ (Unicode) angegeben ist, wird es als UTF-16 behandelt und eine Codepage-Wandlung nach UTF-8 durchgeführt.

- Wenn der Datentyp des JSON-Dokuments (*operand1*) als „A“ (Alphanumerisch) angegeben ist, wird die Codepage für das JSON-Dokument (*operand1*) anhand der folgenden Kriterien für die Wandlung des JSON-Dokuments (*operand1*) in UTF-8 ermittelt:
  - Die Standard-Codepage wird verwendet, wenn der Natural-Profilparameter `CP` definiert ist. Die Standard-Codepage kann durch die Systemvariable `*CODEPAGE` überprüft werden.
  - Wenn keine Standard-Codepage verfügbar ist, wird die Fehlermeldung NAT1328 ausgegeben.

Anmerkung: Verschachtelte Parse-Schleifen sind in Natural for z/OS nicht zulässig.

Syntax PARSE JSON

Die Syntax des `PARSE JSON`-Statements und konkrete Anwendungshinweise finden Sie in der *Statements*-Dokumentation.

Plattform-Unterstützung für PARSE JSON

Das `PARSE JSON`-Statement wird auf folgenden Großrechner-Plattformen unterstützt:

z/OS Batch, TSO, CICS, Com-plete und IMS TM.

Siehe ***Einschränkung bezüglich IMS TM*** weiter unten.

Darüber hinaus steht Ihnen dieses Statement auch auf allen von Natural unterstützten Linux- und Windows-Plattformen zur Verfügung.

Generelle Voraussetzungen

Dieser Abschnitt beschreibt die generellen Voraussetzungen, die erfüllt sein müssen, um die Natural-Statements `REQUEST DOCUMENT` und `PARSE XML` oder `PARSE JSON` benutzen zu können.

- Installationserfordernisse erfüllen
- Grundsätzliche Profileinstellungen vornehmen
- REQUEST DOCUMENT und PARSE XML aktivieren/deaktivieren

- Unicode-Unterstützung einschalten

Installationserfordernisse erfüllen

Damit die Natural-Statements `REQUEST DOCUMENT` und `PARSE XML` oder `PARSE JSON` überhaupt benutzt werden können, müssen zunächst die in der *Installation*-Dokumentation beschrieben Installationsschritte ausgeführt werden, siehe:

- *Installation for REQUEST DOCUMENT and PARSE JSON and XML Statements on z/OS*.
- *Installation for REQUEST DOCUMENT and PARSE JSON and XML Statements*.

Da die Statements `REQUEST DOCUMENT` und `PARSE XML` oder `PARSE JSON` zumindest intern immer Daten von einer Kodierung in eine andere zu konvertieren haben, muss Natural mit ICU-Support betrieben werden. Dazu muss die ICU Library installiert sein.

Damit `REQUEST DOCUMENT` oder `PARSE XML` ausgeführt werden kann, müssen folgende Voraussetzungen erfüllt sein:

- Ein TCP/IP-Stack muss verfügbar und für die Ausführungsumgebung freigegeben sein.
- Ein Domain Name System Server oder DNS Services müssen in der Ausführungsumgebung vorhanden sein, um die Internet-Adressauflösungsanforderungen (Funktion `gehthostbyname`) aufzulösen.
- Ein Natural-Treiber muss installiert und für LE (Language Environment, in IBM-Umgebungen) freigegeben sein,
- Die Unterstützung von HTTPS unter Com-plete erfordert APS Version 2.7.2 Patch Level 16.

Grundsätzliche Profileinstellungen vornehmen

Übersicht über die relevanten Natural-Parameter

Nachfolgend erhalten Sie eine Übersicht über die Natural-Profil- und/oder Session-Parameter, die die Unterstützung der Statements `REQUEST DOCUMENT` and/or `PARSE XML` ein- oder ausschalten bzw. anderweitig beeinflussen.

Anmerkung: Es gibt keinen spezifischen Profilparameter zur Aktivierung oder Deaktivierung des Statements `PARSE JSON`. Ab Natural Version 9.2.3 ist das Statements `PARSE JSON` standardmäßig enthalten.

Parameter	Zweck
XML	Dieser Natural-Profilparameter bzw. das entsprechende Parameter-Makro NTXML mit zugehörigen Schlüsselwort-Subparametern dient zum gemeinsamen oder separaten Aktivieren/Deaktivieren der Unterstützung der Statements REQUEST DOCUMENT und PARSE XML. Mit den Schlüsselwort-Subparametern können außerdem verschiedene Optionen eingestellt werden, zum Beispiel getrenntes Aktivieren/Deaktivieren der beiden Statements, Name der Default-Codepage, URL des (Intranet-)Proxy-Servers, Port-Nummer des Proxy, URL und Port-Nummer des (Intranet-)SSL-Proxy-Servers, Name(n) der lokalen Domain(s,) die direkt angesprochen werden soll(en), usw. Als Voraussetzung für die Benutzung des Profilparameters XML bzw. des Parameter-Makros NTXML muss der Profilparameter CFICU auf ON gesetzt sein.
CFICU	Dieser Natural-Profilparameter bzw. das entsprechende Parameter-Makro NTCFICU mit zugehörigen Schlüsselwort-Subparametern dient zum Einschalten der Unicode- und Codepage-Unterstützung.
CP	Dieser Natural-Profilparameter dient zum Festlegen der Default-Codepage für Natural-Daten und Natural-Quellcode.
CPCVERR	Dieser Natural-Profil- und Session-Parameter legt fest, ob ein während des Konvertierens auftretender Konvertierungsfehler zu einer Natural-Fehlermeldung führt oder nicht.

Ausführliche Beschreibungen dieser Parameter finden Sie in der *Parameter-Referenz*-Dokumentation.

REQUEST DOCUMENT und PARSE XML aktivieren/deaktivieren

Um die Unterstützung der Statements REQUEST DOCUMENT und PARSE XML für die *aktuelle* Sitzung einzuschalten:

1 Um beide Statements *gemeinsam* einzuschalten, setzen Sie den Natural-Profilparameter XML (oder das entsprechende Parameter-Makro NTXML) und außerdem die Schlüsselwort-Subparameter RDOC und PARSE auf ON.

Oder:

Um die Unterstützung der Statements REQUEST DOCUMENT und PARSE XML *einzeln* einzuschalten, setzen Sie nur den betreffenden Schlüsselwort-Subparameter auf ON:

RDOC zur Unterstützung des REQUEST DOCUMENT-Statements

PARSE zur Unterstützung des PARSE XML-Statements

2 Falls die Installationsplattform hinter einer Internet Firewall betrieben wird oder falls der Internet-Datenverkehr über einen Proxy-Server geleitet wird, müssen Sie bei den XML/NTXML Schlüsselwort-Subparametern für Proxy und Proxy Port entsprechende Angaben machen.

Um die Unterstützung der Statements `REQUEST DOCUMENT` und `PARSE XML` für *alle* Sitzung einzuschalten:

- Wenden Sie sich an Ihren System-Administrator, damit er die oben genannten Parameter bzw. Parameter-Makros (siehe *Übersicht über die relevanten Natural-Parameter*) im Natural-Parametermodul hinzufügt und die Werte entsprechend setzt.

Um die Unterstützung der Statements `REQUEST DOCUMENT` und `PARSE XML` auszuschalten:

- Um beide Statements *gemeinsam* auszuschalten, müssen Sie den Natural-Profilparameter `XML` oder das Parameter-Makro `NTXML` auf `OFF` setzen.

  Oder:

  Um die Unterstützung der Statements `REQUEST DOCUMENT` und `PARSE XML` *einzeln* auszuschalten, müssen Sie nur den betreffenden Schlüsselwort-Subparameter auf `OFF` setzen:

  `RDOC` zum Ausschalten der Unterstützung des `REQUEST DOCUMENT`-Statement

  `PARSE` zum Ausschalten der Unterstützung des `PARSE XML`-Statements

Ausführliche Informationen hierzu finden Sie unter *XML - Statements PARSE XML und REQUEST DOCUMENT aktivieren/deaktivieren* in der *Parameter-Referenz*-Dokumentation

Unicode-Unterstützung einschalten

Um die Unicode-Unterstützung einzuschalten:

- Setzen Sie den Profilparameter `CFICU` auf `ON`.

Informationen zu den verschiedenen Optionen, die mit den Schlüsselwort-Subparametern des Profilparameters `CFICU` gesetzt werden können, finden Sie unter *CFICU - Unicode- und Codepage-Unterstützung* in der *Parameter-Referenz*-Dokumentation.

Siehe auch die Abschnitte bezüglich der Statements `PARSE XML` und `REQUEST DOCUMENT` im Abschnitt *Natural-Statements*, der Teil des Abschnitts *Unicode- und Codepage-Unterstützung in der Natural-Programmiersprache* in der *Unicode- und Codepage-Unterstützung*-Dokumentation ist.

HTTPS-Unterstützung für das REQUEST DOCUMENT-Statement unter z/OS

Kurze Einführung in HTTPS

HTTPS (Hypertext Transfer Protocol Secure), das sichere HTTP-Übertragungsprotokoll, ist eine zusätzliche Sicherheitsschicht zwischen dem HTTP- und dem TCP/IP-Protokoll-Stack:

Schicht	Protokoll
Anwendungsschicht	HTTP(S)
Sicherheitsschicht	TLS/SSL
Transportschicht	TCP
Netzwerkschicht	IP

HTTPS wurde eingeführt, um eine Verschlüsselung und eine Authentifizierung von Kommunikationspartnern für eine sichere Datenkommunikation über das Internet zu ermöglichen.

Das HTTPS-URI-Schema wird verwendet um anzeigen, dass die HTTP-Kommunikation gesichert ist. Für die Verschlüsselung der Daten wird das SSL-Protokoll (SSL = Secure Socket Layer) oder sein Nachfolger, das TSL-Protokoll (TSL = Transport Layer Security), verwendet. Die Authentifizierung erfolgt dabei durch den Austausch von Zertifikaten (Certificates), die die Identität der Kommunikationspartner garantieren.

In den meisten Fällen von HTTPS-Kommunikation identifiziert sich jedoch nur der Server gegenüber dem Client. Eine Identifizierung des Clients mittels eines Client-Zertifikats erfolgt relativ selten.

Eine SSL-Kommunikation wird in mehreren Schritten aufgebaut:

- Sie beginnt mit der Identifizierung und Authentifizierung der Kommunikationspartner über das so genannte SSL-Handshake-Protokoll (Client Hello, Server Hello).
- Auf diesen „Handshake“ folgt der Austausch eines symmetrischen Sitzungsschlüssels über eine asymmetrische Verschlüsselung (Private – Public Key Proceeding). Der Public Key, der dabei vom Client verwendet wird, ist ein wesentlicher Bestandteil des Server-Zertifikats.
- Nach erfolgtem „Handshake“ und Austausch der Schlüssel, werden die verschlüsselten Payload Request Messages übermittelt. Der in den vorangegangenen Schritten ausgehandelte symmetrische Sitzungsschlüssel wird zur Ver- bzw Entschlüsselung dieser Mitteilungen verwendet.

Beim HTTPS-Protokoll werden andere Port-Nummern als beim Standard-HTTP-Protokoll verwendet. Während HTTP normalerweise Port 80 verwendet, benutzt HTTPS als Default die Port-Nummer 443.

Der HTTP-Zugang zum Internet von einem Client, der an ein LAN (Local Area Network) angeschlossen ist, wird normalerweise über spezielle HTTP-Server, so genannte Proxy-Server, abgewickelt. Proxy-Server sind Gateways vom LAN. Sie dienen zur Durchführung von Sicherheitsmaßnahmen, stellen Platz zur Zwischenspeicherung (Cache) zu Verfügung, führen Validierungsroutinen oder Filterfunktionen aus und wirken als Firewall. zum Internet. Durch HTTPS gesicherter Internet-Zugang erfolgt meistens über einen eigenen Proxy-Server, der die Verbindungen zu den Remote Servern aufrechterhält. Dieser Proxy ist als „SSL Proxy“ bekannt.

Zertifikate sind binäre Dokumente, die unter anderem auch Informationen über den Besitzer und die Ausgabestelle des Zertifikats, den Public Key für die Verschlüsselung der Schlüsseldaten der Sitzung, das Verfallsdatum und eine digitale Signatur enthalten. Die von HTTPS-Servern vorgelegten Zertifikate sind normalerweise das letzte Glied in einer kompletten Kette von Zertifikaten. Eine solche Kette von Zertifikaten bezeichnet man als Public Key Infrastructure (PKI). Das Zertifikat am oberen Ende der Kette bezeichnet man als Root-Zertifikate. Diese Root-Zertifikate werden grundsätzlich von speziellen Organisationen ausgegeben, die man als Certificate Authorities (CA) bezeichnet. Root-Zertifikate, die von einer CA ausgegeben und signiert werden, nennt man auch CA-(Root-)Zertifikate. Weitere Informationen siehe *HTTP Developers Manual* und andere Informationsquellen im Internet.

HTTPS über AT-TLS

Die HTTPS-Unterstützung für das Natural-Statement `REQUEST DOCUMENT` basiert auf der z/OS Communication Server Component AT-TLS (Application Transparent-Transport Layer Security).

AT-TLS bietet eine TLS/SSL-Verschlüsselung als konfigurierbaren Dienst für Socket-Anwendungen. Realisiert ist es als zusätzliche Schicht über dem TCP/IP Protocol Stack, die die SSL-Funktionalität in nahezu transparentem oder sogar voll transparentem Modus für Socket-Anwendungen nutzt. AT-TLS bietet drei Betriebsarten. Siehe *z/OS Communications Server, IP Programmer's Guide and Reference. Version 1, Release 9*, Chapter 15, IBM manual SC31-8787-09).

Diese Betriebsarten sind:

- **Basic**

  Die Socket-Anwendung läuft unverändert im transparenten Modus, ohne zu „wissen“, dass eine verschlüsselte Kommunikation über AT-TLS durchgeführt wird. Auf diese Weise können Altanwendungen ohne Änderungen am Quellcode im gesicherten Modus laufen.

- **Aware**

  Die Anwendung „weiß“, dass sie im gesicherten Modus läuft, und kann TLS-Statusinformationen abfragen.

- **Controlling**

  Die Socket-Anwendung „weiß“ von der Verwendung von AT-TLS und steuert die Verwendung von AT-TLS-Verschlüsselungsdiensten selbst. Das bedeutet, die Anwendung kann zwischen gesicherter und ungesicherter Kommunikation hin- und herschalten.

Natural for z/OS verwendet den *Controlling*-Modus nur, um den gesicherten Modus für HTTPS-Anfragen einzuschalten, wohingegen HTTP-Anfragen unverschlüsselt bleiben.

Verwaltung von Zertifikaten unter z/OS

Zertifikate, die mit AT-TLS verwendet werden sollen, können unter z/OS auf zwei Arten verwaltet werden. Sie werden in RACF Key Rings oder in Key-Datenbanken gespeichert, die auf dem z/OS UNIX Services-Dateisystem liegen. Welche Vorgehensweise tatsächlich gilt, wird in der Konfigurationsdatei des AT-TLS Policy Agent für den vom Natural HTTPS Client verwendeten z/OS TCP/IP Stack festgelegt.

IBM liefert bei jeder z/OS-Systemauslieferung einen Satz normalerweise verwendeter CA-Root-Zertifikate mit. Wenn Key Rings zum Vorhalten von Server-Zertifikaten benutzt werden sollen, müssen diese Root-Zertifikate durch den Systemadministrator manuell in die Key Rings importiert werden. Wenn IBM neuere Ersatzzertifikate für abgelaufene Root-Zertifikate liefert, müssen alle betroffenen Key Rings entsprechend aktualisiert werden.

Im Gegensatz zu Key Rings enthalten Key-Datenbanken automatisch den aktuellen Satz Root-Zertifikate, nachdem diese neu erstellt worden sind. Jedoch besteht auch bei der Key-Datenbankalternative die Notwendigkeit, immer den neuesten Satz Root-Zertifikate zu pflegen.

Vom Natural HTTPS Client zu verwendende Zertifikate müssen per Flag als „Trusted“ gekennzeichnet werden. Wenn sie Teil der Public Key Infrastructure sind, muss das entsprechende CA-Root-Zertifikat als „Trusted“ gekennzeichnet sein.

Verwendung von RACF Key Rings

In RACF werden digitale Zertifikate in so genannten Key Rings gespeichert. Das RACF-Kommando `RACDCERT` wird verwendet, um Key Rings und die in diesen Key Rings enthaltenen Zertifikate zu verwalten.

Siehe *z/OS Security Server RACF Security Administrator's Guide*, IBM manual SA22-7683-11, und *z/OS Security Server RACF Command Language Reference*, IBM manual SA22-7687-11.

Verwendung von Key-Datenbanken

Alternativ zu RACF können Zertifikate in Key-Datenbanken vorgehalten werden, die in dem z/OS UNIX Services-Dateisystem liegen. Zum Erstellen und Verwalten von Key-Datenbanken muss die `GSKKYMAN`-Utility benutzt werden.

Siehe *z/OS Cryptographic Services PKI Services Guide and Reference*, IBM manual SA22-7693-10.

Einschränkung bezüglich IMS TM

Wenn Sie die Natural-Statements `REQUEST DOCUMENT`, `PARSE JSON` und `PARSE XML` in einer IMS TM-Umgebung verwenden wollen, gelten folgende Einschränkungen:

- Das `PARSE JSON` und `PARSE XML`-Statement kann unter dem TP Monitor IMS TM mit der Einschränkung verwendet werden, dass in einer aktiven `PARSE`-Schleife kein Eingabe-/Ausgabe-Statement benutzt werden darf. Falls in einer aktiven `PARSE`-Schleife eine Ein- oder Ausgabe erfolgt, wird der Fehler NAT0967 ausgegeben.

Bezüglich weiterer Einschränkungen siehe die entsprechenden Anmerkungen in den Statement-Beschreibungen.

Programmbeispiel

Das folgende Programm ist ein Beispiel für die Anwendung der Statements `REQUEST DOCUMENT` und `PARSE XML`.

Weitere Programmbeispiele finden Sie jeweils am Ende einer Statement-Beschreibung in der *Statements*-Dokumentation und in der Natural Library `SYSEXV`.

```
DEFINE DATA
LOCAL
1 #FROM   (A) DYNAMIC
1 #HEADER (A) DYNAMIC
1 #PAGE   (A) DYNAMIC
1 #RC     (I4)
1 #COL    (N8)
1 #COL1   (I4)
1 #COL2   (I4)
1 #COL3   (I4)
1 #LOC    (A30)
1 #CP     (A) DYNAMIC
1 #PATH   (A) DYNAMIC
1 #NAME   (A) DYNAMIC
1 #VALUE  (A) DYNAMIC
```

```
1 #RTERR  (I4)
END-DEFINE
*
ASSIGN #FROM = 'HTTP://SI15.HQ.SAG/autos6.xml'
**
REQUEST DOCUMENT FROM #FROM
RETURN
HEADER ALL #HEADER
PAGE #PAGE    ENCODED    FOR TYPES 'TEXT/XML'
 CODEPAGE ' '
RESPONSE #RC
GIVING #RTERR
**
IF #RC NE 200 /* TEST FOR HTTP RESPONSE 200 = 'OK'
WRITE 'HTTP RESPONSE' #RC 'RECEIVED'
ESCAPE ROUTINE
END-IF
EJECT
PRINT #HEADER
/ '_'(79)
PRINT #PAGE
/ '_'(79)
/ '_'(79)
ASSIGN #CP = *CODEPAGE
EXAMINE #PAGE FOR 'encoding' GIVING POSITION  #COL1
 IF #COL1 GT 0
   EXAMINE #PAGE FOR '?>' GIVING POSITION #COL3
   IF #COL3 GT #COL1
     EXAMINE #PAGE FOR 'ISO-8859-1' GIVING POSITION #COL2
   END-IF
   IF #COL2 GT #COL1 AND #COL2 LT  #COL3
     EXAMINE #PAGE FOR 'ISO-8859-1' REPLACE #CP
   END-IF
 END-IF
PRINT #PAGE
/ '_'(79)
EJECT
PARSE XML #PAGE INTO PATH #PATH NAME #NAME VALUE #VALUE
 PRINT #PATH / 'NAME=' #NAME / 'VALUE=' #VALUE / '_'(79)
END-PARSE
END
```

Anmerkung: Die URL, auf die im obigen Programm zugegriffen wird, adressiert eine Intranet Site und kann nicht aus dem Internet aufgerufen werden.

Ausgabe des Beispielprogramms:

```
HTTP/1.1 200 OK?Date: Thu, 10 Aug 2006 16:26:22 GMT?Server: Apache/1.3.19
?Last-Modified: Thu, 27 Jul 2006 16:44:42 GMT?ETag: "2602c-111-44c8ed7a"
?Accept-Ranges: bytes?Content-Length: 273?Connection: close?Content-Type: text/
xml??
_____________________________________________________________________________
<?xml version="1.0" encoding="ISO-8859-1" ?><autos>?<make></make>?<make>Ford</
make>?<model>Thunderbird</model>?<make>Merceds-Benz</make><model>S400</model><
make>BWM</make><model version="latest">330I</model>?<make><label><company>
Mercedes</company></label></make>?</autos>?
_____________________________________________________________________________
<?xml version="1.0" encoding="IBM01140" ?><autos>?<make></make>?<make>Ford</
make>?<model>Thunderbird</model>?<make>Merceds-Benz</make><model>S400</model><
make>BWM</make><model version="latest">330I</model>?<make><label><company>
Mercedes</company></label></make>?</autos>?
_____________________________________________________________________________
```

```
MORE
```

```
autos
Name= autos
Value=
_____________________________________________________________________________
autos/$
Name=
Value= ?
_____________________________________________________________________________
autos/make
Name= make
Value=
_____________________________________________________________________________
autos/make//
Name= make
Value=
_____________________________________________________________________________
autos/$
Name=
Value= ?
_____________________________________________________________________________
autos/make
Name= make
Value=
VVVV
Name= autos
Value=
_____________________________________________________________________________
autos/$
Name=
Value= ?
_____________________________________________________________________________
autos/make
Name= make
```

```
Value=
_____________________________________________________________________________
```

Häufig gestellte Fragen

Warum muss Codepage-Unterstützung eingeschaltet sein?

In der Installationsdokumentation für Natural for z/OS steht, dass der ICU Handler zum Natural-Nukleus gelinkt sein muss.

PARSE JSON-Statement

Die Codepage-Unterstützung wird benötigt, weil auf Großrechnerplattformen das zu parsende Dokument intern immer nach UTF-8 konvertiert wird (falls das Dokument nicht schon in UTF-8 kodiert ist). Meistens jedoch ist das Dokument nicht in UTF-8, und es findet eine Konvertierung statt. Ausführliche Informationen hierzu finden Sie in der Beschreibung des `PARSE JSON`-Statements in der *Statements*-Dokumentation und im Abschnitt `PARSE XML` in der Dokumentation *Unicode- und Codepage-Unterstützung*.

PARSE XML-Statement

Die Codepage-Unterstützung wird benötigt, weil auf Großrechnerplattformen das zu parsende Dokument intern immer nach UTF-16 konvertiert wird (falls das Dokument nicht schon in UTF-16 kodiert ist). Meistens jedoch ist das Dokument nicht in UTF-16, und es findet eine Konvertierung statt. Ausführliche Informationen hierzu finden Sie in der Beschreibung des `PARSE XML`-Statements

in der *Statements*-Dokumentation und im Abschnitt PARSE XML in der Dokumentation *Unicode- und Codepage-Unterstützung*.

REQUEST DOCUMENT-Statement

Die ICU Library wird benötigt, um von Außen eintreffende HTTP Headers zu interpretieren und nach Außen abgehende HTTP Headers zu konvertieren. Die herein kommenden Headers sind normalerweise in ISO 8859-1 kodiert und müssen auf dem Großrechner immer in die Natural-Default-Codepage (siehe auch Natural-Systemvariable *CODEPAGE) konvertiert werden - auf dem PC ist eine Konvertierung dagegen nicht immer nötig.

Wie sind die XML-Schlüsselwort-Parameter zu benutzen (z.B. RDP und RDNOP)

Auf dem PC führt das REQUEST DOCUMENT-Statement den Browser (Internet Explorer) aus und verwendet dabei die dort vorhandenen Einstellungen.

Auf dem Großrechner muss die URL des (Intranet-)Proxy-Servers, über den alle Anfragen geleitet werden müssen, mit dem NTXML/XML-Schlüsselwort-Subparameter RDP angegeben werden. Mit dem Schlüsselwort-Subparameter RDNOP kann man eine (oder mehrere) lokale Domain(s) festlegen, die direkt und nicht über den Proxy-Server adressiert werden soll(en).

Wie wird das PARSE JSON-Statement aktiviert? Ist dazu ein spezifischer Profilparameter erforderlich?

Es gibt keinen spezifischen Profilparameter zur Aktivierung bzw. Deaktivierung des PARSE JSON-Statements. Ab Natural-Version 9.2.3 wird das PARSE JSON-Statement standardmäßig unterstützt.

Wo erhalte ich Angaben zu Proxy-Server, Port-Nummer und HTTP-Server einer Site?

Informationen über Proxy-Server, Port-Nummer und HTTP-Server einer Site müssen Sie beim Netzwerk-Administrator erfragen.

Sie können aber auch in Ihrem Browser nachsehen, welcher Proxy-Server dort für Ihre Site definiert ist.

Zum Beispiel im Internet Explorer unter: Tools > Internet Options > Lan Settings > Advanced

bzw. bei deutscher Oberfläche unter: Extras > Internetoptionen > Verbindungen > Einstellungen > Einstellungen für lokales Netzwerk (LAN)

Außerdem können Sie im Web nach Tools suchen, die Ihnen diese Informationen liefern. Zum Beispiel hier (ungeprüft): ***http://www.sharewareconnection.com/titles/proxy-settings.htm***

Woran erkenne ich, ob ein Problem mit TCP/IP, mit HTTP oder mit Natural vorliegt?

HTTP Response Codes

Die HTTP-Rückmeldung erfolgt über die `RESPONSE`-Klausel des `REQUEST DOCUMENT`-Statements. Siehe auch *HTTP/HTTPS-Statuscodes "weitergeleitet" und "verweigert"*

TCP/IP-Fehler

Der Nummernkreis für diese Fehler beginnt bei NAT8300.

Insbesondere der Fehler NAT8304 liefert ausführlichere Angaben zu einer fehlgeschlagenen HTTP-Anfrage.

Da die TCP/IP-Fehlermeldungen je nach Installationsumgebung unterschiedlich sein können, ist der in NAT8304 zurück gegebene Text die beste Informationsquelle.

Weitere Informationen:

- Siehe Buffer `RDOCWA` bei Offset 480
- Sehr oft handelt es sich bei diesen Fehlern um ICU-Fehler. Daher wird empfohlen, den Natural-Session-Parameter `CPCVERR` auf `OFF` zu setzen..

Kann ich prüfen, ob ich eine Website von meinem Großrechner aus erreiche, ohne dazu Natural zu benutzen?

Um festzustellen, ob ein Problem an der Natural-Installation liegt oder ob es sich um ein generelleres Problem handelt, können Sie einen `PING` aus TSO heraus absetzen.

Geben Sie zum Beispiel in der TSO Command Shell folgendes Kommando ein:

```
TSO PING www.google.com
```

Die Antwort lautet:

```
CS V1R9: Pinging host WWW.GOOGLE.COM (66.249.91.99)
Ping #1 response took 0.018 seconds.
```

Aus der Natural-Session heraus können Sie den Zugang zu dieser Website mit dem folgenden kleinen Programm testen.

Starten Sie Natural zum Beispiel mit:

```
NATvr CFICU=ON
XML=(ON,RDOC=ON,PARSE=ON,RDP='HTTPPROX.HQ.SAG',RDPPORT=8080,RDNOP='*.EUR.AD.SAG;
*.HQ.SAG;*.SOFTWAREAG.COM')
```

Dabei steht *vr* für die Natural-Release- und Versionsnummer.

Diese Werte einer internen Umgebung und ein Profil wurden benutzt, um es zu speichern. Für Ihre Zwecke müssen Sie die korrekten Werte für die Schlüsselwort-Subparameter `RDP`, `RDPPORT` und `RDNOP` bei Ihrem Netzwerk-Administrator erfragen oder probieren Sie es mit den in Ihrem Browser (Internet Explorer) definierten Werten.

Führen Sie das folgende Programm aus:

```
DEFINE DATA LOCAL
1 #RESULTXML (A) DYNAMIC
1 #RC (I4)
END-DEFINE
REQUEST DOCUMENT FROM "HTTP://WWW.GOOGLE.DE"
RETURN HEADER ALL #HEADER RESPONSE #RC
WRITE #RC
WRITE #HEADER (AL=79)
END
```

Wird NAT2TCP korrekt geladen?

Um zu überprüfen, ob das Modul `NAT2TCP` korrekt geladen wird, können Sie die Utility `SYSPROD` benutzen.

In `SYSPROD` geben Sie das Kommando `SC` (`Display subcomponents`) für das Produkt Natural ein. Wenn Sie die installierten Subkomponenten durchblättern, finden Sie einen Eintrag für `Nat Request Document` (Product ID .... TCP).

Ich erhalte eine Meldung "unsupported coding"

Der Grund für diese Meldung liegt in einem häufig gemachten Benutzerfehler: Ein XML-Dokument wird implizit oder explizit von einer Codepage in eine andere konvertiert, zum Beispiel von ISO-8859-1 in die Codepage, die in der Systemvariablen `*CODEPAGE` vorgefunden wurde. Die Kodier-PI des Dokuments `PI encoding="ISO-8859-1"` wurde jedoch nicht an die geänderte Kodierung angepasst. In diesem Fall beendet der Parser den Vorgang und gibt bereit beim ersten Zeichen des zu parsenden Dokuments eine Fehlermeldung aus.

Wie kann ich bei REQUEST DOCUMENT die Ausgabe des Natural-Fehlers NAT3411 vermeiden?

Setzen Sie den Session-Parameter `CPCVERR` auf `OFF`.

Kann ich selbstsignierte Zertifikate benutzen?

Selbstsignierte Zertifikate können auf einem Intranet-Server unter Verwendung des open ssl sdk für Testzwecke benutzt werden. Nachdem sie in eine Key-Datenbank oder einen RACF Key Ring importiert wurden, müssen sie mit einem "Trusted" Flag versehen werden.

Welche Methode ist für die Pflege von Zertifikaten zu bevorzugen?

Der notwendige Pflegeaufwand für RACF Key Rings scheint viel höher zu sein als bei der Verwendung von Key-Datenbanken. Key Rings müssen für jeden Benutzer erstellt werden, der auf einen HTTPS-Server zugreifen möchte, während Key-Datenbanken von mehreren Benutzer gemeinsam genutzt werden können.

Wie konfiguriere ich TCP/IP für AT-TLS?

Dazu führen Sie folgende Schritte aus:

1. In der TCP/IP-Konfigurationsdatei: Setzen Sie die Option `TTLS` im `TCPCONFIG`-Statement.
2. Konfigurieren und starten Sie den AT-TLS Policy Agent. Dieser Agent wird von TCP/IP bei jeder neuen TCP-Verbindung aufgerufen um zu prüfen, ob es sich um eine SSL-Verbindung handelt.
3. Erstellen Sie die Policy Agent-Datei, die die AT-TLS-Regeln enthält. Diese Datei enthält Regeln zum Festlegen, welche Verbindung über SSL erfolgt. T

Siehe auch *z/OS Communications Server: IP Configuration Guide*, Chapter 18 *Application Transparent Transport Layer Security (AT-TLS) data protection*.

Die folgende Sample Policy Agent-Datei definiert alle nach Außen abgehenden Verbindungen als anwendungsgesteuerte TLS. Dies sollte außer der Natural-`REQUEST DOCUMENT`-Unterstützung keine andere TCP/IP-Anwendung betreffen, weill die Regel als anwendungsgesteuert definiert ist. Das bedeutet, dass die Anwendung den Verbindungsstatus auf SSL setzen kann. Solange wie die Anwendung diesen Status nicht setzt, ist sie nicht betroffen. Die Policy Agent-Datei gestattet es außerdem, die anwendungsgesteuerten SSL-Verbindungen auf bestimmte Ports, Benutzer oder Adressräume zu beschränken. Bei diesem Beispiel wird vorausgesetzt, dass die Zertifikat-Datenbank in der HFS-Datei `/ u/admin/CERT.kdb` liegt.

```
TTLSRule                          ConnRule01~1
{
  LocalAddrSetRef                 addr1
  RemoteAddrSetRef                addr1
  LocalPortRangeRef               portR1
  Direction                       Outbound
  Priority                        255
  TTLSGroupActionRef              gAct1~AllUsersAsClient
  TTLSEnvironmentActionRef        eAct1~AllUsersAsClient
  TTLSConnectionActionRef         cAct1~AllUsersAsClient
}
TTLSGroupAction                   gAct1~AllUsersAsClient
{
  TTLSEnabled                     On
  Trace                           6
}
TTLSEnvironmentAction             eAct1~AllUsersAsClient
{
  HandshakeRole                   Client
  EnvironmentUserInstance         0
  TTLSKeyringParmsRef             keyR1
}
TTLSConnectionAction              cAct1~AllUsersAsClient
{
  HandshakeRole                   Client
  TTLSCipherParmsRef              cipher1~AT-TLS__Silver
  TTLSConnectionAdvancedParmsRef  cAdv1~AllUsersAsClient
  Trace                           0
}
TTLSConnectionAdvancedParms       cAdv1~AllUsersAsClient
{
  ApplicationControlled           On
}
TTLSKeyringParms                  keyR1
{
  Keyring                         /u/admin/CERT.kdb
  KeyringStashFile                /u/admin/CERT.sth
}
TTLSCipherParms                   cipher1~AT-TLS__Silver
{
  V3CipherSuites                  TLS_RSA_WITH_DES_CBC_SHA
  V3CipherSuites                  TLS_RSA_WITH_3DES_EDE_CBC_SHA
  V3CipherSuites                  TLS_RSA_WITH_AES_128_CBC_SHA
}
IpAddrSet                         addr1
{
  Prefix                          0.0.0.0/0
}
PortRange                         portR1
{
  Port                            1024-65535
}
```

Wie überprüfe ich die AT-TLS-Konfiguration?

Prüfen Sie den Policy-Agent Job Output `JESMSGLG` auf:

```
EZZ8771I PAGENT CONFIG POLICY PROCESSING COMPLETE FOR <your TCP/IP address space>: ↩
TTLS
```

Diese Meldung zeigt die erfolgreiche Initialisierung an.

Prüfen Sie den Policy-Agent Job Output `JESMSGLG` auf:

```
EZZ8438I PAGENT POLICY DEFINITIONS CONTAIN ERRORS FOR <your TCP/IP address space>: ↩
TTLS
```

Diese Meldung weist auf Fehler in der Konfigurationsdatei hin. Prüfen Sie bitte die Datei `syslog.log` auf weitere Hinweise.

Umfasst die Konfigurationsregel auch den Client?

Prüfen Sie `syslog.log` auf:

```
EZD1281I TTLS Map   CONNID: 00002909 LOCAL: 10.20.91.61..1751 REMOTE: ↩
10.20.91.117..443
JOBNAME: KSP USERID: KSP TYPE: OutBound STATUS: Appl Control RULE: ConnRule01
ACTIONS: gAct1 eAct1 AllUsersAsClient
```

Der obige Eintrag zeigt an, dass die Verbindung zu Port 443 des Benutzers `KSP` von der Anwendung gesteuert wird.

Wo finde ich weitere Informationen zur Problembestimmung?

Siehe auch *z/OS V1R8.0 Comm Svr: IP Diagnosis Guide: 3.23*, Chapter 29 *Diagnosing Application Transparent Transport Layer Security (AT-TLS)*

Wie schalte ich den P-Agent Trace ein?

Informationen hierzu finden Sie im *Comm Svr: IP Configuration Reference*, Chapter 20 *Syslog deamon and Comm Svr: IP Configuration Guide*, Chapter 1.5.1 *Configuring the syslog daemon (syslogd)*

Fehler beim Verbindungsaufbau

Suchen Sie den Return Code `RC` und den entsprechenden `GSK_`-Funktionsnamen im P-Agent Trace.

Schlagen Sie im *System SSL Programming* im Chapter 12.1 *SSL Function Return Codes* den Return Code `RC` nach.

Beispiel-Trace zu `trace=255`:

```
EZD1281I TTLS Map   CONNID: 00002909 LOCAL: 10.20.91.61..1751 REMOTE: ↩
10.20.91.117..443 JOBNAME: KSP USERID: KSP TYPE: OutBound STATUS: A
EZD1283I TTLS Event GRPID: 00000003 ENVID: 00000000 CONNID: 00002909  RC:    0 ↩
Connection Init
EZD1282I TTLS Start GRPID: 00000003 ENVID: 00000002 CONNID: 00002909 Initial ↩
Handshake ACTIONS: gAct1 eAct1 AllUsersAsClient HS-Client
EZD1284I TTLS Flow  GRPID: 00000003 ENVID: 00000002 CONNID: 00002909  RC:    0 Call ↩
GSK_SECURE_SOCKET_OPEN - 7EE4F718
EZD1284I TTLS Flow  GRPID: 00000003 ENVID: 00000002 CONNID: 00002909  RC:    0 Set ↩
GSK_SESSION_TYPE -  CLIENT
EZD1284I TTLS Flow  GRPID: 00000003 ENVID: 00000002 CONNID: 00002909  RC:    0 Set ↩
GSK_V3_CIPHER_SPECS -  090A2F
EZD1284I TTLS Flow  GRPID: 00000003 ENVID: 00000002 CONNID: 00002909  RC:    0 Set ↩
GSK_FD - 00002909
EZD1284I TTLS Flow  GRPID: 00000003 ENVID: 00000002 CONNID: 00002909  RC:    0 Set ↩
GSK_USER_DATA - 7EEE9B50
EZD1284I TTLS Flow  GRPID: 00000003 ENVID: 00000002 CONNID: 00002909  RC:  435 Call ↩
GSK_SECURE_SOCKET_INIT - 7EE4F718
EZD1283I TTLS Event GRPID: 00000003 ENVID: 00000002 CONNID: 00002909  RC:  435 ↩
Initial Handshake 00000000 7EEE8118
EZD1286I TTLS Error GRPID: 00000003 ENVID: 00000002 CONNID: 00002909 JOBNAME: KSP ↩
USERID: KSP RULE: ConnRule01  RC:  435 Initial Handshake
EZD1283I TTLS Event GRPID: 00000003 ENVID: 00000002 CONNID: 00002909  RC:    0 ↩
Connection Close 00000000 7EEE8118 ↩
```

Weitere Informationsquellen

Die folgende Aufstellung enthält Links auf weitere nützliche Informationsquellen:

- World Wide Web Consortium (W3C): ***https://www.w3.org/***
- Extensible Markup Language (XML): ***https://www.w3.org/XML/***
- HyperText Markup Language (HTML) Home Page: ***https://www.w3.org/MarkUp/***
- W3 Schools: ***https://www.w3schools.com/***

IX Gestaltung der Benutzungsoberflächen von Anwendungen

Die Benutzerschnittstelle (Benutzeroberfläche) einer Anwendung, d.h. die Art und Weise, in der sich eine Anwendung dem Benutzer präsentiert, ist von entscheidender Bedeutung beim Erstellen einer Anwendung.

Dieser Teil beschreibt die verschiedenen Möglichkeiten, die Natural bietet, um Benutzeroberflächen zu erstellen, die ein einheitliches Erscheinungsbild haben und eine einfache, jedoch flexible Benutzerführung bieten.

Beim Entwurf von Benutzeroberflächen spielen Standards und Standardisierung eine wichtige Rolle.

Mit Natural ist es möglich, dem Benutzer eine einheitliche Benutzeroberfläche zu präsentieren, auch über Hardware- und Betriebssystemgrenzen hinaus.

Zum Design einer solchen Oberfläche gehören der allgemeine Bildschirmaufbau (Informations-, Daten- und Meldezeilenbereich), die Funktionstastenbelegung und der Aufbau von Fenstern.

Bildschirmgestaltung — Definition des allgemeinen Layouts von Bildschirmen

Dialog-Gestaltung — Gestaltung von Benutzungsschnittstellen

62 Bildschirmgestaltung

Dieses Kapitel beschreibt die Möglichkeiten zur Gestaltung des Bildschirm-Layouts.

Steuerung der Funktionstastenleiste — Terminalkommando %Y

Mit dem Terminalkommando `%Y` geben Sie an, wie und wo die Natural-Funktionstastenleiste angezeigt werden soll.

Dieser Abschnitt enthält Informationen zu folgenden Themen:

Format der Funktionstastenleiste

Die folgenden Terminalkommandos legen das Format der Funktionstastenleiste fest:

%YN
Die Funktionstastenleiste wird im tabellarischen Software-AG-Format angezeigt:

```
Kommando ===>
Enter-PF1---PF2---PF3---PF4---PF5---PF6---PF7---PF8---PF9---PF10--PF11--PF12---
      Help        Exit                                                 Canc   ↩
```

%YS
Die Funktionstastenleiste zeigt die einzelnen Tasten und ihre Belegung nacheinander, und zwar nur die Tasten, die mit Funktionen belegt sind:

(PF1=*Funktion*,PF2=*Funktion* usw.):

```
Kommando ===>
PF1=Help,PF3=Exit,PF12=Canc                ↩
```

%YP
Die Funktionstastenleiste erscheint im PC-Format, d.h. die einzelnen Tasten werden nacheinander angezeigt, und zwar nur die Tasten, denen Funktionen zugewiesen wurden:

(F1=*Funktion*,F2=*Funktion* usw.)

```
Kommando ===>
F1=Help,F3=Exit,F12=Canc                    ↩
```

Weitere Anzeige-Optionen

Es stehen verschiedene andere Optionen zur Anzeige der Funktionstastenleiste zur Verfügung, wie:

- ein- und zweizeilige Darstellung
- intensivierte Darstellung
- inverse Darstellung
- farbige Darstellung

Einzelheiten zu diesen Optionen entnehmen Sie dem Abschnitt *%Y - Steuerung der PF-Tastenleiste* in der *Terminalkommandos*-Dokumentation.

Positionierung der Funktionstastenleiste

%YB

Die Funktionstastenleiste wird am unteren Bildschirmrand angezeigt:

```
13:33:52                    *****  NATURAL  *****                    2022-11-24
Benutzer SAG                   - Hauptmenue -                Library SAGTEST



                   Funktion

                _  Entwicklungsfunktionen
                _  Einstellungen der Entwicklungsumgebung
                _  Pflegen und Uebertragen von Objekten
                _  Fehlersuche und Systemueberwachung
                _  Beispiel-Libraries
                _  Andere Produkte
                _  Hilfe
                _  Natural-Session beenden









Kommando ===>
Enter-PF1---PF2---PF3---PF4---PF5---PF6---PF7---PF8---PF9---PF10--PF11--PF12---
      Help        Exit                                                 Canc   ↩
```

%YT

Die Funktionstastenleiste wird am oberen Bildschirmrand angezeigt:

```
Enter-PF1---PF2---PF3---PF4---PF5---PF6---PF7---PF8---PF9---PF10--PF11--PF12---
      Help        Exit                                                   Canc
13:33:52                    *****  NATURAL  *****                   2022-11-24
Benutzer SAG                   - Hauptmenue -                Library SAGTEST


                   Funktion

                _  Entwicklungsfunktionen
                _  Einstellungen der Entwicklungsumgebung
                _  Pflegen und Uebertragen von Objekten
                _  Fehlersuche und Systemueberwachung
                _  Beispiel-Libraries
                _  Andere Produkte
                _  Hilfe
                _  Natural-Session beenden









Kommando ===>                                                             ↩
```

%Y*nn*

Die Funktionstastenleiste wird in der *nn*-ten Zeile des Bildschirms angezeigt. Im nachfolgenden Beispiel befindet sich die Funktionstastenleiste in Zeile 10:

```
13:33:52                    *****  NATURAL  *****                    2022-11-24
Benutzer SAG                   - Hauptmenue -                Library SAGTEST


                    Funktion

                _  Entwicklungsfunktionen
                _  Einstellungen der Entwicklungsumgebung
                _  Pflegen und Uebertragen von Objekten
Enter-PF1---PF2---PF3---PF4---PF5---PF6---PF7---PF8---PF9---PF10--PF11--PF12---
      Help        Exit                                                  Canc
                _  Fehlersuche und Systemueberwachung
                _  Beispiel-Libraries
                _  Andere Produkte
                _  Hilfe
                _  Natural-Session beenden









Kommando ===>                                                             ↩
```

Cursor-Sensitivität

%YC

Dieses Kommando macht die Funktionstastenleiste Cursor-empfindlich. Sie reagiert dann wie eine Aktionsleiste auf einem PC-Bildschirm: der Benutzer wählt mit dem Cursor lediglich den Namen oder die Nummer der gewünschten Funktionstaste aus und drückt EINGABE, und Natural reagiert, als ob die betreffende Funktionstaste gedrückt worden wäre.

Durch nochmaliges Eingeben von `%YC` schalten Sie die Cursor-Sensitivität wieder aus.

Durch Verwendung von `%YC` in Verbindung mit tabellarischem Anzeigeformat (`%YN`) und einzeiliger Anzeige (`%YH`) können Sie Ihre Anwendungen mit einer sehr komfortablen Aktionsleisten-Verarbeitung ausstatten: der Benutzer wählt dann nur noch den Namen einer Funktion mit dem Cursor aus und drückt EINGABE, und die Funktion wird ausgeführt.

Steuerung der Meldungszeile — Terminalkommando %M

Mit dem Terminalkommando `%M` geben Sie an, wie und wo die Natural-Meldungszeile angezeigt werden soll.

Im folgenden finden Sie Informationen zu:

- Positionierung der Meldungszeile
- Schützen der Meldungszeile
- Farbe der Meldungszeile

Positionierung der Meldungszeile

%MB
Die Meldungszeile wird am unteren Bildschirmrand angezeigt:

```
13:43:40                     *****  NATURAL  *****                    2022-11-24
Benutzer SAG                    - Hauptmenue -                Library SAGTEST



                   Funktion

                _  Entwicklungsfunktionen
                _  Einstellungen der Entwicklungsumgebung
                _  Pflegen und Uebertragen von Objekten
                _  Fehlersuche und Systemueberwachung
                _  Beispiel-Libraries
                _  Andere Produkte
                _  Hilfe
                _  Natural-Session beenden









Kommando ===>
Enter-PF1---PF2---PF3---PF4---PF5---PF6---PF7---PF8---PF9---PF10--PF11--PF12---
      Help        Exit                                                  Canc
NAT4492 Bitte eine Funktion selektieren/eingeben.                            ↩
```

%MT
Die Meldungszeile wird am oberen Bildschirmrand angezeigt:

```
NAT4492 Bitte eine Funktion selektieren/eingeben.
13:43:40                    *****  NATURAL  *****                  2022-11-24
Benutzer SAG                   - Hauptmenue -               Library SAGTEST


                   Funktion

                _  Entwicklungsfunktionen
                _  Einstellungen der Entwicklungsumgebung
                _  Pflegen und Uebertragen von Objekten
                _  Fehlersuche und Systemueberwachung
                _  Beispiel-Libraries
                _  Andere Produkte
                _  Hilfe
                _  Natural-Session beenden









Enter-PF1---PF2---PF3---PF4---PF5---PF6---PF7---PF8---PF9---PF10--PF11--PF12---
      Help        Exit                                                 Canc
Kommando ===>                                                              ↩
```

Weitere Optionen zur Positionierung der Meldungszeile sind im Abschnitt *%M - Steuerung der Meldungszeile* in der *Terminalkommandos*-Dokumentation beschrieben.

Schützen der Meldungszeile

%MP
Der Schutz der Meldungszeile wird ein- bzw. ausgeschaltet. Ist die Meldungszeile nicht geschützt, kann sie auch für Bildschirmeingaben benutzt werden.

Farbe der Meldungszeile

%M=*color-code*
Die Meldungszeile wird in der angegebenen Farbe angezeigt (eine Beschreibung der Farbcodes finden Sie unter Session-Parameter `CD` in der *Parameter-Referenz*-Dokumentation.

Zuweisen von Farben zu Feldern — Terminalkommando %=

Mit dem Terminalkommando %= können Sie bestimmten Feldern bestimmte Farben zuweisen, und zwar für Programme, die ursprünglich ohne Berücksichtigung von Farbgebung geschrieben wurden. Sie geben einen Feldtyp und/oder ein Feldattribut an sowie eine Farbe. Alle Felder/Texte dieses Typs/Attributs werden dann in dieser Farbe angezeigt.

Außerdem können Sie bestehende Farbzuweisungen ändern, falls bereits vordefinierte Farbgebungen ungeeignet sind.

Darüber hinaus können Sie das Terminalkommando %= in den Natural-Editoren benutzen, um Farben dynamisch zuordnen, z.B. beim Erstellen einer Maske (Map).

Codes	Beschreibung
leer	Bestehende Farbzuweisungen werden gelöscht.
F	Neu definierte Farbzuweisungen gelten statt denen des Programms.
N	Im Programm definierte Farbzuweisungen behalten ihre Gültigkeit.
O	Ausgabefeld (Output).
M	Modifizierbares Feld (Aus- und Eingabe).
T	Textkonstante.
B	Blinkend.
C	Kursiv.
D	Standard (Default).
I	Intensiviert.
U	Unterstrichen.
V	Invers.
BG	Bildschirmhintergrund (Background).
BL	Blau.
GR	Grün.
NE	Neutral.
PI	Rosa (Pink).
RE	Rot (Red).
TU	Türkis.
YE	Gelb (Yellow).

Beispiel:

```
%=TI=RE,OB=YE
```

Dieses Beispiel ordnet die Farbe Rot allen intensivierten Text-Feldern und Gelb allen blinkenden Ausgabefeldern zu.

Outlining (Umrahmung) — Terminalkommando %D=B

„Outlining“ (Boxing) ist die Möglichkeit, bestimmte Felder auf dem Bildschirm „eingerahmt“ (d.h. von einer Linie umgeben) anzuzeigen. Diese Form der Anzeige ist eine weitere Möglichkeit, dem Benutzer Länge und Position von Feldern auf dem Bildschirm deutlich zu machen.

„Outlining“ ist nur auf bestimmten Terminaltypen möglich, in der Regel auf solchen, die auch Doppelbyte-Zeichensätze (Kanji) unterstützen.

Mit dem Terminalkommando `%D=B` steuern Sie das „Outlining“. Einzelheiten zu diesem Kommando entnehmen Sie dem entsprechenden Abschnitt in der *Terminalkommandos*-Dokumentation.

Statistikzeile/Infoline — Terminalkommando %X

Dieses Terminalkommando steuert die Anzeige der Natural-Statistikzeile/Infoline. Die Zeile kann als Statistikzeile oder als Infoline benutzt werden, aber nicht beides gleichzeitig.

Im Folgenden finden Sie Informationen zu:

- Statistikzeile
- Infoline

Statistikzeile

`%X` schaltet die Anzeige der Statistikzeile/Infoline ein bzw. wieder aus. Schalten Sie die Statistikzeile ein, so können Sie statistische Informationen sehen wie:

- die Anzahl der während der letzten Bildschirmoperation an den Bildschirm übergebenen Bytes,
- die logische Zeilenlänge der aktuellen logischen Seite,
- die physische Zeilenlänge des aktiven Natural-Fensters.

Weitere Einzelheiten zur Statistikzeile können Sie der Beschreibung des Terminalkommandos `%X` in der *Terminalkommandos*-Dokumentation.

Das nachstehende Beispiel zeigt die Statistikzeile, wie sie unten am Bildschirmrand angezeigt wird:

```
 >                                      > +  Program     POS      Lib SAG
 All    ....+....1....+....2....+....3....+....4....+....5....+....6....+....7..
   0010 SET CONTROL 'XB'
   0020 SET CONTROL 'XI-'
   0030 DEFINE PRINTER (2) OUTPUT 'INFOLINE'
   0040 WRITE (2) 'EXECUTING' *PROGRAM 'BY' *INIT-USER
   0050 WRITE 'TEST OUTPUT'
   0070 END
   0080
   0090
   0100
   0110
   0120
   0130
   0140
   0150
   0160
   0170
   0180
   0190
   0200
IO=264,AI =292,L=0 C= ,LS=80,P =23,PLS=80,PCS=24,FLD=82,CLS=1,ADA=0
```

Infoline

Sie können die Statistikzeile auch als *Infoline* benutzen, in der Sie Status-Informationen ausgeben können, z.B. bei der Fehlersuche. Alternativ können Sie die Infoline als Trennlinie (wie in den SAA-Standards definiert) verwenden.

`%XI+` definiert die Statistikzeile als Infoline.

Sobald Sie die Infoline mit dem oben erwähnten Kommando aktiviert haben, können Sie die Infoline als Ausgabemedium für Daten mit dem `DEFINE PRINTER`-Statement definieren, wie im folgenden Beispiel veranschaulicht:

```
SET CONTROL 'XT'
SET CONTROL 'XI+'
DEFINE PRINTER (2) OUTPUT 'INFOLINE'
WRITE (2) 'EXECUTING' *PROGRAM 'BY' *INIT-USER
WRITE 'TEST OUTPUT'
END
```

Wenn dieses Programm gestartet wird, werden die Status-Informationen in der Infoline am oberen Rand des Ausgabeschirms angezeigt:

```
EXECUTING POS      BY SAG
Page      1                                              2014-02-07  09:34:08

TEST OUTPUT
```

Weitere Einzelheiten zur Statistikzeile/Infoline, siehe das Terminalkommando `%X` in der *Terminalkommandos*-Dokumentation.

Fenster

Im folgenden finden Sie Informationen zu:

- Was ist ein Fenster?
- DEFINE WINDOW-Statement
- INPUT WINDOW-Statement

Was ist ein Fenster?

Ein *Fenster* ist jener, von einem Programm aufgebaute Abschnitt einer logischen Seite, der auf dem Terminal-Bildschirm angezeigt wird.

Eine *logische Seite* ist der Ausgabebereich für Natural; mit anderen Worten enthält die logische Seite den/die vom Natural-Programm für die Anzeige erzeugte/n Report/Map. Diese logische Seite kann breiter als der physische Bildschirm sein.

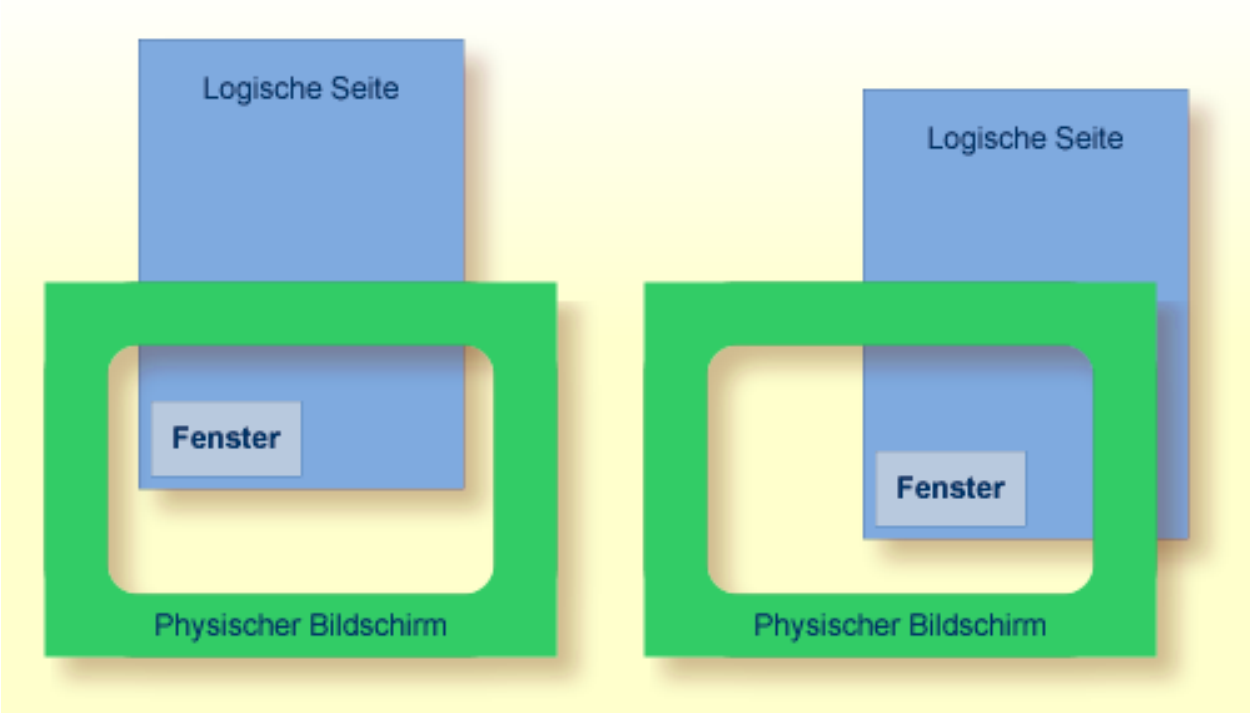
Es ist immer ein Fenster vorhanden, auch wenn dessen Vorhandensein Ihnen nicht bewusst sein mag. Wenn es (durch ein `DEFINE WINDOW`-Statement) nicht anders angegeben ist, ist die Größe des Fensters mit der physischen Größe Ihres Terminal-Bildschirms identisch.

Sie können ein Fenster auf zwei Arten handhaben:

- Sie können die Größe und Position des Fensters auf dem *physischen Bildschirm* steuern.
- Sie können die Position des Fensters auf der *logischen Seite* steuern.

Positionierung auf dem physischen Bildschirm

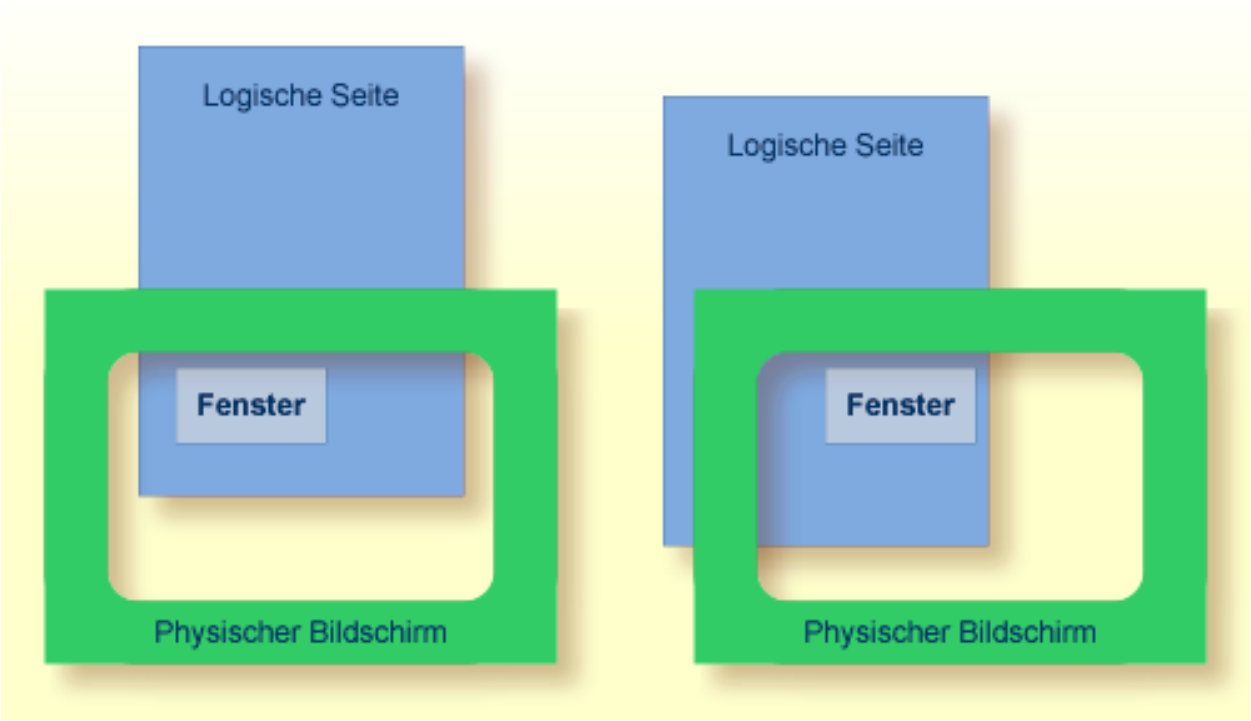
Die Abbildung unten veranschaulicht die Positionierung des Fensters auf dem physischen Bildschirm. Beachten Sie, dass in beiden Fällen der gleiche Ausschnitt der logischen Seite angezeigt wird; es wird lediglich die Position des Fensters auf dem physischen Bildschirm geändert.



Positionierung auf der logischen Seite

Die Abbildung unten veranschaulicht die Positionierung des Fensters auf der logischen Seite.

Wenn Sie die Position des Fensters auf der *logischen Seite* verändern, bleiben Position und Größe des Fensters auf dem *physischen Bildschirm* gleich; d.h. das Fenster wird nicht über der logischen Seite bewegt, sondern die logische Seite wird sozusagen „unter“ dem Fenster verschoben.



DEFINE WINDOW-Statement

Das `DEFINE WINDOW`-Statement dient dazu, die Größe, Position und Attribute eines Bildschirmfensters auf dem *physischen Bildschirm* zu definieren.

Mit einem `DEFINE WINDOW`-Statement wird ein Fenster nicht aktiviert; dies geschieht mit einem `SET WINDOW`-Statement oder der `WINDOW`-Klausel eines `INPUT`-Statements.

Das `DEFINE WINDOW`-Statement bietet verschiedene Optionen. Diese werden anhand des nachstehenden Beispiels erläutert. Es bezieht sich auf die standardmäßige Terminaltyp-Einstellung von Natural; siehe auch Terminalkommando `%T=` und Profilparameter `TTYPE`).

Das folgende Programm definiert ein Fenster auf dem physischen Bildschirm.

```
** Example 'WINDX01': DEFINE WINDOW
************************************************************************
DEFINE DATA LOCAL
1 COMMAND (A10)
END-DEFINE
*
DEFINE WINDOW TEST
       SIZE 5*25
       BASE 5/40
       TITLE 'Sample Window'
       CONTROL WINDOW
```

```
       FRAMED POSITION SYMBOL BOTTOM LEFT
*
INPUT WINDOW='TEST' WITH TEXT 'message line'
      COMMAND (AD=I'_') /
      'dataline 1' /
      'dataline 2' /
      'dataline 3' 'long data line'
*
IF COMMAND = 'TEST2'
  FETCH 'WINDX02'
ELSE
  IF COMMAND = '.'
    STOP
  ELSE
    REINPUT 'invalid command'
  END-IF
END-IF
END
```

Der Window-Name identifiziert das Fenster. Der Name darf bis zu 32 Stellen lang sein. Für Fensternamen gelten die gleichen Namenskonventionen wie für Benutzervariablen. Siehe *Namenskonventionen für Benutzervariablen* in der Dokumentation *Natural benutzen*.

Hier lautet der Name `TEST`.

Mit der `SIZE`-Klausel bestimmen Sie die Größe des Fensters. In dem Beispiel ist das Fenster 5 Zeilen hoch und 25 Spalten (Stellen) breit.

Mit der `BASE`-Klausel bestimmen Sie die Position des Fensters auf dem physischen Bildschirm. In dem Beispiel ist die obere linke Ecke des Fensters auf Zeile 5, Spalte 40 positioniert.

Mit der `TITLE`-Klausel können Sie eine Überschrift für das Fenster angeben. Die angegebene Überschrift wird zentriert in der oberen Rahmenzeile des Fensters angezeigt (natürlich nur, wenn ein Rahmen für das Fenster definiert ist).

Mit der `CONTROL`-Klausel können sie festlegen, ob die PF-Tastenzeile, die Meldungs- und die Statistikzeile in dem Fenster oder auf dem vollen physischen Bildschirm angezeigt werden. In diesem Fall bewirkt `CONTROL WINDOW`, dass die Meldungszeile im Fenster angezeigt wird. `CONTROL SCREEN` hingegen bewirkt, dass die Zeilen auf dem vollen physischen Bildschirm außerhalb des Fenster angezeigt werden. Wenn Sie die `CONTROL`-Klausel weglassen, gilt standardmäßig `CONTROL WINDOW`.

Mit der `FRAMED`-Option geben Sie an, dass das Fenster eingerahmt werden soll.

Dieser Rahmen ist dann cursor-sensitiv. Sie können gegebenenfalls im Fenster vor, zurück, nach rechts oder links blättern, indem Sie den Cursor einfach auf das entsprechende Symbol `<`, `–`, `+` oder `>` (vgl. `POSITION`-Klausel weiter unten) platzieren und EINGABE drücken. Anders ausgedrückt, bewegen Sie damit die logische Seite unter dem Fenster auf dem physischen Bildschirm. Werden keine Symbole angezeigt, können Sie vor- und zurückblättern, indem Sie den Cursor in die obere (zum Zurückblättern) bzw. untere (zum Vorblättern) Rahmenzeile platzieren und EINGABE drücken.

Mit der POSITION-Klausel der FRAMED-Option bestimmen Sie, dass Informationen über die Position des Fensters auf der logischen Seite in dem Fensterrahmen angezeigt werden. Diese Positionsangaben werden nur angezeigt, wenn die logische Seite größer ist als das Fenster; sonst wird die POSITION-Klausel ignoriert. Die Positionsangaben geben an, in welche Richtungen die logische Seite nach oben, unten, links und rechts über das aktuelle Fenster hinausgeht.

Wird keine POSITION-Klausel angegeben, gilt standardmäßig POSITION SYMBOL TOP RIGHT.

POSITION SYMBOL bewirkt, dass die Positionsangaben als Symbole More: < − + > angezeigt werden. Die Angaben werden in der oberen und/oder unteren Rahmenzeile angezeigt.

TOP/BOTTOM bestimmt, in welcher Rahmenzeile (oben oder unten) die Angaben erscheinen sollen.

LEFT/RIGHT bestimmt, ob die Positionsangaben im linken oder rechten Teil der Rahmenzeile angezeigt werden.

Mit dem Terminalkommando %F=*chv* bestimmen Sie, aus welchen Zeichen der Fensterrahmen zusammengesetzt werden soll.

c	Das erste Zeichen wird für die vier *Ecken* (**c**orners) des Rahmens verwendet.
h	Das zweite Zeichen wird für die *horizontalen* Linien des Rahmens verwendet.
v	Das dritte Zeichen wird für die *vertikalen* Linien des Rahmens verwendet.

Beispiel:

```
%F=+-!
```

Der mit dem obigen Kommando definierte Fensterrahmen sieht wie folgt aus:

```
+-------------------------+
!                         !
!                         !
!                         !
!                         !
+-------------------------+
```

INPUT WINDOW-Statement

Das `INPUT WINDOW`-Statement aktiviert das in dem `DEFINE WINDOW`-Statement definierte Fenster. In dem Beispiel wird das Fenster `TEST` aktiviert. Wollen Sie Daten in einem Fenster ausgeben (z.B. mit einem `WRITE`-Statement), benutzen Sie das `SET WINDOW`-Statement.

Wenn das obige Programm ausgeführt wird, wird das Fenster mit einem Eingabefeld `COMMAND` angezeigt. Mit dem Session-Parameter `AD` legen Sie fest, dass der Wert des Feldes intensiviert dargestellt und ein Unterstrich als Füllzeichen benutzt wird.

Ausgabe des Programms `WINDX01`:

```
> r                                    > +  Program     WINDX01  Lib SYSEXPG
Top    ....+....1....+....2....+....3....+....4....+....5....+....6....+....7..
  0010 ** Example 'WINDX01': DEFINE WINDOW
  0020 ****************************** +----Sample Window-----+ ***************
  0030 DEFINE DATA LOCAL               ! message line         !
  0040 1 COMMAND (A10)                 ! COMMAND __________   !
  0050 END-DEFINE                      ! dataline 1           !
  0060 *                               +More:     + >---------+
  0070 DEFINE WINDOW TEST
  0080        SIZE 5*25
  0090        BASE 5/40
  0100        TITLE 'Sample Window'
  0110        CONTROL WINDOW
  0120        FRAMED POSITION SYMBOL BOTTOM LEFT
  0130 *
  0140 INPUT WINDOW='TEST' WITH TEXT 'message line'
  0150       COMMAND (AD=I'_') /
  0160       'dataline 1' /
  0170       'dataline 2' /
  0180       'dataline 3' 'long data line'
  0190 *
  0200 IF COMMAND = 'TEST2'
       ....+....1....+....2....+....3....+....4....+....5....+... S 29   L 1
```

Die in der unteren Rahmenzeile erscheinenden Positionsangaben `More + >` zeigen an, dass die logische Seite mehr Informationen enthält, als im Fenster zu sehen sind.

Um die Informationen anzuzeigen, die sich weiter unten auf der logischen Seite befinden, platzieren Sie den Cursor in die untere Rahmenzeile auf das +-Symbol und drücken EINGABE.

Dadurch wird das Fenster nach unten bewegt. Beachten Sie, dass der Text `long data line` (lange Datenzeile) nicht in das Fenster passt und daher nicht vollständig angezeigt wird.

```
> r                                      > +  Program     WINDX01  Lib SYSEXPG
Top    ....+....1....+....2....+....3....+....4....+....5....+....6....+....7..
  0010 ** Example 'WINDX01': DEFINE WINDOW
  0020 ****************************** +----Sample Window-----+ ***************
  0030 DEFINE DATA LOCAL               ! message line         !
  0040 1 COMMAND (A10)                 ! dataline 3 long data !
  0050 END-DEFINE                      ! dataline 2           !
  0060 *                               +More:   -   >---------+
  0070 DEFINE WINDOW TEST
  0080        SIZE 5*25
  0090        BASE 5/40
  0100        TITLE 'Sample Window'
  0110        CONTROL WINDOW
  0120        FRAMED POSITION SYMBOL BOTTOM LEFT
  0130 *
  0140 INPUT WINDOW='TEST' WITH TEXT 'message line'
  0150       COMMAND (AD=I'_') /
  0160       'dataline 1' /
  0170       'dataline 2' /
  0180       'dataline 3' 'long data line'
  0190 *
  0200 IF COMMAND = 'TEST2'
       ....+....1....+....2....+....3....+....4....+....5....+... S 29   L 1
```

Um diese *verborgenen* Informationen auf der rechten Seite zu sehen, platzieren Sie den Cursor in die untere Rahmenzeile auf das >-Symbol und drücken EINGABE. Das Fenster wird dadurch auf der logischen Seite nach rechts bewegt; das vorher unsichtbare Wort `line` (Zeile) wird angezeigt:

```
> r                                    > +  Program     WINDX01  Lib SYSEXPG
Top    ....+....1....+....2....+....3....+....4....+....5....+....6....+....7..
  0010 ** Example 'WINDX01': DEFINE WINDOW
  0020 ****************************** +----Sample Window-----+ ***************
  0030 DEFINE DATA LOCAL               ! message line         !
  0040 1 COMMAND (A10)                 !  line                ! <==
  0050 END-DEFINE                      !                      !
  0060 *                               +More: < -    ---------+
  0070 DEFINE WINDOW TEST
  0080        SIZE 5*25
  0090        BASE 5/40
  0100        TITLE 'Sample Window'
  0110        CONTROL WINDOW
  0120        FRAMED POSITION SYMBOL BOTTOM LEFT
  0130 *
  0140 INPUT WINDOW='TEST' WITH TEXT 'message line'
  0150       COMMAND (AD=I'_') /
  0160       'dataline 1' /
  0170       'dataline 2' /
  0180       'dataline 3' 'long data line'
  0190 *
  0200 IF COMMAND = 'TEST2'
       ....+....1....+....2....+....3....+....4....+....5....+... S 29   L 1
```

Mehrere Fenster

Sie können mehrere Fenster öffnen. Allerdings ist jeweils nur ein Natural-Fenster aktiv, und zwar das letzte. Vorherige Fenster mögen auf dem Bildschirm noch sichtbar sein, sind aber nicht mehr aktiv und werden von Natural ignoriert. Sie können Eingaben nur im jeweils letzten Fenster machen. Sollte der Platz hierzu nicht ausreichen, müssen Sie das Fenster vorher entsprechend vergrößern.

Wird `TEST2` in das `COMMAND`-Feld eingegeben, wird das Programm `WINDX02` ausgeführt.

```
** Example 'WINDX02': DEFINE WINDOW
************************************************************************
DEFINE DATA LOCAL
1 COMMAND (A10)
END-DEFINE
*
DEFINE WINDOW TEST2
       SIZE 5*30
       BASE 15/40
       TITLE 'Another Window'
       CONTROL SCREEN
       FRAMED POSITION SYMBOL BOTTOM LEFT
*
INPUT WINDOW='TEST2' WITH TEXT 'message line'
```

```
      COMMAND (AD=I'_') /
      'dataline 1' /
      'dataline 2' /
      'dataline 3' 'long data line'
*
IF COMMAND = 'TEST'
  FETCH 'WINDX01'
ELSE
  IF COMMAND = '.'
    STOP
  ELSE
    REINPUT 'invalid command'
  END-IF
END-IF
END
```

Ein zweites Fenster wird nun geöffnet. Das andere Fenster ist zwar noch sichtbar, jedoch inaktiv.

```
message line
> r                                    > +  Program     WINDX01  Lib SYSEXPG
Top    ....+....1....+....2....+....3....+....4....+....5....+....6....+....7..
  0010 ** Example 'WINDX01': DEFINE WINDOW
  0020 ****************************** +----Sample Window-----+ ***************
  0030 DEFINE DATA LOCAL               ! message line         !  Inactive
  0040 1 COMMAND (A10)                 ! COMMAND TEST2_____   !  Window
  0050 END-DEFINE                      ! dataline 1           !  <==
  0060 *                               +More:     + >---------+
  0070 DEFINE WINDOW TEST
  0080        SIZE 5*25
  0090        BASE 5/40
  0100        TITLE 'Sample Window'
  0110        CONTROL WINDOW
  0120        FRAMED POSITION SYMBOL B +------Another Window-------+  Currently   ↩

  0130 *                               ! COMMAND ___________         !  Active
  0140 INPUT WINDOW='TEST' WITH TEXT ' ! dataline 1                  !  Window
  0150       COMMAND (AD=I'_') /       ! dataline 2                  !  <==
  0160       'dataline 1' /            +More:   +------------------+
  0170       'dataline 2' /
  0180       'dataline 3' 'long data line'
  0190 *
  0200 IF COMMAND = 'TEST2'                                                    ↩
```

Beachten Sie, dass die Meldungszeile (message line) für das neue Fenster außerhalb des Fensters (oben auf dem physischen Bildschirm)angezeigt wird. Dies wurde mit der `CONTROL SCREEN`-Klausel im Programm `WINDX02` definiert.

Weitere Einzelheiten zu den Statements `DEFINE WINDOW`, `INPUT WINDOW` und `SET WINDOW` finden Sie in den entsprechenden Beschreibungen in der *Statements*-Dokumentation.

Standard-/Dynamische Layout-Maps

Ein Standard-Layout kann im Map-Editor definiert werden. Wird dieses Layout bei der Erstellung aller Maps (Masken) verwendet, so wird gewährleistet, dass die gesamte Anwendung ein einheitliches Erscheinungsbild aufweist.

Wenn eine Map, die ein Standard-Layout referenziert, initialisiert wird, wird dieses Layout zum festen Bestandteil der Map. Falls das Standard-Layout nachträglich geändert wird, bedeutet dies allerdings, dass alle Maps neu katalogisiert werden müssen, damit die Änderungen greifen.

Im Gegensatz zum Standard-Layout, wird ein dynamisches Layout nicht zum festen Bestandteil einer sich darauf beziehenden Map; vielmehr wird das Layout jeweils zur Laufzeit generiert.

Wenn Sie also im Map-Editor die Layout-Map als dynamisch definieren, werden alle Änderungen der Layout-Map automatisch auch bei allen Maps, die sich darauf beziehen, durchgeführt. Die Maps müssen nicht neu katalogisiert werden.

Weitere Einzelheiten über Standard-Layouts und dynamische Layouts für Maps finden Sie im Abschnitt *Format* unter *Masken-Editor* in der *Editoren*-Dokumentation.

Mehrsprachige Benutzeroberflächen

Mit Natural können Sie mehrsprachige Anwendungen für den internationalen Einsatz erstellen.

Maps, Helproutinen, Fehlermeldungen, Programme, Functions,Subprogramme und Copycodes können in bis zu 60 verschiedenen Sprachen (inklusive Sprachen, die einen Doppelbyte-Zeichensatz benutzen) definiert werden.

Im folgenden finden Sie Informationen zu:

- Sprachcodes
- Definition der Sprache eines Natural-Objektes
- Definition der Benutzersprache
- Referenzieren von mehrsprachigen Objekten
- Programme
- Fehlermeldungen

- Editiermasken für Datums- und Uhrzeitfelder

Sprachcodes

In Natural hat jede Sprache einen *Sprachcode* (von `1` bis `60`). Die folgende Tabelle ist ein Auszug der Gesamttabelle der Sprachcodes.

Eine vollständige Aufstellung der Sprachcodes finden Sie in der Beschreibung der Systemvariablen `*LANGUAGE` in der *Systemvariablen*-Dokumentation.

Sprachcode	Sprache	Mapcode in Objekt-Namen
1	Englisch	1
2	Deutsch	2
3	Französisch	3
4	Spanisch	4
5	Italienisch	5
6	Niederländisch	6
7	Türkisch	7
8	Dänisch	8
9	Norwegisch	9
10	Albanisch	A
11	Portugiesisch	B

Der Sprachcode (linke Spalte) ist der Code, der in der Systemvariable `*LANGUAGE` enthalten ist. Dieser Code wird von Natural intern benutzt. Es ist der Code, den Sie zur Definition der Benutzersprache benutzen (siehe ***Definition der Benutzersprache*** weiter unten). Der Code, den Sie zur Identifikation der Sprache eines Natural-Objekts angeben, ist der *Mapcode* in der rechten Spalte der Tabelle.

Beispiel:

Der Sprachcode für Portugiesisch ist `11`. Der Code, den Sie beim Katalogisieren eines portugiesischen Natural-Objekts angeben, ist `B`.

Definition der Sprache eines Natural-Objektes

Sie definieren die Sprache eines Natural-Objektes (Map, Helproutine, Programm, Subprogramm oder Copycode), indem Sie den entsprechenden Mapcode dem Objektnamen hinzufügen. Bis auf diesen Mapcode muss der Objektname identisch für alle Sprachen sein.

Beim folgenden Beispiel wurden zwei Maps erzeugt, und zwar eine englische und eine deutsche. Um die Sprachen der Maps zu identifizieren, wurde der entsprechende Mapcode jeweils den Mapnamen hinzugefügt.

Beispiel für Mapnamen bei einer mehrsprachigen Anwendung:

`DEMO1` = englische Map (Mapcode 1)

`DEMO2` = deutsche Map (Mapcode 2)

Definition von Sprachen mit alphabetischen Mapcodes

Mapcodes können im Bereich `1-9`, `A-Z` oder `a-y` liegen. Die alphabetischen Mapcodes bedürfen einer besonderen Handhabung.

Normalerweise ist es nicht möglich, ein Objekt zu katalogisieren, das einen Kleinbuchstaben im Namen hat — alle Buchstaben werden automatisch in Großbuchstaben umgewandelt.

Genau dies ist aber erforderlich, wenn Sie z.B. ein Objekt als japanisch (Kanji) definieren wollen. Japanisch hat den Sprachcode `59` und den Mapcode `x`.

Um ein solches Objekt zu katalogisieren, müssen Sie zuerst den Sprachcode (in diesem Fall Sprachcode `59`) richtig setzen, indem Sie das Terminalkommando `%L=`*`nn`* (*nn* entspricht dem Sprachcode) benutzen.

Jetzt können Sie das Objekt katalogisieren, wobei Sie das Und-Zeichen (&) anstelle des eigentlichen Mapcodes im Objektnamen benutzen. Um eine japanische Version der Map `DEMO` zu erhalten, katalogisieren Sie die Map also unter dem Namen `DEMO&`.

Wenn Sie jetzt in der Liste der Natural-Objekte nachschauen, wird die Map richtigerweise als `DEMOx` aufgelistet.

Sie können Objekte mit Sprachcode `1` bis `9` und `A` bis `Z` direkt katalogisieren, d.h. ohne die `&`-Notation.

Im nachfolgenden Beispiel sehen Sie die drei Maps `DEMO1`, `DEMO2` und `DEMOx`. Um die Map `DEMOx` zu löschen, benutzen Sie die gleiche Technik wie bei der Definition des Mapnamens, d.h. Sie setzen zuerst die richtige Sprache mit dem Terminalkommando `%L=59`, dann bestätigen Sie den Löschvorgang mit der `&`-Notation (`DEMO&`).

```
11:48:55               ***** NATURAL LIST COMMAND *****             2014-02-17
User SAG                 - LIST Objects in a Library -         Library SAGTEST

Cmd  Name       Type         S/C  SM Version  User ID    Date        Time
---  *_________ M__________  *__  *  *______  *________  *__________ *________
__   DEMO       Ma +---------------DELETE---------------+ 013-10-24  11:31:51
de   DEMOx      Ma !    +----------DELETE----------+    ! 013-10-24  11:31:51
__   DEMOB      Ma !    !                           !    ! 013-10-24  11:31:51
__   DEMO1      Ma !    !  Please confirm deletion !    ! 013-10-24  11:31:51
__   DEMO2      Ma !    !  with name DEMOx         !    ! 013-10-24  11:31:51
__   MAPTEST    Ma !    !            DEMO&___      ! n  ! 012-07-24  10:04:44
__   MAP01      Ma !    +---------------------------+    ! 012-07-24  10:04:25
                   !                                     !
                   +-------------------------------------+

                                                                7 Objects found

Command ===>
Enter-PF1---PF2---PF3---PF4---PF5---PF6---PF7---PF8---PF9---PF10--PF11--PF12---
      Help  Print Exit  Sort        --    -     +     ++          >     Canc
```

Definition der Benutzersprache

Sie können pro Benutzer bestimmen, welche Sprache (wie in der Systemvariablen `*LANGUAGE` definiert) benutzt wird, und zwar mit dem Profilparameter `ULANG` (der in der *Parameter-Referenz*-Dokumentation beschrieben ist) oder mit dem Terminalkommando `%L=`*`nn`* (wobei *nn* der Sprachcode ist).

Referenzieren von mehrsprachigen Objekten

Um in einem Programm mehrsprachige Objekte zu referenzieren, benutzen Sie das `&`-Zeichen im Namen des Objektes.

Das Programm unten benutzt die zwei Maps `DEMO1` und `DEMO2`. Das `&`-Zeichen am Ende des Mapnamens steht anstelle des Mapcodes und bedeutet, dass die Map mit der Sprache, die dem Wert in der Systemvariable `*LANGUAGE` entspricht, benutzt werden soll.

```
DEFINE DATA LOCAL
1 PERSONNEL VIEW OF EMPLOYEES
  2 NAME (A20)
  2 PERSONNEL-ID (A8)
1 CAR VIEW OF VEHICLES
   2 REG-NUM (A15)
1 #CODE (N1)
END-DEFINE
*
INPUT USING MAP 'DEMO&' /* <--- INVOKE MAP WITH CURRENT LANGUAGE CODE
...
```

Wird dieses Programm ausgeführt, so wird die englische Map (DEMO1) ausgegeben, denn der Wert von *LANGUAGE ist 1 = englisch.

```
                                                    MAP DEMO1

                          SAMPLE MAP



                Please select a function!


                1.) Employee information

                2.) Vehicle information


                Enter code here: _
```

Im Beispiel unten wird der Sprachcode auf 2 = deutsch umgesetzt und zwar mit dem Terminalkommando %L=2.

Wird das Programm nun ausgeführt, wird die deutsche Map (DEMO2) ausgegeben.

```
                          BEISPIEL-MAP



                Bitte wählen Sie eine Funktion!


                1.) Mitarbeiterdaten

                2.) Fahrzeugdaten


              Code hier eingeben: _
```

Programme

Bei manchen Anwendungen kann es nützlich sein, mehrsprachige Programme zu definieren. Ein Fakturierungsprogramm könnte z.B. verschiedene Subprogramme benutzen, um gewisse steuerliche Aspekte zu berücksichtigen, je nachdem in welchem Land die Rechnung erstellt werden soll.

Mehrsprachige Programme werden genauso definiert wie Maps (siehe Beschreibung oben).

Fehlermeldungen

Mit der Natural-Utility `SYSERR` können Sie die Natural-Fehlermeldungen in bis zu 60 Sprachen übersetzen. Sie können aber auch eigene Fehlermeldungen erstellen.

Die Sprache der ausgegebenen Fehlermeldungen wird durch den Inhalt der Systemvariable `*LANGUAGE` bestimmt.

Einzelheiten über Fehlermeldungen finden Sie in der *SYSERR Utility* in der *Utilities*-Dokumentation.

Editiermasken für Datums- und Uhrzeitfelder

Die Sprache für Datums- und Uhrzeitfelder, die mit Editiermasken definiert wurden, wird auch durch die Systemvariable `*LANGUAGE` festgelegt.

Weitere Einzelheiten zu Editiermasken entnehmen Sie dem Session-Parameter `EM` in der *Parameter Reference*.

Kenntnisabhängige Benutzeroberflächen (Expertenmodus)

Es kann sinnvoll sein, Benutzern mit unterschiedlicher Erfahrung bei der Benutzung derselben Anwendung verschiedene Maps mit unterschiedlichem Informationsgehalt zu bieten.

Ist Ihre Anwendung *nicht* für den internationalen Einsatz bestimmt, können Sie die gleichen Techniken, die zur Unterstützung von mehrsprachigen Anwendungen benutzt werden, auch zur Definition von unterschiedlich detaillierten Maps verwenden.

Sie können z.B. Sprachcode `1` als Kenntnisstufe 1 = Kenntnisstand eines Anfängers, und Sprachcode `2` als Kenntnisstufe 2 = Kenntnisstand eines fortgeschrittenen Benutzers definieren. Die Anwendung dieser einfachen Technik wird im Folgenden veranschaulicht.

Die folgende Map (`PERS1`) enthält ausführliche Anweisungen, die dem Endbenutzer sagen, wie eine Funktion ausgewählt wird. Die Informationen sind sehr detailliert. Der Name dieser Map enthält den Mapcode `1`.

```
                                                    MAP PERS1

                            SAMPLE MAP

                  Please select a function


                  1.) Employee information  _

                  2.) Vehicle information   _

                      Enter code:  _

To select a function, do one of the following:

- place the cursor on the input field next to desired function and press ENTER
- mark the input field next to desired function with an X and press ENTER
- enter the desired function code (1 or 2) in the 'Enter code' field and press
  ENTER
```

Die gleiche Map - aber ohne die ausführlichen Anweisungen - wird unter dem gleichen Namen, aber mit Mapcode `2` gespeichert.

```
                                                MAP PERS2


                            SAMPLE MAP

                  Please select a function


                  1.) Employee information  _

                  2.) Vehicle information   _

                      Enter code:  _
```

In dem obigen Beispiel wird die Map mit den vollständigen Anweisungen dann ausgegeben, wenn der `ULANG`-Profilparameter auf `1` gesetzt ist und die Map ohne die Anweisungen, wenn er auf `2` gesetzt ist.

63 Dialog-Gestaltung

Dieses Kapitel beschreibt, wie Benutzeroberflächen erstellt werden können, die einen einfachen und flexiblen Benutzerdialog ermöglichen.

Feldabhängige Verarbeitung

Feldabhängige Verarbeitung — *CURS-FIELD und POS(fieldname)

Mit der Systemvariablen `*CURS-FIELD` zusammen mit der Systemfunktion `POS(field-name)` können Sie eine Verarbeitung davon abhängig machen, in welchem Feld der Cursor sich gerade befindet, wenn der Benutzer EINGABE drückt.

`*CURS-FIELD` enthält die interne Identifikation des Feldes, in dem der Cursor sich gerade befindet; diese Systemvariable kann nicht allein, sondern nur zusammen mit `POS(field-name)` benutzt werden.

Mit `*CURS-FIELD` und `POS(field-name)` können Sie z.B. dem Benutzer die Möglichkeit geben, eine Funktion auszuwählen, indem er einfach den Cursor auf ein bestimmtes Feld platziert und EINGABE drückt.

Das Beispiel unten demonstriert eine solche Anwendung:

```
DEFINE DATA LOCAL
1 #EMP (A1)
1 #CAR (A1)
1 #CODE (N1)
END-DEFINE
*
INPUT USING MAP 'CURS'
*
DECIDE FOR FIRST CONDITION
  WHEN *CURS-FIELD = POS(#EMP) OR #EMP = 'X' OR #CODE = 1
    FETCH 'LISTEMP'
  WHEN *CURS-FIELD = POS(#CAR) OR #CAR = 'X' OR #CODE = 2
    FETCH 'LISTCAR'
   WHEN NONE
    REINPUT 'PLEASE MAKE A VALID SELECTION'
END-DECIDE
END
```

Und das Ergebnis:

```
                                SAMPLE MAP

                   Please select a function


                   1.) Employee information  _ <== Cursor auf Feld
                   2.) Vehicle information   _


                       Enter code:  _

To select a function, do one of the following:

- place the cursor on the input field next to desired function and press ENTER
- mark the input field next to desired function with an X and press ENTER
- enter the desired function code (1 or 2) in the 'Enter code' field and press
  ENTER
```

Wenn der Benutzer den Cursor auf das Eingabefeld (#EMP) neben `Employee information` platziert und EINGABE drückt, gibt das Programm `LISTEMP` eine Liste der Mitarbeiter aus:

```
Page      1                                                2001-01-22  09:39:32

        NAME
--------------------

ABELLAN
ACHIESON
ADAM
ADKINSON
ADKINSON
ADKINSON
ADKINSON
ADKINSON
ADKINSON
ADKINSON
ADKINSON
ADKINSON
AECKERLE
AFANASSIEV
AFANASSIEV
AHL
AKROYD
```

Anmerkungen:

1. In Natural for Ajax-Anwendungen dient `*CURS-FIELD` zur Identifikation des Operanden, welcher den Wert des Control darstellt, welches den Eingabefokus hat. Sie können `*CURS-FIELD` in Verbindung mit der `POS`-Funktion benutzen, um eine Prüfung auf das Control, das den Einga-

befokus hat, zu veranlassen und die Verarbeitung in Abhängigkeit von diesem Zustand durchzuführen.

2. Die Werte von *CURS-FIELD und POS(*field-name*) dienen nur der internen Identifikation der Felder. Sie können für arithmetische Operationen nicht verwendet werden.

Einfachere Programmierung

Systemfunktion POS

Die Natural-Systemfunktion POS(*field-name*) enthält die interne Identifikation des Feldes, dessen Name mit der Systemfunktion angegeben wird.

POS(*field-name*) identifiziert ein bestimmtes Feld, unabhängig von seiner Position in einer Map. Auch wenn sich die Reihenfolge und Anzahl der Felder in einer Map ändert, identifiziert POS(*field-name*) nach wie vor eindeutig dasselbe Feld. Damit genügt zum Beispiel ein einziges REINPUT-Statement, um es von der Programmlogik abhängig zu machen, welches Feld MARKiert werden soll.

Anmerkung: Der Wert von POS(*field-name*) dient nur zur internen Identifikation der Felder. Er kann nicht für arithmetische Operationen benutzt werden.

Beispiel:

```
...
DECIDE ON FIRST VALUE OF ...
   VALUE ...
      COMPUTE #FIELDX = POS(FIELD1)
   VALUE ...
      COMPUTE #FIELDX = POS(FIELD2)
   ...
END-DECIDE
...
REINPUT ... MARK #FIELDX
...
```

Weitere Einzelheiten zu *CURS-FIELD und POS(*field-name*) siehe *Systemvariablen* und *Systemfunktionen*.

Zeilenabhängige Verarbeitung

Systemvariable *CURS-LINE

Mit der Systemvariablen `*CURS-LINE` können Sie eine Verarbeitung davon abhängig machen, in welcher Zeile der Cursor gerade steht, wenn der Benutzer EINGABE drückt.

Mit dieser Variablen können Sie z.B. benutzerfreundliche Menüs gestalten. Bei entsprechender Programmierung braucht der Benutzer lediglich den Cursor in die Zeile der gewünschten Menü-Funktion zu platzieren und EINGABE zu drücken, um die Funktion auszuführen.

Die Cursor-Position bezieht sich auf das aktive Fenster, unabhängig von der Position auf dem physischen Bildschirm.

Anmerkung: Die Meldungszeile, Funktionstastenleiste und Statistikzeile/Infoline zählen nicht als Datenzeilen auf dem Bildschirm.

Das Beispiel unten veranschaulicht die Möglichkeiten einer zeilenabhängigen Verarbeitung, die die Systemvariable `*CURS-LINE` bietet. Wenn der Benutzer in der Map EINGABE drückt, prüft das Programm, ob der Cursor in Zeile 8 des Bildschirms steht. Diese Zeile enthält die Option `Employee information`. Trifft dies zu, wird das Programm `LISTEMP` ausgeführt, das eine Liste der Mitarbeiter ausgibt.

```
DEFINE DATA LOCAL
1 #EMP (A1)
1 #CAR (A1)
1 #CODE (N1)
END-DEFINE
*
INPUT USING MAP 'CURS'
*
DECIDE FOR FIRST CONDITION
  WHEN *CURS-LINE = 8
     FETCH 'LISTEMP'
   WHEN NONE
     REINPUT 'PLACE CURSOR ON LINE OF OPTION YOU WISH TO SELECT'
END-DECIDE
END
```

Ausgabe:

```
                              Company Information


                   Please select a function


                [] 1.) Employee information

                   2.) Vehicle information









   Place the cursor on the line of the option you wish to select and press
   ENTER
```

Der Benutzer platziert den durch **[ ]** dargestellten Cursor in die Zeile der gewünschten Option und drückt EINGABE: das entsprechende Programm wird ausgeführt.

Spaltenabhängige Verarbeitung

Systemvariable *CURS-COL

Die Systemvariable `*CURS-COL` wird analog zu der Systemvariablen `*CURS-LINE` (wie oben beschrieben) benutzt. Mit `*CURS-COL` können Sie eine Verarbeitung davon abhängig machen, in welcher Spalte der Cursor steht.

Verarbeitung aufgrund von Funktionstasten

Systemvariable *PF-KEY

Oft ist eine funktionstastenabhängige Verarbeitung erwünscht.

Diese wird mittels des `SET KEY`-Statements und der Systemvariablen `*PF-KEY` und einer Änderung der Standard-Einstellungen der Map (`Standard Keys` = `Y`) erreicht.

Das `SET KEY`-Statement weist während der Ausführung des Programms Funktionstasten Funktionen zu. Die Systemvariable `*PF-KEY` enthält die Identifikation der vom Benutzer zuletzt gedrückten Funktionstaste.

Das nachstehende Beispiel veranschaulicht die Anwendung von SET KEY mit *PF-KEY.

```
...
SET KEY PF1
*
INPUT USING MAP 'DEMO&'
IF *PF-KEY = 'PF1'
  WRITE 'Help is currently not active'
END-IF
...
```

Das SET KEY-Statement aktiviert PF1 als Funktionstaste.

Das IF-Statement bestimmt, welche Aktionen erfolgen sollen, wenn der Benutzer PF1 drückt.

Beim Programmablauf wird der aktuelle Inhalt der Systemvariablen *PF-KEY geprüft; wenn sie PF1 enthält, wird die entsprechende Aktion ausgeführt.

Weitere Einzelheiten zum Statement SET KEY und der Systemvariable *PF-KEY finden Sie in der *Statements*- bzw. *Systemvariablen*-Dokumentation.

Verarbeitung aufgrund der Namen von Funktionstasten

Systemvariable *PF-NAME

Oft wird eine bestimmte Verarbeitung durch Drücken einer Funktionstaste ausgelöst. Noch mehr Komfort wird durch die Systemvariable *PF-NAME geboten. Mit ihr können Sie eine Verarbeitung von dem Namen einer Funktion abhängig machen, anstatt von einer Funktion.

Die Systemvariable *PF-NAME enthält den Namen der zuletzt gedrückten Funktionstaste (d.h. den Namen, der der Funktionstaste mit der NAMED-Klausel des SET KEY-Statements zugewiesen wurde).

Wünschen Sie z.B., dass der Benutzer die Hilfe-Funktion durch Drücken von wahlweise PF3 oder PF12 aufrufen kann, weisen Sie beiden Tasten den gleichen Namen (im folgenden Beispiel: INFO) zu. Drückt der Benutzer eine dieser beiden Funktionstasten, wird die im IF-Statement definierte Verarbeitung ausgelöst.

```
...
SET KEY PF3  NAMED 'INFO'
        PF12 NAMED 'INFO'
INPUT USING MAP 'DEMO&'
IF *PF-NAME = 'INFO'
  WRITE 'Help is currently not active'
END-IF
...
```

Die mit `NAMED` definierten Funktionsnamen erscheinen in der Funktionstastenleiste:

```
Enter-PF1---PF2---PF3---PF4---PF5---PF6---PF7---PF8---PF9---PF10--PF11--PF12---
                  INFO                                                  INFO
```

Verarbeitung von Daten außerhalb des aktiven Fensters

Die folgenden Themen werden behandelt:

- Systemvariable *COM
- Beispiel für die Benutzung von *COM
- Positionierung des Cursors auf *COM — Terminalkommando %T*

Systemvariable *COM

Wie im Abschnitt *Bildschirm-Gestaltung — **Fenster*** weiter oben beschrieben, ist jeweils nur *ein* Fenster aktiv. In der Regel bedeutet dies, dass Eingaben nur innerhalb dieses Fensters möglich sind.

Mit der Systemvariablen `*COM`, die als Kommunikationsbereich betrachtet werden kann, ist es möglich, Eingaben auch außerhalb des aktiven Fensters zu machen.

Die Voraussetzung hierfür ist, dass die Map `*COM` als modifizierbares Feld enthält. Der Benutzer kann dann in dieses Feld Daten eingeben, auch wenn ein Fenster aktiv ist. Eine weitere Verarbeitung kann vom Inhalt der `*COM`-Variablen abhängig gemacht werden.

Auf diese Weise können Sie Benutzeroberflächen einrichten, wie sie z.B. bei dem Büro-Kommunikationssystem Con-nect implementiert sind: hier hat der Benutzer immer die Möglichkeit, Daten in die Kommandozeile einzugeben, auch wenn ein Fenster mit eigenen Eingabefeldern aktiv ist.

Beachten Sie, dass der Inhalt von `*COM` nur dann gelöscht wird, wenn die Natural-Session beendet wird.

Beispiel für die Benutzung von *COM

Im folgenden Beispiel führt das Programm ADD eine einfache Addition der Daten, die in der Map eingegeben werden, durch. In dieser Map wurde *COM als modifizierbares Feld definiert (am unteren Rand der Map), und zwar mit der im AL-Feld des Extended Field Editing angegebenen Länge. Das Ergebnis der Berechnung wird in einem Fenster ausgegeben. Obwohl dieses Fenster keine Eingabefelder hat, kann der Benutzer dennoch Eingaben machen, und zwar in das *COM-Feld außerhalb des Fensters.

Programm ADD:

```
DEFINE DATA LOCAL
1 #VALUE1 (N4)
1 #VALUE2 (N4)
1 #SUM3 (N8)
END-DEFINE
*
DEFINE WINDOW EMP
  SIZE 8*17
  BASE 10/2
  TITLE 'Total of Add'
  CONTROL SCREEN
  FRAMED POSITION SYMBOL BOT LEFT
*
INPUT USING MAP 'WINDOW'
*
COMPUTE #SUM3 = #VALUE1 + #VALUE2
*
SET WINDOW 'EMP'
INPUT (AD=O) / 'Value 1 +' /
               'Value 2 =' //
               ' ' #SUM3
*
IF *COM = 'M'
  FETCH 'MULTIPLY' #VALUE1 #VALUE2
END-IF
END
```

Ausgabe des Programms ADD:

```
                Map to Demonstrate Windows with *COM

                                CALCULATOR

                     Enter values you wish to calculate


                     Value 1:  12__
                     Value 2:  12__
+-Total of Add-+
!              !
! Value 1 +    !
! Value 2 =    !
!              !
!           24 !
!              !
+--------------+


 Next line is input field (*COM) for input outside the window:
```

Durch Eingabe von `M` wird die Funktion Multiplikation angestoßen; die beiden Werte aus der Eingabemaske werden miteinander multipliziert und das Ergebnis in einem zweiten Fenster ausgegeben:

```
                Map to Demonstrate Windows with *COM

                                CALCULATOR

                     Enter values you wish to calculate


                     Value 1:  12__
                     Value 2:  12__
+-Total of Add-+                              +--------------+
!              !                              !              !
! Value 1 +    !                              ! Value 1 x    !
! Value 2 =    !                              ! Value 2 =    !
!              !                              !              !
!           24 !                              !       144    !
!              !                              !              !
+--------------+                              +--------------+


 Next line is input field (*COM) for input outside the window:
             M
```

Positionierung des Cursors auf *COM — Terminalkommando %T*

Wenn ein Fenster aktiv ist und keine Eingabefelder (`AD=M` oder `AD=A`) enthält, wird der Cursor standardmäßig in die obere linke Ecke des Fensters positioniert.

Mit dem Terminalkommando `%T*` können Sie den Cursor auf die Systemvariable `*COM` außerhalb des Fensters positionieren, wenn das aktive Fenster keine Eingabefelder enthält.

Bei erneuter Eingabe von `%T*` wird die standardmäßige Positionierung des Cursors wieder aktiv.

Beispiel:

```
...
INPUT USING MAP 'WINDOW'
*
COMPUTE #SUM3 = #VALUE1 + #VALUE2
*
SET CONTROL 'T*'
SET WINDOW 'EMP'
INPUT (AD=O) / 'Value 1 +' /
               'Value 2 =' //
               ' ' #SUM3
...
```

Daten vom Bildschirm kopieren

Folgende Themen werden behandelt:

- Terminalkommandos %CS und %CC
- Eine Ausgabezeile eines Reports zur weiteren Verarbeitung auswählen

Terminalkommandos %CS und %CC

Mit diesen Terminalkommandos können sie Teile eines Bildschirms auf dem **Natural-Stack** (`%CS`) ablegen bzw. in die Systemvariable `*COM` (`%CC`) kopieren. Die geschützten Daten einer bestimmten Bildschirmzeile werden Feld für Feld kopiert.

Eine vollständige Beschreibung dieser Terminalkommandos finden Sie in der *Terminalkommandos*-Dokumentation.

Befinden sich die Daten auf dem Natural-Stack oder in `*COM`, stehen sie zur weiteren Verarbeitung zur Verfügung. Mit diesen Kommandos können sie benutzerfreundliche Anwendungen wie im nachfolgenden Beispiel erstellen.

Eine Ausgabezeile eines Reports zur weiteren Verarbeitung auswählen

Im folgenden Beispiel gibt das Programm `COM1` eine Liste der Mitarbeiter von `Abellan` bis `Alestia` aus.

Programm COM1:

```
DEFINE DATA LOCAL
1 EMP VIEW OF EMPLOYEES
  2 NAME(A20)
  2 MIDDLE-NAME (A20)
  2 PERSONNEL-ID (A8)
END-DEFINE
*
READ EMP BY NAME STARTING FROM 'ABELLAN' THRU 'ALESTIA'
  DISPLAY NAME
END-READ
FETCH 'COM2'
END
```

Ausgabe des Programms `COM1`:

```
  Page      1                                              2014-02-17  09:41:21

          NAME
  --------------------

  ABELLAN
  ACHIESON
  ADAM
  ADKINSON
  ADKINSON
  ADKINSON
  ADKINSON
  ADKINSON
  ADKINSON
  ADKINSON
  ADKINSON
  AECKERLE
  AFANASSIEV
  AFANASSIEV
  AHL
  AKROYD
  ALEMAN
  ALESTIA
  MORE
```

Die Kontrolle wird nun an das Programm `COM2` übergeben.

Programm COM2:

```
DEFINE DATA LOCAL
1 EMP VIEW OF EMPLOYEES
  2 NAME(A20)
  2 MIDDLE-NAME (A20)
  2 PERSONNEL-ID (A8)
1 SELECTNAME (A20)
END-DEFINE
*
SET KEY PF5 = '%CCC'
*
INPUT NO ERASE 'SELECT FIELD WITH CURSOR AND PRESS PF5'
*
MOVE *COM TO SELECTNAME
FIND EMP WITH NAME = SELECTNAME
  DISPLAY NAME PERSONNEL-ID
END-FIND
END
```

In diesem Programm ist das Terminalkommando `%CCC` der Funktionstaste PF5 zugeordnet. Das Terminalkommando kopiert alle geschützten Daten von der Zeile, in der sich der Cursor befindet, in die Systemvariable `*COM`. Diese Daten stehen dann zur weiteren Verarbeitung zur Verfügung. Die Art der Verarbeitung wird in den fett hervorgehobenen Zeilen des Beispielprogramms festgelegt.

Jetzt platziert der Benutzer den Cursor auf den Namen, der ihn interessiert, drückt PF5, und weitere Daten dieses Mitarbeiters werden ausgegeben.

```
SELECT FIELD WITH CURSOR AND PRESS PF5                    2014-02-17  09:44:25

         NAME
  --------------------

  ABELLAN
  ACHIESON
  ADAM  <==  Cursor positioned on name for which more information is required
  ADKINSON
  ADKINSON
  ADKINSON
  ADKINSON
  ADKINSON
  ADKINSON
  ADKINSON
  ADKINSON
  AECKERLE
  AFANASSIEV
  AFANASSIEV
  AHL
  AKROYD
  ALEMAN
  ALESTIA
```

In diesem Fall wird die Personalnummer (`PERSONNEL ID`) des ausgewählten Mitarbeiters angezeigt:

```
Page      1                                               2014-02-17  09:44:52

         NAME          PERSONNEL
                          ID
  -------------------- ---------

  ADAM                 50005800
```

Statements REINPUT/REINPUT FULL

Das Statement `REINPUT` dient dazu, zu einem `INPUT`-Statement zurückzukehren und dieses erneut auszuführen. In der Regel wird es dazu benutzt, eine Fehlermeldung auszugeben, die dem Benutzer sagt, dass auf das `INPUT`-Statement hin ungültige Daten eingegeben wurden.

Wenn Sie die Option `FULL` in einem `REINPUT`-Statement angeben, wird das entsprechende `INPUT`-Statement vollständig neu ausgeführt:

- Bei einem normalen `REINPUT`-Statement (ohne `FULL`-Option) werden Inhalte von Variablen, die zwischen `INPUT`- und `REINPUT`-Statement geändert wurden, nicht angezeigt; d.h. alle Variablen

auf dem Schirm zeigen den Inhalt, den Sie hatten, als das INPUT-Statement ursprünglich ausgeführt wurde.

- Bei einem REINPUT FULL-Statement werden alle nach der ersten Ausführung des INPUT-Statements gemachten Änderungen sichtbar, wenn das INPUT-Statement erneut ausgeführt wird; d.h. alle Variablen auf dem Schirm haben den Inhalt, den sie zum Zeitpunkt der Ausführung des REINPUT-Statements hatten.
- Wollen Sie zusätzlich den Cursor auf ein bestimmtes Feld positionieren, so können Sie die MARK-Option verwenden. Um auf eine bestimmte Position innerhalb eines Feldes zu positionieren, geben Sie die MARK POSITION-Option an.

Das Beispiel unten veranschaulicht die Anwendung von REINPUT FULL mit MARK POSITION.

```
DEFINE DATA LOCAL
1 #A (A10)
1 #B (N4)
1 #C (N4)
END-DEFINE
*
INPUT (AD=M) #A #B #C
IF #A = ' '
  COMPUTE #B = #B + #C
  RESET #C
  REINPUT FULL 'Enter a value' MARK POSITION 5 IN *#A
END-IF
END
```

Der Benutzer gibt 3 in das Feld #B und 3 in das Feld #C ein und drückt EINGABE.

```
  #A            #B     3 #C     3
```

Das Programm verlangt, dass das Feld #A einen Wert enthält. Das Statement REINPUT FULL mit MARK POSITION 5 IN *#A gibt den Eingabeschirm erneut aus, und zwar mit der geänderten Variable #B, die jetzt den Wert 6 enthält (Stand nach der COMPUTE-Berechnung). Der Cursor wird an die 5. Stelle in das für Neueingaben bereite Feld #A platziert.

```
Enter name of field
  #A    _       #B     6 #C     0



Enter a value
```

Das gleiche Statement ohne die FULL-Option würde folgenden Bildschirm ausgeben. Beachten Sie, dass die Variablen #B and #C auf den Stand zur Ausführungszeit des INPUT-Statements zurückgesetzt wurden (beide Felder enthalten den Wert 3).

```
#A     _          #B       3 #C       3
```

Objektorientierte Datenverarbeitung — der Natural-Kommando-Prozessor

Der Natural-Kommando-Prozessor wird zur Definition und Steuerung der Navigation innerhalb einer Anwendung benutzt.

- Der *Entwicklungsteil* ist die Utility SYSNCP. Mit dieser Utility definieren Sie Kommandos, d.h. Kombinationen von Schlüsselwörtern, und die Aktionen, die als Reaktion auf die Ausführung dieser Kommandos ausgeführt werden sollen. Aus Ihren Definitionen erzeugt SYSNCP Entscheidungstabellen, die festlegen, was passiert, wenn ein Benutzer ein Kommando eingibt.
- Der *Laufzeitteil* ist das Statement `PROCESS COMMAND`. Dieses Statement wird benutzt, um den Kommando-Prozessor in einem Natural-Programm aufzurufen. Im Statement geben Sie den Namen der `SYSNCP`-Tabelle an, die benutzt wird, um die Dateneingabe eines Benutzers zu diesem Zeitpunkt zu verarbeiten.

Weitere Informationen zum Natural-Kommando-Prozessor siehe *SYSNCP Utility* in der *Utilities*-Dokumentation und das Statement `PROCESS COMMAND` in der *Statements*-Dokumentation.

X NaturalX

Dieser Teil beschreibt, wie objektbasierte Anwendungen entwickelt werden.

64 Einführung in NaturalX

Dieses Kapitel enthält eine kurze Einführung in die komponentenbasierte Programmierung, und damit einhergehend die Benutzung der NaturalX-Schnittstelle und einem speziell für diesen Zweck vorgesehenen Satz Natural-Statements.

Warum NaturalX?

Auf Komponenten-Architektur basierende Software-Anwendungen bieten viele Vorteile gegenüber traditionellen Designs. Diese sind u.a. die Folgenden:

- Schnellere Entwicklung. Die Programmierer können Anwendungen schneller erstellen, indem sie die Software aus vorerstellten Komponenten zusammensetzen.
- Geringere Entwicklungskosten. Eine allgemeine Menge von Schnittstellen für Programme zur Verfügung zu haben, bedeutet weniger Arbeit bei der Integration der Komponenten in vollständige Lösungen.
- Höhere Flexibilität. Es ist leichter, Software für unterschiedliche Abteilungen innerhalb eines Unternehmens zu standardisieren, indem Sie einfach einige der Komponenten verändern, aus denen die Anwendung besteht.
- Reduzierte Wartungskosten. Im Falle eines Umstiegs auf eine neue Version ist es häufig ausreichend, einige der Komponenten zu ändern, anstatt die gesamte Anwendung ändern zu müssen.

Mit NaturalX können Sie komponentenbasierte Anwendungen erstellen.

Sie können NaturalX einsetzen, um einen komponentenbasierten Programmierstil zu pflegen. Allerdings können auf Großrechner- und UNIX-Plattform die Komponenten nicht verteilt werden und nur in einer lokalen Natural-Session laufen.

65 NaturalX-Anwendungen entwickeln

Dieses Dokument beschreibt, wie eine Anwendung durch Definition und Benutzung von Klassen entwickelt wird.

Entwickungsumgebungen

- **Klassen auf Windows-Plattformen entwickeln**
  Auf Windows-Plattformen stellt Natural den Class Builder als Werkzeug zur Entwicklung von Natural-Klassen zur Verfügung. Der Class Builder präsentiert eine Natural-Klasse in einer strukturierten hierarchischen Reihenfolge und ermöglicht es dem Benutzer, die Klassen und ihre Komponenten effizient zu verwalten. Wenn Sie den Class Builder benutzen, sind überhaupt keine Vorkenntnisse oder auch nur Grundkenntnisse der unten beschriebenen Syntax-Elemente erforderlich.
- **Klassen mit SPoD entwickeln**
  In einer Natural Single Point of Development-Umgebung (SPoD), der zentralen Umgebung für Entwicklungen einschließlich eines Großrechners und/oder UNIX Remote Development Servers (DFÜ-Entwicklungsserver) können Sie den Class Builder benutzen, der zusammen mit der Natural Studio-Benutzeroberfläche zur Entwicklung von Klassen auf Großrechnern und/oder UNIX-Plattformen zur Verfügung steht. In diesem Fall sind keine Vorkenntnisse oder auch nur Grundkenntnisse der im Folgenden beschriebenen Syntax-Elemente erforderlich.
- **Klassen auf Großrechner- oder UNIX-Plattformen entwickeln**
  Wenn Sie SpoD nicht benutzen, entwickeln Sie mittels des Natural-Programm-Editors Klassen auf diesen Plattformen. In diesem Fall sollten Sie die Syntax der unten beschriebenen Klassen-Definition kennen.

Klassen definieren

Wenn Sie eine Klasse definieren, müssen Sie ein Natural-Klassenmodul definieren, innerhalb dessen Sie ein `DEFINE CLASS`-Statement erstellen. Mittels des `DEFINE CLASS`-Statements können Sie der Klasse einen extern verwendbaren Namen zuweisen und ihre Schnittstellen, Methoden und Eigenschaften definieren. Der Klasse kann auch ein Objekt-Datenbereich zugewiesen werden, der das Layout einer Instanz der Klasse beschreibt.

Dieser Abschnitt umfasst die folgenden Themen:

- Methoden implementieren

Natural-Klassenmodul erstellen

Um ein Natural-Klassenmodul zu erstellen

- Benutzen Sie das `CREATE OBJECT`-Statement zur Erstellung eines Natural-Objekts des Typs Klasse.

Klasse spezifizieren

Das `DEFINE CLASS`-Statement definiert den Namen der Klasse, die Schnittstellen, die die Klasse unterstützt, und die Struktur ihrer Objekte.

Um eine Klasse zu spezifizieren

- Benutzen Sie das `DEFINE CLASS`-Statement, das in der *Statements*-Dokumentation beschrieben ist.

Schnittstelle definieren

Jede Schnittstelle einer Klasse wird mit einem `INTERFACE`-Statement innerhalb der Klassen-Definition angegeben. Ein `INTERFACE`-Statement gibt den Namen der Schnittstelle und eine Anzahl von Eigenschaften und Methoden an. Für Klassen, die als COM−Klassen registriert werden sollen, gibt es auch die Globally Unique ID der Schnittstelle an.

Eine Klasse kann eine oder mehrere Schnittstellen haben. Für jede Schnittstelle wird ein `INTERFACE`-Statement in der Klassen-Definition kodiert. Jedes `INTERFACE`-Statement enthält eine oder mehrere `PROPERTY`- und `METHOD`-Klauseln. Gewöhnlich stehen die in einer Schnittstelle enthaltenen Eigenschaften und Methoden miteinander in einem technischen oder betriebswirtschaftlichen Zusammenhang.

Die `PROPERTY`-Klausel definiert den Namen einer Eigenschaft und weist der Eigenschaft eine Variable vom Objekt-Datenbereich zu. Diese Variable wird zum Speichern des Wertes der Eigenschaft benutzt.

Die `METHOD`-Klausel definiert den Namen einer Methode und weist der Methode ein Subprogramm zu. Dieses Subprogramm wird zum Implementieren der Methode benutzt.

Um eine Schnittstelle zu definieren

- Benutzen Sie das `INTERFACE`-Statement (siehe *Statements*-Dokumentation).

Objekt-Datenvariable einer Property zuweisen

Das PROPERTY-Statement wird nur benutzt, wenn mehrere Klassen dieselbe Schnittstelle auf verschiedene Art und Weise implementieren sollen. In diesem Fall benutzen die Klassen dieselbe Schnittstellen-Definition gemeinsam und beziehen sie von einem Natural-**Copycode** ein. Das PROPERTY-Statement wird dann benutzt, um *außerhalb* der Schnittstellen-Definition eine Variable vom Objekt-Datenbereich einer Eigenschaft zuzuweisen. Wie die PROPERTY-Klausel des INTERFACE-Statements definiert das PROPERTY-Statement den Namen einer Eigenschaft und weist eine Variable vom Objekt-Datenbereich der Eigenschaft zu. Diese Variable wird zum Speichern des Wertes der Eigenschaft benutzt.

Um eine Objekt-Datenvariable einer Eigenschaft zuzuweisen

- Benutzen Sie das PROPERTY-Statement (siehe *Statements*-Dokumentation).

Subprogramm einer Methode zuweisen

Das METHOD-Statement wird nur benutzt, wenn mehrere Klassen dieselbe Schnittstelle auf verschiedene Arten implementieren sollen. In diesem Fall benutzen die Klassen dieselbe Schnittstellen-Definition gemeinsam und beziehen sie von einem Natural-**Copycode** ein. Das METHOD-Statement wird dann benutzt, um außerhalb der Schnittstellen-Definition der Methode ein Subprogramm zuzuweisen. Wie die METHOD-Klausel des INTERFACE-Statements definiert das METHOD-Statement den Namen einer Methode und weist der Methode ein Subprogramm zu. Dieses Subprogramm wird zum Implementieren der Methode benutzt.

Um einer Methode ein Subprogramm zuzuweisen

- Benutzen Sie das METHOD-Statement (siehe *Statements*-Dokumentation).

Methoden implementieren

Eine Methode wird in der folgenden allgemeinen Form als ein Natural-Subprogramm implementiert:

```
DEFINE DATA
*
* Implementation code of the method
*
END
```

Informationen zum DEFINE DATA-Statement siehe *Statements*-Dokumentation.

Alle Klauseln des DEFINE DATA-Statements sind optional.

Um die Daten-Konsistenz sicherzustellen, empfiehlt es sich, dass Sie keine Inline-Datendefinitionen, sondern Data Areas benutzen.

Wenn eine PARAMETER clause-Klausel angegeben wird, kann die Methode Parameter und/oder einen Rückgabewert haben.

Mit BY VALUE in der Parameter Data Area markierte Parameter sind Eingabe-Parameter der Methode.

Nicht mit BY VALUE markierte Parameter werden „By Reference“ übergeben und sind Eingabe/Ausgabe-Parameter. Dies ist die Voreinstellung.

Der erste mit BY VALUE RESULT markierte Parameter wird als Rückgabewert für die Methode zurückgegeben. Wenn mehr als ein Parameter auf diese Art markiert wird, werden die anderen als Eingabe/Ausgabe-Parameter behandelt.

Als OPTIONAL markierte Parameter brauchen nicht angegeben zu werden, wenn die Methode aufgerufen wird und Sie die *n*X-Notation im SEND METHOD-Statement benutzen.

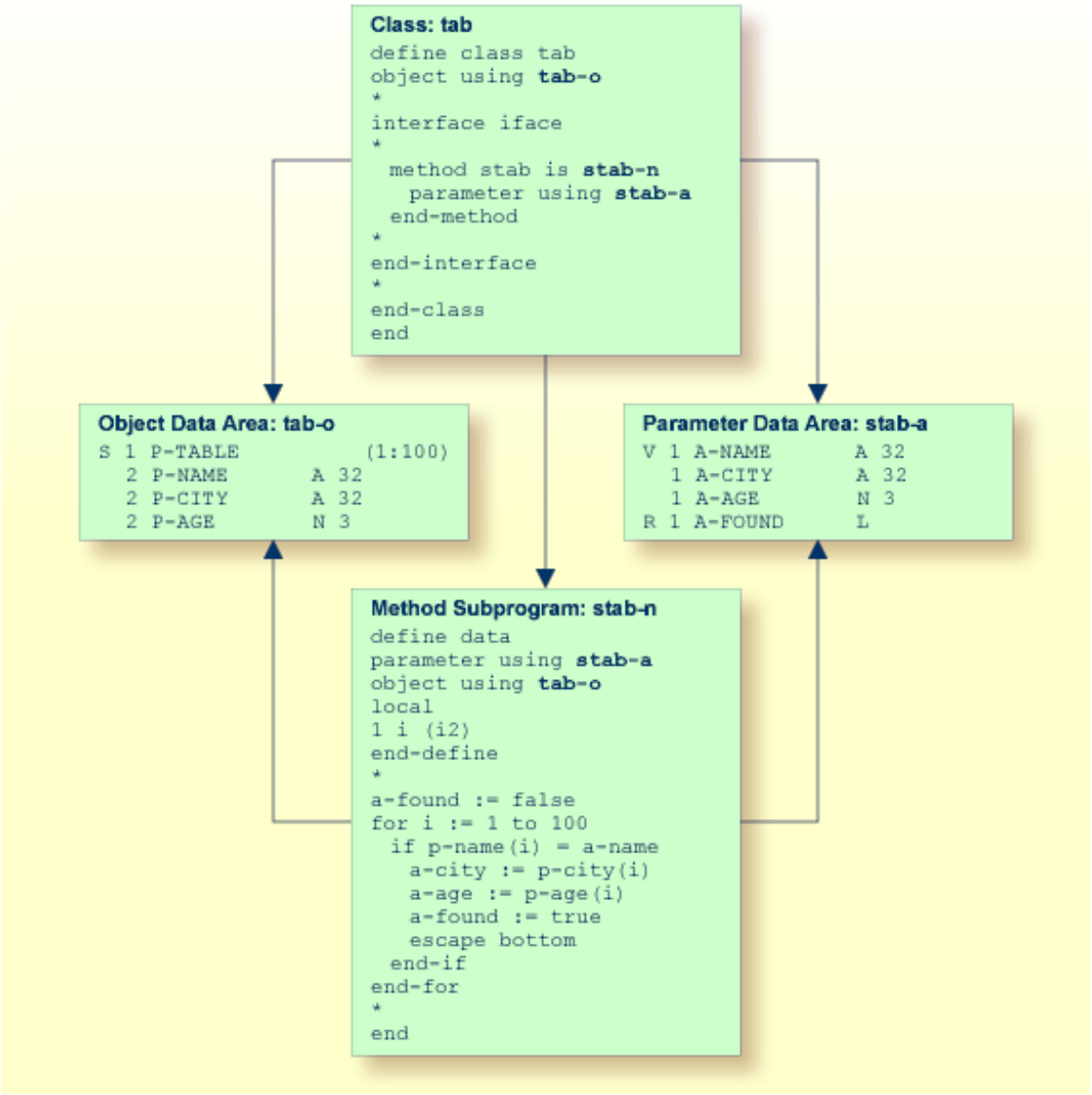
Um sicherzustellen, dass das Methoden-Subprogramm genau dieselben Parameter akzeptiert, wie in dem entsprechenden METHOD-Statement in der Klassen-Definition angegeben, benutzen Sie eine Parameter Data Area anstatt von Inline-Datendefinitionen. Benutzen Sie dieselbe Parameter Data Area wie in dem betreffenden METHOD-Statement.

Um dem Methoden-Subprogramm Zugriff auf die Objektdaten-Struktur zu geben, kann die OBJECT-Klausel angegeben werden. Um sicherzustellen, dass das Methoden-Subprogramm korrekt auf die Objektdaten zugreifen kann, benutzen Sie eine Local Data Area anstatt von Inline-Datendefinitionen. Verwenden Sie dieselbe Local Data Area, wie in der OBJECT-Klausel des DEFINE CLASS-Statements angegeben.

Die Klauseln GLOBAL, LOCAL und INDEPENDENT können wie in jedem anderen Natural-Programm benutzt werden.

Obwohl es technisch möglich ist, macht es gewöhnlich keinen Sinn, eine CONTEXT-Klausel in einem Methoden-Subprogramm zu benutzen.

In dem folgenden Beispiel werden Daten über eine vorgegebene Person von einer Tabelle eingelesen. Der Suchschlüssel wird als ein BY VALUE-Parameter übergeben. Die sich daraus ergebenden Daten werden über “by reference”-Parameter zurückgegeben („By Reference“ ist die Standard-Definition). Der Rückgabewert der Methode wird durch die Spezifikation BY VALUE RESULT definiert.



Klassen und Objekte benutzen

Auf in einer lokalen Natural-Session erstellte Objekte kann von anderen Modulen aus in derselben Natural-Session zugegriffen werden.

Das Statement `CREATE OBJECT` wird benutzt, um ein Objekt (auch als *Instance* bezeichnet) einer gegebenen Klasse zu erstellen.

Um Objekte in Natural-Programmen zu referenzieren, müssen Object-Handles im `DEFINE DATA`-Statement definiert werden. Methoden eines Objekts werden mit dem Statement `SEND METHOD`

aufgerufen. Objekte können Eigenschaften haben, auf die mit der normalen Zuweisungssyntax zugegriffen wird.

Diese Schritte sind im Folgenden beschrieben:

- Objekt-Handles definieren
- Instanz einer Klasse erstellen
- Bestimmte Methode eines Objekts aufrufen
- Properties aufrufen

Objekt-Handles definieren

Um Objekte in Natural-Programmen zu referenzieren, müssen Object-Handles im `DEFINE DATA`-Statement wie folgt definiert werden:

```
DEFINE DATA
  level-handle-name [(array-definition)] HANDLE OF OBJECT
  ...
END-DEFINE
```

Beispiel:

```
DEFINE DATA LOCAL
1 #MYOBJ1 HANDLE OF OBJECT
1 #MYOBJ2 (1:5) HANDLE OF OBJECT
END-DEFINE
```

Instanz einer Klasse erstellen

Um eine Instanz einer Klasse zu erstellen

- Benutzen Sie das `CREATE OBJECT`-Statement (siehe Natural *Statements*-Dokumentation).

Bestimmte Methode eines Objekts aufrufen

Um eine bestimmte Methode eines Objekts aufzurufen

- Benutzen Sie das `SEND METHOD`-Statement (siehe Natural *Statements*-Dokumentation).

Properties aufrufen

Auf Properties kann mit dem Statement `ASSIGN` (oder `COMPUTE`) wie folgt zugegriffen werden:

```
ASSIGN operand1.property-name = operand2
ASSIGN operand2 = operand1.property-name
```

Object Handle — *operand1*

Operand1 muss als eine Object-Handle definiert werden und identifiziert das Objekt, dessen Eigenschaft aufgerufen werden soll. Das Objekt muss bereits vorhanden sein.

operand2

Als *operand2* geben Sie einen Operanden an, dessen Format datenübertragungskompatibel zu dem Format der Eigenschaft sein muss. Weitere Informationen siehe ***Kompatibilitätsregeln zur Datenübertragung***.

property-name

Der Name einer Property des Objekts.

Stimmt der Name der Property mit der Natural Identifier Syntax überein, kann er wie folgt angegeben werden:

```
create object #o1 of class "Employee"
  #age := #o1.Age
```

Wenn der Name der Property nicht mit der Natural Identifier Syntax übereinstimmt, muss er in spitze Klammern gesetzt werden:

```
create object #o1 of class "Employee"
  #salary := #o1.<<%Salary>>
```

Der Name der Property kann auch mit einem Schnittstellen-Namen qualifiziert werden. Dies ist erforderlich, wenn das Objekt mehr als eine Schnittstelle hat, die eine Property mit demselben Namen enthält. In diesem Fall muss der qualifizierte Name der Property in spitzen Klammern stehen:

```
create object #o1 of class "Employee"
  #age := #o1.<<PersonalData.Age>>
```

Beispiel:

```
define data
  local
  1 #i         (i2)
  1 #o         handle of object
  1 #p         (5) handle of object
  1 #q         (5) handle of object
  1 #salary    (p7.2)
  1 #history   (p7.2/1:10)
  end-define
  * ...
  * Code omitted for brevity.
  * ...
  * Set/Read the Salary property of the object #o.
  #o.Salary := #salary
  #salary := #o.Salary
  * Set/Read the Salary property of
  * the second object of the array #p.
  #p.Salary(2) := #salary
  #salary := #p.Salary(2)
  *
  * Set/Read the SalaryHistory property of the object #o.
  #o.SalaryHistory := #history(1:10)
  #history(1:10) := #o.SalaryHistory
  * Set/Read the SalaryHistory property of
  * the second object of the array #p.
  #p.SalaryHistory(2) := #history(1:10)
  #history(1:10) := #p.SalaryHistory(2)
  *
  * Set the Salary property of each object in #p to the same value.
  #p.Salary(*) := #salary
  * Set the SalaryHistory property of each object in #p
  * to the same value.
  #p.SalaryHistory(*) := #history(1:10)
  *
  * Set the Salary property of each object in #p to the value
  * of the Salary property of the corresponding object in #q.
  #p.Salary(*) := #q.Salary(*)
  * Set the SalaryHistory property of each object in #p to the value
  * of the SalaryHistory property of the corresponding object in #q.
  #p.SalaryHistory(*) := #q.SalaryHistory(*)
  *
  end
```

Um Arrays von Objekt-Handles und Eigenschaften mit Arrays als Werte korrekt benutzen zu können, ist es wichtig, Folgendes zu wissen:

Eine Eigenschaft einer Array-Ausprägung von Objekt-Handles wird mit der folgenden Index-Notation adressiert:

```
#p.Salary(2) := #salary
```

Auf eine Eigenschaft, die ein Array als Wert hat, wird stets als Ganzes zugegriffen. Deshalb ist keine Index-Notation mit dem Namen der Eigenschaft erforderlich:

```
#o.SalaryHistory := #history(1:10)
```

Eine Eigenschaft einer Array-Ausprägung von Objekt-Handles, die ein Array als Wert hat, wird deswegen wie folgt adressiert:

```
#p.SalaryHistory(2) := #history(1:10)
```

XI

66 Für Natural reservierte Schlüsselwörter

Dieses Kapitel enthält eine Liste aller in der Natural-Programmiersprache reservierten Schlüsselwörter.

Wichtig: Um mögliche Namenskonflikte zu vermeiden, empfiehlt es sich sehr, diese für Natural reservierten Schlüsselwörter nicht als Namen für Variablen zu benutzen.

Alphabetische Liste der für Natural reservierten Schlüsselwörter

Die folgende Liste ist eine Übersicht der für Natural reservierten Schlüsselwörter und dient nur der allgemeinen Information. Benutzen Sie in Zweifelsfällen die **Schlüsselwort-Prüffunktion** des Compilers.

[A | B | C | D | E | F | G | H | I | J | K | L | M | N | O | P | Q | R | S | T | U | V | W | X | Y | Z]

- A -

ABS
ABSOLUTE
ACCEPT
ACTION
ACTIVATION
AD
ADD
AFTER
AL
ALARM
ALL
ALPHA
ALPHABETICALLY
AND
ANY
APPL
APPLICATION
ARRAY
AS
ASC
ASCENDING
ASSIGN
ASSIGNING
ASYNC
AT
ATN
ATT

ATTRIBUTES
AUTH
AUTHORIZATION
AUTO
AVER
AVG

- B -

BACKOUT
BACKWARD
BASE
BEFORE
BETWEEN
BLOCK
BOT
BOTTOM
BREAK
BROWSE
BUT
BX
BY

- C -

CABINET
CALL
CALLDBPROC
CALLING
CALLNAT
CAP
CAPTIONED
CASE
CC
CD
CDID
CF
CHAR
CHARLENGTH
CHARPOSITION
CHILD
CIPH
CIPHER
CLASS
CLOSE
CLR

COALESCE
CODEPAGE
COMMAND
COMMIT
COMPOSE
COMPRESS
COMPUTE
CONCAT
CONDITION
CONST
CONSTANT
CONTEXT
CONTROL
CONVERSATION
COPIES
COPY
COS
COUNT
COUPLED
CS
CURRENT
CURSOR
CV

- D -

DATA
DATAAREA
DATE
DAY
DAYS
DC
DECIDE
DECIMAL
DEFINE
DEFINITION
DELETE
DELIMITED
DELIMITER
DELIMITERS
DESC
DESCENDING
DIALOG
DIALOG-ID
DIGITS

DIRECTION
DISABLED
DISP
DISPLAY
DISTINCT
DIVIDE
DL
DLOGOFF
DLOGON
DNATIVE
DNRET
DO
DOCUMENT
DOEND
DOWNLOAD
DU
DY
DYNAMIC

- E -

EDITED
EJ
EJECT
ELSE
EM
ENCODED
END
END-ALL
END-BEFORE
END-BREAK
END-BROWSE
END-CLASS
END-DECIDE
END-DEFINE
END-ENDDATA
END-ENDFILE
END-ENDPAGE
END-ERROR
END-FILE
END-FIND
END-FOR
END-FUNCTION
END-HISTOGRAM
ENDHOC

END-IF
END-INTERFACE
END-LOOP
END-METHOD
END-NOREC
END-PARAMETERS
END-PARSE
END-PROCESS
END-PROPERTY
END-PROTOTYPE
END-READ
END-REPEAT
END-RESULT
END-SELECT
END-SORT
END-START
END-SUBROUTINE
END-TOPPAGE
END-WORK
ENDING
ENTER
ENTIRE
ENTR
EQ
EQUAL
ERASE
ERROR
ERRORS
ES
ESCAPE
EVEN
EVENT
EVERY
EXAMINE
EXCEPT
EXISTS
EXIT
EXP
EXPAND
EXPORT
EXTERNAL
EXTRACTING

- F -

FALSE
FC
FETCH
FIELD
FIELDS
FILE
FILL
FILLER
FINAL
FIND
FIRST
FL
FLOAT
FOR
FORM
FORMAT
FORMATTED
FORMATTING
FORMS
FORWARD
FOUND
FRAC
FRAMED
FROM
FS
FULL
FUNCTION
FUNCTIONS

- G -

GC
GE
GEN
GENERATED
GET
GFID
GIVE
GIVING
GLOBAL
GLOBALS
GREATER
GT
GUI

- H -

HANDLE
HAVING
HC
HD
HE
HEADER
HEX
HISTOGRAM
HOLD
HORIZ
HORIZONTALLY
HOUR
HOURS
HW

- I -

IA
IC
ID
IDENTICAL
IF
IGNORE
IM
IMMEDIATE
IMPORT
IN
INC
INCCONT
INCDIC
INCDIR
INCLUDE
INCLUDED
INCLUDING
INCMAC
INDEPENDENT
INDEX
INDEXED
INDICATOR
INIT
INITIAL
INNER
INPUT
INSENSITIVE

INSERT
INT
INTEGER
INTERCEPTED
INTERFACE
INTERFACE4
INTERMEDIATE
INTERSECT
INTO
INVERTED
INVESTIGATE
IP
IS
ISN

- J -

JOIN
JSON
JUST
JUSTIFIED

- K -

KD
KEEP
KEY
KEYS

- L -

LANGUAGE
LAST
LC
LE
LEAVE
LEAVING
LEFT
LENGTH
LESS
LEVEL
LIB
LIBPW
LIBRARY
LIBRARY-PASSWORD
LIKE

LIMIT
LINDICATOR
LINES
LISTED
LOCAL
LOCKS
LOG
LOG-LS
LOG-PS
LOGICAL
LOOP
LOWER
LS
LT

- M -

MACROAREA
MAP
MARK
MASK
MAX
MC
MCG
MESSAGES
METHOD
MGID
MICROSECOND
MIN
MINUTE
MODAL
MODIFIED
MODULE
MONTH
MORE
MOVE
MOVING
MP
MS
MT
MULTI-FETCH
MULTIPLY

- N -

NAME

NAMED
NAMESPACE
NATIVE
NAVER
NC
NCOUNT
NE
NEWPAGE
NL
NMIN
NO
NODE
NOHDR
NONE
NORMALIZE
NORMALIZED
NOT
NOTIT
NOTITLE
NULL
NULL-HANDLE
NUMBER
NUMERIC

- O -

OBJECT
OBTAIN
OCCURRENCES
OF
OFF
OFFSET
OLD
ON
ONCE
ONLY
OPEN
OPTIMIZE
OPTIONAL
OPTIONS
OR
ORDER
OUTER
OUTPUT

- P -

PACKAGESET
PAGE
PARAMETER
PARAMETERS
PARENT
PARSE
PASS
PASSW
PASSWORD
PATH
PATTERN
PA1
PA2
PA3
PC
PD
PEN
PERFORM
PF*n* (*n* = 1 to 9)
PF*nn*(*nn* = 10 to 99)
PGDN
PGUP
PGM
PHYSICAL
PM
POLICY
POS
POSITION
PREFIX
PRINT
PRINTER
PROCESS
PROCESSING
PROFILE
PROGRAM
PROPERTY
PROTOTYPE
PRTY
PS
PT
PW

- Q -

QUARTER
QUERYNO

- R -

RD
READ
READONLY
REC
RECORD
RECORDS
RECURSIVELY
REDEFINE
REDUCE
REFERENCED
REFERENCING
REINPUT
REJECT
REL
RELATION
RELATIONSHIP
RELEASE
REMAINDER
REPEAT
REPLACE
REPORT
REPORTER
REPOSITION
REQUEST
REQUIRED
RESET
RESETTING
RESIZE
RESPONSE
RESTORE
RESULT
RET
RETAIN
RETAINED
RETRY
RETURN
RETURNS
REVERSED
RG

RIGHT
ROLLBACK
ROUNDED
ROUTINE
ROW
ROWS
RR
RS
RULEVAR
RUN

- S -

SA
SAME
SCAN
SCREEN
SCROLL
SECOND
SELECT
SELECTION
SEND
SENSITIVE
SEPARATE
SEQUENCE
SERVER
SET
SETS
SETTIME
SF
SG
SGN
SHORT
SHOW
SIN
SINGLE
SIZE
SKIP
SL
SM
SOME
SORT
SORTED
SORTKEY
SOUND

SPACE
SPECIFIED
SQL
SQLID
SQRT
STACK
START
STARTING
STATEMENT
STATIC
STATUS
STEP
STOP
STORE
SUBPROGRAM
SUBPROGRAMS
SUBROUTINE
SUBSTR
SUBSTRING
SUBTRACT
SUM
SUPPRESS
SUPPRESSED
SUSPEND
SYMBOL
SYNC
SYSTEM

- T -

TAN
TC
TERMINATE
TEXT
TEXTAREA
TEXTVARIABLE
THAN
THEM
THEN
THRU
TIME
TIMESTAMP
TIMEZONE
TITLE
TO

TOP
TOTAL
TP
TR
TRAILER
TRANSACTION
TRANSFER
TRANSLATE
TREQ
TRUE
TS
TYPE
TYPES

- U -

UC
UNDERLINED
UNION
UNIQUE
UNKNOWN
UNTIL
UPDATE
UPLOAD
UPPER
UR
USED
USER
USING

- V -

VAL
VALUE
VALUES
VARGRAPHIC
VARIABLE
VARIABLES
VERT
VERTICALLY
VIA
VIEW

- W -

WH

WHEN
WHERE
WHILE
WINDOW
WITH
WORK
WRITE
WITH_CTE

- X -

XML

- Y -

YEAR

- Z -

ZD
ZP

Prüfung auf für Natural reservierte Schlüsselwörter durchführen

Es gibt eine Untermenge von Natural-Schlüsselwörtern, die zweideutig wären, wenn sie als Namen für Variablen benutzt würden. Dies sind insbesondere Schlüsselwörter, die Natural-Statements (`ADD`, `FIND` usw.) oder System-Funktionen (`ABS`, `SUM` usw.) identifizieren. Wenn Sie ein solches Schlüsselwort als Namen einer Variable benutzen, können Sie diese Variable nicht im Zusammenhang mit optionalen Operanden (mit `CALLNAT`, `WRITE` usw.) benutzen.

Beispiel:

```
DEFINE DATA LOCAL
1 ADD (A10)
END-DEFINE
CALLNAT 'MYSUB' ADD 4     /* ADD is regarded as ADD statement
END
```

Um Variablennamen in einem Programmierobjekt auf solche für Natural reservierten Schlüsselwörter zu überprüfen, können Sie eine der folgenden Funktionen benutzen:

- den `KCHECK`-Schlüsselwort-Parameter des `CMPO`-Profilparameters oder des `NTCMPO`-Parametermakros oder
- die `KCHECK`-Option des `COMPOPT`-Systemkommandos.

Die folgende Tabelle enthält eine Liste der für Natural reservierten Schlüsselwörter, die von KCHECK überprüft werden.

A - D	E - F	G - P	R - S	T - W
ABS	EJECT	GET	READ	TAN
ACCEPT	ELSE	HISTOGRAM	REDEFINE	TERMINATE
ADD	END	IF	REDUCE	TOP
ALL	END-ALL	IGNORE	REINPUT	TOTAL
ANY	END-BEFORE	IMPORT	REJECT	TRANSFER
ASSIGN	END-BREAK	INCCONT	RELEASE	TRUE
AT	END-BROWSE	INCDIC	REPEAT	UNTIL
ATN	END-DECIDE	INCDIR	REQUEST	UPDATE
AVER	END-ENDDATA	INCLUDE	RESET	UPLOAD
BACKOUT	END-ENDFILE	INCMAC	RESIZE	VAL
BEFORE	END-ENDPAGE	INPUT	RESTORE	VALUE
BREAK	END-ERROR	INSERT	RET	VALUES
BROWSE	END-FILE	INT	RETRY	WASTE
CALL	END-FIND	INVESTIGATE	RETURN	WHEN
CALLDBPROC	END-FOR	LIMIT	ROLLBACK	WHILE
CALLNAT	END-FUNCTION	LOG	ROUNDED	WITH_CTE
CLOSE	END-HISTOGRAM	LOOP	RULEVAR	WRITE
COMMIT	ENDHOC	MAP	RUN	
COMPOSE	END-IF	MAX	SELECT	
COMPRESS	END-LOOP	MIN	SEND	
COMPUTE	END-NOREC	MOVE	SEPARATE	
COPY	END-PARSE	MULTIPLY	SET	
COS	END-PROCESS	NAVER	SETTIME	
COUNT	END-READ	NCOUNT	SGN	
CREATE	END-REPEAT	NEWPAGE	SHOW	
DECIDE	END-RESULT	NMIN	SIN	
DEFINE	END-SELECT	NONE	SKIP	
DELETE	END-SORT	NULL-HANDLE	SORT	
DISPLAY	END-START	OBTAIN	SORTKEY	
DIVIDE	END-SUBROUTINE	OLD	SQRT	
DLOGOFF	END-TOPPAGE	ON	STACK	
DLOGON	END-WORK	OPEN	START	
DNATIVE	ENTIRE	OPTIONS	STOP	
DO	ESCAPE	PARSE	STORE	
DOEND	EXAMINE	PASSW	SUBSTR	
DOWNLOAD	EXP	PERFORM	SUBSTRING	
	EXPAND	POS	SUBTRACT	
	EXPORT	PRINT	SUM	
	FALSE	PROCESS	SUSPEND	
	FETCH			
	FIND			
	FOR			
	FORMAT			
	FRAC			

Standardmäßig wird keine Schlüsselwort-Prüfung durchgeführt.

67 Referenzierte Beispielprogramme

Dieses Kapitel enthält einige zusätzliche Beispielprogramme, auf die im *Leitfaden zur Programmierung* verwiesen wird.

Anmerkung zu Beispiel-Libraries:

- Den Quellcode der Beispielprogramme finden Sie in der Natural-Library SYSEXPG. Die Beispielprogramme greifen auf die Daten der Dateien EMPLOYEES (Personaldaten) und VEHICLES (Fahrzeugdaten) zu, die zu Demonstrationszwecken erstellt wurden. Die Natural-Library SYSEXPG enthält auch Beispielprogramme für ***Functions***.
- Weitere Beispielprogramme für die Verwendung von Natural-Statements finden Sie in der Natural-Library SYSEXSYN. Diese Beispiele sind außerdem im Abschnitt *Referenzierte Beispielprogramme* in der *Statements*-Dokumentation enthalten.
- Wenden Sie sich wegen der Verfügbarkeit dieser Bibliotheken in Ihrer Umgebung an Ihren Natural-Administrator.
- Damit die Natural-Beispielprogramme auf eine Adabas-Datenbank zugreifen können, muss der Adabas-Nucleus-Parameter OPTIONS auf TRUNCATION gesetzt sein.

READ-Statement

Auf das folgende Beispiel wird im Abschnitt ***Datenbankzugriffe*** verwiesen.

READX03 - READ-Statement (mit LOGICAL-Klausel)

```
** Example 'READX03': READ (with LOGICAL clause)
************************************************************************
DEFINE DATA LOCAL
1 EMPLOY-VIEW VIEW OF EMPLOYEES
  2 NAME
  2 PERSONNEL-ID
  2 JOB-TITLE
END-DEFINE
*
LIMIT 8
READ EMPLOY-VIEW LOGICAL BY PERSONNEL-ID
  DISPLAY NOTITLE  *ISN      NAME
                  'PERS-NO'  PERSONNEL-ID
                  'POSITION' JOB-TITLE
END-READ
END
```

Ausgabe des Programms READX03:

```
   ISN              NAME         PERS-NO          POSITION
---------- -------------------- -------- ------------------------

        204 SCHINDLER            11100102 PROGRAMMIERER
        205 SCHIRM               11100105 SYSTEMPROGRAMMIERER
        206 SCHMITT              11100106 OPERATOR
        207 SCHMIDT              11100107 SEKRETAERIN
        208 SCHNEIDER            11100108 SACHBEARBEITER
        209 SCHNEIDER            11100109 SEKRETAERIN
        210 BUNGERT              11100110 SYSTEMPROGRAMMIERER
        211 THIELE               11100111 SEKRETAERIN
```

FIND-Statement

Auf die folgenden Beispiele wird im Abschnitt *Datenbankzugriffe* verwiesen.

FINDX07 - FIND-Statement (mit mehreren Klauseln)

```
** Example 'FINDX07': FIND  (with several clauses)
************************************************************************
DEFINE DATA LOCAL
1 EMPLOY-VIEW VIEW OF EMPLOYEES
  2 NAME
  2 FIRST-NAME
  2 CITY
  2 SALARY    (1)
  2 CURR-CODE (1)
END-DEFINE
*
FIND EMPLOY-VIEW WITH  PHONETIC-NAME = 'JONES' OR = 'BECKR'
                 AND   CITY          = 'BOSTON' THRU 'NEW YORK'
                 BUT NOT               'CHAPEL HILL'
                 SORTED BY NAME
                 WHERE SALARY (1)    < 28000
  DISPLAY NOTITLE NAME FIRST-NAME CITY SALARY (1)
END-FIND
END
```

Ausgabe des Programms `FINDX07`:

```
        NAME              FIRST-NAME              CITY           ANNUAL
                                                                 SALARY
-------------------- -------------------- -------------------- ----------

BAKER                PAULINE              DERBY                      4450
JONES                MARTHA               KALAMAZOO                 21000
JONES                KEVIN                DERBY                      7000
```

FINDX08 - FIND-Statement (mit LIMIT)

```
** Example 'FINDX08': FIND (with LIMIT)
**                     Demonstrates FIND statement with LIMIT option to
**                     terminate program with an error message if the
**                     number of records selected exceeds a specified
**                     limit (no output).
************************************************************************
DEFINE DATA LOCAL
1 EMPLOY-VIEW VIEW OF EMPLOYEES
  2 NAME
  2 JOB-TITLE
END-DEFINE
*
FIND EMPLOY-VIEW WITH LIMIT (5) JOB-TITLE = 'SALES PERSON'
  DISPLAY NAME JOB-TITLE
END-FIND
END
```

Von Programm `FINDX08` verursachter Laufzeitfehler:

```
NAT1005 More records found than specified in search limit.
```

FINDX09 - FIND-Statement (unter Verwendung von *NUMBER, *COUNTER, *ISN)

```
** Example 'FINDX09': FIND (using *NUMBER, *COUNTER, *ISN)
************************************************************************
DEFINE DATA LOCAL
1 EMPLOY-VIEW VIEW OF EMPLOYEES
  2 DEPT
  2 NAME
END-DEFINE
*
FIND EMPLOY-VIEW WITH CITY  = 'BOSTON'
                 WHERE DEPT = 'TECH00' THRU 'TECH10'
  DISPLAY NOTITLE
          'COUNTER' *COUNTER NAME DEPT 'ISN' *ISN
  AT START OF DATA
    WRITE '(TOTAL NUMBER IN BOSTON:' *NUMBER ')' /
  END-START
```

```
END-FIND
END
```

Ausgabe des Programms `FINDX09`:

```
  COUNTER           NAME         DEPARTMENT     ISN
                                    CODE
---------- -------------------- ---------- -----------

(TOTAL NUMBER IN BOSTON:           7 )

         1 STANWOOD             TECH10             782
         2 PERREAULT            TECH10             842
```

FINDX10 – FIND-Statement (in Kombination mit READ)

```
** Example 'FINDX10': FIND (combined with READ)
************************************************************************
DEFINE DATA LOCAL
1 EMPLOY-VIEW VIEW OF EMPLOYEES
  2 PERSONNEL-ID
  2 NAME
  2 FIRST-NAME
1 VEHIC-VIEW VIEW OF VEHICLES
  2 PERSONNEL-ID
  2 MAKE
END-DEFINE
*
LIMIT 15
*
EMP. READ EMPLOY-VIEW BY NAME STARTING FROM 'JONES'
  VEH. FIND VEHIC-VIEW WITH PERSONNEL-ID = PERSONNEL-ID (EMP.)
    IF NO RECORDS FOUND
      MOVE '*** NO CAR ***' TO MAKE
    END-NOREC
    DISPLAY NOTITLE
            NAME (EMP.) (IS=ON)
            FIRST-NAME (EMP.) (IS=ON)
            MAKE (VEH.)
  END-FIND
END-READ
END
```

Ausgabe des Programms `FINDX10`:

```
        NAME             FIRST-NAME             MAKE
-------------------- -------------------- --------------------

JONES                VIRGINIA             CHRYSLER
                     MARSHA               CHRYSLER
                                          CHRYSLER
                     ROBERT               GENERAL MOTORS
                     LILLY                FORD
                                          MG
                     EDWARD               GENERAL MOTORS
                     MARTHA               GENERAL MOTORS
                     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 ***
```

FINDX11 - FIND NUMBER-Statement (mit *NUMBER)

```
** Example 'FINDX11': FIND NUMBER (with *NUMBER)
************************************************************************
DEFINE DATA LOCAL
1 EMPLOY-VIEW VIEW OF EMPLOYEES
  2 FIRST-NAME
  2 NAME
  2 CITY
  2 JOB-TITLE
  2 SALARY        (1)
*
1 #PERCENT        (N.2)
1 REDEFINE #PERCENT
  2 #WHOLE-NBR    (N2)
1 #ALL-BOST       (N3.2)
1 #SECR-ONLY      (N3.2)
1 #BOST-NBR       (N3)
1 #SECR-NBR       (N3)
END-DEFINE
*
F1. FIND NUMBER EMPLOY-VIEW WITH CITY = 'BOSTON'
F2. FIND NUMBER EMPLOY-VIEW WITH CITY = 'BOSTON'
                           AND JOB-TITLE = 'SECRETARY'
*
MOVE *NUMBER(F1.) TO #ALL-BOST  #BOST-NBR
MOVE *NUMBER(F2.) TO #SECR-ONLY #SECR-NBR
DIVIDE #ALL-BOST INTO #SECR-ONLY GIVING #PERCENT
*
```

```
WRITE TITLE LEFT JUSTIFIED UNDERLINED
  'There are' #BOST-NBR 'employees in the Boston offices.' /
  #SECR-NBR '(=' #WHOLE-NBR (EM=99%')') 'are secretaries.'
*
SKIP 1
FIND EMPLOY-VIEW WITH CITY      = 'BOSTON'
                 AND  JOB-TITLE = 'SECRETARY'
  DISPLAY NAME FIRST-NAME JOB-TITLE SALARY (1)
END-FIND
END
```

Ausgabe des Programms `FINDX11`:

```
There are    7 employees in the Boston offices.
   3 (= 42%) are secretaries.
------------------------------------------------------------------------------

        NAME              FIRST-NAME              CURRENT             ANNUAL
                                                  POSITION            SALARY
-------------------- -------------------- ------------------------ ----------

SHAW                 LESLIE               SECRETARY                     18000
CREMER               WALT                 SECRETARY                     20000
COHEN                JOHN                 SECRETARY                     16000
```

Geschachtelte READ- und FIND-Statements

Auf die folgenden Beispiele wird im Abschnitt *Datenbank-Verarbeitungsschleifen* verwiesen.

READX04 – READ-Statement (in Kombination mit FIND und den Systemvariablen *NUMBER und *COUNTER)

```
** Example 'READX04': READ (in combination with FIND and the system
**                     variables *NUMBER and *COUNTER)
************************************************************************
DEFINE DATA LOCAL
1 EMPLOY-VIEW VIEW OF EMPLOYEES
  2 PERSONNEL-ID
  2 NAME
  2 FIRST-NAME
1 VEHIC-VIEW VIEW OF VEHICLES
  2 PERSONNEL-ID
  2 MAKE
END-DEFINE
*
LIMIT 10
```

```
RD. READ EMPLOY-VIEW BY NAME STARTING FROM 'JONES'
  FD. FIND VEHIC-VIEW WITH PERSONNEL-ID = PERSONNEL-ID (RD.)
    IF NO RECORDS FOUND
      ENTER
    END-NOREC
    /*
    DISPLAY NOTITLE
            *COUNTER (RD.)(NL=8) NAME         (AL=15) FIRST-NAME (AL=10)
            *NUMBER  (FD.)(NL=8) *COUNTER (FD.)(NL=8) MAKE
  END-FIND
END-READ
END
```

Ausgabe des Programms `READX04`:

```
  CNT          NAME       FIRST-NAME   NMBR      CNT             MAKE
--------- --------------- ---------- --------- --------- --------------------

        1 JONES           VIRGINIA           1         1 CHRYSLER
        2 JONES           MARSHA             2         1 CHRYSLER
        2 JONES           MARSHA             2         2 CHRYSLER
        3 JONES           ROBERT             1         1 GENERAL MOTORS
        4 JONES           LILLY              2         1 FORD
        4 JONES           LILLY              2         2 MG
        5 JONES           EDWARD             1         1 GENERAL MOTORS
        6 JONES           MARTHA             1         1 GENERAL MOTORS
        7 JONES           LAUREL             1         1 GENERAL MOTORS
        8 JONES           KEVIN              1         1 DATSUN
        9 JONES           GREGORY            1         1 FORD
       10 JOPER           MANFRED            0         0
```

LIMITX01 - LIMIT-Statement (für READ- und FIND-Schleifenverarbeitung)

```
** Example 'LIMITX01': LIMIT  (for READ, FIND loop processing)
************************************************************************
DEFINE DATA LOCAL
1 EMPLOY-VIEW VIEW OF EMPLOYEES
  2 PERSONNEL-ID
  2 FIRST-NAME
  2 NAME
1 VEH-VIEW VIEW OF VEHICLES
  2 PERSONNEL-ID
  2 MAKE
END-DEFINE
*
LIMIT 4
*
READ EMPLOY-VIEW BY NAME STARTING FROM 'A'
  FIND VEH-VIEW WITH PERSONNEL-ID = EMPLOY-VIEW.PERSONNEL-ID
```

```
    IF NO RECORDS FOUND
      MOVE 'NO CAR' TO MAKE
    END-NOREC
    DISPLAY PERSONNEL-ID NAME FIRST-NAME MAKE
  END-FIND
END-READ
END
```

Ausgabe des Programms `LIMITX01`:

```
Page      1                                              04-12-13  14:01:57

PERSONNEL-ID         NAME                FIRST-NAME                MAKE
------------ -------------------- -------------------- --------------------

             ABELLAN              KEPA                 NO CAR
30000231     ACHIESON             ROBERT               FORD
             ADAM                 SIMONE               NO CAR
20008800     ADKINSON             JEFF                 GENERAL MOTORS
```

ACCEPT- und REJECT-Statements

Auf die folgenden Beispiele wird im Abschnitt *Datensätze mit ACCEPT / REJECT auswählen* verwiesen.

ACCEPX04 - ACCEPT IF ... LESS THAN ...-Statement

```
** Example 'ACCEPX04': ACCEPT IF ... LES THAN ...
************************************************************************
DEFINE DATA LOCAL
1 EMPLOY-VIEW VIEW OF EMPLOYEES
  2 PERSONNEL-ID
  2 NAME
  2 JOB-TITLE
  2 SALARY (1)
END-DEFINE
*
LIMIT 20
READ EMPLOY-VIEW BY PERSONNEL-ID = '20017000'
  ACCEPT IF SALARY (1) LESS THAN 38000
  DISPLAY NOTITLE PERSONNEL-ID NAME JOB-TITLE SALARY (1)
END-READ
END
```

Ausgabe des Programms `ACCEPX04`:

```
PERSONNEL         NAME                 CURRENT            ANNUAL
   ID                                 POSITION           SALARY
--------- -------------------- ------------------------- ----------

20017000  CREMER               ANALYST                        34000
20017100  MARKUSH              TRAINEE                        22000
20017400  NEEDHAM              PROGRAMMER                     32500
20017500  JACKSON              PROGRAMMER                     33000
20017600  PIETSCH              SECRETARY                      22000
20017700  PAUL                 SECRETARY                      23000
20018000  FARRIS               PROGRAMMER                     30500
20018100  EVANS                PROGRAMMER                     31000
20018200  HERZOG               PROGRAMMER                     31500
20018300  LORIE                SALES PERSON                   28000
20018400  HALL                 SALES PERSON                   30000
20018500  JACKSON              MANAGER                        36000
20018800  SMITH                SECRETARY                      24000
20018900  LOWRY                SECRETARY                      25000
```

ACCEPX05 – ACCEPT IF ... AND ...-Statement

```
** Example 'ACCEPX05': ACCEPT IF ... AND ...
************************************************************************
DEFINE DATA LOCAL
1 EMPLOY-VIEW VIEW OF EMPLOYEES
  2 NAME
  2 CITY
  2 JOB-TITLE
  2 SALARY (1:2)
END-DEFINE
*
LIMIT 6
READ EMPLOY-VIEW PHYSICAL WHERE SALARY(2) > 0
  ACCEPT IF  SALARY(1) > 10000
         AND SALARY(1) < 50000
  DISPLAY (AL=15) 'SALARY I' SALARY (1) 'SALARY II'  SALARY (2)
                   NAME JOB-TITLE  CITY
END-READ
END
```

Ausgabe des Programms `ACCEPX05`:

```
Page      1                                              04-12-13  14:05:28

 SALARY I  SALARY II       NAME           CURRENT          CITY
                                         POSITION
---------- ---------- --------------- --------------- ---------------

     48000      46000 SPENGLER        SACHBEARBEITER  DARMSTADT
     45000      40000 SPECK           SACHBEARBEITER  DARMSTADT
     48000      46000 SCHINDLER       PROGRAMMIERER   HEPPENHEIM
     36000      32000 SCHMIDT         SEKRETAERIN     HEPPENHEIM
```

ACCEPX06 – REJECT IF ... OR ...-Statement

```
** Example 'ACCEPX06': REJECT IF ... OR ...
************************************************************************
DEFINE DATA LOCAL
1 EMPLOY-VIEW VIEW OF EMPLOYEES
  2 PERSONNEL-ID
  2 SALARY (1)
  2 JOB-TITLE
  2 CITY
  2 NAME
END-DEFINE
*
LIMIT 20
READ EMPLOY-VIEW LOGICAL BY PERSONNEL-ID = '20017000'
  REJECT IF SALARY (1) < 20000
         OR SALARY (1) > 26000
  DISPLAY NOTITLE SALARY (1) NAME JOB-TITLE CITY
END-READ
END
```

Ausgabe des Programms `ACCEPX06`:

```
  ANNUAL           NAME                 CURRENT                  CITY
  SALARY                               POSITION
---------- -------------------- ------------------------ --------------------

     22000 MARKUSH              TRAINEE                  LOS ANGELES
     22000 PIETSCH              SECRETARY                VISTA
     23000 PAUL                 SECRETARY                NORFOLK
     24000 SMITH                SECRETARY                SILVER SPRING
     25000 LOWRY                SECRETARY                LEXINGTON
```

AT START OF DATA- und AT END OF DATA-Statements

Auf die folgenden Beispiele wird im Abschnitt *AT START/END OF DATA-Statements* verwiesen.

ATENDX01 – AT END OF DATA-Statement

```
** Example 'ATENDX01': AT END OF DATA
************************************************************************
DEFINE DATA LOCAL
1 EMPLOY-VIEW VIEW OF EMPLOYEES
  2 PERSONNEL-ID
  2 NAME
  2 JOB-TITLE
END-DEFINE
*
READ (6) EMPLOY-VIEW BY PERSONNEL-ID FROM '20017000'
  DISPLAY NOTITLE NAME JOB-TITLE
  AT END OF DATA
    WRITE / 'LAST PERSON SELECTED:' OLD(NAME)
  END-ENDDATA
END-READ
END
```

Ausgabe des Programms `ATENDX01`:

```
        NAME                  CURRENT
                             POSITION
-------------------- -------------------------

CREMER               ANALYST
MARKUSH              TRAINEE
GEE                  MANAGER
KUNEY                DBA
NEEDHAM              PROGRAMMER
JACKSON              PROGRAMMER

LAST PERSON SELECTED: JACKSON
```

ATSTAX02 – AT START OF DATA-Statement

```
** Example 'ATSTAX02': AT START OF DATA
************************************************************************
DEFINE DATA LOCAL
1 EMPLOY-VIEW VIEW OF EMPLOYEES
  2 PERSONNEL-ID
  2 FIRST-NAME
  2 NAME
  2 SALARY    (1)
  2 CURR-CODE (1)
  2 BONUS     (1,1)
END-DEFINE
*
LIMIT 3
FIND EMPLOY-VIEW WITH CITY = 'MADRID'
  DISPLAY NAME FIRST-NAME SALARY(1) BONUS(1,1) CURR-CODE (1)
  /*
  AT START OF DATA
    WRITE NOTITLE *DAT4E /
  END-START
END-FIND
END
```

Ausgabe des Programms `ATSTAX02`:

```
        NAME              FIRST-NAME        ANNUAL     BONUS    CURRENCY
                                            SALARY               CODE
-------------------- -------------------- ---------- ---------- --------

13/12/2004

DE JUAN              JAVIER                1988000          0 PTA
DE LA MADRID         ANSELMO               3120000          0 PTA
PINERO               PAULA                 1756000          0 PTA
```

WRITEX09 – WRITE-Statement (in Kombination mit AT END OF DATA)

```
** Example 'WRITEX09': WRITE (in combination with AT END OF DATA )
************************************************************************
DEFINE DATA LOCAL
1 EMPLOY-VIEW VIEW OF EMPLOYEES
  2 CITY
  2 NAME
  2 BIRTH
  2 JOB-TITLE
  2 DEPT
END-DEFINE
```

```
*
READ (3) EMPLOY-VIEW BY CITY
  DISPLAY NOTITLE NAME BIRTH (EM=YYYY-MM-DD) JOB-TITLE
  WRITE 38T 'DEPT CODE:' DEPT
  /*
  AT END OF DATA
    WRITE / 'LAST PERSON SELECTED:' OLD(NAME)
  END-ENDDATA
  SKIP 1
END-READ
END
```

Ausgabe des Programms `WRITEX09`:

```
        NAME            DATE             CURRENT
                         OF             POSITION
                       BIRTH
-------------------- ---------- -------------------------

SENKO                1971-09-11 PROGRAMMER
                                      DEPT CODE: TECH10

GODEFROY             1949-01-09 COMPTABLE
                                      DEPT CODE: COMP02

CANALE               1942-01-01 CONSULTANT
                                      DEPT CODE: TECH03


LAST PERSON SELECTED: CANALE
```

DISPLAY- und WRITE-Statements

Auf die folgenden Beispiele wird im Abschnitt *Statements DISPLAY und WRITE* verwiesen.

DISPLX13 – DISPLAY-Statement (zum Vergleich mit WRITEX08 mit WRITE)

```
** Example 'DISPLX13': DISPLAY (compare with WRITEX08 using WRITE)
************************************************************************
DEFINE DATA LOCAL
1 EMPLOY-VIEW VIEW OF EMPLOYEES
  2 PERSONNEL-ID
  2 FIRST-NAME
  2 NAME
  2 SALARY (2)
  2 BONUS  (1,1)
```

```
  2 CITY
END-DEFINE
*
LIMIT 2
READ EMPLOY-VIEW WITH CITY = 'CHAPEL HILL' WHERE BONUS(1,1) NE 0
  /*
  DISPLAY 'PERS/ID' PERSONNEL-ID  NAME / FIRST-NAME
          '**' '=' SALARY(1:2) 'BONUS' BONUS(1,1) CITY (AL=15)
  /*
  SKIP 1
END-READ
END
```

Ausgabe des Programms `DISPLX13`:

```
Page      1                                                 04-12-13  14:11:28

  PERS           NAME             ANNUAL       BONUS         CITY
   ID          FIRST-NAME          SALARY
-------- -------------------- ------------- ---------- ---------------

20027000 CUMMINGS             **      41000       1500 CHAPEL HILL
         PUALA                        38900

20000200 WOOLSEY              **      26000       3000 CHAPEL HILL
         LOUISE                       24700
```

WRITEX08 – WRITE-Statement (zum Vergleich mit DISPLX13 mit DISPLAY)

```
** Example 'WRITEX08': WRITE  (compare with DISPLX13 using DISPLAY)
************************************************************************
DEFINE DATA LOCAL
1 EMPLOY-VIEW VIEW OF EMPLOYEES
  2 PERSONNEL-ID
  2 FIRST-NAME
  2 NAME
  2 SALARY (2)
  2 BONUS  (1,1)
  2 CITY
END-DEFINE
*
LIMIT 2
READ EMPLOY-VIEW WITH CITY = 'CHAPEL HILL' WHERE BONUS(1,1) NE 0
  /*
  WRITE 'PERS/ID' PERSONNEL-ID  NAME / FIRST-NAME
        '**' '=' SALARY(1:2) 'BONUS' BONUS(1,1) CITY (AL=15)
  /*
  SKIP 1
```

```
END-READ
END
```

Ausgabe des Programms `WRITEX08`:

```
Page      1                                                 04-12-13  14:12:43

PERS/ID 20027000 CUMMINGS
PUALA                ** ANNUAL SALARY:      41000      38900 BONUS       1500
CHAPEL HILL

PERS/ID 20000200 WOOLSEY
LOUISE               ** ANNUAL SALARY:      26000      24700 BONUS       3000
CHAPEL HILL
```

DISPLX14 – DISPLAY-Statement (mit AL, SF und *n*X)

```
** Example 'DISPLX14': DISPLAY  (with AL, SF and nX)
************************************************************************
DEFINE DATA LOCAL
1 EMPLOY-VIEW VIEW OF EMPLOYEES
  2 FIRST-NAME
  2 NAME
  2 ADDRESS-LINE (1)
  2 TELEPHONE
    3 AREA-CODE
    3 PHONE
  2 CITY
END-DEFINE
*
READ (3) EMPLOY-VIEW BY NAME STARTING FROM 'W'
  DISPLAY (AL=15 SF=5) NAME CITY / ADDRESS-LINE(1) 2X TELEPHONE
  SKIP 1
END-READ
END
```

Ausgabe des Programms `DISPLX14`:

```
Page      1                                                 04-12-13  14:14:00

     NAME               CITY                     TELEPHONE
                       ADDRESS
                                              AREA              TELEPHONE
                                              CODE
---------------     ---------------  ---------------     ---------------

WABER              HEIDELBERG       06221               456452
                   ERBACHERSTR. 78
```

```
WADSWORTH           DERBY            0332               515365
                    56 PINECROFT CO

WAGENBACH           FRANKFURT        069                983218
                    BECKERSTR. 4
```

WRITEX09 – WRITE-Statement (in Kombination mit AT END OF DATA)

```
** Example 'WRITEX09': WRITE (in combination with AT END OF DATA )
************************************************************************
DEFINE DATA LOCAL
1 EMPLOY-VIEW VIEW OF EMPLOYEES
  2 CITY
  2 NAME
  2 BIRTH
  2 JOB-TITLE
  2 DEPT
END-DEFINE
*
READ (3) EMPLOY-VIEW BY CITY
  DISPLAY NOTITLE NAME BIRTH (EM=YYYY-MM-DD) JOB-TITLE
  WRITE 38T 'DEPT CODE:' DEPT
  /*
  AT END OF DATA
    WRITE / 'LAST PERSON SELECTED:' OLD(NAME)
  END-ENDDATA
  SKIP 1
END-READ
END
```

Ausgabe des Programms `WRITEX09`:

```
        NAME            DATE             CURRENT
                         OF             POSITION
                       BIRTH
-------------------- ---------- -------------------------

SENKO                1971-09-11 PROGRAMMER
                                DEPT CODE: TECH10

GODEFROY             1949-01-09 COMPTABLE
                                DEPT CODE: COMP02

CANALE               1942-01-01 CONSULTANT
                                DEPT CODE: TECH03


LAST PERSON SELECTED: CANALE
```

DISPLAY-Statement

Auf folgendes Beispiel wird im Abschnitt *Seitenüberschriften, Seitenvorschübe und Leerzeilen* verwiesen.

DISPLX21 – DISPLAY-Statement (mit Schrägstrich '/' und zum Vergleich mit WRITE)

```
** Example 'DISPLX21': DISPLAY (usage of slash '/' in DISPLAY and WRITE)
************************************************************************
DEFINE DATA LOCAL
1 EMPLOY-VIEW VIEW OF EMPLOYEES
  2 CITY
  2 NAME
  2 FIRST-NAME
  2 ADDRESS-LINE (1)
END-DEFINE
*
WRITE TITLE LEFT JUSTIFIED UNDERLINED
       *TIME
   5X  'PEOPLE LIVING IN SALT LAKE CITY'
   21X 'PAGE:' *PAGE-NUMBER /
   15X 'AS OF' *DAT4E //
*
WRITE TRAILER UNDERLINED 'REGISTER OF' / 'SALT LAKE CITY'
*
READ (2) EMPLOY-VIEW WITH CITY = 'SALT LAKE CITY'
  DISPLAY  NAME /
           FIRST-NAME
           'HOME/CITY' CITY
           'STREET/OR BOX NO.' ADDRESS-LINE (1)
  SKIP 1
END-READ
END
```

Ausgabe des Programms `DISPLX21`:

```
14:15:50.1     PEOPLE LIVING IN SALT LAKE CITY                    PAGE:      1
               AS OF 13/12/2004


------------------------------------------------------------------------------
        NAME                 HOME                 STREET
     FIRST-NAME              CITY              OR BOX NO.
-------------------- -------------------- --------------------

ANDERSON             SALT LAKE CITY       3701 S. GEORGE MASON
JENNY
```

```
SAMUELSON            SALT LAKE CITY       7610 W. 86TH STREET
MARTIN

                                  REGISTER OF
                                SALT LAKE CITY
------------------------------------------------------------------------------
```

Spaltenüberschriften

Auf das folgende Beispiel wird im Abschnitt *Spaltenüberschriften* verwiesen.

DISPLX15 – DISPLAY-Statement (mit FC, UC)

```
** Example 'DISPLX15': DISPLAY  (with FC, UC)
************************************************************************
DEFINE DATA LOCAL
1 EMPLOY-VIEW VIEW OF EMPLOYEES
  2 FIRST-NAME
  2 NAME
  2 ADDRESS-LINE (1)
  2 CITY
  2 TELEPHONE
    3 AREA-CODE
    3 PHONE
END-DEFINE
*
FORMAT AL=12 GC== UC=%
*
READ (3) EMPLOY-VIEW BY NAME STARTING FROM 'R'
  DISPLAY NOTITLE (FC=*)
          NAME FIRST-NAME CITY  (FC=- UC=-) /
          ADDRESS-LINE(1) TELEPHONE
  SKIP 1
END-READ
END
```

Ausgabe des Programms `DISPLX15`:

```
****NAME**** *FIRST-NAME* ----CITY---- ========TELEPHONE========
                          **ADDRESS***
                                        ****AREA**** *TELEPHONE**
                                        ****CODE****
%%%%%%%%%%% %%%%%%%%%%% %%%%%%%%%%%% %%%%%%%%%%% %%%%%%%%%%%

RACKMANN    MARIAN       FRANKFURT    069          375849
```

```
                          FINKENSTR. 1

RAMAMOORTHY  TY           SEPULVEDA    209          175-1885
                          12018 BROOKS

RAMAMOORTHY  TIMMIE       SEATTLE      206          151-4673
                          921-178TH PL
```

DISPLX16 – DISPLAY-Statement (mit '/', '*text*', '*text*/*text*')

```
** Example 'DISPLX16': DISPLAY  (with '/', 'text', 'text/text')
************************************************************************
DEFINE DATA LOCAL
1 EMPLOY-VIEW VIEW OF EMPLOYEES
  2 FIRST-NAME
  2 NAME
  2 ADDRESS-LINE (1)
  2 CITY
  2 TELEPHONE
    3 AREA-CODE
    3 PHONE
END-DEFINE
*
READ (5) EMPLOY-VIEW BY NAME STARTING FROM 'E'
  DISPLAY NOTITLE
    '/'          NAME       (AL=12) /* suppressed header
    'FIRST/NAME' FIRST-NAME (AL=10) /* two-line user-defined header
    'ADDRESS'    CITY /             /* user-defined header
    ' '          ADDRESS-LINE(1)    /* 'blank' header
                 TELEPHONE (HC=L)   /* default header
  SKIP 1
END-READ
END
```

Ausgabe des Programms `DISPLX16`:

```
               FIRST          ADDRESS        TELEPHONE
                NAME
                                              AREA   TELEPHONE
                                              CODE
            ---------- -------------------- ------ ---------------

EAVES        TREVOR     DERBY                0332   657623
                        17 HARTON ROAD

ECKERT       KARL       OBERRAMSTADT         06154  99722
                        FORSTWEG 22

ECKHARDT     RICHARD    DARMSTADT
```

```
                      BRESLAUERPL. 4

EDMUNDSON    LES      TULSA                918    945-4916
                      2415 ALSOP CT.

EGGERT       HERMANN  STUTTGART            0711   981237
                      RABENGASSE 8
```

Feldausgabe-relevante Parameter

Auf die folgenden Beispiele wird im Abschnitt *Parameter zur Beeinflussung der Ausgabe von Feldern* verwiesen.

Sie stehen zur Verfügung, um die Benutzung der Parameter LC, IC, TC, AL, NL, IS, ZP und ES und des Statements SUSPEND IDENTICAL SUPPRESS zu demonstrieren:

DISPLX17 - DISPLAY-Statement (mit NL, AL, IC, LC, TC)

```
** Example 'DISPLX17': DISPLAY (with NL, AL, IC, LC, TC)
************************************************************************
DEFINE DATA LOCAL
1 EMPLOY-VIEW VIEW OF EMPLOYEES
  2 FIRST-NAME
  2 NAME
  2 SALARY (1)
  2 BONUS  (1,1)
END-DEFINE
*
READ (3) EMPLOY-VIEW BY NAME STARTING FROM 'JONES'
  DISPLAY NOTITLE (IS=ON NL=15)
                    NAME
    '-' '='         FIRST-NAME (AL=12)
    'ANNUAL SALARY' SALARY(1)  (LC=USD   TC=.00)   /
    '+ BONUSES'     BONUS(1,1) (IC='+  ' TC=.00)
  SKIP 1
END-READ
END
```

Ausgabe des Programms DISPLX17:

```
        NAME           FIRST-NAME       ANNUAL SALARY
                                          + BONUSES
-------------------- -------------- ----------------------

JONES                - VIRGINIA     USD           46000.00
                                                +  9000.00

                     - MARSHA       USD           50000.00
                                                   +  0.00

                     - ROBERT       USD           31000.00
                                                   +  0.00
```

DISPLX18 – DISPLAY-Statement (Benutzung von Voreinstellungen für SF, AL, UC, LC, IC, TC und zum Vergleich mit DISPLX19)

```
** Example 'DISPLX18': DISPLAY  (using default settings for SF, AL, UC,
**                     LC, IC, TC and compare with DISPLX19)
************************************************************************
DEFINE DATA LOCAL
1 EMPLOY-VIEW VIEW OF EMPLOYEES
  2 NAME
  2 FIRST-NAME
  2 CITY
  2 SALARY    (1)
  2 BONUS     (1,1)
END-DEFINE
*
FIND (6) EMPLOY-VIEW WITH CITY = 'CHAPEL HILL'
  DISPLAY NAME FIRST-NAME SALARY(1) BONUS(1,1)
END-FIND
END
```

Ausgabe des Programms `DISPLX18`:

```
Page      1                                                 04-12-13  14:20:48

        NAME             FIRST-NAME        ANNUAL     BONUS
                                           SALARY
-------------------- -------------------- ---------- ----------

KESSLER              CLARE                     41000          0
ADKINSON             DAVID                     24000          0
GEE                  TOMMIE                    39500          0
HERZOG               JOHN                      31500          0
QUILLION             TIMOTHY                   30500          0
CUMMINGS             PUALA                     41000       1500
```

DISPLX19 - DISPLAY-Statement (mit SF, AL, LC, IC, TC und zum Vergleich mit DISPLX18)

```
** Example 'DISPLX19': DISPLAY  (with SF, AL, LC, IC, TC and compare
**                      with DISPLX19)
************************************************************************
DEFINE DATA LOCAL
1 EMPLOY-VIEW VIEW OF EMPLOYEES
  2 NAME
  2 FIRST-NAME
  2 CITY
  2 SALARY (1)
  2 BONUS  (1,1)
END-DEFINE
*
FORMAT SF=3 AL=15 UC==
*
FIND (6) EMPLOY-VIEW WITH CITY = 'CHAPEL HILL'
  DISPLAY (NL=10)
    NAME
    FIRST-NAME  (LC='- ' UC=-)
    SALARY (1)  (LC=USD)
    BONUS (1,1) (IC='*** ' TC=' ***')
END-FIND
END
```

Ausgabe des Programms `DISPLX19`:

```
Page      1                                              04-12-13  14:21:57

     NAME            FIRST-NAME           ANNUAL              BONUS
                                          SALARY
===============   -----------------   ==============   ====================

KESSLER           - CLARE             USD      41000             *** 0 ***
ADKINSON          - DAVID             USD      24000             *** 0 ***
GEE               - TOMMIE            USD      39500             *** 0 ***
HERZOG            - JOHN              USD      31500             *** 0 ***
QUILLION          - TIMOTHY           USD      30500             *** 0 ***
CUMMINGS          - PUALA             USD      41000          *** 1500 ***
```

SUSPEX01 - SUSPEND IDENTICAL SUPPRESS-Statement (in Verbindung mit den Parametern IS, ES, ZP bei DISPLAY)

```
** Example 'SUSPEX01': SUSPEND IDENTICAL SUPPRESS (in conjunction with
**                      parameters IS, ES, ZP in DISPLAY)
************************************************************************
DEFINE DATA LOCAL
1 EMPLOY-VIEW VIEW OF EMPLOYEES
  2 PERSONNEL-ID
  2 FIRST-NAME
  2 NAME
  2 CITY
1 VEH-VIEW VIEW OF VEHICLES
  2 PERSONNEL-ID
  2 MAKE
END-DEFINE
*
LIMIT 15
RD. READ EMPLOY-VIEW BY NAME STARTING FROM 'JONES'
  SUSPEND IDENTICAL SUPPRESS
  FD. FIND VEH-VIEW WITH PERSONNEL-ID = PERSONNEL-ID (RD.)
    IF NO RECORDS FOUND
      MOVE '******' TO MAKE
    END-NOREC
    DISPLAY NOTITLE (ES=OFF IS=ON ZP=ON AL=15)
            NAME       (RD.)
            FIRST-NAME (RD.)
            MAKE       (FD.) (IS=OFF)
  END-FIND
END-READ
END
```

Ausgabe des Programms `SUSPEX01`:

```
     NAME         FIRST-NAME         MAKE
--------------- --------------- ---------------

JONES           VIRGINIA        CHRYSLER
JONES           MARSHA          CHRYSLER
                                CHRYSLER
JONES           ROBERT          GENERAL MOTORS
JONES           LILLY           FORD
                                MG
JONES           EDWARD          GENERAL MOTORS
JONES           MARTHA          GENERAL MOTORS
JONES           LAUREL          GENERAL MOTORS
JONES           KEVIN           DATSUN
JONES           GREGORY         FORD
JOPER           MANFRED         ******
```

```
JOUSSELIN       DANIEL          RENAULT
JUBE            GABRIEL         ******
JUNG            ERNST           ******
JUNKIN          JEREMY          ******
KAISER          REINER          ******
```

SUSPEX02 - SUSPEND IDENTICAL SUPPRESS-Statement (in Verbindung mit den Parametern IS, ES, ZP in DISPLAY). Identisch mit SUSPEX01, aber mit IS=OFF.

```
** Example 'SUSPEX02': SUSPEND IDENTICAL SUPPRESS (in conjunction with
**                     parameters IS, ES, ZP in DISPLAY)
**                     Identical to SUSPEX01, but with IS=OFF.
************************************************************************
DEFINE DATA LOCAL
1 EMPLOY-VIEW VIEW OF EMPLOYEES
  2 PERSONNEL-ID
  2 FIRST-NAME
  2 NAME
  2 CITY
1 VEH-VIEW VIEW OF VEHICLES
  2 PERSONNEL-ID
  2 MAKE
END-DEFINE
*
LIMIT 15
RD. READ EMPLOY-VIEW BY NAME STARTING FROM 'JONES'
  SUSPEND IDENTICAL SUPPRESS
  FD. FIND VEH-VIEW WITH PERSONNEL-ID = PERSONNEL-ID (RD.)
    IF NO RECORDS FOUND
      MOVE '******' TO MAKE
    END-NOREC
    DISPLAY NOTITLE (ES=OFF IS=OFF ZP=ON AL=15)
            NAME       (RD.)
            FIRST-NAME (RD.)
            MAKE       (FD.) (IS=OFF)
  END-FIND
END-READ
END
```

Ausgabe des Programms `SUSPEX02`:

```
     NAME         FIRST-NAME         MAKE
--------------- --------------- ---------------

JONES           VIRGINIA        CHRYSLER
JONES           MARSHA          CHRYSLER
JONES           MARSHA          CHRYSLER
JONES           ROBERT          GENERAL MOTORS
JONES           LILLY           FORD
```

```
JONES           LILLY           MG
JONES           EDWARD          GENERAL MOTORS
JONES           MARTHA          GENERAL MOTORS
JONES           LAUREL          GENERAL MOTORS
JONES           KEVIN           DATSUN
JONES           GREGORY         FORD
JOPER           MANFRED         ******
JOUSSELIN       DANIEL          RENAULT
JUBE            GABRIEL         ******
JUNG            ERNST           ******
JUNKIN          JEREMY          ******
KAISER          REINER          ******
```

COMPRX03 - COMPRESS-Statement (in Verbindung mit LC und TC)

```
** Example 'COMPRX03': COMPRESS (using parameters LC and TC)
************************************************************************
DEFINE DATA LOCAL
1 EMPLOY-VIEW VIEW OF EMPLOYEES
  2 CITY
  2 SALARY     (1)
  2 CURR-CODE  (1)
  2 LEAVE-DUE
  2 NAME
  2 FIRST-NAME
  2 JOB-TITLE
*
1 #SALARY      (N9)
1 #FULL-SALARY (A25)
1 #VACATION    (A11)
END-DEFINE
*
READ (3) EMPLOY-VIEW WITH CITY = 'BOSTON'
  MOVE SALARY(1) TO #SALARY
  COMPRESS 'SALARY  :' CURR-CODE(1) #SALARY INTO #FULL-SALARY
  COMPRESS 'VACATION:' LEAVE-DUE            INTO #VACATION
  /*
  DISPLAY NOTITLE NAME FIRST-NAME
           'JOB DESCRIPTION' JOB-TITLE  (LC='JOB     : ')  /
           '/'               #FULL-SALARY                   /
           '/'               #VACATION  (TC='DAYS')
  SKIP 1
END-READ
END
```

Ausgabe des Programms `COMPRX03`:

```
        NAME              FIRST-NAME               JOB DESCRIPTION
-------------------- -------------------- ----------------------------------

SHAW                 LESLIE               JOB     : SECRETARY
                                          SALARY  : USD 18000
                                          VACATION: 2DAYS

STANWOOD             VERNON               JOB     : PROGRAMMER
                                          SALARY  : USD 31000
                                          VACATION: 1DAYS

CREMER               WALT                 JOB     : SECRETARY
                                          SALARY  : USD 20000
                                          VACATION: 3DAYS
```

Editiermasken

Auf folgende Beispiele wird im Abschnitt *Editiermasken – der EM-Parameter* verwiesen.

EDITMX03 - Editiermaske (unterschiedliche EM-Angabe bei alphanumerischen Feldern)

```
** Example 'EDITMX03': Edit mask (different EM for alpha-numeric fields)
************************************************************************
DEFINE DATA LOCAL
1 EMPLOY-VIEW VIEW OF EMPLOYEES
  2 PERSONNEL-ID
  2 FIRST-NAME
  2 NAME
  2 CITY
  2 SALARY(1)
END-DEFINE
*
LIMIT 3
READ EMPLOY-VIEW BY PERSONNEL-ID FROM '20018000'
                 WHERE SALARY(1) = 28000 THRU 30000
  DISPLAY 'N A M E'  NAME       (EM=X^^X^^X^^X^^X^^X^^X^^X^^X^^X) /
          'NAME HEX' NAME       (EM=H^H^H^H^H^H^H^H^H^H^H)
                     FIRST-NAME (EM=' - 'X(15)*)
                     CITY       (EM=X..X(10))
  SKIP 1
END-READ
END
```

Ausgabe des Programms `EDITMX03`:

```
Page      1                                                  04-12-13  14:26:57

            N A M E                  FIRST-NAME         CITY
            NAME HEX
------------------------------ ------------------ -------------

L  O  R  I  E                    - JEAN-PAUL      * C..LEVELAND
D3 D6 D9 C9 C5 40 40 40 40 40 40

H  A  L  L                       - ARTHUR         * A..NN ARBER
C8 C1 D3 D3 40 40 40 40 40 40 40

V  A  S  W  A  N  I              - TOMMIE         * M..ONTERREY
E5 C1 E2 E6 C1 D5 C9 40 40 40 40
```

EDITMX04 - Editiermaske (unterschiedliche EM-Angaben bei numerischen Feldern)

```
** Example 'EDITMX04': Edit mask (different EM for numeric fields)
************************************************************************
DEFINE DATA LOCAL
1 EMPLOY-VIEW VIEW OF EMPLOYEES
  2 PERSONNEL-ID
  2 FIRST-NAME
  2 NAME
  2 SALARY (1)
  2 BONUS  (1,1)
  2 LEAVE-DUE
END-DEFINE
*
LIMIT 2
READ EMPLOY-VIEW BY PERSONNEL-ID = '20018000'
                 WHERE SALARY(1) = 28000 THRU 30000
  DISPLAY (SF=4)
          'N A M E'    NAME
          'SALARY'     SALARY(1)  (EM=*USD^ZZZ,999)
          'BONUS (ZZ)' BONUS(1,1) (EM=S*ZZZ,999) /
          'BONUS (Z9)' BONUS(1,1) (EM=SZ99,999+) /
          '->' '='     BONUS(1,1) (EM=-999,999)
          'VAC/DUE'    LEAVE-DUE  (EM=+999)
  SKIP 1
END-READ
END
```

Ausgabe des Programms `EDITMX04`:

```
Page      1                                                 04-12-13  14:27:43

      N A M E             SALARY       BONUS (ZZ)     VAC
                                       BONUS (Z9)     DUE
                                          BONUS
--------------------    -----------    -----------    ---

LORIE                   USD *28,000    +**4,000       +13
                                       + 04,000+
                                       ->  004,000

HALL                    USD *30,000    +**5,000       +14
                                       + 05,000+
                                       ->  005,000
```

EDITMX05 - Editiermaske (EM-Angaben für Datums- und Uhrzeit-Systemvariablen)

```
** Example 'EDITMX05': Edit mask (EM for date and time system variables)
************************************************************************
WRITE NOTITLE //
  'DATE INTERNAL :' *DATX (DF=L) /
  '              :' *DATX (EM=N(9)' 'ZW.'WEEK 'YYYY)  /
  '              :' *DATX (EM=ZZJ'.DAY 'YYYY)        /
  '     ROMAN    :' *DATX (EM=R)  /
  '     AMERICAN :' *DATX (EM=MM/DD/YYYY)      12X 'OR  ' *DAT4U /
  '     JULIAN   :' *DATX (EM=YYYYJJJ)         15X 'OR  ' *DAT4J /
  '     GREGORIAN:' *DATX (EM=ZD.''L(10)''YYYY) 5X 'OR  ' *DATG ///
  'TIME INTERNAL :' *TIMX                      14X 'OR  ' *TIME /
  '              :' *TIMX (EM=HH.II.SS.T) /
  '              :' *TIMX (EM=HH.II.SS' 'AP) /
  '              :' *TIMX (EM=HH)
END
```

Ausgabe des Programms `EDITMX05`:

```
DATE INTERNAL : 2004-12-13
              : Monday 51.WEEK 2004
              : 348.DAY 2004
     ROMAN    : MMIV
     AMERICAN : 12/13/2004           OR   12/13/2004
     JULIAN   : 2004348              OR   2004348
     GREGORIAN: 13.December2004      OR   13December 2004


TIME INTERNAL : 14:28:49             OR   14:28:49.1
              : 14.28.49.1
```

```
              : 02.28.49 PM
              : 14
```

DISPLAY VERT mit WRITE-Statement

WRITEX10 - WRITE-Statement (mit *n*T, T**field* und P**field*)

```
** Example 'WRITEX10': WRITE (with nT, T*field and P*field)
************************************************************************
DEFINE DATA LOCAL
1 EMPLOY-VIEW VIEW OF EMPLOYEES
  2 JOB-TITLE
  2 NAME
  2 SALARY (1)
  2 BONUS  (1,1)
END-DEFINE
*
READ (3) EMPLOY-VIEW WITH JOB-TITLE FROM 'SALES PERSON'
  DISPLAY NOTITLE NAME 30T JOB-TITLE
          VERT AS 'SALARY/BONUS' SALARY(1) BONUS(1,1)
  AT BREAK OF JOB-TITLE
    WRITE 20T 'AVERAGE' T*JOB-TITLE OLD(JOB-TITLE) (AL=15)
              '(SAL)'   P*SALARY AVER(SALARY(1))  /
          46T '(BON)'   P*BONUS  AVER(BONUS(1,1)) /
  END-BREAK
  SKIP 1
END-READ
END
```

Ausgabe des Programms `WRITEX10`:

```
        NAME                          CURRENT            SALARY
                                     POSITION            BONUS
--------------------          ------------------------- ----------

SAMUELSON                     SALES PERSON                  32000
                                                             6000

PAPAYANOPOULOS                SALES PERSON                  34000
                                                             7000

HELL                          SALES PERSON                  38000
                                                             9000

                   AVERAGE    SALES PERSON    (SAL)         34666
                                              (BON)          7333
```

AT BREAK-Statement

Auf das folgende Beispiel wird im Abschnitt *Gruppenwechsel* verwiesen.

ATBREX06 - AT BREAK OF-Statement (zum Vergleichen von NMIN, NAVER, NCOUNT mit MIN, AVER, COUNT)

```
** Example 'ATBREX06': AT BREAK OF (comparing NMIN, NAVER, NCOUNT with
**                     MIN, AVER, COUNT)
************************************************************************
DEFINE DATA LOCAL
1 EMPLOY-VIEW VIEW OF EMPLOYEES
  2 CITY
  2 SALARY (1:2)
END-DEFINE
*
WRITE TITLE '-- SALARY STATISTICS BY CITY --' /
*
READ (2) EMPLOY-VIEW WITH CITY = 'NEW YORK'
  DISPLAY CITY  'SALARY (1)' SALARY(1) 15X 'SALARY (2)' SALARY(2)
  AT BREAK OF CITY
    WRITE /
      14T 'S A L A R Y   (1)'         39T 'S A L A R Y  (2)'          /
      13T '-   MIN:' MIN(SALARY(1))   38T '-   MIN:' MIN(SALARY(2))   /
      13T '-  AVER:' AVER(SALARY(1))  38T '-  AVER:' AVER(SALARY(2))  /
      16T COUNT(SALARY(1)) 'RECORDS'  41T COUNT(SALARY(2)) 'RECORDS'  //
      13T '-  NMIN:' NMIN(SALARY(1))  38T '-  NMIN:' NMIN(SALARY(2))  /
      13T '- NAVER:' NAVER(SALARY(1)) 38T '- NAVER:' NAVER(SALARY(2)) /
      16T NCOUNT(SALARY(1)) 'RECORDS' 41T NCOUNT(SALARY(2)) 'RECORDS'
  END-BREAK
END-READ
END
```

Ausgabe des Programms `ATBREX06`:

```
                       -- SALARY STATISTICS BY CITY --

        CITY         SALARY (1)               SALARY (2)
-------------------- ----------               ----------

NEW YORK                  17000                    16100
NEW YORK                  38000                    34900

             S A L A R Y   (1)        S A L A R Y  (2)
            -   MIN:      17000      -   MIN:      16100
            -  AVER:      27500      -  AVER:      25500
                     2 RECORDS                2 RECORDS
```

```
           -  NMIN:      17000      -  NMIN:      16100
           - NAVER:      27500      - NAVER:      25500
                    2 RECORDS                 2 RECORDS
```

Statements COMPUTE, MOVE und COMPRESS

Auf die folgenden Beispiele wird im Abschnitt *Datenberechnungen* verwiesen.

WRITEX11 – WRITE-Statement (mit *n*X, *n*/*n* und COMPRESS)

```
** Example 'WRITEX11': WRITE (with nX, n/n and COMPRESS)
************************************************************************
DEFINE DATA LOCAL
1 EMPLOY-VIEW VIEW OF EMPLOYEES
  2 PERSONNEL-ID
  2 SALARY       (1)
  2 FIRST-NAME
  2 NAME
  2 CITY
  2 ZIP
  2 CURR-CODE    (1)
  2 JOB-TITLE
  2 LEAVE-DUE
  2 ADDRESS-LINE (1)
*
1 #SALARY        (A8)
1 #FULL-NAME     (A25)
1 #FULL-CITY     (A25)
1 #FULL-SALARY   (A25)
1 #VACATION      (A16)
END-DEFINE
*
READ (3) EMPLOY-VIEW LOGICAL BY PERSONNEL-ID = '2001800'
  MOVE SALARY(1) TO  #SALARY
  COMPRESS FIRST-NAME NAME                    INTO #FULL-NAME
  COMPRESS ZIP        CITY                    INTO #FULL-CITY
  COMPRESS 'SALARY  :'  CURR-CODE(1)  #SALARY INTO #FULL-SALARY
  COMPRESS 'VACATION:'  LEAVE-DUE    'DAYS'   INTO #VACATION
  /*
  DISPLAY NOTITLE 'NAME AND ADDRESS' NAME
               5X 'PERS-NO.'         PERSONNEL-ID
               3X 'JOB TITLE'        JOB-TITLE (LC='JOB     : ')
  WRITE   1/5 #FULL-NAME        1/37  #FULL-SALARY
          2/5 ADDRESS-LINE(1)   2/37  #VACATION
          3/5 #FULL-CITY
  SKIP 1
```

```
END-READ
END
```

Ausgabe des Programms `WRITEX11`:

```
  NAME AND ADDRESS       PERS-NO.               JOB TITLE
--------------------     --------   -----------------------------------

FARRIS                   20018000   JOB     : PROGRAMMER
    JACKIE FARRIS                   SALARY  : USD 30500
    918 ELM STREET                  VACATION: 10 DAY
    32306 TALLAHASSEE

EVANS                    20018100   JOB     : PROGRAMMER
    JO EVANS                        SALARY  : USD 31000
    1058 REDSTONE LANE              VACATION: 11 DAY
    68508 LINCOLN

HERZOG                   20018200   JOB     : PROGRAMMER
    JOHN HERZOG                     SALARY  : USD 31500
    255 ZANG STREET #253            VACATION: 12 DAY
    27514 CHAPEL HILL
```

IFX03 - IF-Statement

```
** Example 'IFX03': IF
************************************************************************
DEFINE DATA LOCAL
1 EMPLOY-VIEW VIEW OF EMPLOYEES
  2 NAME
  2 CITY
  2 BONUS  (1,1)
  2 SALARY (1)
*
1 #INCOME  (N9)
1 #TEXT    (A26)
END-DEFINE
*
WRITE TITLE '-- DISTRIBUTION OF CATALOGS I AND II --' /
*
READ (3) EMPLOY-VIEW BY CITY = 'SAN FRANSISCO'
  COMPUTE #INCOME = BONUS(1,1) + SALARY(1)
  /*
  IF #INCOME > 40000
    MOVE 'CATALOGS I AND II' TO #TEXT
  ELSE
    MOVE 'CATALOG I'         TO #TEXT
  END-IF
  /*
```

```
  DISPLAY NAME 5X 'SALARY' SALARY(1) / BONUS(1,1)
  WRITE T*SALARY '-'(10) /
        16X 'INCOME:' T*SALARY #INCOME 3X #TEXT /
        16X '='(19)
  SKIP 1
END-READ
END
```

Ausgabe des Programms `IFX03`:

```
                    -- DISTRIBUTION OF CATALOGS I AND II --

        NAME               SALARY
                           BONUS
--------------------     ----------

COLVILLE JR                   56000
                                  0
                           ----------
                INCOME:       56000   CATALOGS I AND II
                ===================

RICHMOND                       9150
                                  0
                           ----------
                INCOME:        9150   CATALOG I
                ===================

MONKTON                       13500
                                600
                           ----------
                INCOME:       14100   CATALOG I
                ===================
```

COMPRX03 - COMPRESS-Statement (mit Parametern LC and TC)

```
** Example 'COMPRX03': COMPRESS (using parameters LC and TC)
************************************************************************
DEFINE DATA LOCAL
1 EMPLOY-VIEW VIEW OF EMPLOYEES
  2 CITY
  2 SALARY     (1)
  2 CURR-CODE  (1)
  2 LEAVE-DUE
  2 NAME
  2 FIRST-NAME
  2 JOB-TITLE
*
1 #SALARY      (N9)
```

```
1 #FULL-SALARY (A25)
1 #VACATION    (A11)
END-DEFINE
*
READ (3) EMPLOY-VIEW WITH CITY = 'BOSTON'
  MOVE SALARY(1) TO #SALARY
  COMPRESS 'SALARY  :' CURR-CODE(1) #SALARY INTO #FULL-SALARY
  COMPRESS 'VACATION:' LEAVE-DUE            INTO #VACATION
  /*
  DISPLAY NOTITLE NAME FIRST-NAME
          'JOB DESCRIPTION' JOB-TITLE  (LC='JOB     : ')  /
          '/'               #FULL-SALARY                  /
          '/'               #VACATION  (TC='DAYS')
  SKIP 1
END-READ
END
```

Ausgabe des Programms `COMPRX03`:

```
        NAME             FIRST-NAME               JOB DESCRIPTION
-------------------- -------------------- ----------------------------------

SHAW                 LESLIE               JOB     : SECRETARY
                                          SALARY  : USD 18000
                                          VACATION: 2DAYS

STANWOOD             VERNON               JOB     : PROGRAMMER
                                          SALARY  : USD 31000
                                          VACATION: 1DAYS

CREMER               WALT                 JOB     : SECRETARY
                                          SALARY  : USD 20000
                                          VACATION: 3DAYS
```

Systemvariablen

Auf die folgenden Beispiele wird im Abschnitt *Systemvariablen und Systemfunktionen* verwiesen.

EDITMX05 – Editiermaske (EM bei Datums- und Uhrzeit-Systemvariablen)

```
** Example 'EDITMX05': Edit mask (EM for date and time system variables)
************************************************************************
WRITE NOTITLE //
  'DATE INTERNAL :' *DATX (DF=L) /
  '              :' *DATX (EM=N(9)' 'ZW.'WEEK 'YYYY)  /
  '              :' *DATX (EM=ZZJ'.DAY 'YYYY)        /
  '     ROMAN    :' *DATX (EM=R)  /
  '     AMERICAN :' *DATX (EM=MM/DD/YYYY)      12X 'OR  ' *DAT4U /
  '     JULIAN   :' *DATX (EM=YYYYJJJ)         15X 'OR  ' *DAT4J /
  '     GREGORIAN:' *DATX (EM=ZD.''L(10)''YYYY) 5X 'OR  ' *DATG ///
  'TIME INTERNAL :' *TIMX                      14X 'OR  ' *TIME /
  '              :' *TIMX (EM=HH.II.SS.T) /
  '              :' *TIMX (EM=HH.II.SS' 'AP) /
  '              :' *TIMX (EM=HH)
END
```

Ausgabe des Programms `EDITMX05`:

```
DATE INTERNAL : 2004-12-13
              : Monday 51.WEEK 2004
              : 348.DAY 2004
     ROMAN    : MMIV
     AMERICAN : 12/13/2004           OR   12/13/2004
     JULIAN   : 2004348              OR   2004348
     GREGORIAN: 13.December2004      OR   13December 2004


TIME INTERNAL : 14:36:58             OR   14:36:58.8
              : 14.36.58.8
              : 02.36.58 PM
              : 14
```

READX04 - READ-Statement (in Verbindung mit FIND und den Systemvariablen *NUMBER und *COUNTER)

```
** Example 'READX04': READ (in combination with FIND and the system
**                    variables *NUMBER and *COUNTER)
************************************************************************
DEFINE DATA LOCAL
1 EMPLOY-VIEW VIEW OF EMPLOYEES
  2 PERSONNEL-ID
  2 NAME
  2 FIRST-NAME
1 VEHIC-VIEW VIEW OF VEHICLES
```

```
  2 PERSONNEL-ID
  2 MAKE
END-DEFINE
*
LIMIT 10
RD. READ EMPLOY-VIEW BY NAME STARTING FROM 'JONES'
  FD. FIND VEHIC-VIEW WITH PERSONNEL-ID = PERSONNEL-ID (RD.)
    IF NO RECORDS FOUND
      ENTER
    END-NOREC
    /*
    DISPLAY NOTITLE
            *COUNTER (RD.)(NL=8) NAME         (AL=15) FIRST-NAME (AL=10)
            *NUMBER  (FD.)(NL=8) *COUNTER (FD.)(NL=8) MAKE
  END-FIND
END-READ
END
```

Ausgabe des Programms `READX04`:

```
  CNT          NAME       FIRST-NAME   NMBR      CNT             MAKE
--------- --------------- ---------- --------- --------- --------------------

        1 JONES           VIRGINIA           1         1 CHRYSLER
        2 JONES           MARSHA             2         1 CHRYSLER
        2 JONES           MARSHA             2         2 CHRYSLER
        3 JONES           ROBERT             1         1 GENERAL MOTORS
        4 JONES           LILLY              2         1 FORD
        4 JONES           LILLY              2         2 MG
        5 JONES           EDWARD             1         1 GENERAL MOTORS
        6 JONES           MARTHA             1         1 GENERAL MOTORS
        7 JONES           LAUREL             1         1 GENERAL MOTORS
        8 JONES           KEVIN              1         1 DATSUN
        9 JONES           GREGORY            1         1 FORD
       10 JOPER           MANFRED            0         0
```

WTITLX01 – WRITE TITLE-Statement (mit *PAGE–NUMBER)

```
** Example 'WTITLX01': WRITE TITLE (with *PAGE-NUMBER)
************************************************************************
DEFINE DATA LOCAL
1 VEHIC-VIEW VIEW OF VEHICLES
  2 MAKE
  2 YEAR
  2 MAINT-COST (1)
END-DEFINE
*
LIMIT 5
*
```

```
READ VEHIC-VIEW
END-ALL
SORT BY YEAR USING MAKE MAINT-COST (1)
  DISPLAY NOTITLE YEAR MAKE MAINT-COST (1)
  AT BREAK OF YEAR
    MOVE 1 TO *PAGE-NUMBER
    NEWPAGE
  END-BREAK
  /*
  WRITE TITLE LEFT JUSTIFIED
        'YEAR:' YEAR 15X 'PAGE' *PAGE-NUMBER
END-SORT
END
```

Ausgabe des Programms `WTITLX01`:

```
YEAR:  1980               PAGE      1
YEAR          MAKE         MAINT-COST
----- -------------------- ----------

 1980 RENAULT                  20000
 1980 RENAULT                  20000
 1980 PEUGEOT                  20000
```

Systemfunktionen

Auf die folgenden Beispiele wird im Abschnitt *Systemvariablen und Systemfunktionen* verwiesen.

ATBREX06 - AT BREAK OF-Statement (zum Vergleichen von NMIN, NAVER, NCOUNT mit MIN, AVER, COUNT)

```
** Example 'ATBREX06': AT BREAK OF (comparing NMIN, NAVER, NCOUNT with
**                      MIN, AVER, COUNT)
************************************************************************
DEFINE DATA LOCAL
1 EMPLOY-VIEW VIEW OF EMPLOYEES
  2 CITY
  2 SALARY (1:2)
END-DEFINE
*
WRITE TITLE '-- SALARY STATISTICS BY CITY --' /
*
READ (2) EMPLOY-VIEW WITH CITY = 'NEW YORK'
  DISPLAY CITY  'SALARY (1)' SALARY(1) 15X 'SALARY (2)' SALARY(2)
  AT BREAK OF CITY
    WRITE /
```

```
      14T 'S A L A R Y   (1)'         39T 'S A L A R Y  (2)'          /
      13T '-   MIN:' MIN(SALARY(1))   38T '-   MIN:' MIN(SALARY(2))   /
      13T '-  AVER:' AVER(SALARY(1))  38T '-  AVER:' AVER(SALARY(2))  /
      16T COUNT(SALARY(1)) 'RECORDS'  41T COUNT(SALARY(2)) 'RECORDS'  //
      13T '-  NMIN:' NMIN(SALARY(1))  38T '-  NMIN:' NMIN(SALARY(2))  /
      13T '- NAVER:' NAVER(SALARY(1)) 38T '- NAVER:' NAVER(SALARY(2)) /
      16T NCOUNT(SALARY(1)) 'RECORDS' 41T NCOUNT(SALARY(2)) 'RECORDS'
  END-BREAK
END-READ
END
```

Ausgabe des Programms ATBREX06:

```
                      -- SALARY STATISTICS BY CITY --

        CITY         SALARY (1)               SALARY (2)
-------------------- ----------               ----------

NEW YORK                  17000                    16100
NEW YORK                  38000                    34900

             S A L A R Y   (1)        S A L A R Y  (2)
            -   MIN:      17000      -   MIN:      16100
            -  AVER:      27500      -  AVER:      25500
                     2 RECORDS                2 RECORDS

            -  NMIN:      17000      -  NMIN:      16100
            - NAVER:      27500      - NAVER:      25500
                     2 RECORDS                2 RECORDS
```

ATENPX01 - AT END OF PAGE-Statement (mit der durch GIVE SYSTEM FUNCTIONS in DISPLAY verfügbaren Systemfunktion)

```
** Example 'ATENPX01': AT END OF PAGE (with system function available
**                     via GIVE SYSTEM FUNCTIONS in DISPLAY)
************************************************************************
DEFINE DATA LOCAL
1 EMPLOY-VIEW VIEW OF EMPLOYEES
  2 PERSONNEL-ID
  2 NAME
  2 JOB-TITLE
  2 SALARY (1)
END-DEFINE
*
READ (10) EMPLOY-VIEW BY PERSONNEL-ID = '20017000'
  DISPLAY NOTITLE GIVE SYSTEM FUNCTIONS
          NAME JOB-TITLE 'SALARY' SALARY(1)
  /*
  AT END OF PAGE
```

```
    WRITE / 24T 'AVERAGE SALARY: ...' AVER(SALARY(1))
  END-ENDPAGE
END-READ
END
```

Ausgabe des Programms `ATENPX01`:

```
        NAME                  CURRENT             SALARY
                             POSITION
-------------------- ------------------------- ----------

CREMER               ANALYST                        34000
MARKUSH              TRAINEE                        22000
GEE                  MANAGER                        39500
KUNEY                DBA                            40200
NEEDHAM              PROGRAMMER                     32500
JACKSON              PROGRAMMER                     33000
PIETSCH              SECRETARY                      22000
PAUL                 SECRETARY                      23000
HERZOG               MANAGER                        48500
DEKKER               DBA                            48000

                       AVERAGE SALARY: ...      34270
```

Stichwortverzeichnis

Symbole

A

B

C

D

E

M

N

O

P

R

S

T

U

V

W

X

Y

Z