COBOL

Altsysteme warten und erweitern

Das umfassende Praxis-Handbuch

DIE LESEPROBE

Kapitel 1

Was unterscheidet COBOL von modernen, objektorientierten Sprachen?

1.1 Die Geschichte von COBOL

An dieser Stelle soll nicht bis ins kleinste Detail beschrieben werden, an welchem Tag welches Feature in die Programmiersprache COBOL aufgenommen wurde, vielmehr soll ein grober Überblick über die Entstehung dieser Sprache und die verschiedenen Standards gegeben werden. In Tabelle 1.1 findet sich eine chronologische, wenn auch oberflächliche Aufstellung der Entstehungsgeschichte.

Jahr	Entwicklungsschritt
1960	Unter dem Namen COBOL-60 wurde eine erste Version von COBOL von einem Gremium mit den Namen CODASYL verabschiedet. Dieses Gremium bestand aus Vertretern von Regierung, Militär und Privatwirtschaft der USA und hat es sich zur Aufgabe gemacht, eine gemeinsame Programmiersprache zu entwickeln, die auf unterschiedlichen Computersystemen lauffähig sein sollte.
1961	Die Programmiersprache wurde komplett überarbeitet und war nicht abwärtskompatibel. Aus dieser Erfahrung heraus hat man festgelegt, künftig dafür zu sorgen, dass ältere Programme mit neuen Compiler-Versionen noch übersetzt werden können. In diesem Jahr wurden der COBOL-SORT und REPORT WRITER aufgenommen. Es handelt sich dabei um Module, um Dateien zu sortieren und Auswertungen zu erstellen.
1965	Definition von Tabellen und die Möglichkeit, Dateien zu bearbeiten, haben den Sprachumfang erweitert.
1968	In diesem Jahr wurde der ANSI-Standard X3.23-1968 für COBOL verabschiedet und seither ständig weiterentwickelt.
1974	Einzug der strukturierten Programmierung bestehend aus internen Unterprogrammen
1985	Unter der Bezeichnung COBOL-85 wurden beispielsweise die Begrenzer `END-IF` und `END-PERFORM` eingeführt. Auch die internen Funktionen wurden zu dieser Zeit definiert.
2002	Die Verarbeitung von Unicode und die objektorientierte Programmierung wurden aufgenommen.

Tabelle 1.1: Grobe Entstehungsgeschichte

Wichtig zu wissen ist, dass es sich bei COBOL um eine standardisierte Programmiersprache handelt und es unterschiedliche Hersteller von Compilern gibt, die natürlich auch immer eigene Erweiterungen eingebracht haben.

Interessant ist auch, dass es zuletzt eine standardisierte, objektorientierte Version von COBOL gegeben hat, die aber kaum zum Einsatz kam.

1.2 Fest definierter Sprachumfang

Moderne, objektorientierte Programmiersprachen kennen oft nur sehr wenige Befehle wie `if`, `while`, `for` usw. Ihre ganz große Stärke liegt darin, dass sie über mächtige Klassenbibliotheken mit einer Unzahl an Methoden verfügen, die durch eigene Entwicklungen permanent erweitert werden. Jedes noch so komplizierte Problem kann durch teilweise simple Methodenaufrufe gelöst werden.

Klassisches COBOL kennt solche Bibliotheken nicht. Hier werden alle Anforderungen durch fest vorgegebene Befehle gelöst. Will man beispielsweise in einer Zeichenkette den Buchstaben Ä durch AE ersetzen, geht das in Java recht einfach, wie in Listing 1.1 zu sehen.

```
String zeichenkette = "ÄÖÜ";
String neu = zeichenkette.replace("Ä", "AE");
```

Listing 1.1: Ersetzen von Zeichen in Java

Um dieselbe Aufgabe in COBOL zu lösen, ist weit mehr Text erforderlich, wie in Listing 1.2 abgedruckt. Da sich in diesem Beispiel der Ausgangstext in seiner Länge maximal verdoppeln kann, muss das bei der Definition des neuen Feldes berücksichtigt werden. Ein Java-Entwickler macht sich darüber eher wenig Gedanken. In COBOL haben alle Datenfelder eine feste Länge. Die Angabe `PIC X(3)` bestimmt, dass das Datenfeld drei alphanumerische Zeichen beinhalten kann.

```
WORKING-STORAGE SECTION.
01  ZEICHENKETTE          PIC X(3) VALUE "ÄÖÜ".
01  NEU                   PIC X(6) VALUE SPACE.
01  NEU-TABELLE REDEFINES NEU.
    05  NEU-ELEMENT       PIC X OCCURS 6.
01  I                     PIC 9.
01  K                     PIC 9.
PROCEDURE DIVISION.
    MOVE 1 TO K.
    MOVE SPACE TO NEU.
    PERFORM VARYING I FROM 1 BY 1 UNTIL I > 3
        IF ZEICHENKETTE(I:1) = "Ä"
            MOVE "A" TO NEU-ELEMENT(K)
            ADD 1 TO K
```

```
            MOVE "E" TO NEU-ELEMENT(K)
            ADD 1 TO K
        ELSE
            MOVE ZEICHENKETTE(I:1) TO NEU-ELEMENT(K)
            ADD 1 TO K
        END-IF
    END-PERFORM.
```

Listing 1.2: Ersetzen von Zeichen in COBOL

Das Problem bei dem hier gezeigten Beispiel ist, dass aus einem Zeichen zwei Zeichen werden können und sich die restlichen Zeichen daran anschließen müssen. Ist dagegen gefordert, aus jedem Ä ein einfaches A zu machen, lässt sich die Aufgabe auch in COBOL viel kürzer lösen (siehe Listing 1.3).

```
WORKING-STORAGE SECTION.
01  ZEICHENKETTE          PIC X(3) VALUE "ÄÖÜ".
01  NEU                   PIC X(3) VALUE SPACE.
PROCEDURE DIVISION.
    MOVE ZEICHENKETTE TO NEU.
    INSPECT NEU REPLACING ALL "Ä" BY "A".
```

Listing 1.3: Ersetzen einzelner Zeichen in COBOL

Für die unterschiedlichen Aufgaben stehen in COBOL verschiedene Befehle zur Verfügung. Diese muss man geschickt kombinieren, um das gewünschte Ziel zu erreichen.

1.3 Prozedurale Programmierung

COBOL ist eine prozedurale Programmiersprache, was meint, dass man alle Anweisungen in einer einzigen Prozedur hintereinander schreibt und diese linear abgearbeitet werden.

In einer funktionsorientierten Programmiersprache wie beispielsweise C schreibt man eine Reihe von Funktionen und kombiniert diese geschickt. In objektorientierten Sprachen programmiert man dagegen Klassen, die mithilfe von Methoden die Attribute der Klasseninstanzen manipulieren.

In COBOL definiert man seine Daten in der `DATA DIVISION` und dort meist in der `WORKING-STORAGE SECTION`. Alle Datenfelder stehen allen Anweisungen innerhalb der Prozedur zur Verfügung, die in der `PROCEDURE DIVISION` programmiert werden. Es gibt keine Kapselung der Datenfelder wie beispielsweise die innerhalb einer Funktion oder wie die Attribute einer Instanz. Es ist auch nicht möglich, lokale Variablen innerhalb einer Schleife zu programmieren, deren Sichtbarkeit dann auf die Anweisungen innerhalb dieser Schleife begrenzt wäre.

In COBOL können externe Unterprogramme geschrieben werden, um die Komplexität einer Aufgabe in mehrere kleinere Programme aufzuteilen und um die Wiederverwendbarkeit von Logik zu ermöglichen. Typische COBOL-Programme sind aber dennoch meist mehrere Tausend Zeilen lang und bestehen nicht aus einer Unzahl an Unterprogrammaufrufen.

Die Möglichkeiten, heute in COBOL ebenfalls objektorientiert zu programmieren oder eigene Funktionen zu implementieren, findet man so gut wie gar nicht.

1.4 Linearer Programmablauf

Alle Anweisungen eines COBOL-Programms stehen in der `PROCEDURE DIVISION`. Das Programm beginnt mit der ersten dort stehenden Anweisung, die eine nach der anderen abgearbeitet werden.

Dieser lineare Programmablauf wird nur durch Befehle wie `GO TO` oder `PERFORM` unterbrochen. Während man mit `GO TO` schlicht zu einer anderen Stelle innerhalb der Prozedur springt, um von dort an wieder linear weiterzulaufen, ruft man mit `PERFORM` ein internes Unterprogramm auf, an dessen Ende man wieder an die rufende Stelle zurückspringt und es dann mit der Anweisung weitergeht, die auf die `PERFORM`-Anweisung folgt. In den Genen der Programmiersprache COBOL findet man eine solche Aufteilung in interne Unterprogramme jedoch nicht. Wenn sich der Programmierer nicht um den ordentlichen Ablauf innerhalb seines Programms kümmert, passieren durchaus überraschende Dinge.

Um interne Unterprogramme zu programmieren, benötigt man Sprungmarken, bei denen es sich um Sections oder Paragraphen handeln kann. Die Details werden in späteren Kapiteln erklärt. In Listing 1.4 ist ein COBOL-Programm zu sehen, das über drei interne Unterprogramme verfügt, die mithilfe der Anweisung `PERFORM` auch aufgerufen werden.

```
WORKING-STORAGE SECTION.
01  FELD                PIC 99.
PROCEDURE DIVISION.
STEUER SECTION.
    MOVE 0 TO FELD.
    PERFORM UPRO-01.
    PERFORM UPRO-02.
UPRO-01 SECTION.
    ADD 1 TO FELD.
UPRO-02 SECTION.
    ADD 2 TO FELD.
ENDE SECTION.
    DISPLAY FELD.
```

Listing 1.4: COBOL-Programm mit internen Unterprogrammen

Was passiert in Listing 1.4 genau? Das Programm beginnt mit der Anweisung `MOVE 0 TO FELD`, was dieses mit `0` initialisiert. Danach wird das interne Unterprogramm `UPRO-01` aufgerufen. Da es sich hier um eine `SECTION` handelt, endet das interne Unterprogramm mit dem Beginn der nächsten `SECTION`. Durch den Aufruf `PERFORM UPRO-01` wird also lediglich die Anweisung `ADD 1 TO FELD` ausgeführt, Der Inhalt der Variablen `FELD` ist jetzt `1`. Die Steuerung geht an die nächste Anweisung nach dem `PERFORM` zurück. Dort steht `PERFORM UPRO-02`. Einzige Anweisung dieses internen Unterprogramms ist `ADD 2 TO FELD`. Diese wird ausgeführt und in der Variablen `FELD` steht jetzt der Wert `3`. Die Steuerung geht wieder an die nächste Anweisung nach dem `PERFORM` zurück und die nächste Anweisung ist tatsächlich `ADD 1 TO FELD`. Diese wird ausgeführt und in `FELD` steht jetzt `4`. Jetzt kommt auch noch `ADD 2 TO FELD` dran, womit wir schon bei dem Wert `6` sind. Am Ende wird dann noch `DISPLAY FELD` ausgeführt, was den Wert `6` auf dem Bildschirm ausgibt oder in die Standardausgabe schreibt. Danach folgt nichts mehr und die Steuerung geht an das Betriebssystem zurück.

Das meint COBOL mit einem linearen Programmablauf.

Um die erneuten Additionen, also den erneuten linearen Ablauf der internen Unterprogramme, zu verhindern, müsste nach dem zweiten `PERFORM` entweder ein `GO TO ENDE` stehen oder ein `PERFORM ENDE`, gefolgt von der Anweisung `STOP RUN`, die das Programm jetzt beendet und die Steuerung an das Betriebssystem zurückgibt.

1.5 Datenfelder mit fester Länge

Alle Datenfelder, mit denen man in einem COBOL-Programm arbeiten will, müssen in der `DATA DIVISION` definiert werden. Je nach Verwendung findet man diese Felder dort zum Beispiel in der `FILE SECTION` oder der `WORKING-STORAGE SECTION`.

Für alle Datenfelder gilt, dass ihr Datentyp und ihre Länge in Byte fest definiert sind. Datenfelder mit variabler Länge, die sich erst zur Laufzeit ergibt, gibt es in COBOL nicht.

Manchmal spricht man in COBOL von einem Feld oder einer Tabelle mit variabler Länge, meint damit aber nicht dasselbe wie ein Entwickler in einer objektorientierten Sprache. Will man beispielsweise in COBOL mit einem Datenbankfeld arbeiten, das vom Typ `VARCHAR` ist, muss man Folgendes definieren:

```
01  FELDNAME.
    05  FELDLAENGE         PIC S9(4) COMP.
    05  FELDINHALT         PIC X(200).
```

Zu dem eigentlichen Datenfeld gehört zunächst ein Längenfeld, gefolgt von einem weiteren Feld für den Feldinhalt. Dieses ist aber in dem Beispiel immer 200 Byte lang, egal, welcher Wert in `FELDLAENGE` steht. Der COBOL-Programmierer muss vielmehr selbst darauf achten, dass er maximal so viele Bytes verarbeitet, wie das Längenfeld angibt.

Nicht selten füllt er FELDINHALT vor einem Zugriff mit lauter Leerzeichen, um so die Werte aus einem vorangegangenen Zugriff zu überschreiben.

Auch eine Tabelle mit einer variablen Anzahl an Elementen lässt sich in COBOL zwar definieren, dennoch belegt eine solche Tabelle im Hauptspeicher immer den maximal benötigten Platz.

```
01  ANZAHL              PIC 99.
01  TABELLE.
    05  ELEMENT         OCCURS 1 TO 20 DEPENDING ON ANZAHL.
        10  DATENFELD   PIC X(20).
```

Listing 1.5: Tabelle mit variabler Elementanzahl

In Listing 1.5 ist eine solche Tabelle definiert. Sie soll mindestens ein, maximal zwanzig Elemente besitzen, je nachdem, was zur Laufzeit in dem Feld ANZAHL steht. Und tatsächlich kommt es zu einem schweren Fehler, wenn das Programm auf ein ungültiges Element zugreift. Im Hauptspeicher befinden sich aber immer 20 Elemente, die jeweils 20 Byte lang sind.

Dieser Umstand muss bei dem Design einer COBOL-Anwendung bedacht werden, auch wenn heute der zur Verfügung stehende Hauptspeicher viel größer ist als früher.

Es gibt aber nicht nur das Problem, dass die Datenfelder in Summe zu groß sein könnten, manchmal sind sie schlicht auch zu klein. Will man beispielsweise eine XML-Datei lesen, weiß man gar nicht, wie groß die einzelnen Felder für die Aufnahme der Daten definiert werden müssen. Moderne COBOL-Compiler bieten tatsächlich die Möglichkeit, solche Dateien zu lesen, über entsprechende Statusfelder bekommt man dabei die Information, ob es dabei dazu gekommen ist, dass Feldinhalte abgeschnitten werden mussten.

1.6 Module statt Instanzen

Eine komplexe Anwendung besteht aus einer Menge einzelner COBOL-Programme, die sich untereinander aufrufen können. Typischerweise sind sie oft mehrere Hundert oder gar Tausende Zeilen lang. Das hängt einerseits damit zusammen, dass die Programmiersprache COBOL sehr geschwätzig ist, man also viel Quellcode für relativ wenig Funktion schreiben muss, andererseits gibt es aber auch keine wirkliche Motivation, stark zu modularisieren.

Ein COBOL-Programm erledigt typischerweise eine Aufgabe, und diese komplett. Dabei greift es gleichzeitig auf alle Datenfelder zu, die es dafür benötigt.

Objektorientierte Sprachen kapseln zusammengehörige Daten in Klassen in Form von Attributen. Diese Klassen bieten eine Reihe von Methoden, um ihre Attribute zu manipulieren. Benötigt man zur Laufzeit mehrere Daten desselben Typs, erzeugt man die passende Anzahl von Instanzen dieser Klassen.

Eine komplexe objektorientierte Anwendung besteht daher aus einer umfangreichen Menge von Klassen, die sich gegenseitig benutzen und ihre Daten vor anderen schützen.

Benötigt man in COBOL mehrere Daten desselben Typs, definiert man sich eine Tabelle, die groß genug ist. Die eigenen Daten werden an externe Unterprogramme übergeben, die diese manipulieren können. Von einer Kapselung der Daten kann hier nicht die Rede sein.

Um ein COBOL-Programm zu verstehen und um es ändern zu können, muss man seinen Blickwinkel ändern. In COBOL stehen nicht die Daten im Vordergrund, sondern die Befehle der programmierten Prozedur. Ein COBOL-Programmierer fragt sich, welche Daten er in Gänze zur Bewältigung seiner Aufgabe benötigt, und definiert diese dann komplett in seiner `DATA DIVISION`. Er wird nur dann ein externes Unterprogramm aufrufen, wenn er die dort programmierte Prozedur auch in anderen COBOL-Programmen verwenden will. Für das eigene Programm ist das externe lediglich eine Art verlängerte Werkbank.

Um zu verstehen, was das bedeutet, stellen Sie sich ein Hauptprogramm vor, das in Folge seiner Verarbeitung dasselbe externe Unterprogramm zweimal aufruft. Beim ersten Aufruf befindet es sich noch in seinem initialen Zustand. Die Datenfelder sind leer oder mit Initialwerten gefüllt. Beim zweiten Aufruf stehen in den Feldern jedoch noch genau die Werte, die die Felder am Ende des ersten Aufrufs hatten, außer man gibt das externe Modul nach dem Aufruf explizit wieder frei. Über die `LINKAGE SECTION` teilen sich beide Module einen Teil der Daten, die sie beide manipulieren können. Es gibt auch keine Instanzen von externen Unterprogrammen.

Kapitel 2

Programmstruktur und grundlegende Sprachelemente

Die meisten Programmiersprachen erlauben die Definition von Variablen, Konstanten und Dateien, aber auch die Codierung von ausführbaren Anweisungen an beliebigen Stellen im Quellprogramm. In COBOL sieht es dagegen anders aus. In diesem Kapitel werden die Struktur und die Elemente eines COBOL-Programms beschrieben, sodass Sie sich in einem bestehenden COBOL-Programm leichter zurechtfinden.

2.1 COBOL-Programmstruktur

In der Programmiersprache COBOL hat man für Übersicht im Quellprogramm gesorgt, indem man das Quellprogramm in vier Programmteile, `DIVISION`s genannt, untergliedert hat. Jedem Programmteil wurde ein fester Name als Überschrift und ein Verwendungszweck gegeben:

```
IDENTIFICATION DIVISION.
ENVIRONMENT DIVISION.
DATA DIVISION.
PROCEDURE DIVISION.
```

Listing 2.1: Die vier Teile eines COBOL-Programms

Diese `DIVISION`s müssen in der hier angegebenen Reihenfolge im Programm erscheinen. Manche sind optional und können weggelassen werden.

`DIVISION`s unterteilen sich weiter in `SECTION`s, deren Namen durch die COBOL-Syntax vorgegeben sind, außer in der `PROCEDURE DIVISION`.

`SECTION`s (auch Kapitel genannt) können sich weiter in Paragraphen unterteilen. Auch hier gilt, dass deren Namen vorgeschrieben sind, außer in der `PROCEDURE DIVISION`.

Die eigentlichen COBOL-Definitionen und Anweisungen schreibt man schließlich in Sätzen, wobei diese Bezeichnung wörtlich zu nehmen ist. Sätze enden mit einem Punkt und der spielt in der Syntax von COBOL eine wichtige Rolle. Spätestens in dem Kapitel 9 über Verzweigungen und interne Unterprogramme wird das deutlich.

Sätze bestehen aus Klauseln und Wörtern, die teils durch die Syntax vorgegeben und teils frei durch den Programmierer wählbar sind.

2.1.1 Die Bedeutung der Programmteile (DIVISIONs)

Der Erkennungsteil `IDENTIFICATION DIVISION` enthält eine Reihe von Informationen zur Benennung und Dokumentation des Quellprogramms. Dieser Teil hat wenig Einfluss auf das Programm. Die hier gemachten Angaben werden – für spätere Bezugnahme – *Kommentareintragungen* genannt. Der Maschinenteil `ENVIRONMENT DIVISION` beschreibt die für das Programm notwendige Umgebung. Zusätzlich werden Beziehungen zwischen den logischen Dateien, die im Quellprogramm definiert sind, und den tatsächlichen Ein/Ausgabeeinheiten, auf denen sich diese Dateien befinden, hergestellt. Die an dieser Stelle gemachten Angaben werden *Klauseln* genannt.

Der Datenteil `DATA DIVISION` dient dazu, die Daten zu beschreiben, die im Programm verarbeitet werden sollen. Das umfasst Dateisatzbeschreibungen, Konstanten und Variablen. Diese Angaben werden *Definitionen* und *Klauseln* genannt.

Der Prozedurteil `PROCEDURE DIVISION` enthält eine Reihe von ausführbaren Anweisungen, die zusammen mit den definierten Daten das Objektprogramm bilden. Die in diesem Teil gemachten Angaben werden *Anweisungen* genannt.

2.1.2 Die Hierarchie in einem COBOL-Programm

```
IDENTIFICATION DIVISION
     |--> Paragraphen
            |--> Kommentare

ENVIRONMENT DIVISION
     |--> SECTIONs
            |--> Paragraphen
                   |--> Sätze
                          |--> Klauseln
                                 |--> Wörter

DATA DIVISION
     |--> SECTIONs
            |--> Definitionen
                   |--> Sätze
                          |--> Klauseln
                                 |--> Wörter

PROCEDURE DIVISION
     |--> SECTIONs
            |--> Paragraphen
                   |--> Sätze
                          |--> Anweisungen
                                 |--> Wörter
```

Listing 2.2: COBOL-Hierarchie

Alle Namen der SECTIONs und der Paragraphen in den ersten drei DIVISIONs sind von COBOL fest vorgegeben. In der PROCEDURE DIVISION können Sie beliebige Namen verwenden.

2.1.3 Das COBOL-Programm im Überblick

```
(1)    IDENTIFICATION DIVISION.            Kommentareintragungen ...

(2)    ENVIRONMENT DIVISION.
       CONFIGURATION SECTION.
       SOURCE-COMPUTER.
             Name des Umwandlungssystems
       OBJECT-COMPUTER.
             Name des ausführenden Systems
       SPECIAL-NAMES.
             Speziell vom Programmierer
             festzulegende Namen und Regeln
       REPOSITORY.
             Namen von objektorientierten Klassen,
             Funktionen, Interfaces, die in diesem
             Programm benutzt werden sollen. Diese sind
             nicht Gegenstand dieses Buches
       INPUT-OUTPUT SECTION.
       FILE-CONTROL.
             Klauseln zur Definition von Dateien
       I-O-CONTROL.
             Klauseln zu speziellen Ein/Ausgabetechniken

(3)    DATA DIVISION.
       FILE SECTION.
             Definition von Datensätzen
       WORKING-STORAGE SECTION.
             Definition von Konstanten und Variablen
       LOCAL-STORAGE SECTION.
             Definition von Variablen, die bei jedem
             Programmaufruf dynamisch angelegt werden
       LINKAGE SECTION.
             Felder für den Datenaustausch in einem
             Unterprogramm
       REPORT SECTION.
             Für Definitionen für den REPORT-WRITER
       SCREEN SECTION.
             Definition von Eingabemasken

(4)    PROCEDURE DIVISION.
             Anweisungen für die Verarbeitung
```

Listing 2.3: Aufbau eines COBOL-Programms

Eine detaillierte Beschreibung der einzelnen DIVISIONs und SECTIONs finden Sie in Kapitel 3.

2.2 COBOL-Sprachelemente

Sie haben im vorangehenden Abschnitt gesehen, dass ein COBOL-Programm letztendlich aus Wörtern besteht.

2.2.1 Reservierte Wörter

Unter einem *reservierten Wort* versteht man ein Wort, das für die Darstellung einer Klausel oder einer Anweisung reserviert worden ist. Diese Wörter umfassen:

Schlüsselwörter

Dies sind Wörter, die vorhanden sein müssen, um eine korrekte Anweisung zu programmieren. Es gibt drei Arten von Schlüsselwörtern:

- Verben wie `MOVE`, `PERFORM`, `COMPUTE`
- Notwendige Wörter, die in den Klauseln und Anweisungen vorkommen, z.B. `TO`, `FROM`
- Wörter, die eine besondere funktionelle Bedeutung haben, z.B. `NEGATIVE`, `NUMERIC`

Kontextsensitive Schlüsselwörter

Verschiedene Schlüsselwörter sind nur dann reserviert, wenn sie innerhalb einer Anweisung verwendet werden, für die sie als Schlüsselwort vorgesehen sind. Wird dasselbe Wort in einem anderen Zusammenhang genutzt, wird es als Programmiererwort betrachtet (Programmiererwörter). Beispiele für solche Wörter sind `ARITHMETIC`, `BACKGROUND-COLOR`, `BYTE-LENGTH`.

Wahlwörter

Die Wahlwörter können wahlweise, wo sie erlaubt sind, verwendet werden. Sie haben keinen Einfluss auf die Wirkung einer Klausel oder einer Anweisung und dienen ausschließlich der besseren Lesbarkeit des Programms, z.B. `IS`, `ARE`.

Verknüpfer

Ein Verknüpfer kann sein

- ein *Kennzeichnerbindewort*: `IN`, `OF` verknüpft einen Datennamen oder Paragraphennamen mit seinem Kennzeichner, z.B. `NAME IN KUNDENSATZ`
- oder ein *boolescher Operator*: `AND`, `OR`, `AND NOT`, `OR NOT` wird verwendet zur Herstellung von zusammengesetzten Bedingungen.

2.2.2 Programmiererwörter

Ein Programmiererwort ist ein COBOL-Wort, das vom Programmierer selbst gewählt werden kann. Es wird als symbolische Adresse zur Benennung von Datenbereichen, Dateien oder Programm-Verzweigungszielen verwendet.

Aufbau

1. Ein Wort besteht aus 1 bis 31 Zeichen des folgenden Vorrates: A bis Z, 0 bis 9, – (Bindestrich) und _ (Unterstrich).
2. Ein Wort darf nicht mit einem Bindestrich beginnen oder enden.
3. Es darf kein Leerzeichen enthalten.
4. Alle Programmiererwörter, ausgenommen Segmentnummern, Stufennummern und Paragraphennamen in der `PROCEDURE DIVISION`, müssen eindeutig sein. Paragraphennamen innerhalb einer einzigen `SECTION` dürfen sich nicht wiederholen.

   Es ist zwar möglich, mehrere Variablen mit demselben Namen zu definieren; um sie dann aber auch verwenden zu können, müssen sie sich eindeutig qualifizieren lassen, also zum Beispiel innerhalb verschiedener Datengruppen angelegt worden sein.
5. Alle Programmiererwörter, ausgenommen Paragraphennamen, SECTION-Namen, Stufennummern und Segmentnummern, müssen mindestens ein alphabetisches Zeichen enthalten. Stufennummern dienen der hierarchischen Deklaration von Variablen und Segmentnummern der Segmentierung der `PROCEDURE DIVISION`. Auf Letzteres wird im Rahmen dieses Buches nicht weiter eingegangen.

Beispiele

```
Datennamen   ---->    BETRAG    MWST   KUNDEN-SATZ
Kapitel u. Paragraphennamen ----> VERARBEITUNG LESEN
```

Listing 2.4: Beispiele für Programmiererwörter

2.2.3 Literale

Ein Literal ist eine Konstante, die in der Form einer Zeichenfolge angegeben wird. Diese kann mittels einer `MOVE`-Anweisung übertragen oder für die Vorbesetzung eines Datenfelds mittels der `VALUE`-Klausel verwendet werden.

Nicht numerische Literale

Ein nicht numerisches Literal ist eine Zeichenfolge von 1 bis 160 Zeichen, die in einfachen oder doppelten Anführungszeichen eingeschlossen ist.

Das Literal kann alle Zeichen aus dem aktuellen Zeichenvorrat enthalten. Sollen die Anführungszeichen selbst Bestandteil der Zeichenkette sein, müssen sie verdoppelt werden.

Beispiele

```
"Nachricht ""in Anführungszeichen"""
'16%'
"Umsatzliste"
```

Listing 2.5: Beispiele für nicht numerische Variable

Hexadezimale Literale

Damit Sie jeden Wert aus der aktuellen Codetabelle von 0 bis 255 in einem Literal im Programm verwenden können, bietet COBOL die Möglichkeit, ein hexadezimales Literal von `X"00"` bis `X"FF"` anzugeben. Das Literal wird als nicht numerisches betrachtet und muss für jedes Byte zwei hexadezimale Ziffern beinhalten.

Beispiele

```
X"00000F"          (3 Byte)
X"2020202020"      (5 Byte)
X'313233343536'    (6 Byte)
```

Listing 2.6: Beispiele für hexadezimale Literale

Verketten von nicht numerischen Literalen

Mit dem Ampersandzeichen `&` lassen sich sowohl nicht numerische als auch hexadezimale Literale in verschiedenen Befehlen verketten.

Beispiele

```
01 EINGABE-NAME   PIC X(40) VALUE "PETER" & " SCHULZ".
```

Auch in der `MOVE`- oder in jeder anderen Anweisung lässt sich die Verkettung anwenden:

```
MOVE "PETER" & " SCHULZ" TO EINGABE-NAME
DISPLAY "PETER" & " SCHULZ" AT 1001
```

Listing 2.7: Beispiele für das Verketten nicht numerischer Literale

Numerische Literale (Festkommazahlen)

Ein numerisches Literal ist eine Folge aus den Zeichen:

1. Ziffern von `0` bis `9`
2. Vorzeichen `+` oder `-`
3. Dezimalpunkt `.`

Aufbau

1. Das Literal darf maximal 1 bis 31 Ziffern enthalten.
2. Das Literal darf nur ein Vorzeichen enthalten und muss dann auch mit diesem beginnen. Wird das Vorzeichen weggelassen, wird + angenommen.
3. Das Literal darf nur einen Dezimalpunkt enthalten, der nie als letztes Zeichen angegeben werden darf.
4. Es muss mindestens eine Ziffer verwendet werden.

Beispiele

```
-123.45
+9876
.99
00000
```

Listing 2.8: Beispiele für numerische Literale

Numerische Literale (Fließkommazahlen)

Ein solches Literal besteht aus zwei Festpunktzahlen, die durch den Buchstaben `E` getrennt sind.

Aufbau

1. Das erste Literal darf maximal 1 bis 31 Ziffern enthalten und kann mit einem Vorzeichen und einem Dezimalpunkt ausgestattet sein.
2. Das zweite Literal ist der Exponent. Es kann vorzeichenbehaftet sein, darf aber maximal drei Ziffern umfassen. Die Angabe eines Dezimalpunkts ist nicht erlaubt.
3. Es muss jeweils mindestens eine Ziffer verwendet werden.

Beispiele

```
-123.45E5
+9876E123
.99E-3
0E0
```

Listing 2.9: Beispiele für Fließkommazahlen

Boolesche Literale

Ein boolesches Literal besteht aus den Zeichen `0` und `1` oder einer hexadezimalen Ziffer `0` bis `F` und kann eine Gesamtlänge von 160 Zeichen annehmen. Es beginnt mit einem `B` bzw. `BX` und ist in einfachen oder doppelten Anführungszeichen eingeschlossen.

Boolesche Literale können zusammen mit den booleschen Operatoren `B-XOR`, `B-AND`, `B-OR` und `B-NOT` verwendet werden.

Beispiele

```
B"1100"
B'11110000'
BX"8"
```

Listing 2.10: Beispiele für boolesche Literale

Nationale Literale

COBOL kennt neben den alphanumerischen Literalen (`USAGE DISPLAY`) auch nationale Literale (`USAGE NATIONAL`). Letztere kennzeichnen sich besonders dadurch, dass sie für die interne Darstellung jedes Zeichens mehr Speicherplatz benötigen als alphanumerische Zeichen. Bei Verwendung von UTF-16 ist jedes Zeichen zwei Byte groß.

Nationale Literale lassen sich auch in hexadezimaler Form darstellen.

Beispiele

```
N"Text mit nationalem Zeichensatz"
N'123'
NX"02A102A2"
```

Listing 2.11: Beispiele für nationale Literale

2.2.4 Figurative Konstanten

Eine figurative Konstante ist ein COBOL-Wort, für das vom Compiler ein bestimmter Wert erzeugt wird.

ZERO bzw. ZEROS bzw. ZEROES

Der Inhalt des Datenfelds, das diese figurative Konstante enthält, hängt von seinem Attribut ab.

Beispiel

Feldbeschreibung	Binär	entpackt	Gepackt
Inhalt (hex)	"000000"	"303030"	"00000F"

SPACE bzw. SPACES

Eine oder mehrere Wiederholungen des Zeichens »Leerzeichen«.

Beispiel

Löschen des Bereichs `KUNDEN-SATZ`:

```
MOVE SPACE TO KUNDEN-SATZ
```

Inhalt des Bereichs in hexadezimaler Schreibweise (lauter Leerzeichen):

```
X"20202020202020202020202020202020"
```

Die MOVE-Anweisung überträgt Daten zu einem Feld und wird im 6 detailliert ausgeführt.

HIGH-VALUE bzw. HIGH-VALUES

Damit ist das Zeichen mit der höchsten Ordnungsnummer im aktuell verwendeten Zeichensatz gemeint. Im ASCII-Zeichensatz entspricht das X"FF".

Beispiel

```
MOVE HIGH-VALUE TO KENNZEICHEN
```

Inhalt des Datenfelds:

```
X"FFFFFF"
```

LOW-VALUE bzw. LOW-VALUES

Eine oder mehrere Wiederholungen des Zeichens X"00". Damit ist das Zeichen mit der niedrigsten Ordnungsnummer im aktuell verwendeten Zeichensatz gemeint.

Beispiel

```
MOVE LOW-VALUE TO KENNZEICHEN.
```

Inhalt des Datenfelds:

```
X"000000"
```

QUOTE bzw. QUOTES

Eine oder mehrere Wiederholungen des Zeichens »Anführungszeichen«.

Beispiel

```
MOVE QUOTE TO DATENFELD
```

ALL Literal

Die figurative Konstante ALL Literal wurde für den Programmierer freigelassen. Er kann damit bestimmen, welches Zeichen hier eingesetzt werden soll.

Beispiel 1

Aufbauen einer Linie

```
MOVE ALL "-" TO LINIE
```

Inhalt des Datenfelds:

```
"-------------------------"
```

Beispiel 2

Aufbauen einer Tabulatorzelle

```
MOVE ALL "I----" TO TABZEILE
```

Inhalt des Datenfelds:

```
"I----I----I----I----I----I"
```

Sie können anhand dieses Beispiels sehen, dass dies so lange wiederholt wird, bis die Feldlänge nicht mehr ausreicht.

2.2.5 Trennsymbole

Interpunktion

Leerzeichen

Das Leerzeichen muss immer nach jedem COBOL-Element angegeben werden. Wo ein Leerzeichen vorhanden ist, können auch mehrere angegeben werden.

Beispiel

```
COMPUTE SUMME = ZAHL1 + ZAHL2
```

Komma und Semikolon

Diese Zeichen haben keine Bedeutung für die Interpretation des Quellprogramms. Sie verbessern lediglich die Lesbarkeit der Klauseln und Anweisungen.

Beispiel

```
ADD 1 TO ZAHL1, ZAHL2, ZAHL3.
01 STEUER PIC S9(0); COMP;    VALUE ZERO.
```

Listing 2.12: Interpunktionsbeispiel

Punkt

Der Punkt stellt das Endkriterium einer Aussage dar, z.B. das Ende

- einer Teil- oder Kapitelüberschrift
- einer Dateibeschreibung
- einer Feldbeschreibung
- einer Anweisung

Beispiel

```
WORKING-STORAGE SECTION.
01 SCHALTER      PIC 9 VALUE ZERO.

PROCEDURE DIVISION.
VERARBEITUNG SECTION.
    IF SCHALTER = 1
        DISPLAY "ENDE"
        STOP RUN.
```

Listing 2.13: Korrekte Verwendung des COBOL-Punkts

Insbesondere zeigt sich die Bedeutung des Punkts in der Beendigung von bedingten Anweisungen, wie in Listing 2.14.

```
    IF ZZ = 50
        MOVE ZERO TO ZZ
        ADD 1 TO SZ
        WRITE A-SATZ AFTER PAGE.

    WRITE A-SATZ FROM POSTENZEILE
```

Listing 2.14: Fehlerhafte Verwendung des COBOL-Punkts

Der Punkt beendet in diesem Beispiel die `IF`-Anweisung; eine nachfolgende Anweisung wird in jedem Fall ausgeführt.

Anweisungsbegrenzer

In ANSI85 wurden einige Anweisungen um einen Anweisungsbegrenzer erweitert. Dieser Begrenzer hat die Aufgabe, eine Anweisung zu beenden und syntaxmäßig von der nachfolgenden Anweisung zu trennen. Der Begrenzer ersetzt damit die Funktion des Punkts, wie in Listing 2.15.

```
    IF ZZ = 50
        MOVE ZERO TO ZZ
        ADD 1 TO SZ
        WRITE A-SATZ AFTER PAGE
```

```
    END-IF
    WRITE A-SATZ FROM POSTENZEILE
```

Listing 2.15: Verwendung von Anweisungsbegrenzern

Ohne Anweisungsbegrenzer war es oft problematisch, ein nicht so sauber geschriebenes COBOL-Programm richtig zu lesen. Wenn der Entwickler nicht auf korrekte Einrückungen in seinem Quellcode geachtet hat, konnte ein COBOL-Punkt schnell mal überlesen werden.

```
    IF ZZ = 50
        MOVE ZERO TO ZZ
        ADD 1 TO SZ.
        WRITE A-SATZ AFTER PAGE.   *> ACHTUNG !!
    WRITE A-SATZ FROM POSTENZEILE.
```

Listing 2.16: Gefährliche Verwendung des COBOL-Punkts

Verwendet man für jede entsprechende Anweisung ihren Anweisungsbegrenzer, wird man vom Compiler auf fehlerhaft eingesetzte COBOL-Punkte aufmerksam gemacht. Beispiel: Am Ende eines Befehls einer `IF`-Anweisung wird versehentlich ein Punkt gesetzt. Außerdem wird das `IF` durch ein `END-IF` beendet. Der COBOL-Compiler wird nun dieses `END-IF` als fehlerhaft markieren, weil das `IF` ja bereits durch den Punkt beendet wurde.

Wird also ein Anweisungsbegrenzer als fehlerhaft betrachtet, muss man den darüber liegenden Block auf versehentlich gesetzte COBOL-Punkte oder natürlich auch auf fehlerhaft benutzte Anweisungsbegrenzer hin untersuchen.

```
    IF ZZ = 50
        MOVE ZERO TO ZZ
        ADD 1 TO SZ.
        WRITE A-SATZ AFTER PAGE
    END-IF.                        *> Syntaxfehler !!
    WRITE A-SATZ FROM POSTENZEILE.
```

Listing 2.17: Syntaktisch fehlerhafte Verwendung des COBOL-Punkts

Tabelle 2.1 enthält alle Anweisungsbegrenzer.

Begrenzer	Anweisung
END-ACCEPT	ACCEPT
END-ADD	ADD
END-CALL	CALL
END-COMPUTE	COMPUTE

Tabelle 2.1: Die Anweisungsbegrenzer von COBOL

Begrenzer	Anweisung
END-DELETE	DELETE
END-DISPLAY	DISPLAY
END-DIVIDE	DIVIDE
END-EVALUATE	EVALUATE
END-IF	IF
END-MULTIPLY	MULTIPLY
END-PERFORM	PERFORM
END-READ	READ
END-RECEIVE	RECEIVE
END-RETURN	RETURN
END-REWRITE	REWRITE
END-SEARCH	SEARCH
END-START	START
END-STRING	STRING
END-SUBTRACT	SUBTRACT
ENO-UNSTRING	UNSTRING
END-WRITE	WRITE

Tabelle 2.1: Die Anweisungsbegrenzer von COBOL (Forts.)

Anführungszeichen

Diese dürfen nur paarweise zur Begrenzung von nicht numerischen Literalen auftreten, außer wenn das Literal fortgesetzt wird. Es können wahlweise die doppelten oder das einfache Anführungszeichen paarig verwendet werden.

Einem öffnenden Anführungszeichen muss ein Leerzeichen oder eine runde Klammer unmittelbar vorausgehen. Einem schließenden Anführungszeichen muss eines der folgenden Trennzeichen unmittelbar folgen:

Leerzeichen, Komma, Semikolon, Punkt oder schließende runde Klammer.

Beispiel

```
MOVE "FALSCHES KENNZEICHEN" TO FEHLER-MELDUNG.
```

Linke und rechte Rundklammern

Diese dürfen nur paarweise als Begrenzer von Normal- und Spezialindizes, arithmetischen Ausdrücken oder Bedingungen verwendet werden.

Runden Klammern können Leerzeichen vorausgehen und/oder folgen; sie müssen es aber nicht.

Den Rundklammern kommt beim Aufruf von Funktionen eine besondere Bedeutung zu, da sie hier dazu verwendet werden können, die Parameterliste der Funktion zu umschließen.

2.2.6 Operatoren

Arithmetische Operatoren

Folgende arithmetische Operatoren sind in COBOL bekannt und müssen immer durch mindestens ein Leerzeichen getrennt angegeben werden:

`+` Addition

`-` Subtraktion

`*` Multiplikation

`/` Division

`**` Potenzierung

Außerdem kann das Additions- und Subtraktionszeichen auch als Vorzeichen für eine Konstante oder Variable verwendet werden.

Boolesche Operatoren

Diese Operatoren dienen zur bitweisen Verknüpfung zweier boolescher Datenfelder beziehungsweise Konstanten.

`B-AND` Boolesche UND-Verknüpfung

`B-OR` Boolesche ODER-Verknüpfung

`B-XOR` Boolesche EXKLUSIV-ODER-Verknüpfung

`B-NOT` Boolesche Negation (nur ein Operand erlaubt)

Vergleichsoperatoren

Auch hier muss vor und nach jedem Operator ein Leerzeichen vorhanden sein. Einen eigenen Operator für »ungleich« gibt es nicht. Die Abfrage auf »gleich« muss mit dem Schlüsselwort `NOT` negiert werden.

`>` größer

`<` kleiner

`=` gleich

`>=` größer gleich

`<=` kleiner gleich

`NOT =` nicht gleich