Dr.sc.nat. Michael Wagner
michael@wagnertech.de

COBOL Grundlagenkurs

für Ein- und Umsteiger

Ralph Steyer

1. Ausgabe, März 2017

ISBN 978-3-86249-690-7

TE-COBOL_RS

TRAINER-EDITION published by HERDT

Dr.sc.nat. Michael Wagner

Impressum

Matchcode: TE-COBOL_RS

Autor: Ralph Steyer

Herausgeber: Dipl. Math. Ralph Steyer – www.rjs.de

1. Ausgabe, März 2017

Produziert im HERDT-Digitaldruck

HERDT-Verlag für Bildungsmedien GmbH
Am Kümmerling 21-25
55294 Bodenheim
Internet: www.herdt.com
E-Mail: info@herdt.com

Inhaltsverzeichnis

1 Zu den Unterlagen

Diese Unterlagen sind zum Lernen von COBOL gedacht. Entweder in Form des Selbststudiums oder als Begleitmaterial in COBOL-Kursen. Vermittelt werden die elementaren Grundlagen, um Programme mit COBOL erstellen als auch pflegen zu können. Dabei wird Wert auf die grundsätzliche Anwendung der verschiedenen Techniken und einfache Beispiele gelegt und nicht auf Vollständigkeit aller möglichen Anweisungen, Befehle oder Parameter. Insbesondere werden keine Dialekt- oder Distributions-spezifischen Besonderheiten beachtet.

Unter einer **Distribution** versteht man eine Zusammenstellung von Software, die als Komplettpaket weitergegeben wird. Verschiedene Distributionen der gleichen Software können unterschiedliche Bestandteile beinhalten und sich auch in anderen spezifischen Details unterscheiden.

1.1 Was benötigen Sie zum Arbeiten mit den Unterlagen?

COBOL hat seine Hauptanwendung auf Mainframes / Großrechnern. Jedoch kann man mittlerweile auch sehr gut mit einem normalen PC mit COBOL programmieren und COBOL-Programme ausführen. Als Basis für die Unterlagen wird deshalb explizit nur ein PC vorausgesetzt. Als Betriebssysteme werden Windows 10 und Linux die Referenzen sein, wobei auch ein Mac eingeschlossen wird. Die Ausführungen zu den eigentlichen Programmcodes sind jedoch unabhängig von der Plattform und lassen sich – bis auf PC-spezifische Details und Ausführungen zu einer konkreten Distribution – auf das Umfeld von Großrechnern übertragen.

1.2 Was sollten Sie bereits können?

Der Kurs ist als Einsteigerkurs konzipiert, der die Grundlagen der Sprache COBOL von Grund auf erarbeitet. Allerdings wird COBOL sehr selten als erste Programmiersprache gelernt. Umsteiger aus anderen Sprachen sind also explizit als Zielgruppe einkalkuliert. Deshalb werden zumindest elementare, einfache Erfahrungen in der Programmierung vorausgesetzt. Ebenso sollten Sie mit einem Texteditor umgehen können. Kenntnisse der wichtigsten Befehle Ihres Betriebssystems (Windows oder Linux / MacOS) zum Umgang mit Dateien und Verzeichnissen in der Konsole sind hilfreich.

1.3 Schreibkonventionen

Gehen wir noch kurz auf einige Schreibkonventionen ein, die in den Unterlagen eingehalten werden, und die Ihnen helfen sollen, die Übersicht zu bewahren. Wichtige Begriffe werden hervorgehoben. Vor allem sollten Sie erkennen können, ob es sich um normalen Text oder Programmcode handelt. Diese Kennzeichnungen finden Sie im Fließtext, aber auch absatzweise. Ebenso werden Bereiche verwendet, die über die Markierung mit Symbolen besondere Aufmerksamkeit erzeugen sollen.

Das ist ein besonderer oder wichtiger Hinweis, den Sie an der Stelle beachten sollten. Oder eine Information, die Sie sich gut merken sollten.

Das ist ein Tipp, der Ratschläge oder besondere Tricks zu einer jeweiligen Situation zeigt.

Das ist eine Aufgabe, die Sie an der Stelle lösen sollten. Bei einigen Aufgaben (etwa dem Erstellen eines Programms) wird explizit auf eine Lösung im Anhang verwiesen, wenn es notwendig ist und die Ausführungen an der Stelle die Lösung der Aufgabe nicht weiter erklären oder beschreiben.

Das ist eine Lösung zu einer Aufgabe. Dieses Symbol wird nur im Anhang verwendet.

2 COBOL – Grundlagen

In diesem einleitenden Kapitel werden erste Grundlagen zu COBOL vermittelt und unter anderem alles Wichtige erklärt, was Sie grundsätzlich zum Erstellen und Übersetzen eines COBOL-Programms benötigen. COBOL-Syntax steht noch nicht im Fokus.

2.1 Was ist COBOL?

Zuerst einmal ein paar Fakten zur Sprache:

- COBOL steht für **Co**mmon **B**usiness **O**riented **L**anguage.
- Zentrales Ziel der Sprache war (und ist) die Lösung von kaufmännischen, betriebswirtschaftlichen Problemen.
- Eingeführt wurde COBOL Ende der 1950er-Jahre.
- Die Sprache wurde als höhere Programmiersprache konzipiert, was sie klar gegenüber damals gebräuchlichem Maschinencode als auch Assembler abgrenzt.
- Es handelt sich um eine hardwareunabhängige, standardisierte und problemorientierte Sprache. Dabei ist es eine **kompilierte** Programmiersprache, die an natürliche Sprache angelehnt ist. COBOL gilt als eine robuste Sprache, was im kaufmännischen, betriebswirtschaftlichen Umfeld von elementarer Wichtigkeit ist.
- Obwohl COBOL seit vielen Jahren, wenn nicht Jahrzehnten, als veraltet diskreditiert wird, ist die Sprache immer noch im Einsatz bei großen Firmen. Insbesondere im Versicherungs- und Bankenumfeld. Es werden zwar so gut wie keine neuen COBOL-Programme erstellt, aber vermutlich werden COBOL-Programme noch auf Jahrzehnte hinaus im Einsatz bleiben und diese müssen gepflegt werden.
- Die ursprüngliche Zielplattform von COBOL waren große Mainframes, was sich aufgrund des Erscheinungsdatums nahezu zwangsläufig ergibt. Dort liegt auch heute noch der Schwerpunkt von COBOL-Programmen. Aber es gibt mittlerweile Distributionen für die unterschiedlichsten Architekturen und Betriebssysteme.
- Der Zweck bzw. die Stärke von COBOL besteht in der Ausführung von Batch-Programmen (sequentiell abgearbeiteten Befehlsschritten) und Transaktionen. Ebenso geht die Verarbeitung großer Datenmengen mit COBOL hervorragend. Hierbei ist COBOL auch heute noch kaum gegenüber einer modernen Sprache im Hintertreffen. Ganz im Gegenteil. Die Erstellung von komfortablen Benutzerschnittstellen zählt hingegen weniger zu den Stärken von COBOL.
- In der Originalversion war COBOL eine prozedurale Sprache, aber mittlerweile gibt es eine objektorientierte Abwandlung bzw. Erweiterung von COBOL (OO Cobol), was aber in den Unterlagen nicht behandelt wird.
- Die Namen der verschiedenen Versionen von COBOL orientieren sich am Erscheinungsjahr.

2.2 Die Evolution von COBOL

- 1959 wurde COBOL entwickelt durch CODASYL.
- Die nächste Version ist COBOL-61 (1961).
- 1968 wurde COBOL von ANSI als Standardsprache für die kommerzielle Nutzung zugelassen (COBOL-68).
- Zwei wesentliche Überarbeitungen erfolgten im Jahr 1974 (COBOL-74) und 1985 (COBOL-85).

- 2002: Objektorientiertes COBOL.
- Im Jahr 2014 wurde die derzeit neueste Aktualisierung veröffentlicht. Unter http://www.iso.org/iso/iso_catalogue/catalogue_tc/catalogue_detail.htm?csnumber=51416 sind Details zu COBOL 2014 zu finden. Die Neuerungen sind im Einsteigerbereich in der Regel nicht wichtig und werden in den Unterlagen keine Rolle spielen.

2.3 Wichtige Quellen zu COBOL

Grundsätzlich findet man im Internet zu COBOL ungewöhnlich wenige Quellen – insbesondere im Vergleich zu anderen Programmiersprachen. Aber es gibt dennoch Webseiten und Projekte zu dem Thema. Sowohl was Software als auch Informationen und Dokumentationen angeht. Beachten Sie allerdings, dass hier keine Gewähr für die Aktualität der folgenden Angaben gegeben werden kann. Die Rechte und Verantwortung obliegen den Betreibern der Webseiten und Projekte. In der folgenden Tabelle finden Sie eine Auswahl an Quellen rund um COBOL.

URL	Beschreibung
http://opencobol.add1tocobol.com/OpenCOBOL%20Programmers%20Guide.pdf	OpenCOBOL 1.1 Programmer's Guide
http://open-cobol.sourceforge.io/	GnuCOBOL Guides
http://www.ibm.com/software/products/de/cobolaix	IBM - COBOL for AIX
http://www.mingw.org/	MinGW
https://de.wikipedia.org/wiki/COBOL	Erklärungen zu COBOL bei Wikipedia
https://github.com/OpenCobolIDE/OpenCobolIDE	OpenCOBOLIDE
https://notepad-plus-plus.org/	Notepad++
https://public.support.unisys.com/aseries/docs/clearpath-mcp-14.0/pdf/86001518-312.pdf	COBOL ANSI-85 Programming Reference Manual - unisys
https://sourceforge.net/projects/open-cobol/	GnuCOBOL
https://supportline.microfocus.com/documentation/books/sx50/lrpubb.htm	COBOL Language Reference – Micro Focus
https://www.cygwin.com/	Cygwin
https://www.ibm.com/support/knowledgecenter/SSLTBW_2.1.0/com.ibm.zos.v2r1.halc001/cob.htm	Original COBOL application programming interface (EZACICAL)
https://www.video2brain.com/de/videotraining/cobol-grundlagen	COBOL – Grundlagen-Programmierung – Onlinetraining bei Video2brain
https://www2.southeastern.edu/Academics/Faculty/kyang/2007/Fall/CMPS401/Doc/aa-q2g0h-tk.pdf	HP COBOL - Reference Manual

Tabelle 2-1: Quellen zu COBOL

3 COBOL-Distributionen

Zum Erstellen von COBOL-Programmen braucht man minimal eine COBOL-Distribution und mindestens einen Editor.

Im Umfeld von Mainframes gibt es besondere COBOL-Installationen, die hier nicht behandelt werden (beispielsweise Enterprise COBOL for z/OS von IBM).

In den Unterlagen wird eine ***freie*** COBOL-Distribution vorausgesetzt, die unter anderem für Windows, Linux oder MacOS zur Verfügung steht.

Dabei sollte man vorabschicken, dass COBOL für PCs im Wesentlichen ein Unix-artiges Betriebssystem voraussetzt. Das ist erklärbar, denn Unix ist auch auf Mainframes / Großrechner verbreitet und da liegt ja das ursprüngliche Habitat von COBOL. Unter Windows gibt es damit gewisse Probleme, um COBOL dort einzusetzen.

Ein Weg führt über **Cygwin**, was für COBOL-Distributionen unter Windows die notwendigen Voraussetzungen schafft. Genau genommen lassen sich mit Cygwin Computerprogramme, die üblicherweise unter POSIX-Systemen wie GNU/Linux, BSD und Unix laufen, auf Windows portieren. Mittels Cygwin portierte Programme laufen derzeit unter allen Windows-Versionen ab Windows Vista.

Cygwin ist jedoch noch keine COBOL-Distribution selbst, sondern schafft eben nur unter Windows die Voraussetzungen, dass man eine solche unter Windows verwenden kann. Eine COBOL-Distribution, die hier in den Unterlagen vorausgesetzt werden soll, ist **GnuCOBOL**.

Abb. 3.1: Die Webseite von GnuCOBOL-Projekts

Diese Distribution wird typischer Weise im Quellcode geladen und dann auf dem lokalen Rechner individuell konfiguriert, kompiliert und eingerichtet. Es gibt aber auch vorgefertigte Versionen mit Standardeinstellungen, die zum Lernen von COBOL auf jeden Fall genügen und viel einfacher einzurichten sind. Insbesondere unter Windows.

3.1 GnuCOBOL

GnuCOBOL (https://sourceforge.net/projects/open-cobol/ - früher bekannt als OpenCOBOL - Abb. 3.1) ist ein modernes System auf Open Source-Basis mit einem COBOL-Compiler und diversen weiteren Tools. Dieser COBOL-Compiler übersetzt COBOL-Quellcode zuerst in C-Quellcode und kompiliert diesen gene-

rierten Code mittels dem C/C++-Compiler GCC (GNU Compiler Collection) dann in endgültig lauffähigen Code. Allerdings ist die vollständige Kompatibilität mit allen COBOL-Standards nicht garantiert, obgleich zumindest so gut wie alle Regeln von COBOL-85 eingehalten werden. Für diese Unterlagen soll GnuCOBOL das Referenzsystem sein, denn neben der Tatsache, dass es sich um eine freie Implementierung handelt, ist die Einrichtung sehr einfach.

3.1.1 GnuCOBOL unter Windows

Zwar gab es in früheren Versionen von GnuCOBOL unter Windows ein paar Probleme mit der Installation, aber das hat sich mittlerweile weitgehend erledigt. Zumindest wenn Sie nicht den „normalen" Weg mit der individuellen Kompilierung des COBOL-Compilers aus den Quellcodes auf Ihrem System gehen, sondern mit einer vorgefertigten Version wie **MinGW** vorlieb nehmen. Diese genügt zum Lernen von COBOL allemal.

Sie zu finden ist aber nicht ganz einfach, denn sie wird auf den Projektseiten von GnuCOBOL etwas versteckt angeboten. Unter der URL https://sourceforge.net/p/open-cobol/faq/?source=navbar, die Sie z.B. auf der Einstiegsseite von GnuCOBOL unter dem Link *FAQ and How-To* finden, gelangen Sie auf eine Folgeseite mit Hinweisen zu verschiedenen vorkompilierten Versionen von COBOL für mehrere Betriebssysteme – auch Windows (Abb. 3.2).

Derzeit kommen Sie mit dem Link http://sourceforge.net/projects/open-cobol/files/gnu-cobol/1.1/ zum Download eines passenden Archivs (https://sourceforge.net/projects/open-cobol/files/gnu-cobol/1.1/GnuCOBOL_1.1_MinGW_BDB_PDcurses_MPIR.7z/download), das eine komplett lauffähige, stabile COBOL-Version enthält (Abb. 3.3).

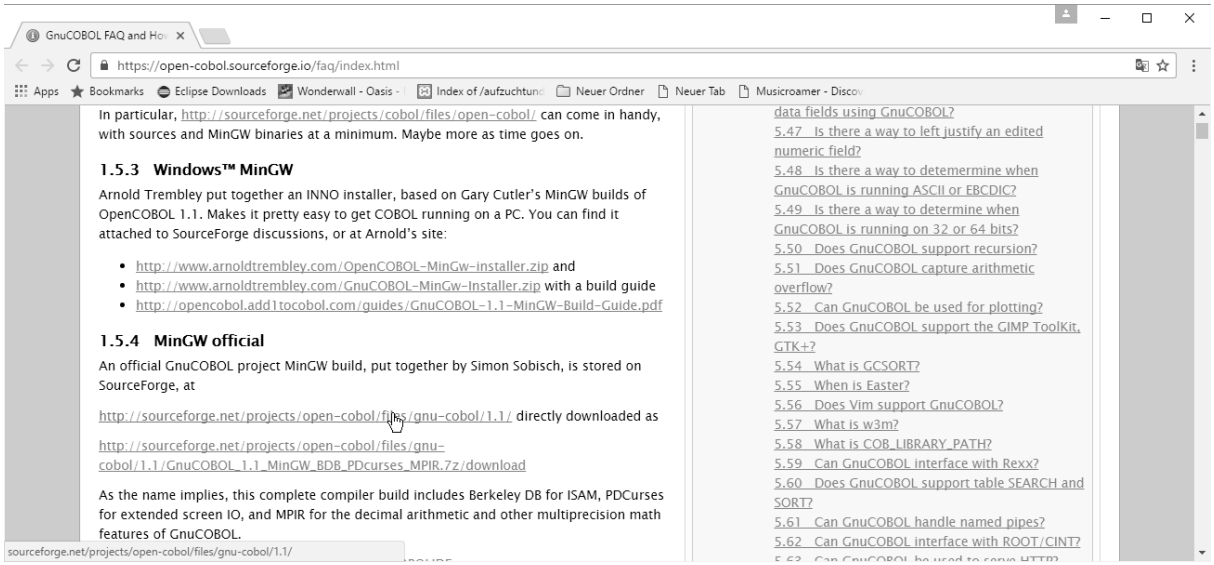

Abb. 3.2: Hier kommt man zu MinGW

Die aktuellste Version ist unter https://sourceforge.net/projects/open-cobol/files/latest/download?source=files zu finden, wobei Sie bei allen Verweisen auf Internet-Quellen wie gesagt damit rechnen müssen, dass diese sich über die Zeit ändern und auch bestimmte Programmversionen nicht mehr bereitstehen.

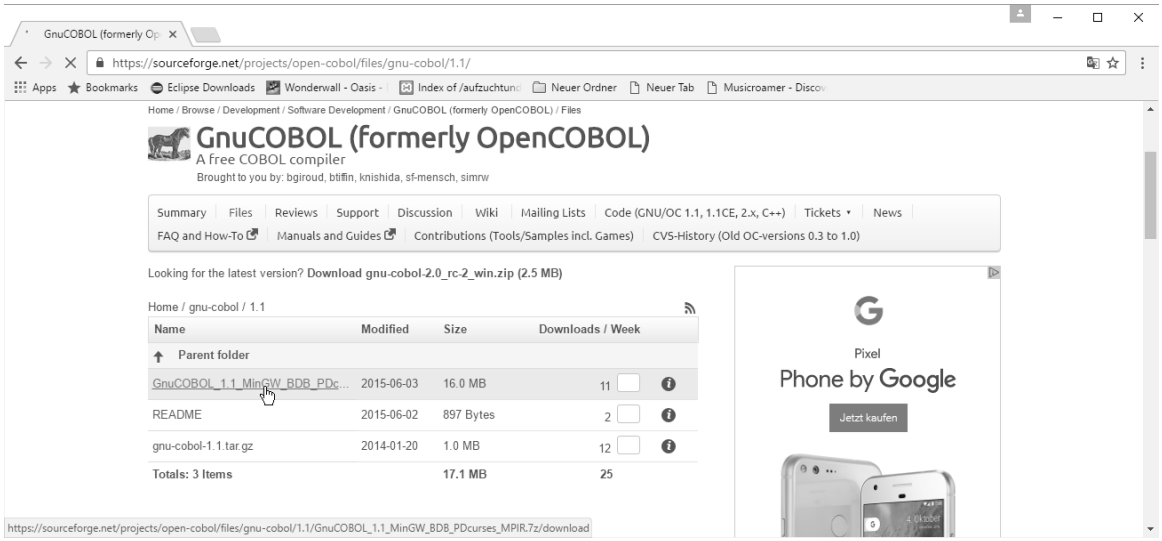

Abb. 3.3: Das Archiv von MinGW, das unter Windows nur noch extrahiert werden muss

Wenn Sie allerdings ein Archiv mit einer vorkompilierten Version von COBOL für Windows geladen haben, muss das aus dem Internet geladene Archiv nur extrahiert werden und ist out-of-the-box lauffähig. Maximal sind einige Umgebungsvariablen anzupassen und auch dafür gibt es in dem extrahierten Verzeichnis von MinGW ein vorgefertigtes Einrichtungsprogramm, das in der Konsole mit *set_env.cmd* aufgerufen werden kann (Abb. 3.4).

```
F:\MinGW>set_env.cmd

Setting environment for GnuCOBOL 1.1 with MinGW binaries
(GCC 4.8.1, BDB 6.1.23, PDcurses 3.4, MPIR 2.7.0)

cobc (GNU Cobol) 1.1.0
Copyright (C) 2001,2002,2003,2004,2005,2006,2007 Keisuke Nishida
Copyright (C) 2006-2012 Roger While
This is free software; see the source for copying conditions.  There is NO
warranty; not even for MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE.
Built     May 30 2015 23:52:03
Packaged  Jan 20 2014 07:40:53 UTC
C version "4.8.1"

F:\MinGW>
```

Abb. 3.4: In der Konsole wurde das Einrichtungsskript aufgerufen

3.1.2 Installieren auf einem Unix-artigen System

Während unter Windows die Verwendung eines bereits kompilierten Systems meist der bessere und vor allen Dingen einfachere Weg ist, installiert man unter einem Unix-artigen System wie Linux oder MacOS in der Regel GnuCOBOL direkt aus dem Internet mit dem jeweils eigenen Paketmanager. Dabei ist die Installation in der Konsole sogar oft einfacher als mit den grafischen Tools, aber von der konkreten Version des Betriebssystems abhängig. Etwa geht das so bei Fedora Linux:

sudo dnf install open-cobol

Für Debian-basierte Distributionen wie Mint Linux oder Ubuntu (Abb. 3.5) geht das beispielsweise auf diese Weise:

sudo apt-get install open-cobol

Die Installation unter den meisten anderen Linux-Distributionen und anderen Unix-artigen Systeme ist ebenso einfach. Mac-Benutzer können etwa Homebrew zum Installieren von GnuCOBOL verwenden. Insbesondere muss man nach der Installation nicht einmal Konfigurationen vornehmen – das System sollte bereits vollständig eingerichtet sein.

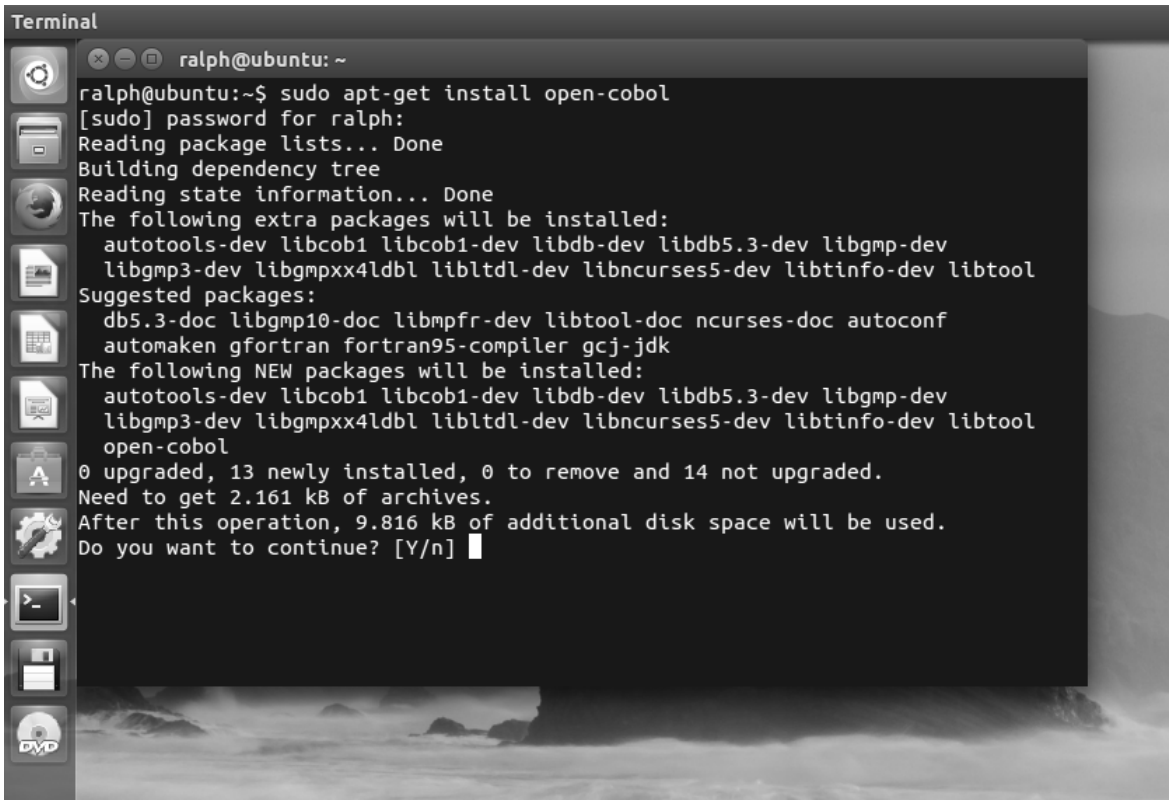

Abb. 3.5: Die Installation von GnuCOBOL unter Ubuntu

3.1.3 OpenCobolIDE

Zum Schreiben von COBOL-Quellcode genügt ein ganz einfacher Editor, aber mit Unterstützung der Syntax und anderen Features ist das Kodieren sicherer, schneller und bequemer.

Bereits ein allgemeiner Editor wie Notepad++, der unter Windows sehr beliebt ist, bietet bereits Unterstützung für COBOL.

Spezielle IDEs für COBOL bieten aber noch mehr Komfort. Die Auswahl ist nicht sonderlich groß, aber **OpenCobolIDE** ist eine solche spezialisierte, freie IDE mit Codevervollständigung, Syntax highlighting, einem Navigationsbaum etc. (https://github.com/OpenCobolIDE/OpenCobolIDE). Für die verschiedenen Betriebssysteme gibt es unter https://launchpad.net/cobcide/+download passende Installationsdateien.

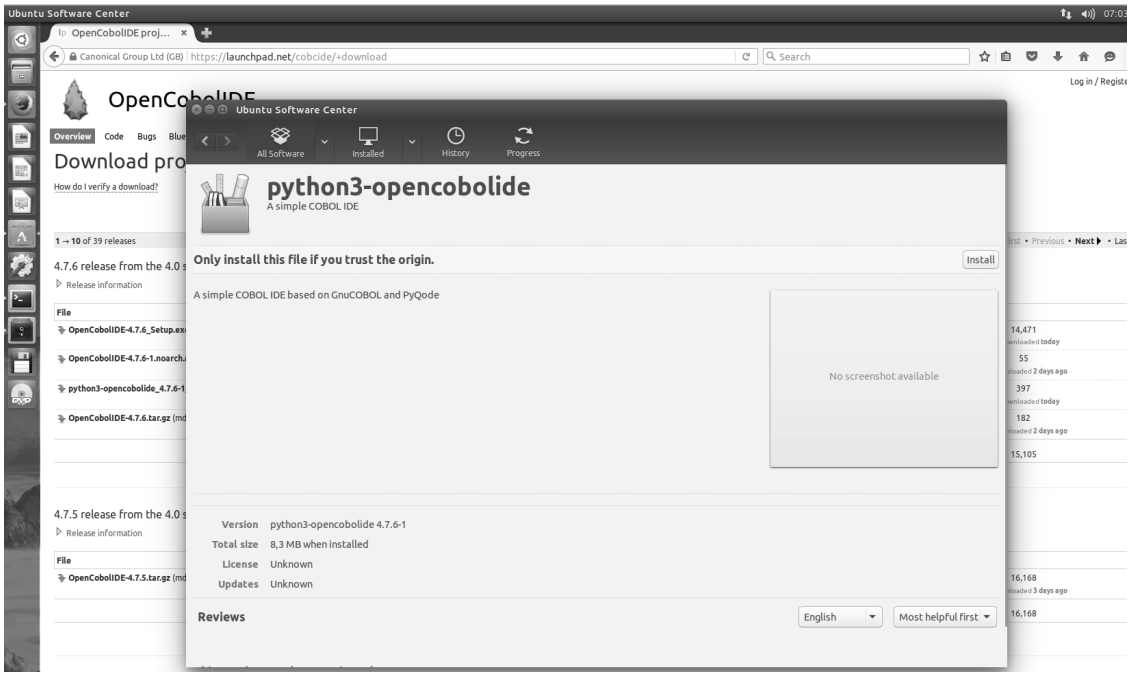

Abb. 3.6: Die Installation von OpenCOBOLIDE unter Ubuntu

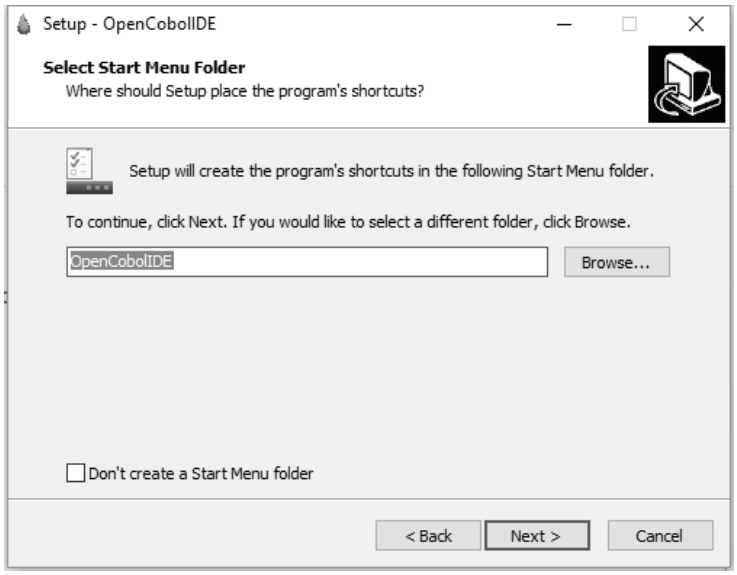

Abb. 3.7: Die Installation von OpenCOBOLIDE unter Windows erfolgt mit einem typischen Assistenten

4 Vom Quellcode zum ersten lauffähigen Programm

Kommen wir zum Erstellen des ersten COBOL-Programms und wie Sie es übersetzen und natürlich auch ausführen können.

4.1 Erstellen des Quellcodes

Nachfolgend sehen Sie ein ganz einfaches COBOL-Programm mit minimaler Syntax.

Beachten Sie, dass die Codes im Editor erst ab Spalte 8 notiert werden dürfen. Die ersten 7 Zeichen einer jeden Zeile müssen leer bleiben bzw. haben in COBOL eine besondere Bedeutung, die hier noch nicht ausgeführt, aber später geklärt wird (siehe Seite 17).

Geben Sie das nachfolgende Listing in einem Texteditor ein:

```
IDENTIFICATION DIVISION.
PROGRAM-ID. HALLOWELT.
PROCEDURE DIVISION.
  DISPLAY "Hallo Welt".
STOP RUN.
END PROGRAM HALLOWELT.
```

4.2 Übersetzen und Ausführen des Programms

Jeder Quellcode muss nach dem Speichern unter einem geeigneten Namen (etwa *erstes_beispiel.cbl*) zuerst **kompiliert** werden, damit lauffähiger Code entsteht.

So erfolgt etwa das Aufrufen des COBOL-Compilers in der Konsole (MinGW):

cobc –x erstes_beispiel.cbl

Damit wird unter Windows eine EXE-Datei erzeugt, die danach wie üblich direkt über den Namen der EXE-Datei aufgerufen werden kann (Abb. 4.1).

```
F:\cobolprojekte\kap4>cobc -x erstes_beispiel.cbl

F:\cobolprojekte\kap4>erstes_beispiel
Hallo Welt

F:\cobolprojekte\kap4>_
```

Abb. 4.1: Kompilieren und Ausführen in der Windows-Konsole

Unter einem Unix-artigen System wird keine EXE-Datei, sondern eine „normale“ ausführbare Datei ohne Dateierweiterung erstellt[1]. Der Aufruf erfolgt in dem Beispiel dann über **./erstes_beispiel** (beachten Sie den Punkt und den / vor dem Dateinamen).

Wenn Sie mit GnuCOBOLIDE arbeiten, können Sie den Quelltext auch direkt aus der IDE heraus übersetzen und gleich ausführen (Abb. 4.2). Allerdings müssen dazu der Compiler und die Umgebung in der IDE korrekt eingerichtet sein, was hier nicht vertieft wird.

Wenn Sie den Parameter ***–x*** beim Aufruf des Compilers weglassen, wird eine dynamisch ladbare, lauffähige Bibliothek erzeugt – ein sogenanntes **Module** (engl.) bzw. Modul (deutsch). Unter Windows ist das dann eine DLL-Datei (Dynamic Link Library). So etwas braucht man etwa, wenn man Unterprogramme erstellen möchte, die dann aus einem Hauptprogramm dynamisch

[1] Der Parameter –x steht ja auch für „e**x**ecutable“ – „ausführbar“ - und das ist nur unter Windows an eine bestimmte Dateiendung gebunden.

aufgerufen werden.

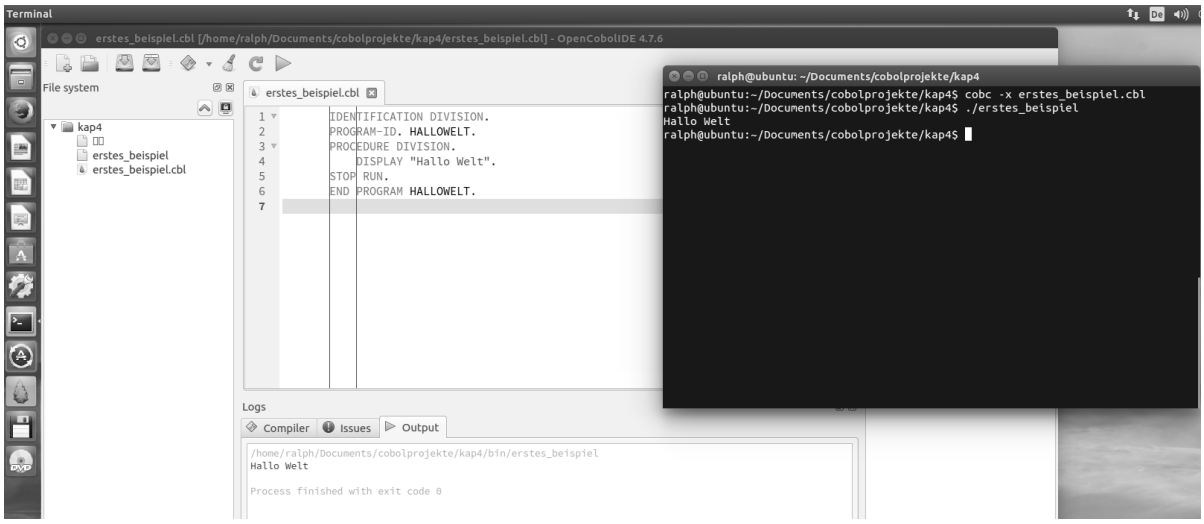

Abb. 4.2: Kompilieren und Ausführen unter Linux – sowohl in der Konsole als auch aus der IDE

4.3 Hilfe zum Compiler

Der COBOL-Compiler besitzt eine ganze Reihe an weiteren Parametern. Die Anweisung

cobc –h

zeigt eine Hilfe zum Compiler und dessen weiteren Parametern und Einstellungen an (Abb. 4.3).

Abb. 4.3: Hilfe zum Compiler

5 Kodierungsregeln und Kodierungsblätter

Die Syntax von COBOL beinhaltet sehr strenge und unflexible Kodierungsregeln, die sich aufgrund der Historie der Sprache ergeben. Es gibt sowohl zwingende Bereiche in einem COBOL-Programm, aber auch starre Regeln in den sogenannten **Kodierungsblättern**, die sogar die Spalten im Quellcode betreffen. Darüber hinaus ist in COBOL nur eine sehr eingeschränkte Menge an Zeichen erlaubt.

Dieser Abschnitt beschreibt diese grundsätzlichen Kodierungsregeln, die insbesondere für Umsteiger aus modernen Sprachen oft recht befremdlich erscheinen.

5.1 Grundlegende COBOL-Syntax - Überblick

- Die Dateiendung von COBOL-Quelltext ist *.cbl* oder *.col*.
- Der **Punkt** beendet COBOL-Anweisungen.
- Das Quellprogramm von COBOL muss in einem festen Format geschrieben werden. COBOL-Programme werden dazu auf **Kodierungsblätter** geschrieben.
- Es gibt in den Kodierungsblättern 80 Zeichenpositionen (Spalten) in jeder Zeile eines kodierenden Blatts.

5.2 Die Kodierungsblätter

Der Quellcode in COBOL ist grundsätzlich in Kodierungsblättern einzugeben. Die **Zeichenpositionen** (Spalten) werden dabei in die folgenden fünf Felder gruppiert:

- Der Bereich der Zeilennummern
- Der Bereich des Indikators
- Bereich A
- Bereich B
- Identifizierungsbereich

Viele Umsteiger aus „modernen“ Programmiersprachen machen Fehler bei der Einhaltung der Spalten bzw. Bereiche. Oft kann man in einem Editor aber auch nicht deutlich erkennen, ob man außerhalb des gültigen Bereichs für eine bestimmte Aktion ist. Etwa verschleiert der Tabulator, dass man noch nicht in Bereich A oder B, sondern noch im Bereich der Zeilennummern oder des Indikators ist oder nicht sichtbare Zeichen befinden sich rechts jenseits der gültigen Spalten. Eine IDE wie GnuOPENIDE hilft hier sehr viel, weil da auch die Bereiche angezeigt werden (Abb. 4.2). Bei anderen Editoren sollte man unbedingt die Anzeige der Spalten aktivieren und beachten. Ebenso hilft es, wenn man den Tabulator vermeidet oder auf 1 Zeichen einstellt.

Nachfolgend sehen Sie die Bedeutungen der verschiedenen Regionen in einem Kodierungsblatt:

Positionen	Feld	Beschreibung
1-6	Zeilennummern	Für Zeilennummern reserviert.
7	Indikator	Asterisk (*) für Kommentare. Bindestrich für die Fortsetzung. Slash (/) für Seitenvorschub.
8-11	Bereich A	Alle COBOL Divisionen, Abschnitte, Absätze und einige spezielle Einträge müssen in Bereich A beginnen.

12-72	Bereich B	Alle COBOL-Statements müssen in Bereich B beginnen und dürfen auch nicht über den Bereich (also Spalte 73 oder folgende) hinausgehen.
73-80	Identifizierungsbereich	Optionaler Bereich.

Tabelle 5-1: Die Zeichenpositionen in COBOL

Manche COBOL-Compiler unterscheiden nicht so streng zwischen Bereich A und Bereich B. Etwa der Compiler von GnuCOBOL. Aber um die Kompatibilität des Quellcodes mit anderen COBOL-Compilern zu gewährleisten, sollten Sie sich an diese Bereichsgrenzen halten. Und diese „laxe" Regel gilt in keinem Fall für die ersten 7 Spalten und alle Spalten ab Position 73. Diese Regionen müssen dringend korrekt eingehalten werden. Das bedeutet auch, dass Anweisungen in einer Zeile in COBOL nicht zu lang sein dürfen.

5.2.1 Ein weiteres, vollständigeres Beispiel

Kommen wir zu einem zweiten Beispiel, in dem von dem Benutzer eine Eingabe entgegengenommen und aufbereitet ausgegeben werden soll.

Beachten Sie, dass wieder die Spalten 1 bis 7 im Editor leer bleiben müssen. Erst ab Bereich A dürfen Eingaben erfolgen. Ausnahme ist der Kommentarbereich, der in jeder Zeile zwingend mit dem Zeichen * in Spalte 7 (dem Indikator) beginnen muss.

Geben Sie das nachfolgende Listing in einem Texteditor ein und übersetzen Sie es danach mit dem Compiler. Anschließend sollten Sie das Programm ausführen:

```
IDENTIFICATION DIVISION.
  PROGRAM-ID. EINGABENAMEN.
DATA DIVISION.
  WORKING-STORAGE SECTION.
  01 WS-NAME PIC X(5).
PROCEDURE DIVISION.
    DISPLAY "Bitte geben Sie Ihren Namen an".
    ACCEPT WS-NAME.
    DISPLAY "Hallo ", WS-NAME.
  STOP RUN.
  END PROGRAM EINGABENAMEN.
```

```
F:\cobolprojekte\kap5>cobc -x eingabenamen.cbl

F:\cobolprojekte\kap5>eingabenamen
Bitte geben Sie Ihren Namen an
Ralph
Hallo Ralph

F:\cobolprojekte\kap5>
```

Abb. 5.1: Der Quelltext wurde kompiliert und das Programm ausgeführt

Eine kleine Abwandlung des Beispiels, das rein funktional vollkommen identisch ist, zeigt einige weitere Varianten bei der Verwendung von Kodierungsblättern. Dabei werden hier ausdrücklich die ersten 7 Spalten des Kodierungsblatts zu sehen sein.

Geben Sie das nachfolgende Listing in einem Texteditor ein und übersetzen Sie es danach mit dem Compiler. Anschließend sollten Sie das Programm wieder ausführen:

```
      ****************************************
      * Author: Ralph Steyer
      * Date: 02.02.2017
      ****************************************
000001 IDENTIFICATION DIVISION.

000002 PROGRAM-ID. EINGABENAMEN.
000003 AUTHOR. RALPH STEYER.
      ***************************************
      * Das ist ein Kommentarbereich                  *
      ***************************************
000004 DATA DIVISION.
000005 WORKING-STORAGE SECTION.
000006 01 WS-NAME PIC X(5).

000007 PROCEDURE DIVISION.
000008     DISPLAY "Bitte geben Sie Ihren Namen an".
000009     ACCEPT WS-NAME.
000010     DISPLAY "Hallo ", WS-NAME.
000011 STOP RUN.
000012 END PROGRAM EINGABENAMEN.
```

Sie erkennen hier gegenüber dem vorherigen Beispiel einige Besonderheiten bzw. Änderungen.

Besonders auffällig sind die Zeilennummern. Diese sind explizit nicht (!) vor jeder Zeile im Quellcode zu finden. Sondern nur vor denjenigen, die wirklich relevant für den Compiler sind. Sie fehlen also beispielsweise ausdrücklich vor Kommentarzeilen oder Leerzeilen. Beachten Sie, dass die Leerzeilen im Quellcode bewusst gewählt wurden. Die Nummerierungen machen deutlich, welche Zeilen eine relevante „Position" im Quellcode haben. Allerdings ist so eine Nummerierung von Zeilen weder notwendig noch immer gleich gewählt. Ein Programmierer hat hier vollkommene Freiheiten, welche Zeilen er wie nummeriert. Auch Leerzeilen könnten z.B. mit Zeilennummer versehen werden, obgleich das selten sinnvoll ist.

COBOL stammt aus der Zeit von **Lochkarten**. Die optionale Nummerierung der relevanten Zeilen eines Quellcodes entspricht der früher üblichen Nummerierung von Lochkarten. Denn diese mussten unbedingt so gekennzeichnet werden, um sie wieder im Rahmen der Stapelverarbeitung (Batch) in die richtige Reihenfolge sortieren zu können, wenn der Stapel durcheinander geraten war.

Aber auch andere Details der Kodierungsblätter lassen sich aus dem Lochkartenkonzept herleiten. Die Firma IBM ließ sich bereits 1928 ein 80-Spalten-Format patentieren, das bis in die 1970er Jahre hinein weite Verbreitung fand und wohl der Hintergrund der Beschränkung von Kodierungsblättern auf 80 Zeichen begründet.

In dem Listing sollte zudem auffallen, dass Kommentare auch bereits vor der ersten DIVISION auftauchen können. Diese müssen wie gesagt zwingend in Spalte 7 mit dem passenden Indikator beginnen. Die weiteren Sterne in den anderen Spalten einer Kommentarzeile sind optional, aber oft üblich.

Die Angabe AUTHOR in der IDENTIFICATION DIVISION gilt in COBOL mittlerweile als obsolet. In dem Beispiel wird die Angabe des Autors deshalb redundant notiert. In der Praxis sollte man sie explizit nur noch in einem Kommentar notieren, aber viele alte Quellcodes enthalten sie noch in der IDENTIFICATION DIVISION.

5.3 Erlaubter Zeichensatz in COBOL

Bei den Zeichen, die Sie in einem COBOL-Quellcode verwenden können, gibt es einige Besonderheit bzw. Einschränkungen, die man in modernen Programmiersprachen so teils nicht mehr kennt:

- Die Zeichen stehen in der Hierarchie von COBOL am niedrigsten. Sie können nicht weiter unterteilt werden. Das gilt für alle höheren Programmiersprachen.
- Der COBOL-Zeichensatz umfasst nur 78 erlaubte Zeichen und das ist gegenüber modernen Sprachen doch sehr eingeschränkt.

Zeichen	Beschreibung
A-Z	Alphabet (Großbuchstaben)
a-z	Alphabet (Kleinbuchstaben)
0-9	Numerisch
	Leerzeichen
+	Pluszeichen
-	Minuszeichen oder Bindestrich
*	Stern
/	Schrägstrich vorwärts
$	Währungssymbol Dollar
,	Komma
;	Semikolon
.	Dezimalpunkt oder Periode
"	Anführungszeichen
(	Linke Klammer
)	Rechte Klammer
>	Größer als
<	Kleiner als
:	Doppelpunkt
'	Apostroph
=	Gleichheitszeichen

Tabelle 5-2: Zeichen in COBOL

In den folgenden Listings in den Unterlagen werden aus Platzgründen weder Zeilennummern noch der Indikator – also die führenden 7 Spalten jeder Zeile in einem Kodierungsblatt - angegeben. Denken Sie daran, dass Sie diese 7 Spalten auf den Kodierungsblättern aber unbedingt beachten. Und beachten Sie, dass aus drucktechnischen Gründen längere Zeilen möglicherweise auf mehrere Zeilen verteilt werden und Sie diese in einer Zeile eingeben müssen.

6 Grundaufbau eines COBOL-Programms

Ein COBOL-Programm hat immer die gleiche sequentielle Top-Down-Struktur. Strukturierung – von grob bis fein

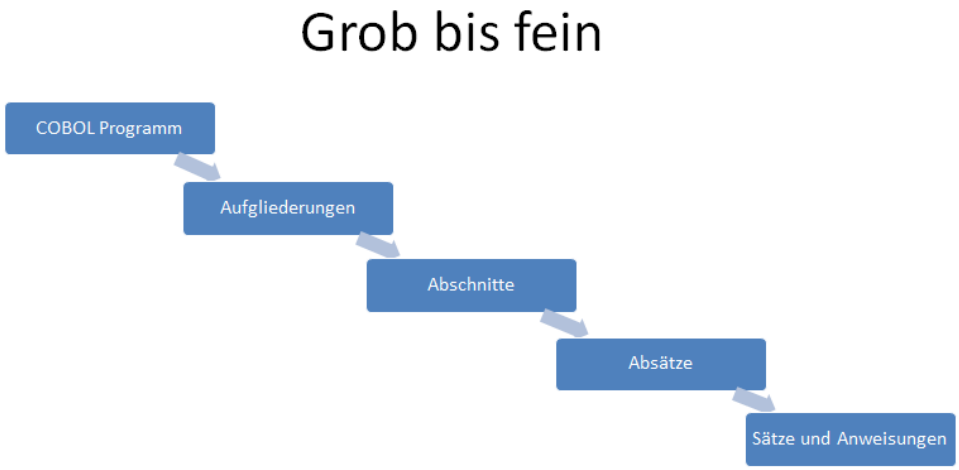

Abb. 6.1: Der Aufbau von COBOL-Programmen

- Das COBOL-Programm splittet sich in **Aufgliederungen** (DIVISONs oder Segmente genannt) auf.
- **Abschnitte/Sektionen** oder **SECTIONs** sind die logische Unterteilung der Programmlogik. Ein Abschnitt ist eine Sammlung von Absätzen.
- Die **Absätze** sind die optionale Unterteilung eines Abschnittes oder auch direkt der Aufgliederung/DIVISION. Man nutzt das in der PROCEDURE DIVISON bei Bedarf. Beachten Sie, dass man nicht zwingend Absätze in einem COBOL-Programm braucht. Er ist entweder benutzerdefiniert gekennzeichnet oder man hat hier zur Kennzeichnung einen vordefinierten COBOL-Name – beides gefolgt von einem Punkt. Ein Absatz besteht aus null oder mehr Sätzen/Einträgen. So könnte das aussehen:

```
...
  0090-MARKER.
...
  0095-MARKER.
...
0100-MARKER.
...
```

- Die **Sätze** sind die Kombination von einer oder mehreren Anweisungen. Die Sätze erscheinen werden nur in der PROCEDURE DIVISION. Ein Satz in COBOL muss mit einem Punkt enden. Mehrere Anweisungen in einem Satz bilden einen **Block**, wie es in anderen Programmiersprachen meist genannt wird.

- **Statements** (Anweisungen) sind die konkreten COBOL-Anweisungen. Beachten Sie, dass Statements nicht mit einem Punkt beendet werden, sondern nur die Sätze. Das ist oft etwas verwirrend, denn wenn ein Satz nur aus einer Anweisung besteht, sieht es so aus als würde die Anweisung mit dem Punkt beendet. Es ist aber der Satz.
- **Zeichen** sind – wie schon erwähnt -nicht mehr teilbare Symbole und am niedrigsten in der Hierarchie. Aus diesen werden die anderen Strukturen aufgebaut.

Ein Satz endet in COBOL wie erwähnt mit einem Punkt. Es ist jedoch ein häufiger Fehler bei Umsteigern, dass Sätze zu früh mit Punkten beendet werden. Viele „Sätze“ in COBOL werden von einer Endanweisung beendet. Oder man kann dies zumindest machen. Etwa die *IF*-Anweisung, die mit einem *END-IF* beendet wird.

Beispiel:

```
IF WS-NUM1 IS GREATER THAN WS-NUM2 THEN
  DISPLAY 'Ja'
ELSE
   DISPLAY 'WS-NUM1 < WS-NUM2'
END-IF.
```

In dem Fall dürfen die Anweisungen im Inneren des Blocks **nicht** mit einem Punkt abgeschlossen werden.

6.1 Vier Hauptsegmente (DIVISIONs)

Die Segmente in COBOL-Programmen gliedern sich wie folgt auf:

- Identifikation
- Umgebung (Environment)
- Daten
- Prozeduren

Man gruppiert diese DIVISIONs in logische Bereiche.

6.1.1 Initialisierungsbereiche

Dazu zählen die folgenden Segmente:

- Identifikation
- Umgebung
- Daten

6.1.2 Ausführung & Ende

Das gehört zu dieser logischen Eingruppierung:

- Verarbeitungsteil
    - Prozedurabschnitt
- Programmende
    - STOP RUN

7 COBOL-Syntax - Vertiefung

Dieses Kapitel führt nun die wesentlichen Elemente und Strukturen der COBOL-Syntax ein.

7.1 Case-Sensitivität in COBOL

Ursprünglich wurden COBOL-Anweisungen nur in Großbuchstaben geschrieben. Aber in COBOL wird **nicht** zwischen Groß- und Kleinschreibung unterschieden. COBOL ist somit **case insensitive** - also **nicht** case-sensitiv, wie die meisten modernen Programmiersprachen. Allerdings hat sich eingebürgert, COBOL-Schüsselworte durchgängig groß zu schreiben. Aber das ist nicht zwingend.

Das folgende Beispiel vertieft Ihre ersten COBOL-Kenntnisse und mischt bewusst – aus rein didaktischen Gründen – dennoch Groß- und Kleinschreibung. Dabei werden auch bestimmte Anweisungen verwendet, die noch nicht eingeführt wurden. Sie sollten aber mit etwas Programmiererfahrung nachvollziehbar sein.

Suchen Sie in dem Beispiel (CASESENSITIV) alle Stellen, an denen COBOL-Bezeichner klein statt groß geschrieben wurden. Die Lösung finden Sie im Anhang auf der Seite 78 ff.

```
IDENTIFICATION DIVISION.
  PROGRAM-ID. CASESENSITIV.
DATA DIVISION.
WORKING-STORAGE SECTION.
01 WS-NUM1 PIC 9(9) VALUE 3.
01 WS-NUM2 PIC 9(9).
PROCEDURE DIVISION.
  MOVE 15 TO WS-NUM2.
  if WS-NUM1 IS GREATER THAN OR EQUAL TO WS-NUM2 THEN
    display 'WS-NUM1 > WS-NUM2'
  ELSE
    DISPLAY 'WS-NUM1 < WS-NUM2'
  end-IF.
  STOP RUN.
  END PROGRAM CASESENSITIV.
```

7.2 Namensregeln für Bezeichner in COBOL

Für die Benennung von Bezeichnern in COBOL gelten einige strenge Regeln.

Alle Bezeichner, die in COBOL auftreten können, verwenden natürlich nur die oben genannten, erlaubten Zeichen. Natürlich dürfen auch keine reservierten Token verwendet werden. Das ist trivial.

Aber Bezeichner für **Variablen** etc. müssen auch zwingend mit einem Buchstaben beginnen. Zahlen oder etwa der Unterstrich sind nicht erlaubt. Sie haben auch nur maximal 30 Zeichen zur Verfügung.

Die Bezeichner von **Absätzen** / **Paragraphs** dürfen jedoch rein nummerisch sein und damit auch mit einer Zahl beginnen. Es ist sogar gängige Praxis in COBOL, dass die Bezeichner von Absätzen nummerisch beginnen.

Beispiel:

```
0050-OPEN-FILE.
   OPEN INPUT SALESFILE.
   OPEN OUTPUT PRINT-FILE.
...
0100-PROCESS-RECORDS.
   PERFORM 0110-WRITE-HEADING-LINE.
...
```

Dabei lässt man in der Regel Lücken in der Nummerierung der Absätze, um später Absätze in chronologischer Folge einfügen zu können.

7.3 Zeichenketten (Character Strings)

Zeichenketten werden in COBOL durch die Kombination von einzelnen Zeichen gebildet. Eine Zeichenfolge kann das sein:

- Kommentar
- Literal
- COBOL-Wort

Alle Zeichenketten müssen mit Separatoren beendet werden. Ein **Separator** (**Begrenzer**) wird zur Trennung von Zeichenketten verwendet.

7.4 Separatoren

Häufig verwendete Separatoren in COBOL sind folgende Zeichen:

- Leerzeichen
- Komma
- Punkt
- Apostrophe
- Klammern
- Anführungszeichen

7.5 Kommentare

Ein Kommentar ist eine Zeichenfolge, die die Ausführung eines Programms nicht beeinflusst. Es kann jede Kombination von Zeichen sein.

Es gibt zwei Arten von Kommentaren:

7.5.1 Kommentarzeile

Eine Kommentarzeile kann überall notiert werden. Sie startet mit dem Zeichen *****. **Dieses muss zwingend in Spalte 7 (der Indikatorspalte) stehen!**

7.5.2 Kommentareintrag

Kommentareinträge sind diejenigen, die in den optionalen Absätzen der IDENTIFICATION DIVISION enthalten sind. Sie werden im Bereich B geschrieben.

Zeichen, die in dem Kodierungsblatt ab der Spalte 73 notiert werden und nicht über die Spalte 80 hinausgehen (Identifizierungsbereich), werden vom Compiler auch ignoriert und sind damit in gewisser Weise auch Kommentare.

7.6 Operatoren in COBOL

Ein **Operator** ist in der Programmierung ein Symbol (bzw. Token), das angibt, welche Operation in einem Ausdruck ausgeführt werden soll. Die Elemente (Variablen oder Literale, das heißt im numerischen Fall Zahlen), mit denen eine solche Operation durchgeführt wird, nennt man **Operanden**. Man unterscheidet die Operatoren nun nach ihrem Typ.

7.6.1 Arithmetische Operatoren

Arithmetische Operatoren benutzen immer eine oder zwei Zahlenoperanden und liefern als Ergebnis einer arithmetischen Operation eine neue Zahl als Wert. COBOL nutzt einfach die Zeichen, die man aus der Schulmathematik kennt:

```
+ - * /
```

Man wendet in COBOL diese Operatoren in Verbindung mit COMPUTE an. Es gibt auch alternative mathematische Verben (siehe Seite 39).

Eine Besonderheit ist in COBOL der Umgang mit Modulo, was – statt über einen Operator - über eine „Intrinsic Function“ mit Namen MOD bereitgestellt wird (siehe Seite 58).

Unter Modulo versteht man die Division mit Rest. Der Rest ist das Ergebnis.

7.6.2 Vergleichsoperatoren

Sie können bei logischen Ausdrücken nicht nur auf Gleichheit von zwei Seiten prüfen, sondern auch auf diverse andere Bedingungen, etwa ob der Wert auf einer Seite kleiner oder größer ist. Dazu stehen Ihnen unter COBOL die folgenden Vergleichsoperatoren zur Verfügung:

```
= < > <= >= <>
```

Beachten Sie, dass COBOL das einfache Gleichheitszeichen für den Test auf Gleichheit verwendet. In viele Programmiersprachen wird hingegen das doppelte Gleichheitszeichen verwendet.

Als Ergebnis gibt es einen sogenannten booleschen Wert (Wahrheitswert), den Sie mit dem Operator IS überprüfen und mit dem Operator NOT negieren können.

Für die Vergleichsoperatoren stellt COBOL auch Token zur Verfügung, die alternativ verwendet werden können.

- EQUAL TO (=)
- GREATER THAN (>)
- LESS THAN (<),
- GREATER THAN OR EQUAL (>=)
- LESS THAN OR EQUAL (<=)]

7.6.3 Zuweisungsoperatoren

In COBOL gibt es in dem Sinn keine Zuweisungsoperatoren im klassischen Sinn. Stattdessen kommen sogenannte Verben (siehe Seite 32) zum Einsatz, die damit eine Art Zwitter zwischen Operatoren und Anweisungen darstellen:

- MOVE
- SET
- COMPUTE

7.6.4 Logische Operatoren

Logische Operatoren verknüpfen Bedingungen bzw. boolesche Ausdrücke. Es gibt in COBOL „und“ (AND) und „oder“ (OR).

7.6.4.1 Bitweise Operatoren

COBOL kennt auch Operatoren auf Bitebene, die der Vollständigkeit halber erwähnt werden sollen:

- B-AND
- B-OR
- B-XOR
- B-NOT
- B-LEFT
- B-RIGHT

7.7 Die DIVISIONs

Schauen wir uns nun die einzelnen DIVISIONs in einem Kodierungsblatt etwas genauer an:

- IDENTIFICATION DIVISION: Erkennungsteil – Pflicht in einem COBOL-Programm
- ENVIRONMENT DIVISION: Beschreibung der Umgebung
- DATA DIVISION: Datenteil
- PROCEDURE DIVISION: Prozedurteil

7.7.1 Die IDENTIFICATION DIVISION

Ein COBOL-Programm kann im Grunde ausschließlich aus dieser DIVSION bestehen, wenngleich das Programm dann keine Funktionalität besitzt und weitgehend nutzlos ist. Aber rein formal genügt diese DIVISION. Inhalte dieses Abschnitts sind folgende:

- IDENTIFICATION DIVISION.
- PROGRAM-ID. Programmname.
- [AUTHOR. Verfasser.]
- [INSTALLATION. Computertyp.]
- [DATE-WRITTEN. Datum der Erstellung.]
- [DATE-COMPILED. Datum der Umwandlung.]
- [SECURITY. Wer darf Programm lesen.]

Der erste Teil der Angabe (Überschriften / Paragrafen) steht im A-Bereich, die Inhalte stehen im B-Bereich.

Beispiel:

```
000001 IDENTIFICATION DIVISION.
000002 PROGRAM-ID. KODIERUNG as "KODIERUNG".
000003 AUTHOR. RALPH STEYER.
```

Für die Reihenfolge der Angaben gibt es Konventionen. Aber der Compiler berücksichtigt die Reihenfolge nicht.

In dem Bereich sollte man nur Mussangaben verwenden – also die PROGRAM-ID. Optionale Angaben wurden von ANSI als „deprecated“ gekennzeichnet. Man findet diese Angaben allerdings noch in vielen Altprogrammen.

7.7.2 Die ENVIRONMENT DIVISION

Dieser optionale Abschnitt enthält eine Beschreibung der COBOL-Umgebung. Das umfasst etwa dies:

- Namen
- Organisationsformen
- Zugriffsmethoden

Oft gibt es zwei Sektionen:

- CONFIGURATION SECTION
- INPUT-OUTPUT SECTION

Beispiel:

```
000004 ENVIRONMENT DIVISION.
000005 CONFIGURATION SECTION.
000006 SOURCE-COMPUTER.
000007 OBJECT-COMPUTER.

000008 INPUT-OUTPUT SECTION.
```

Im **Konfigurationsabschnitt** kann man etwa Spezialnamen festlegen.

Beispiele für SPECIAL-NAMES:

```
DECIMAL-POINT IS COMMA.
SYSOUT IS AUSGABE.
PAGE IS NEUE-SEITE.
ALPHA-TEST IS 'A' ALSO 'a' etc.
```

Im Eingabe-Ausgabe-Abschnitt kann für Dateizugriffe etwa so etwas festlegen:

- Organisationsform der Dateien (direkt, relativ, sequentiell)
- Zugriffsmethode (direkt, sequentiell, dynamisch)

7.7.3 Die DATA DIVISION

Der Datenbereich ist in einem COBOL-Programm zwar optional, aber meist sehr wichtig. Hier werden Felder / Variablen definiert bzw. deklariert. Dabei unterscheidet man in COBOL folgende Begriffe:

- Data Name (Datennamen)
- Level Number (Stufennummer)
- Picture Clause (Picture-Klausel)
- Value Clause (Wertklausel)

Die Struktur ist allgemein folgende:

```
01          TOTAL-STUDENTS      PIC9(5)       VALUE '125'.
 |                |                 |              |
```

Level Number *Data Name* *Picture Clause* *Value Clause*

7.7.3.1 Data Name

Datennamen sind die Bezeichner einer Variablen und müssen in der DATA DIVISION definiert werden, bevor sie in der PROCEDURE DIVISION verwendet werden können. Sie müssen dazu einen benutzerdefinierten Namen angeben. Diese Datennamen verschaffen den Bezug auf den Speicherplatz. Sie können elementar oder ein sogenannter Gruppentyp sein.

Beispiele:

Valid:

```
WS-NAME
TOTAL-STUDENTS
A100
100B
```

Invalid:

`MOVE`	(Reserviert)
`COMPUTE`	(Reserviert)
`100`	(Kein Zeichen aus dem Alphabet)
`100+B`	(+ ist verboten)

7.7.3.2 Level Number (Stufennummern)

Eine Stufennummer wird verwendet, um die Datenebene in einem Datensatz anzugeben. Sie werden ebenso verwendet, um zwischen Elementar- und Gruppenelementen zu unterscheiden und Felder zu gruppieren. Elementare Elemente können verwendet werden, um Gruppenelemente zu erstellen.

Beispiel:

```
01 DATEN.
  02 GRUPPE1.
    05 WS-NUM1 PIC PPP999.
    05 WS-NUM2 PIC X(6) VALUE '123$'.
    05 WS-NUM3 PIC 9(3)V9(2) VALUE 123.45.
  02 GRUPPE2.
    06 WS-NUM4 PIC 999 VALUE 1.
    06 WS-NUM5 PIC 999 VALUE 3.
    06 WS-NUM6 PIC 9999.
```

Für Werte 0.000001 .. 0.000999

Hier sehen Sie die Stufennummern mit einer Beschreibung:

Stufen	Beschreibung
01	Beschreibung von Strukturen (Record).
02 bis 49	Gruppierung und elementare Einträge.
66	Redefinitionen oder Umbenennungen.
77	Nicht mehr teilbare Einträge – wird heutzutage meist durch 01 definiert.
88	Schalter für den Namen von Bedingungen.

Tabelle 7-1: Die Stufennummern bei der Felddeklaration

7.7.3.3 Picture Clause (Feldtypen)

Mit der Picture Clause wird die Art von einem **elementaren** Feld festgelegt. Man nennt das den **Datentyp** oder **Feldtyp**. Dieser kann in COBOL nummerisch, alphabetisch oder alphanummerisch sein. Das ist gegenüber modernen Sprachen eine ziemlich eingeschränkte Auswahl an Datentypen.

- alphanummerisch wird mit **X** bezeichnet. In Klammern folgt die Länge. Alternativ gibt man jedes Zeichen einzeln an. Beispiel:

```
01 FELD1 PIC  X(020).
01 FELD2 PIC  XXXX.
```

- alphabetisch wird mit **A** bezeichnet. In Klammern folgt die Länge oder man gibt auch hier jedes Zeichen einzeln an. Beispiel:

```
01 FELD2 PIC  A(020).
01 FELD2 PIC  AA.
```

- nummerisch beginnt mit der **9**. In Klammern folgt auch hier bei Bedarf die Länge oder man gibt explizit jedes einzelne Zeichen an. Zusätzliche Angaben sind möglich für ein Vorzeichen (**S** - Sign). Nachgestellt kann man angeben, ob das Feld binär gespeichert werden soll (BINARY). Oder implizit nummerisch dezimal (**V**). Beispiel

```
01 FELD-BINAER-OHNE-VORZEICHEN PIC 9(08) BINARY.
01 FELD1-DECIMAL-OHNE-VORZEICHEN PIC 9(5)V99.
01 FELD2-DECIMAL-OHNE-VORZEICHEN PIC 999V9(6).
01 FELD-BINAER-MIT-VORZEICHEN PIC S9(08) BINARY.
01 FELD-DECIMAL-MIT-VORZEICHEN PIC S9(5)V9(9).
```

Die verschiedenen Typangaben lassen sich in einer Deklaration auch mischen. Sie geben einfach für jedes Zeichen sein individuell gewünschtes Format an. Beispiel:

```
GEMISCHTES-FELD1 PIC  AA99.
```

Bei der Spezifikation des Datentyps von Zeichen über die Picture-Klauseln gibt es noch weitere Symbole mit weitergehender Bedeutung, die aber eher selten zum Einsatz kommen. Diese sollen hier nur angedeutet werden. Mit **B** kann man etwa Leerzeichen festlegen, mit **E** den Start eines Exponenten bei einer Gleitkommadarstellung und mit **P** kann man die Position des Dezimalpunkts angeben. Darüber hinaus gibt es die Möglichkeit Währungssymbole anzugeben und noch einige weitere Sonderzeichen. Sie finden z.B. unter dem Link https://supportline.microfocus.com/documentation/books/mx30/lhpdf40g.htm mehr Informationen dazu.

7.7.3.4 Value Clause

Die Value Clause (Wertklausel) gibt mit VALUE bei einem **elementaren** Typ optional einen Defaultwert (Vorgabewert) für eine Variable an. Dieser muss natürlich zum Datentyp, der mit der Picture-Klausel festgelegt wird, passen. Sehr oft verwendet man hier figurative Konstanten (siehe Seite 32). Insbesondere, wenn man alle Stellen eines Feldes mit einem Wert vorbelegen möchte, verwendet man auch VALUES (Plural). Oder man wählt die Pluralversion der figurativen Konstanten. COBOL ist an dieser Stelle ungewöhnlich flexibel, was durchaus zu Verwirrung führen kann.

7.7.3.5 Ein Beispiel zur Deklaration von Variablen

Hier ist ein vollständiges Beispiel (*picture.cbl*), bei dem die Deklaration von Variablen im Fokus stehen soll.

```
F:\cobolprojekte\kap7>cobc -x picture.cbl

F:\cobolprojekte\kap7>picture
Geben Sie die erste zweistellige Zahl ein
12
Geben Sie die zweite dreistellige Zahl ein
123
Geben Sie 5 Buchstaben ein
abcde
Geben Sie 5 Buchstaben oder Zahlen ein
a1b2c
Geben Sie 2 Buchstaben und dann 2 Zahlen ein
ab12
12.00
+123
abcde
a1b2c
ab12

F:\cobolprojekte\kap7>
```

Abb. 7.1: Verschiedene Picture-Klauseln

Erstellen Sie den nachfolgenden Quellcode, kompilieren Sie ihn und führen Sie dann das Programm aus. Achten Sie bei der Ausführung darauf, dass Sie die Zeichen wie gefordert eingeben.

```
IDENTIFICATION DIVISION.
  PROGRAM-ID. PICTURES.
DATA DIVISION.
  WORKING-STORAGE SECTION.
  01 DATEN.
    05 WS-NUM1 PIC 9(2)V99 VALUE ZEROES.
    05 WS-NUM2 PIC S9(3) BINARY VALUE 0.
    05 WS-ALPHA PIC A(5).
    05 WS-ALPHANUM PIC X(5).
    05 WS-GEMISCHTES-FELD PIC  AA99.
PROCEDURE DIVISION.
  DISPLAY "Geben Sie eine zweistellige Zahl ein".
  ACCEPT WS-NUM1.
  DISPLAY "Geben Sie eine dreistellige Zahl ein".
  ACCEPT WS-NUM2.
  DISPLAY "Geben Sie 5 Buchstaben ein".
  ACCEPT WS-ALPHA.
  DISPLAY "Geben Sie 5 Buchstaben oder Zahlen ein".
  ACCEPT WS-ALPHANUM.
  DISPLAY "Geben Sie 2 Buchstaben plus 2 Zahlen ein".
  ACCEPT WS-GEMISCHTES-FELD.
  DISPLAY WS-NUM1.
  DISPLAY WS-NUM2.
  DISPLAY WS-ALPHA.
  DISPLAY WS-ALPHANUM.
  DISPLAY WS-GEMISCHTES-FELD.
  STOP RUN.
  END PROGRAM PICTURES.
```

Führen Sie nun das Programm erneut aus und achten darauf, dass Sie die Zeichen **nicht** (!) wie gefordert eingeben. Also wenn Zahlen gefordert werden, geben Sie Buchstaben ein und umgekehrt. Geben Sie auch mehr oder weniger Zeichen ein als notwendig bzw. erlaubt. Die Lösung finden Sie im Anhang auf der Seite 78.

7.7.3.6 Gruppierung von Daten

Bei einer **Gruppierung** von Daten werden Sie – wie schon oben gesehen - in COBOL keine Picture-Klausel finden. Ebenso keinen Vorgabewert über eine Wertklausel. Erst die enthaltenen elementaren Daten haben wieder so eine Picture-Klausel und eventuell eine Wertklausel. Man kann Gruppierungen von Daten über verschiedene Techniken in COBOL ansprechen und damit gewährleisten, dass die enthaltenen elementaren Daten in einer gewünschten Reihenfolge im Speicher vorliegen. Dazu werden noch Beispiele in späteren Kapiteln folgen, die das erläutern (siehe unter anderem auf Seite 59).

Beispiel:

```
05 SALESPERSON-NAME.
  10 LASTNAME  PIC X(30).
  10 FIRSTNAME PIC X(30).
```

7.7.4 PROCEDURE DIVISION

- Die Logik des Programms oder der Code
- Aufteilung nach Regeln der strukturierten Programmierung

7.8 Literale in COBOL

Literale sind konstante Werte, die in einem Programm verwendet werden. Etwa, um Werte von Variablen zu repräsentieren oder direkt in Berechnungen. In folgendem Beispiel ist *"Hello World"* ein Literal.

```
PROCEDURE DIVISION.
DISPLAY 'Hello World'.
```

Es gibt in COBOL zwei Arten von Literalen.

7.8.1 Alphanumerische LiteraleDie

- Alphanumerische Literale (oft **Strings** genannt) werden in Anführungszeichen (doppelte Hochkommata) oder Apostrophe (einfache Hochkommata) eingeschlossen. Beide Syntaxvarianten sind gleichwertig und erlauben das Verschachteln von Strings.
- Die Länge kann bis zu 160 Zeichen betragen, was gegenüber modernen Sprachen sehr wenig ist. Beachten Sie, dass Sie damit die maximale Breite einer Zeile in einem Kodierungsblatt überschreiten und die Zeichen bei Bedarf auf mehrere Zeilen verteilen müssen.
- Ein Apostroph oder ein Anführungszeichen kann immer ein Teil eines Literals sein, wenn es **maskiert** (gedoppelt) wird.

Die folgenden Beispiele zeigen gültige und ungültige alphanumerische Literale.

Valid:

```
'This is valid'
"This is valid"
'This isn''t invalid'
```

Invalid:

```
'This is invalid"
'This isn't valid'
```

7.8.2 Zahlenliterale

- Ein numerisches Literal ist eine Kombination aus Ziffern von 0 bis 9, +, -, oder Dezimalpunkt.
- Die Länge kann bis zu 18 Zeichen betragen.

- Das Vorzeichen kann nicht ganz rechts stehen.
- Der Dezimalpunkt sollte nicht am Ende stehen.

Die folgenden Beispiele zeigen gültige und ungültige numerische Literale.

Valid:

```
100
+10.9
-1.9
```

Invalid:

```
1,00
10.
10.9-
```

7.9 Grundverben in COBOL

COBOL verwendet für gewisse Aufgaben (Tasks) eine Reihe vordefinierter Schlüsselworte, die hier **Verben** oder **COBOL-Worte**-genannt werden. Dazu zählen etwa DISPLAY, ACCEPT etc.

- Ein COBOL-Wort ist eine Zeichenkette, die ein reserviertes Wort oder ein benutzerdefiniertes Wort sein kann.
- Die Länge kann bis zu 30 Zeichen betragen.

7.9.1 Benutzerdefiniert

- Benutzerdefinierte Wörter werden für die Benennung von Dateien, Daten, Aufzeichnungen, Absatznamen und Abschnitte verwendet.
- Alphabet-Zeichen, Ziffern und Bindestriche sind zulässig
- In COBOL reservierte Wörter sind nicht erlaubt.

7.9.2 Reservierte Wörter

Reservierte Wörter sind vordefinierte Wörter in COBOL. Es gibt verschiedene Arten von reservierten Wörtern:

- Schlüsselwörterwie ADD, ACCEPT, MOVE, etc.
- Sonderzeichen wie +, -, *, & lt;, & lt; =, usw.
- Figürliche (Figurative) Konstanten wie ZERO, SPACES etc.

7.9.3 Figurative Konstanten

Die figurativen Konstanten stehen in COBOL für Werte (meist gibt Singular- und Pluralvarianten) und erlauben eine sprechende Anwendung. Es gibt etwa diese Konstanten, die weitgehend selbsterklärend sind:

- [ALL] ZERO / ZEROS / ZEROES
- [ALL] SPACE / SPACES
- [ALL] HIGH-VALUE / HIGH-VALUES
- [ALL] LOW-VALUE / LOW-VALUES
- [ALL] QUOTE / QUOTES
- ALL '*literal*'

- [ALL] NULL / NULLS

In verschiedenen Beispielen werden die Konstanten zum Einsatz kommen (siehe etwa Seite 37).

Mit den Token TRUE und FALSE stehen auch die typischen Symbole für Wahrheitswerte zur Verfügung.

8 Grundlegende Befehle in COBOL

Nachfolgend finden Sie einige wichtige COBOL-Befehle, die in vielen Programmen in der PROCEDURE DIVISION Anwendung finden.

Viele Verben / Anweisungen in COBOL können von einer Endanweisung beendet werden. Dabei wird dem Verb einfach der Präfix *END* vorangestellt, der mit einem Bindestrich mit dem Verb verbunden wird.

8.1 Allgemeine Verben

Zuerst sollen eine Reihe an Verben vorgestellt werden, die in den meisten COBOL-Programmen auftauchen. Damit werden so grundlegende Dinge wie das Initialisieren von Variablen und deren Wertänderung, das Entgegennehmen von Benutzereingaben und die Anzeige von Daten vorgenommen.

8.1.1 MOVE

Verschiebt bzw. bewegt einen Wert von einer Stelle an die andere. Damit kann man vor allen Dingen Variablen mit einem Wert belegen.

Beispiel:

```
MOVE 15 TO WS-NUM2.
```

8.1.2 DISPLAY

Dient zur Anzeige von Daten. Streng genommen ist das ein Schreiben auf SYSOUT (die Konsole).

Beispiel:

```
DISPLAY 'WS-NUM1 < WS-NUM2'
```

8.1.3 ACCEPT

Die Anweisung dient zum Lesen vom System (von SYSIN). Das ist in der Regel die Konsoleneingabe.

Beispiel:

```
ACCEPT WS-FIRST-NAME.
```

Man kann auch die Konsole direkt angeben.

Beispiel:

```
ACCEPT WS-FIRST-NAME FROM CONSOLE.
```

Leider ist die Anweisung recht unflexibel, denn die Anzahl der Zeichen bei der Eingabe muss exakt der Länge des Feldes entsprechen, in das die Eingabe erfolgen soll. Sind die Eingabedaten kürzer als der Empfangsbereich, wird der Bereich mit Leerzeichen der entsprechenden Darstellung für den Empfangsbereich aufgefüllt.

8.1.4 CONTINUE

Dabei handelt es sich um eine sogenannte Leeranweisung. In Verbindung mit Schleifen und Entscheidungen manchmal sinnvoll.

Beispiel:

```
IF STD  < 34 THEN
  CONTINUE
```

```
ELSE
...
END-IF.
```

Auch kann die Anweisung dann hilfreich sein, wenn COBOL explizit eine Anweisung fordert, diese aber eben nichts tun darf. Das kann in Verbindung mit Sprunganweisungen sinnvoll sein. Siehe dazu auch das Beispiel auf Seite 60.

8.1.5 INITIALIZE

Damit kann man in Verbindung mit REPLACING Werte von Variablen initialisieren.

Beispiel:

```
INITIALIZE WS-AGE REPLACING NUMERIC DATA BY 12.
```

8.1.6 Ein vollständiges Beispiel

Das nachfolgende Beispiel nutzt die meisten der beschriebenen Verben, die Sie aber auch schon an anderer Stelle in der Praxis gesehen haben.

Erstellen Sie das nachfolgende Beispiel, übersetzen Sie es mit dem Compiler und lassen Sie es laufen:

```
IDENTIFICATION DIVISION.
  PROGRAM-ID. VERBEN.
DATA DIVISION.
  WORKING-STORAGE SECTION.
    01 DATEN.
      05 WS-NUM1 PIC 9(2) VALUE 42.
      05 WS-NUM2 PIC 9(2).
PROCEDURE DIVISION.
  INITIALIZE WS-NUM2 REPLACING NUMERIC DATA BY 1.
  DISPLAY WS-NUM2.
  INITIALIZE WS-NUM2 REPLACING NUMERIC DATA BY 2.
  DISPLAY WS-NUM2.
  DISPLAY "Geben Sie eine 2-stellige Zahl ein".
  ACCEPT WS-NUM1.
  MOVE WS-NUM1 TO WS-NUM2.
  DISPLAY WS-NUM2.
  MOVE 44 TO WS-NUM2.
  DISPLAY WS-NUM2.
  STOP RUN.
   END PROGRAM VERBEN.
```

8.2 Mathematische Verben in COBOL

COBOL verfügt für arithmetische Aufgaben (Tasks) über eine Reihe spezieller mathematischer Verben, die in dem Abschnitt vorgestellt werden. Diese Verben lassen sich oft als Alternative zur Verwendung arithmetischer Operatoren einsetzen.

- ADD
- SUBTRACT
- MULTIPLY
- DIVIDE
- COMPUTE

Bis auf das letzte Verb sollten die Verben von der Bedeutung selbsterklärend sein. Aber nicht unbedingt von der Anwendung. Hier gibt es einige Besonderheiten.

8.2.1 Die direkten mathematischen Operationsverben

Betrachten wir zuerst die Verben, die die üblichen mathematischen Operationen beschreiben. Dabei sind allerdings die verbundenen Verben TO, FROM, BY und INTO zu beachten, die in Kombination mit den mathematischen Verben verwendet werden. Das geht so:

• ADD ... **TO** ...

• SUBTRACT ... **FROM** ...

• MULTIPLY ... **BY** ...

• DIVIDE ... **INTO**

Beispiele:

```
ADD W1 TO W2
ADD 12 TO W2
END-ADD
SUBTRACT W1 FROM W2
SUBTRACT 12 FROM W2
END-SUBTRACT
MULTIPLY W1 BY W2
MULTIPLY 12 BY W2
END-MULTIPLY
DIVIDE W1 INTO W2
DIVIDE 12 INTO W2
END-DIVIDE
```

Sie können als Operanden der mathematischen Verben auch mehrere Operanden (entweder durch Kommata oder Leerzeichen getrennt) angeben.

Bei einer Division erhält man den Restwert mit REMAINDER.

Mit der Anweisung GIVING kann man sich das Resultat einer Operation in einer eigenen Zielvariablen ausgeben lassen. Dann werden die Operanden vor GIVING nicht verändert. Ohne GIVING werden die verknüpften Operanden verändert, was teils nicht ganz trivial nachzuvollziehen ist.

Betrachten Sie das folgende Beispiel *calc1.cbl*, um dieses Verhalten nachzuvollziehen.

```
IDENTIFICATION DIVISION.
  PROGRAM-ID. CALC.
DATA DIVISION.
  WORKING-STORAGE SECTION.
    01  A     PIC 9(1) VALUE 3.
    01  B     PIC 9(1)  VALUE 4.
    01  RES   PIC 9(2).
PROCEDURE DIVISION.
  DISPLAY "Wert von A: ", A.
  DISPLAY "Wert von B: ", B.
  DISPLAY "ADD A TO B GIVING RES."
  ADD A TO B GIVING RES.
  DISPLAY "Wert von A: ", A.
```

```
        DISPLAY "Wert von B: ", B.
        DISPLAY "Wert von RES: ", RES.
        DISPLAY "ADD A TO B."
        ADD A TO B.
        DISPLAY "Wert von A: ", A.
        DISPLAY "Wert von B: ", B.
        STOP RUN.
        END PROGRAM CALC.
```

```
F:\cobolprojekte\kap8>calc1
Wert von A: 3
Wert von B: 4
ADD A TO B GIVING RES.
Wert von A: 3
Wert von B: 4
Wert von RES: 07
ADD A TO B.
Wert von A: 3
Wert von B: 7

F:\cobolprojekte\kap8>
```

Abb. 8.1: Mathematische Operationen am Beispiel der Addition mit und ohne GIVING

Bei der Anweisung *ADD A TO B GIVING RES.* steht das Ergebnis der Operation in *RES*, während *A* und *B* unverändert bleiben. Bei der Operation *ADD A TO B.* steht das Ergebnis hingegen über *B* zur Verfügung (Abb. 8.1). Dieses Feld wurde damit verändert.

Hier ist ein noch umfangreicheres Beispiel (*calc2.cbl*). Dabei wird mit einer figürlichen Konstanten eine Trennlinie definiert, um die Ausgaben aufzubereiten (*01 SEP PIC X(20) VALUES ALL "-".*):

```
IDENTIFICATION DIVISION.
  PROGRAM-ID. CALC.
  DATA DIVISION.
    WORKING-STORAGE SECTION.
      01  A     PIC 9(3) VALUE 12.
      01  B     PIC 9(3)  VALUE 5.
      01  C     PIC 9(3)  VALUE 67.
      01  D     PIC 9(3)  VALUE 368.
      01  RES   PIC 9(5).
      01  R     PIC 9(5).
      01  SEP PIC X(20) VALUES ALL "-".
  PROCEDURE DIVISION.
    DISPLAY "Wert von A: ", A.
    DISPLAY "Wert von B: ", B.
    DISPLAY "ADD A TO B GIVING RES."
    ADD A TO B GIVING RES.
    DISPLAY "Wert von RES: ", RES.
    DISPLAY SEP.
    DISPLAY "Wert von C: ", C.
    DISPLAY "ADD A, B TO C GIVING RES."
    ADD A, B TO C GIVING RES.
    DISPLAY "Wert von RES: ", RES.
    DISPLAY SEP.
    DISPLAY "Wert von D: ", D.
    DISPLAY "ADD A B C TO D GIVING RES."
    ADD A B C TO D GIVING RES.
    DISPLAY "Wert von RES: ", RES.
```

```
DISPLAY SEP.
DISPLAY "SUBTRACT A B C FROM D GIVING RES."
SUBTRACT A B C FROM D GIVING RES.
DISPLAY "Wert von RES: ", RES.
DISPLAY SEP.
DISPLAY "A B FROM C D."
SUBTRACT A B FROM C D.
DISPLAY "Neuer Wert von C: ", C.
DISPLAY "Neuer Wert von D: ", D.
DISPLAY SEP.
DISPLAY "Wert von A: ", A.
DISPLAY "Wert von B: ", B.
DISPLAY "MULTIPLY A BY B GIVING RES.".
MULTIPLY A BY B GIVING RES.
DISPLAY "Wert von RES: ", RES.
DISPLAY SEP.
DISPLAY "Wert von A: ", A.
DISPLAY "Wert von B: ", B.
DISPLAY "Wert von C: ", C.
DISPLAY "MULTIPLY A BY B C.".
MULTIPLY A BY B C.
DISPLAY "Neuer Wert von A: ", A.
DISPLAY "Neuer Wert von B: ", B.
DISPLAY "Neuer Wert von C: ", C.
DISPLAY SEP.
DISPLAY "DIVIDE A INTO B".
DIVIDE A INTO B.
DISPLAY "Neuer Wert von A: ", A.
DISPLAY "Neuer Wert von B: ", B.
DISPLAY SEP.
DISPLAY "DIVIDE A BY B GIVING RES REMAINDER R.".
DIVIDE A BY B GIVING RES REMAINDER R.
DISPLAY "Wert von RES: ", RES.
DISPLAY "Wert von R: ", R.
STOP RUN.
END PROGRAM CALC.
```

Versuchen Sie die einzelnen Schritte in dem Beispiel nachzuvollziehen. Betrachten Sie Abb. 8.2 oder führen Sie das Beispiel aus.

Erstellen Sie ein Programm, das mittels der besprochenen mathematischen Verben einen Taschenrechner simuliert. Das Programm soll die Möglichkeit der Addition, Multiplikation, Subtraktion und Division bereitstellen. Das Programm fordert einen Anwender zur Eingabe zweier Zahlen und der gewünschten Rechenoperation auf. Die mathematischen Verben werden je nach gewünschter Rechenoperation aufgerufen werden und das Ergebnis ausgegeben. Eine mögliche Lösung finden Sie im Anhang auf der Seite 78 ff.

```
F:\cobolprojekte\kap8>calc2
Wert von A: 012
Wert von B: 005
ADD A TO B GIVING RES.
Wert von RES: 00017
--------------------
Wert von C: 067
ADD A, B TO C GIVING RES.
Wert von RES: 00084
--------------------
Wert von D: 368
ADD A B C TO D GIVING RES.
Wert von RES: 00452
--------------------
SUBTRACT A B C FROM D GIVING RES.
Wert von RES: 00284
--------------------
A B FROM C D.
Neuer Wert von C: 050
Neuer Wert von D: 351
--------------------
Wert von A: 012
Wert von B: 005
MULTIPLY A BY B GIVING RES.
Wert von RES: 00060
--------------------
Wert von A: 012
Wert von B: 005
Wert von C: 050
MULTIPLY A BY B C.
Neuer Wert von A: 012
Neuer Wert von B: 060
Neuer Wert von C: 600
--------------------
DIVIDE A INTO B
Neuer Wert von A: 012
Neuer Wert von B: 005
--------------------
DIVIDE A BY B GIVING RES REMAINDER R.
Wert von RES: 00002
Wert von R: 00002

F:\cobolprojekte\kap8>_
```

Abb. 8.2: Verschiedene mathematische Operationen

8.2.2 Die Anweisung COMPUTE

Mit COMPUTE kann man die üblichen mathematischen Operatoren verwenden. Das macht die Programmierung von arithmetischen Formeln sogar meist intuitiver als die vorher genannten Verben.

Beispiel:

```
COMPUTE W1 = W1 + W2
COMPUTE W1 = W1 * W2 + W2/4 - 20
END-COMPUTE.
```

Hier ist ein Beispiel *MATHVERBS.cbl*, in dem neben COMPUTE auch noch einmal die anderen mathematischen Verben des Abschnitts davor sowie eine explizite Angabe einer Endanweisung zum Einsatz kommen:

```
IDENTIFICATION DIVISION.
  PROGRAM-ID. MATHVERBS.
DATA DIVISION.
  WORKING-STORAGE SECTION.
  01 DATEN.
    05 WS-NUM1 PIC 9(2) VALUE 8.
    05 WS-NUM2 PIC 9(2) VALUE 3.
      05 WS-NUM3 PIC 9(2).
PROCEDURE DIVISION.
  ADD WS-NUM1 TO WS-NUM2.
  ADD 66 TO WS-NUM2
  END-ADD.
```

```
DISPLAY WS-NUM2.
SUBTRACT WS-NUM1 FROM WS-NUM2
END-SUBTRACT.
DISPLAY WS-NUM2.
MULTIPLY WS-NUM1 BY WS-NUM2
END-MULTIPLY.
DISPLAY WS-NUM2.
DIVIDE WS-NUM1 INTO WS-NUM2
END-DIVIDE.
DISPLAY WS-NUM2.
COMPUTE WS-NUM3 = WS-NUM1 * WS-NUM2
COMPUTE WS-NUM3 = WS-NUM1 * 2
END-COMPUTE.
STOP RUN.
END PROGRAM MATHVERBS.
```

Erstellen Sie ein Programm, das mittels COMPUTE zwei Zahlen addiert, multipliziert, dividiert und subtrahiert. Die Zahlen stehen in zwei Variablen zur Verfügung und sollen die Werte 13 und 5 haben. Die Division von 13 durch den Wert 5 geht nicht ganzzahlig auf. Überlegen Sie, wie Sie den Nachkommateil darstellen können. Wählen Sie geeignet Picture-Klauseln und ggfls. mehrere Variablen. Geben Sie alle Ergebnisse aus. Eine mögliche Lösung finden Sie im Anhang ab der Seite 78.

9 Kontrollstrukturen und der Programmfluss in COBOL

Jede Programmiersprache verfügt über Kontrollstrukturen, um den Programmfluss zu steuern. Diese Kontrollstrukturen sind für Programmiereinsteiger eher anspruchsvoll, wohingegen bereits geringe Programmiererfahrung diese Art der Anweisungen oft trivial erscheinen lassen. Dieses Kapitel behandelt die Kontrollflussanweisungen von COBOL. Dabei unterscheidet man – wie bei fast allen Programmiersprachen - zwischen

- Entscheidungsstrukturen,
- Schleifen (Iterationsanweisungen) und
- Sprunganweisungen.

Entscheidungsstrukturen suchen einen Programmfluss aus einer oder mehreren Alternativen. Die Alternative kann wahr oder falsch (TRUE oder FALSE) sein. Bedingungen können miteinander verknüpft werden. Dazu gibt es die folgenden Schlüsselworte bzw. Operatoren:

- AND
- OR

Dabei gilt es Klammerregeln zu beachten.

Schleifen wiederholen einfach Anweisungen.

Sprunganweisungen verzweigen gezielt zu einer anderen Stelle im Programm.

9.1 Entscheidungsanweisungen

Entscheidungsstrukturen suchen wie gesagt einen Programmfluss aus einer oder mehreren Alternativen. Dazu gibt es in COBOL ein paar Anweisungen, die Sie genauso oder ähnlich in allen höheren Programmiersprachen haben.

9.1.1 Die IF-Bedingung in COBOL

Die IF-Bedingung ist eine der elementarsten Entscheidungsstrukturen nahezu jeder Programmiersprache. Selbst die Syntax ist in den meisten Programmiersprachen gleich oder zumindest ähnlich. In COBOL ist zu beachten, dass IF oft in Kombination mit THEN verwendet wird, worauf neuere Programmiersprachen weitgehend verzichten. Allerdings kann man in COBOL auch darauf verzichten, was eine unangenehme Inkonsequenz in COBOL darstellt. Wie man sich auch entscheidet – man sollte konsequent arbeiten.

Ebenso ist wichtig, dass die IF-Anweisung meist mit END-IF beendet wird. Aber auch darauf kann man verzichten, wenn man nur eine Anweisung bedingt ausführen möchte.

Ansonsten gibt es aber die zu erwartenden Varianten, einschließlich des ELSE-Zweigs, auch in COBOL. Anhand konkreter Beispiele lassen sich die Besonderheiten recht gut erkennen, wobei die grundsätzlichen Dinge so offensichtlich sein sollten, dass weitere Erklärungen nicht notwendig sein dürften.

Beispiel (*if1.cbl* – mit verschachtelten IF-Anweisungen):

```
IDENTIFICATION DIVISION.
  PROGRAM-ID. ENTSCHEIDUNG.
DATA DIVISION.
  WORKING-STORAGE SECTION.
    01 WS-NUM1 PIC 9(9).
    01 WS-NUM2 PIC 9(9).
    01 WS-NUM3 PIC 9(5).
```

```
    01 WS-NUM4 PIC 9(6).
PROCEDURE DIVISION.
  MOVE 25 TO WS-NUM1 WS-NUM3.
  MOVE 15 TO WS-NUM2 WS-NUM4.
  IF WS-NUM1 > WS-NUM2 THEN
    DISPLAY 'In IF-BLOCK 1'
    IF WS-NUM3 = WS-NUM4 THEN
      DISPLAY 'In dem inneren IF-BLOCK 2'
    ELSE
      DISPLAY 'In dem inneren ELSE-BLOCK'
    END-IF
  ELSE
    DISPLAY 'In dem ELSE-BLOCK aussen'
  END-IF.
  STOP RUN.
  END PROGRAM ENTSCHEIDUNG.
```

Beispiel (*if2.cbl* – mit symbolischen Vergleichsverben):

```
IDENTIFICATION DIVISION.
  PROGRAM-ID. ENTSCHEIDUNG.
DATA DIVISION.
  WORKING-STORAGE SECTION.
    01 WS-NUM1 PIC 9(9).
    01 WS-NUM2 PIC 9(9).
PROCEDURE DIVISION.
  MOVE 25 TO WS-NUM1.
  MOVE 15 TO WS-NUM2.
  IF WS-NUM1 IS GREATER THAN OR EQUAL TO WS-NUM2 THEN
    DISPLAY 'WS-NUM1 > WS-NUM2'
  ELSE
    DISPLAY 'WS-NUM1 < WS-NUM2'
  END-IF.
  STOP RUN.
  END PROGRAM ENTSCHEIDUNG.
```

Beispiel (*if3.cbl* – mit figürlichen Konstanten):

```
IDENTIFICATION DIVISION.
  PROGRAM-ID. ENTSCHEIDUNG.
DATA DIVISION.
  WORKING-STORAGE SECTION.
    01 WS-NUM1 PIC S9(9) VALUE -123.
    01 WS-NUM2 PIC S9(9) VALUE 123.
PROCEDURE DIVISION.
  IF WS-NUM1 IS POSITIVE THEN
    DISPLAY 'WS-NUM1 POSITIV'.
  IF WS-NUM1 IS NEGATIVE THEN
    DISPLAY 'WS-NUM1 NEGATIV'.
  IF WS-NUM1 IS ZERO THEN
    DISPLAY 'WS-NUM1 0'.
  IF WS-NUM2 IS POSITIVE THEN
    DISPLAY 'WS-NUM2 POSITIV'.
  STOP RUN.
  END PROGRAM ENTSCHEIDUNG.
```

Beispiel (*if4.cbl* – mit weiteren figürlichen Konstanten):

```
IDENTIFICATION DIVISION.
  PROGRAM-ID. ENTSCHEIDUNG.
DATA DIVISION.
  WORKING-STORAGE SECTION.
    01 WS-NUM1 PIC X(9) VALUE "abc".
    01 WS-NUM2 PIC S9(9) VALUE 123.
PROCEDURE DIVISION.
  IF WS-NUM1 IS ALPHABETIC THEN
    DISPLAY 'WS-NUM1 ALPHABETISCH'.
  IF WS-NUM1 IS NUMERIC THEN
    DISPLAY 'WS-NUM1 NUMERISCH'.
  IF WS-NUM2 IS NUMERIC THEN
    DISPLAY 'WS-NUM2 NUMERISCH'.
  STOP RUN.
  END PROGRAM ENTSCHEIDUNG.
```

Beispiel (*if5.cbl* – mit weiteren figürlichen Konstanten und Negation):

```
IDENTIFICATION DIVISION.
  PROGRAM-ID. ENTSCHEIDUNG.
DATA DIVISION.
  WORKING-STORAGE SECTION.
    01 WS-NUM1 PIC 9(2) VALUE 20.
    01 WS-NUM2 PIC 9(9) VALUE 25.
PROCEDURE DIVISION.
  IF NOT WS-NUM1 IS LESS THAN WS-NUM2 THEN
    DISPLAY 'IF-BLOCK'
  ELSE
    DISPLAY 'ELSE-BLOCK'
  END-IF.
  STOP RUN.
  END PROGRAM ENTSCHEIDUNG.
```

Beispiel (*if6.cbl* – mit Verknüpfung von Bedingungen):

```
IDENTIFICATION DIVISION.
  PROGRAM-ID. ENTSCHEIDUNG.
DATA DIVISION.
  WORKING-STORAGE SECTION.
    01 WS-NUM1 PIC 9(2) VALUE 20.
    01 WS-NUM2 PIC 9(2) VALUE 25.
    01 WS-NUM3 PIC 9(2) VALUE 20.
PROCEDURE DIVISION.
  IF WS-NUM1 IS LESS THAN WS-NUM2 AND WS-NUM1=WS-NUM3 THEN
    DISPLAY 'Beide Bedingungen OK'
  ELSE
    DISPLAY 'Eine oder beide Bedingungen nicht OK'
  END-IF.
  IF WS-NUM1 IS GREATER THAN WS-NUM2 OR WS-NUM1=WS-NUM3 THEN
    DISPLAY 'Mindestens eine der Bedingungen OK'
  ELSE
    DISPLAY 'Keine der Bedingungen OK'
  END-IF.
  IF WS-NUM1 IS GREATER THAN WS-NUM2 OR WS-NUM1=WS-NUM2 THEN
    DISPLAY 'Mindestens eine der Bedingungen OK'
  ELSE
    DISPLAY 'Keine der Bedingungen OK'
```

```
END-IF.
STOP RUN.
END PROGRAM ENTSCHEIDUNG.
```

9.1.2 Vergleiche in COBOL

In vielen Situationen (auch in den IF-Anweisungen eben) muss man Variablen mit anderen Variablen oder auch mit Literalen vergleichen. Dabei sind in COBOL ein paar Besonderheiten zu beachten, die sich insbesondere beim Vergleich von Texten ergeben.

9.1.2.1 Vergleichsbedingungen in COBOL - Vergleich zweier Operanden.

- Algebraische Vergleiche von numerischen Feldern sind unabhängig von ihrer Größe und Nutzungsklausel.
- Wenn zwei nicht-numerische Operanden gleicher Größe verglichen werden, dann Zeichen für Zeichen von links.
- Wenn zwei nicht-numerische Operanden ungleicher Größe verglichen werden, werden an das kürzere Datenelement Leerzeichen angefügt.
- Zeichen besitzen eine Repräsentation in der ANSI-Tabelle. Das erlaubt auch den Vergleich von Buchstaben und Texten auf „Größer“ oder „Kleiner“. Auch unter Berücksichtigung von Groß- und Kleinschreibung. So ist der Buchstabe A an der Stelle 65 zu finden, während a an der Stelle 97 steht.

Erstellen Sie das nachfolgende Programm. Versuchen Sie die Ausgaben zu verstehen. Die Lösung finden Sie im Anhang auf der Seite 78 ff.

```
IDENTIFICATION DIVISION.
  PROGRAM-ID. VERGLEICH.
DATA DIVISION.
  WORKING-STORAGE SECTION.
  01 WS-NUM1 PIC 9(2) VALUE 5.
  01 WS-NUM2 PIC 9(3) VALUE 5.
  01 WS-FELD1 PIC X(4) VALUE "Ralf".
  01 WS-FELD2 PIC X(5) VALUE "Ralph".
  01 WS-FELD3 PIC X(5) VALUE "ralph".
PROCEDURE DIVISION.
  IF WS-NUM1 = WS-NUM2 THEN
    DISPLAY "Zahlen sind gleich.".
  IF WS-FELD1 = WS-FELD2 THEN
   DISPLAY
  "Textfelder WS-FELD1 und WS-FELD2 sind gleich.".
  IF WS-FELD1 > WS-FELD2 THEN
    DISPLAY "Textfeld WS-FELD1 > WS-FELD2.".
  IF WS-FELD2 = WS-FELD3 THEN
    DISPLAY
    "Textfelder WS-FELD2 und WS-FELD3 sind gleich.".
  IF WS-FELD3 > WS-FELD2 THEN
    DISPLAY "Textfeld WS-FELD3 > Textfeld WS-FELD2".
  STOP RUN.
  END PROGRAM VERGLEICH.
```

Erstellen Sie ein Programm, das vom Anwender eine Benutzerkennung und ein Passwort entgegen nimmt. Das Passwort soll mit einem vorgegebenen Text und das Passwort mit einem anderen Text übereinstimmen. Es handelt sich also um eine typische Zugangsüberprüfung. Arbeiten Sie in beiden Fällen mit einem 4-stelligen Feld. Je nach Eingabe werden unterschiedliche Ausgaben ange-

zeigt. Eine mögliche Lösung finden Sie im Anhang auf der Seite 78 ff.

9.1.2.2 Das EVALUATE -Verb in COBOL

Eine Sonderform der IF-Anweisung erlaubt die Angabe von **Wertebereichen**, in denen der Testwert liegen muss. Je nach Zugehörigkeit der Testvariablen zu einem Wertebereich können dann gezielt Anweisungen ausgeführt werden.

Schematisch sieht das so aus:

```
EVALUATE TRUE
WHEN bedingung-1
anw-11 [... anw-1n]
WHEN bedingung-2
anw-21 [... anw-2n]
...
WHEN OTHER
anw-91 [... anw-9n]
END-EVALUATE
```

Entweder sollen die beschriebenen Bedingungen erfüllt sein (TRUE) oder explizit nicht (FALSE).

Erstellen Sie das einfache Beispiel *EVAL.cbl*, kompilieren Sie es und führen Sie es aus.

```
IDENTIFICATION DIVISION.
  PROGRAM-ID. EVAL.
DATA DIVISION.
  WORKING-STORAGE SECTION.
     01 WS-A PIC S9(3) VALUE 2.
PROCEDURE DIVISION.
    EVALUATE TRUE
    WHEN WS-A > 2
      DISPLAY "WA-A > 2"
    WHEN WS-A <= 2
      DISPLAY "WA-A <= 2"
    WHEN OTHER
      DISPLAY "Andere Situation"
    END-EVALUATE.
  STOP RUN.
  END PROGRAM EVAL.
```

Hier ist noch ein umfangreiches Beispiel (*bewerten.cbl*).

Versuchen Sie das nachfolgende Listing nachzuvollziehen.

```
IDENTIFICATION DIVISION.
  PROGRAM-ID. ENTSCHEIDUNG.
DATA DIVISION.
  WORKING-STORAGE SECTION.
  01 WS-A PIC S9(3) VALUE 3.
PROCEDURE DIVISION.
  EVALUATE TRUE
  WHEN WS-A > 2
    DISPLAY 'WS-A > 2'
  WHEN WS-A < 0
```

```
    DISPLAY 'WS-A < 0'
  WHEN OTHER
    DISPLAY 'Sonstiges'
  END-EVALUATE.
  EVALUATE FALSE
  WHEN WS-A > 2
    DISPLAY 'WS-A > 2'
  WHEN WS-A < 0
    DISPLAY 'WS-A < 0'
  WHEN OTHER
   DISPLAY 'Sonstiges'
  END-EVALUATE.
  MOVE -11 TO WS-A.
  EVALUATE TRUE
  WHEN WS-A > 2
    DISPLAY 'WS-A > 2'
  WHEN WS-A < 0
    DISPLAY 'WS-A < 0'
  WHEN OTHER
    DISPLAY 'Sonstiges'
  END-EVALUATE.
  MOVE 1 TO WS-A.
  EVALUATE TRUE
  WHEN WS-A > 2
    DISPLAY 'WS-A > 2'
  WHEN WS-A < 0
    DISPLAY 'WS-A < 0'
  WHEN OTHER
    DISPLAY 'Sonstiges'
  END-EVALUATE.
  STOP RUN.
  END PROGRAM ENTSCHEIDUNG.
```

9.1.3 Bereichsangaben in COBOL

Eine Sonderform der IF-Anweisung erlaubt die Angabe von **Wertebereiche** (Range) mit der Anweisung THRU, in denen der Testwert liegen muss. Je nach Zugehörigkeit der Testvariablen zu einem Wertebereich können dann gezielt Anweisungen ausgeführt werden.

Erstellen Sie das Beispiel (*BEREICHSTEST.cbl*), kompilieren Sie es und führen es aus.

```
IDENTIFICATION DIVISION.
  PROGRAM-ID. BEREICHSTEST.
DATA DIVISION.
  WORKING-STORAGE SECTION.
    01 WS-NUM PIC 9(3) VALUE 15.
    88 JA VALUES ARE 018 THRU 100.
    88 NEIN VALUES ARE 000 THRU 17.
PROCEDURE DIVISION.
  IF JA
    DISPLAY 'Im JA-Bereich'.
  IF NEIN
    DISPLAY 'Im NEIN-Bereich'.
    MOVE 21 TO WS-NUM.
  IF JA
    DISPLAY 'Im JA-Bereich'.
  IF NEIN
```

```
    DISPLAY 'Im NEIN-Bereich'.
STOP RUN.
END PROGRAM BEREICHSTEST.
```

9.2 Sprunganweisungen

Mittels Sprunganweisungen kann man den Programmfluss gezielt an eine bestimmte Stelle weiterreichen, der bei einer sequenziellen Abarbeitung des Quellcodes zu dem Zeitpunkt nicht erreicht würde.

9.2.1 Die GO TO-Anweisung und benannte Absätze

Die GO TO-Anweisung ist über die Jahre in Verruf geraten (Stichwort: Spagetti-Code) und zählt in modernen Programmiersprachen zu den Anweisungen, die entweder explizit verhindert werden oder zumindest vermieden werden sollten. In COBOL ist sie jedoch elementar für die Steuerung des Programmflusses. Mit dem Schlüsselwort GO, dem durch ein Leerzeichen getrennt TO folgt (GO TO), können Sie gezielt benannte Absätze im Quellcode anspringen. Das macht man in typischen COBOL-Programmen recht häufig.

Beispiel:

```
IDENTIFICATION DIVISION.
  PROGRAM-ID. GOTO.
DATA DIVISION.
  WORKING-STORAGE SECTION.
    01 WS-NUM1 PIC 9(2) VALUE 5.
    01 WS-NUM2 PIC 9(2) VALUE 6.
PROCEDURE DIVISION.
  DISPLAY "Gleich passiert das ganz schlimme GO TO"
  IF WS-NUM2 < WS-NUM1 THEN
     GO TO 0100-SPRUNGMARKE.
  DISPLAY "Das wuerde so gerne noch ausgegeben".
0100-SPRUNGMARKE.
  DISPLAY "Aber nur der Text kommt noch".
  STOP RUN.
  END PROGRAM GOTO.
```

Leider ist diese GO TO-Anweisung nicht zu Unrecht in Verruf geraten. Die nachfolgenden Listings sollen Ihnen drastisch deutlich machen, was man an „Unsinn“ damit anstellen kann.

Vermeiden Sie die GO TO-Anweisung! Das ist einer der wichtigsten Tipps im ganzen Kurs. Aber in den bestehenden Codes werden Sie stattdessen eine sehr häufige Verwendung finden und diese Programme müssen gepflegt werden. Deshalb sollten Sie die Anwendung wirklich gut verstehen.

Zuerst sehen Sie, wie man „ganz einfach“ mit GO TO eine Art „Endlosschleife“[2] erstellen kann.

```
IDENTIFICATION DIVISION.
  PROGRAM-ID. OVER.
PROCEDURE DIVISION.
  0100-START.
    GO TO 0100-START.
  STOP RUN.
  END PROGRAM OVER.
```

Die nächsten Beispiele sind nicht wirklich so nutzlos bzw. schlecht wie dieses drastische Listing, aber dennoch zunehmend unübersichtlich.

[2] Es ist natürlich keine **Schleife**!

Versuchen Sie die Arbeitsweise des folgenden Listings nachzuvollziehen.

```
IDENTIFICATION DIVISION.
  PROGRAM-ID. SPAGETTI.
DATA DIVISION.
  WORKING-STORAGE SECTION.
    01 WS-A PIC S9(3) VALUE 20.
    88 JA VALUES ARE 010 THRU 65.
    01 WS-COUNTER PIC 9(1) VALUE 0.
PROCEDURE DIVISION.
  GO TO 0100-MARKER.
  0090-MARKER.
  MOVE -20 TO WS-A.
  GO TO 0100-MARKER.
  0095-MARKER.
  MOVE 70 TO WS-A.
  GO TO 0100-MARKER.
  0100-MARKER.
  ADD 1 TO WS-COUNTER.
  IF JA
    DISPLAY "OK"
  ELSE
    DISPLAY "Nicht OK"
  END-IF.
  IF (WS-COUNTER = 1)
    GO TO 0090-MARKER
  END-IF.
  IF (WS-COUNTER = 2)
    GO TO 0095-MARKER
  END-IF.
  STOP RUN.
  END PROGRAM SPAGETTI.
```

9.2.2 Bedingtes GO TO in COBOL

Von der an sich schon „gefährlichen“ Grundversion der GO TO-Anweisung gibt es in COBOL sogar noch eine bedingte Variante, bei der man in Anhängigkeit vom Wert einer Variablen (mit DEPENDING ON) eine von mehreren Sprungmarken anspringt.

Beispiel:

```
IDENTIFICATION DIVISION.
  PROGRAM-ID. SPAGETTI2.
DATA DIVISION.
  WORKING-STORAGE SECTION.
   01 WS-NUM1 PIC 99 VALUE 2.
PROCEDURE DIVISION.
  0100-START.
  DISPLAY "Programm ist gestartet.".
  GO TO 0110-MARKER 0130-MARKER 0120-MARKER
    DEPENDING ON WS-NUM1.
  0110-MARKER.
  DISPLAY "Im Absatz 0110-MARKER.".
  0120-MARKER.
  DISPLAY "Im Absatz 0120-MARKER.".
  0130-MARKER.
  DISPLAY "Im Absatz 0130-MARKER.".
  STOP RUN.
```

```
END PROGRAM SPAGETTI2.
```

Das Beispiel kann man vielleicht noch nachvollziehen, aber wenn das Listing wirklich länger wird, wird es teils grenzwertig. Das folgende Listing zeigt einen Taschenrechner, dessen Berechnungsschritte in Absätze ausgelagert werden. Diese werden dann mit GO TO angesprungen.

Versuchen Sie die Arbeitsweise des folgenden Listings *SPAGETTITASCHENRECHNER. cbl* nachzuvollziehen. Sie können das als eine Art „Härtetest" sehen, wie gut Sie mittlerweile COBOL verstehen.

```
IDENTIFICATION DIVISION.
  PROGRAM-ID. SPAGETTITASCHENRECHNER.
DATA DIVISION.
  WORKING-STORAGE SECTION.
  01 DATEN.
  05 WS-NUM1 PIC 9(2) VALUE 0.
  05 WS-NUM2 PIC 9(2) VALUE 0.
  05 WS-OP PIC X(1) VALUE NULL.
PROCEDURE DIVISION.

  0100-START.
  DISPLAY "Geben Sie die erste Zahl ein".
  ACCEPT WS-NUM1.
  DISPLAY "Geben Sie die zweite Zahl ein".
  ACCEPT WS-NUM2.
  DISPLAY "Geben Sie die Operation ein".
  ACCEPT WS-OP.
  IF(WS-OP="+")
    MOVE 1 TO WS-OP
  ELSE IF(WS-OP="*")
    MOVE 2 TO WS-OP
  ELSE IF(WS-OP="/")
    MOVE 3 TO WS-OP
  ELSE IF(WS-OP="-")
    MOVE 4 TO WS-OP
  ELSE
   GO TO 0150-MARKER
  END-IF.
  GO TO 0110-MARKER 0120-MARKER 0130-MARKER 0140-MARKER
  DEPENDING ON WS-OP.

  0105-MARKER.
  DISPLAY WS-NUM2.
  GO TO 0100-START.

  0110-MARKER.
    ADD WS-NUM1 TO WS-NUM2
    END-ADD.
    GO TO 0105-MARKER.

  0120-MARKER.
    MULTIPLY WS-NUM1 BY WS-NUM2.
    GO TO 0105-MARKER.

  0130-MARKER.
    DIVIDE WS-NUM1 INTO WS-NUM2.
    GO TO 0105-MARKER.
```

```
0140-MARKER.
  SUBTRACT WS-NUM1 FROM WS-NUM2.
  GO TO 0105-MARKER.

0150-MARKER.
STOP RUN.
END PROGRAM SPAGETTITASCHENRECHNER.
```

9.2.3 Einfaches PERFORM in COBOL

Mit dem PERFORM-Verb kann man gezielt bestimmten Code des COBOL-Programms ausführen. Insbesondere durch Angabe von Absatzbezeichnern kann man damit Code, der an anderer Stelle notiert ist, aufrufen – sogar mehrfach, was die Variante eines Unterprogramms bzw. einer Funktion darstellt und erste Ansätze modularer Programmierung erahnen lässt. Man kann hier auch von einer Art Sprunganweisung sprechen, wobei im Unterschied zu GO TO der Programmfluss nach dem Abarbeiten der PERFORM-Anweisungen wieder zum Aufrufpunkt zurückkehrt und mit dem nächsten Schritt weiter macht!

Nutzen Sie – wenn möglich – PERFORM anstelle von GO TO.

Erstellen Sie das Beispiel (*PERF1.cbl*), kompilieren Sie es und führen es aus.

```
IDENTIFICATION DIVISION.
  PROGRAM-ID. PERF1.
DATA DIVISION.
PROCEDURE DIVISION.
  0100-START.
  DISPLAY "Programm ist gestartet".
  PERFORM 0110-MARKER THRU 0130-MARKER.
  DISPLAY "Programmschritt 2".
  PERFORM 0140-MARKER.
  0110-MARKER.
  DISPLAY "Unterabsatz 0110-MARKER".
  0120-MARKER.
  DISPLAY "Unterabsatz 0120-MARKER".
  0130-MARKER.
  DISPLAY "Unterabsatz 0130-MARKER".
  0140-MARKER.
  STOP RUN.
  END PROGRAM PERF1.
```

Die PERFORM-Anweisung ist wirklich wichtig und mächtig. Deshalb folgt hier noch ein Beispiel zur Verdeutlichung der Arbeitsweise.

Versuchen Sie die Arbeitsweise des folgenden Listings nachzuvollziehen.

```
IDENTIFICATION DIVISION.
  PROGRAM-ID. AUSFUEHREN.
DATA DIVISION.
  WORKING-STORAGE SECTION.
  01 WS-NUM1 PIC 9(2) VALUE 1.
PROCEDURE DIVISION.
  0100-START.
  PERFORM DISPLAY "Im 1. Absatz"
  END-PERFORM.
  0110-ABSATZ.
  PERFORM
```

```
  MOVE 2 TO WS-NUM1
  DISPLAY "In Absatz 0110-ABSATZ"
END-PERFORM.
PERFORM 0130-ABSATZ THRU 0150-ABSATZ.
0120-ABSATZ.
PERFORM
  DISPLAY "In Absatz 0120-ABSATZ"
  STOP RUN
END-PERFORM.
0130-ABSATZ.
DISPLAY WS-NUM1.
DISPLAY "In Absatz 0130-ABSATZ".
MOVE 3 TO WS-NUM1.
0140-ABSATZ.
DISPLAY WS-NUM1.
DISPLAY "In Absatz 0140-ABSATZ".
MOVE 4 TO WS-NUM1.
0150-ABSATZ.
DISPLAY WS-NUM1.
DISPLAY "In Absatz 0150-ABSATZ".
END PROGRAM AUSFUEHREN.
```

9.3 Schleifen

Iterationsanweisungen sind immer dann sinnvoll, wenn Anweisungen wiederholt werden müssen. Wie allgemein üblich gibt es in COBOL annehmende (fußgesteuerte) und ablehnende (kopfgesteuerte) Schleifen.

9.3.1 Ausführen Bis – Schleifen in COBOL

Mit dem PERFORM-Verb kann man in COBOL sowohl kopf- (ablehnende) als auch fußgesteuerte (annehmende) Schleifen (Iterationen) erstellen. In Kombination mit einer Bedingung über UNTIL können Sie festlegen, wie oft eine Schleife wiederholt wird. Mit VARYING kann eine **Zählvariable** angegeben werden.

Schematisch sieht das bei einer **abweisenden** Schleife dann so aus:

```
PERFORM [WITH TEST BEFORE]
UNTIL   Bedingung
anweisung(en)
END-PERFORM
```

Wenn Sie kein WITH TEST BEFORE angeben, ist das die Vorgabe.

Das wäre eine **annehmende** Schleife

```
PERFORM WITH TEST AFTER
UNTIL   Bedingung
anweisung(en)
END-PERFORM
```

So könnte ein schematisches Beispiel aussehen:

```
Anweisung-1
PERFORM WITH TEST AFTER
VARYING I1 FROM 1 BY 1
UNTIL   I1 > 10
```

anweisung(en)

END-PERFORM

Anweisung-2

Das wäre ein vollständiges Beispiel für eine annehmende Schleife:

Erstellen Sie das Beispiel (*ANNEHMENDESCHLEIFE.cbl*), kompilieren Sie es und führen es aus.

```
IDENTIFICATION DIVISION.
  PROGRAM-ID. ANNEHMENDESCHLEIFE.
DATA DIVISION.
  WORKING-STORAGE SECTION.
  01 I PIC 99 VALUE 0.
PROCEDURE DIVISION.
  PERFORM WITH TEST BEFORE
  VARYING I FROM 0 BY 1
  UNTIL I > 10
    DISPLAY I
  END-PERFORM.
  STOP RUN.
  END PROGRAM ANNEHMENDESCHLEIFE.
```

Bei Zählschleifen wird die Anzahl der Schleifendurchläufe fest vorgegeben. Es gibt mehrere Varianten, die alle das PERFORM-Verb verwenden. Man wird oft den Start- und Endwert sowie die Schrittweite für die Zählvariable (positiv wie negativ) angeben – in Kombination mit VARYING.

In dem nachfolgenden Listing wird die Zählvariable rückwärts gezählt.

Versuchen Sie die Arbeitsweise der folgenden Listings nachzuvollziehen.

```
IDENTIFICATION DIVISION.
  PROGRAM-ID. COUNTDOWN.
DATA DIVISION.
WORKING-STORAGE SECTION.
  01 I PIC 99 VALUE 10.
PROCEDURE DIVISION.
  PERFORM WITH TEST AFTER
  VARYING I FROM 10 BY -1
  UNTIL I = 0
    DISPLAY I
  END-PERFORM.
  DISPLAY "Bumm".
  STOP RUN.
  END PROGRAM COUNTDOWN.
```

Das kann man auch so variieren, dass in der Schleife mit einem weiteren PERFORM eine Section gezielt angesprungen wird. Beispiel:

```
IDENTIFICATION DIVISION.
  PROGRAM-ID. COUNTDOWN2.
DATA DIVISION.
  WORKING-STORAGE SECTION.
  01 I PIC 99 VALUE 10.
PROCEDURE DIVISION.
  PERFORM WITH TEST AFTER
  VARYING I FROM 10 BY -1
  UNTIL I = 0
    PERFORM 0100-MARKER
  END-PERFORM.
```

```
  DISPLAY "Bumm".
  0100-MARKER.
  DISPLAY I.
  0110-MARKER.
  STOP RUN.
  END PROGRAM COUNTDOWN2.
```

Hier sehen Sie noch ein etwas komplexeres Beispiel (*perform2.cbl*):

```
IDENTIFICATION DIVISION.
  PROGRAM-ID. SCHLEIFEN.
DATA DIVISION.
  WORKING-STORAGE SECTION.
  01 WS-NUM1 PIC 9(2) VALUE 0.
  01 WS-NUM2 PIC 9(2) VALUE 0.
PROCEDURE DIVISION.
  0100-PARA.
  DISPLAY "Schleife 1 mit TEST AFTER".
  PERFORM 0110-PARA WITH TEST AFTER UNTIL WS-NUM1 > 3.
  DISPLAY "Schleife 2 mit TEST BEFORE".
  PERFORM 0120-PARA WITH TEST BEFORE UNTIL WS-NUM2 > 3.
  STOP RUN.
  0110-PARA.
  DISPLAY 'In 0110-PARA: Wert von WS-NUM1 : 'WS-NUM1.
  ADD 1 TO WS-NUM1.
  0120-PARA.
  DISPLAY 'In 0120-PARA: Wert von WS-NUM2 : 'WS-NUM2.
  ADD 1 TO WS-NUM2.
  END PROGRAM SCHLEIFEN.
```

Mit TIMES kombiniert, gibt man bei Schleifen die Anzahl der gewünschten Schleifendurchläufe an. Mit der Anweisung TIMES kann man auch eine Anzahl an Wiederholungen direkt angeben. Das ist insbesondere mit dem Aufruf von benannten Abschnitten oft zu finden.

Erstellen Sie das folgende Beispiel (*COUNTDOWN2.cbl*), kompilieren Sie es und führen es aus.

```
IDENTIFICATION DIVISION.
  PROGRAM-ID. COUNTDOWN2.
DATA DIVISION.
  WORKING-STORAGE SECTION.
  01 I PIC 99 VALUE 10.
PROCEDURE DIVISION.
  PERFORM 0100-MARKER 9 TIMES.
  DISPLAY "Bumm".
0100-MARKER.
  SUBTRACT 1 FROM I.
  DISPLAY I.
0110-MARKER.
  STOP RUN.
  END PROGRAM COUNTDOWN2.
```

Und hier ist zur weiteren Verdeutlichung ein zusätzliches Beispiel mit Zählvariablen und der Anweisung TIMES (*perform3.cbl*).

Versuchen Sie die Arbeitsweise der folgenden Listings nachzuvollziehen.

```
IDENTIFICATION DIVISION.
  PROGRAM-ID. SCHLEIFEN.
DATA DIVISION.
```

```
  WORKING-STORAGE SECTION.
  01 WS-NUM1 PIC 9(2) VALUE 0.
  01 WSNUM PIC 9(2) VALUE 0.
PROCEDURE DIVISION.
  0100-PARA.
  PERFORM 0110-PARA 5 TIMES.
  PERFORM ITER VARYING WSNUM FROM 1 BY 1 UNTIL
    WSNUM = 5.
  PERFORM ITER VARYING WSNUM FROM 10 BY -2 UNTIL
    WSNUM = 2.
  STOP RUN.
  0110-PARA.
  DISPLAY 'In 0110-PARA: Wert von WS-NUM1 : 'WS-NUM1.
  ADD 1 TO WS-NUM1.
  ITER.
  DISPLAY 'In Absatz ITER: Wert von WSNUM : ' WSNUM.
  END PROGRAM SCHLEIFEN.
```

9.3.2 Sections mit PERFORM anspringen

Man unterteilt ein komplexeres COBOL-Programm so gut wie immer in Sektionen. Aber warum und weshalb? Der Vorzug der strukturierten Programmierung ist es, dass Aufgaben in kleine Module aufzuteilen sind. Und diese Umsetzung erfolgt in COBOL mit Sections.

Der Aufruf erfolgt mit *PERFORM section-name*, was wir oben schon gesehen haben. Das führt zu einer Verzweigung zur Section. Dort erfolgt die Ausführung aller Anweisungen in der Section.

Von dort kann erfolgt ein Rücksprung. Nun kann es sinnvoll sein, dass man einzig oder letzte Anweisung in einer Section CONTINUE notiert. Das ist das Schema für einen einfachen Aufruf:

```
. . .
PERFORM VORLAUF
. . .
VORLAUF SECTION.
[VORLAUF-01.]
anweisung-1
anweisung-2
CONTINUE.
[VORLAUF-EXIT.]
```

Damit man dem Programmfluss die Möglichkeit vor der nächsten Sektion zurückzukehren. Das kann dann nützlich sein, wenn ansonsten das Programm enden würde.

10 Erweiterte COBOL-Syntax

In diesem Kapitel behandeln wir wichtige erweiterte COBOL-Techniken. Das umfasst den Aufruf von Unterprogrammen, eingebaute Funktionen in COBOL, den Umgang mit Strings sowie den Einsatz von Tabellen bzw. Arrays.

10.1 Externe Unterprogramme aufrufen in COBOL

Bei umfangreicheren Programmen muss man in der Regel eine Modularisierung vornehmen. Selbst interne Unterprogramme, die mit PERFORM ausgeführt werden, oder GO TO-Anweisungen sind da in der Regel nicht mehr ausreichend. Mit dem Verb CALL können Sie - in Verbindung mit END-CALL - externe Unterprogramme aufrufen, die jedoch in besonders kompilierter Form vorliegen (als Module) und einen spezifischen Aufbau haben müssen. An solche Unterprogramme kann man Daten per Call-by-Reference oder Call-by-Value übergeben.

Die schematische Syntax bei einem sogenannten dynamischen Aufruf sieht so aus:

```
CALL pgm-name USING var-1 ... var-n
[END-CALL]
```

Die schematische Syntax bei einem sogenannten statischen Aufruf ist die:

```
CALL 'pgm-name' USING var-1 ... var-n
[END-CALL]
```

10.1.1 Dynamischer versus statischer Aufruf

Bei einem **statischen Aufruf** wird ein aufgerufenes Modul zum Aufrufer fest gelinkt. Das hat die Konsequenzen, dass bei einer Änderung des Moduls alle Aufrufer das Modul erneut linken. Der Vorteil ist, dass der Aufruf etwas schneller ist.

Der **dynamische Aufruf** erlaubt den flexiblen Umgang mit Änderungen in Moduln.

Nicht jeder Compiler erlaubt beide Varianten des Aufrufs. Das muss ggfls. in der Dokumentation der konkreten Plattform eruiert werden.

10.1.2 Der Aufbau von Unterprogramme

Interne Unterprogramme in COBOL sind vom Aufbau, der Logik zum Aufruf wie externe Programme zu sehen, aber Bestandteile des Gesamtprogramms.

Es gibt für die Verwendung von Unterprogrammen einige Regeln, die sowohl die Unterprogramme als auch das Hauptprogramm betreffen:

- Jedes Programm und Unterprogramm hat eine PROGRAM-ID mit der Angabe eines Programmnamens in der IDENTIFICATION DIVISION
- Programmnamen müssen eindeutig sein.
- Eine CONFIGURATION SECTION gibt es nur im äußersten Programm – dem Hauptprogramm.
- Jedes Programm und Unterprogramm hat eine eigene Anweisung zum Beenden. Im Hauptprogramm finden Sie "END PROGRAM *programmname*.", während im Unterprogramm "EXIT PROGRAM." steht.

- Ein Unterprogramm (inneres Programm) kann nur vom direkten umgebenden Elternprogramm aufgerufen werden. Das Elternprogramm kann das Hauptprogramm selbst, aber auch ein Unterprogramm sein.
- Es gibt globale und lokale Variablen.
- Im Unterprogramm stellt die LINKAGE SECTION einen Bereich zur Verfügung, über den Funktionalitäten dem Hauptprogramm zur Verfügung gestellt werden können.
- Umgekehrt können Daten darüber an das Unterprogramm übergeben werden. Das macht man mit der USING-Anweisung bei der PROCEDURE DIVISION.

Achten Sie darauf, dass Unterprogramme als **Module** kompiliert werden müssen (also **ohne** den Parameter -x). Nur das Hauptprogramm wird als eigenständig lauffähiges Programm kompiliert.

Beispiel (das Hauptprogramm *main.cbl*):

```
IDENTIFICATION DIVISION.
  PROGRAM-ID. MAIN.
DATA DIVISION.
  WORKING-STORAGE SECTION.
  01 WS-STUDENT-ID PIC 9(4) VALUE 1000.
  01 WS-STUDENT-NAME PIC A(15) VALUE 'Herbi'.
PROCEDURE DIVISION.
  DISPLAY 'Im MAIN-Programm'.
  CALL 'UNTERPROGRAMM1' USING WS-STUDENT-ID, WS-STUDENT-NAME.
  DISPLAY 'Student Id : ' WS-STUDENT-ID
  DISPLAY 'Student Name : ' WS-STUDENT-NAME
  DISPLAY 'Wieder direkt im MAIN-Programm'.
  STOP RUN.
  END PROGRAM MAIN.
```

Das aufgerufene Unterprogramm könnte so aussehen (*unterprogramm1.cbl*):

```
IDENTIFICATION DIVISION.
  PROGRAM-ID. UNTERPROGRAMM1.
DATA DIVISION.
  LINKAGE SECTION.
  01 LS-STUDENT-ID PIC 9(4).
  01 LS-STUDENT-NAME PIC A(15).
PROCEDURE DIVISION USING LS-STUDENT-ID, LS-STUDENT-NAME.
  DISPLAY 'Im Unterprogramm'.
  MOVE 1111 TO LS-STUDENT-ID.
  EXIT PROGRAM.
```

Betrachten wir noch ein weiteres Beispiel, das zwei Unterprogramme verwendet. Dabei soll eine Zugangsbeschränkung umgesetzt werden. Das Hauptprogramm sollte einen Anwender zur Eingabe einer Userid und des Passworts auffordern. Sofern diese Daten als „legitimiert" hinterlegt sind, sollte ein bestimmter Inhalt angezeigt werden und andernfalls eine Fehlermeldung (Abb. 10.1). Die Zugangsdaten werden hartkodiert (also im Quellcode) in dem Unterprogramm zur Zugangsverifizierung gespeichert.

Das ist das Hauptprogramm *ZUGANGHAUPTPROGRAMM.cbl*:

```
IDENTIFICATION DIVISION.
  PROGRAM-ID. ZUGANGHAUPTPROGRAMM.
DATA DIVISION.
  WORKING-STORAGE SECTION.
```

```
  01 DATEN.
    05 WS-USERID PIC X(4) VALUES SPACES.
    05 WS-PASSWORT PIC X(4) VALUES SPACES.
PROCEDURE DIVISION.
  CALL 'ZUGANGDATENENTGEGENNAHME' USING WS-USERID WS-PASSWORT.
  CALL 'ZUGANGVERIFIZIERUNG' USING WS-USERID WS-PASSWORT.
  STOP RUN.
  END PROGRAM ZUGANGHAUPTPROGRAMM.
```

Das ist das Unterprogramm *ZUGANGDATENENTGEGENNAHME.cbl*:

```
IDENTIFICATION DIVISION.
  PROGRAM-ID. ZUGANGDATENENTGEGENNAHME.
DATA DIVISION.
   LINKAGE SECTION.
  01 WS-USERID PIC X(4).
  01 WS-PASSWORT PIC X(4).
PROCEDURE DIVISION USING WS-USERID WS-PASSWORT.
  DISPLAY "Geben Sie die USERID ein".
  ACCEPT WS-USERID.
  DISPLAY "Geben Sie das PASSWORT ein".
  ACCEPT WS-PASSWORT.
  EXIT PROGRAM.
```

Und dies ist das Listing des Unterprogramms *ZUGANGVERIFIZIERUNG.cbl*:

```
IDENTIFICATION DIVISION.
  PROGRAM-ID. ZUGANGVERIFIZIERUNG.
DATA DIVISION.
  LINKAGE SECTION.
  01 WS-USERID PIC X(4).
  01 WS-PASSWORT PIC X(4).
PROCEDURE DIVISION USING WS-USERID WS-PASSWORT.
  IF WS-USERID = "otto" AND WS-PASSWORT = "abcd" THEN
    DISPLAY  "Hallo ", WS-USERID
  ELSE
    DISPLAY "Falsche Zugangsdaten "
  END-IF.
  EXIT PROGRAM.
```

```
F:\cobolprojekte\kap10>ZUGANGHAUPTPROGRAMM
Geben Sie die USERID ein
otto
Geben Sie das PASSWORT ein
abcd
Hallo otto

F:\cobolprojekte\kap10>ZUGANGHAUPTPROGRAMM
Geben Sie die USERID ein
dirk
Geben Sie das PASSWORT ein
acde
Falsche Zugangsdaten

F:\cobolprojekte\kap10>
```

Abb. 10.1: Einmal die korrekten und einmal die falschen Daten

Erstellen Sie ein Taschenrechnerprogramm mit den Funktionalitäten der Addition, Multiplikation, Subtraktion und Division. Das Hauptprogramm fordert einen Anwender zur Eingabe zweier Zahlen und der gewünschten Rechenoperation auf. Die Rechenoperationen werden in vier Unterprogramme ausgelagert, die je nach gewünschter Rechenoperation aufgerufen werden. Dort werden jeweils die Rechenoperationen durchgeführt. Das Ergebnis wird wieder im Hauptprogramm ausgegeben. Eine mögliche Lösung finden Sie im Anhang auf der Seite 78 ff.

10.2 Eingebaute COBOL-Funktionen

In COBOL gibt es eine ganze Reihe an vordefinierten Funktionen für die unterschiedlichsten Zwecke. Diese werden **COBOL Intrinsic Functions** oder auch **built-in functions** genannt und bilden das COBOL-API, was sich aber je nach COBOL-Distribution und Plattform unterscheiden kann.

Sehr ungewöhnlich – im Vergleich zu den meisten anderen Programmiersprachen – ist die Art der Verwendung von Funktionen in COBOL. Wenn Sie eine Intrinsic Function **aufrufen** wollen, müssen Sie immer das Wort FUNCTION verwenden, gefolgt von der jeweiligen Funktion sowie aller erforderlichen oder optionalen Argumente, die dann in Klammern gesetzt werden. Gibt es keine Argumente, können die leeren Klammern entfallen, was auch sehr ungewöhnlich im Vergleich zu modernen Programmiersprachen ist.

In anderen Sprachen muss man zum Aufruf einer Funktion in der Regel nur den Namen verwenden, während so ein Schlüsselwort die **Deklaration** einer Funktion einleitet. In COBOL kann man allerdings derzeit keine eigenen Funktionen im klassischen Sinn deklarieren. Für künftige Versionen ist das jedoch geplant.

Beispiele:

```
MOVE FUNCTION UPPER-CASE(STATE-IN) TO STATE-FOR-FILE.
MOVE FUNCTION CURRENT-DATE TO WS-CURRENT-DATE-FIELDS.
COMPUTE WS-RESULT = FUNCTION SQRT (WS-NUMBER)
COMPUTE WS-RESULT = FUNCTION MOD (WS-INTEGER-1, WS-INTEGER-2)
COMPUTE INT-ANS-WS = FUNCTION INTEGER(2123.456)
COMPUTE TOTAL-WS = FUNCTION SUM(FLDA FLDB FLDC FLDD).
COMPUTE MAX-WS = FUNCTION MAX(FLDA FLDB FLDC FLDD).
COMPUTE MIN-WS = FUNCTION MIN(FLDA FLDB FLDC FLDD).
```

Das wäre ein vollständiges Beispiel:

Erstellen Sie das Beispiel (*MINIMUM.cbl*), kompilieren Sie es und führen es aus.

```
IDENTIFICATION DIVISION.
  PROGRAM-ID. MINIMUM.
DATA DIVISION.
  WORKING-STORAGE SECTION.
  01 WS-NUM1 PIC 9(2) VALUE 6.
  01 WS-NUM2 PIC 9(2) VALUE 3.
PROCEDURE DIVISION.
  DISPLAY FUNCTION MIN(WS-NUM1, WS-NUM2).
  STOP RUN.
  END PROGRAM MINIMUM.
```

Funktionen können natürlich auch verschachtelt werden.

Beispiel:

```
COMPUTE x = FUNCTION MAX((FUNCTION SQRT(5)) 2.5 3.5).
```

Details zu den verschiedenen verfügbaren „Intrinsic Functions“ finden Sie in einer API-Dokumentation von COBOL. Etwa hier: http://opencobol.add1tocobol.com/OpenCOBOL%20Programmers%20Guide.pdf.

Hier sind zwei weitere vollständige Beispiele zum Einsatz dieser eingebauten Standardfunktionen.

Beispiel zum Ziehen von **Zufallszahlen** (*zufall.cbl*):

```
IDENTIFICATION DIVISION.
  PROGRAM-ID. ZUFALL.
DATA DIVISION.
  WORKING-STORAGE SECTION.
  01 COUNTER PIC 9(2) VALUE 0.
  01 WS-NUM1 PIC 9(2) VALUE 0.
  01 ZIEHUNG-TEMP1 PIC S9V9(10) VALUE 0.
  01 ZIEHUNG-TEMP2 PIC S9V9(10) VALUE 0.
PROCEDURE DIVISION.
  0100-PARA.
  DISPLAY "Ziehung Zufallszahlen".
  PERFORM 0110-PARA WITH TEST AFTER UNTIL COUNTER > 1.
  STOP RUN.
  0110-PARA.
  ADD 1 TO COUNTER.
  COMPUTE ZIEHUNG-TEMP1 = FUNCTION RANDOM.
  COMPUTE ZIEHUNG-TEMP2 = FUNCTION RANDOM (100).
  DISPLAY COUNTER ". Zufallszahl 1: " ZIEHUNG-TEMP1.
  DISPLAY COUNTER ". Zufallszahl 2: " ZIEHUNG-TEMP2.

  END PROGRAM ZUFALL.
```

Beispiel zur Ausgabe des aktuellen **Datums** (*datum.cbl*):

```
IDENTIFICATION DIVISION.
  PROGRAM-ID. DATUM.
DATA DIVISION.
  WORKING-STORAGE SECTION.
  01 WS-CURRENT-DATE-DATA.
    05 WS-CURRENT-DATE.
      10 WS-CURRENT-YEAR PIC 9(4).
      10 WS-CURRENT-MONTH PIC 9(2).
      10 WS-CURRENT-DAY PIC 9(2).
    05 WS-CURRENT-TIME.
      10 WS-CURRENT-HOUR PIC 9(2).
      10 WS-CURRENT-MINUTE PIC 9(2).
      10 WS-CURRENT-SECOND PIC 9(2).
      10 WS-CURRENT-MILLISECOND PIC 9(2).
    05 WS-CURRENT-TIMEZONE.
      10 WS-CURRENT-ZONE PIC X(1).
      10 WS-CURRENT-ZONEHOUR PIC 9(4).
PROCEDURE DIVISION.
  DISPLAY FUNCTION CURRENT-DATE.
  MOVE FUNCTION CURRENT-DATE TO WS-CURRENT-DATE-DATA.
  DISPLAY WS-CURRENT-DAY, ".", WS-CURRENT-MONTH, ".",
  WS-CURRENT-YEAR.
  IF WS-CURRENT-ZONE="+"
    DISPLAY WS-CURRENT-ZONEHOUR, " Stunden vor Greenwich."
  ELSE
    DISPLAY WS-CURRENT-ZONEHOUR, " Stunden nach Greenwich."
  END-IF.
  STOP RUN.
  END PROGRAM DATUM.
```

Versuchen Sie die Arbeitsweise des letzten Listings nachzuvollziehen.

Das Beispiel ist insbesondere deshalb interessant, weil die Rückgabe der Funktion CURRENT-DATE keinem elementaren Feld, sondern direkt einem Gruppenfeld zugewiesen wird. Die Funktion liefert eine Folge von Zeichen bzw. Werten, deren Bedeutung durch die Struktur des Gruppenfeldes WS-CURRENT-DATE-DATA wiedergespiegelt wird. Jedes Zeichen bzw. jede Folge von Zeichen des Rückgabewerts der Funktion stehen damit an einer Stelle, dessen Bedeutung über das Gruppenfeld erst verdeutlicht wird.

Wir haben also einen Speicherbereich, dessen Semantik das Gruppenfeld festlegt. Sie können aber auch Teile des Rückgabewerts damit einzeln verwenden. Sie müssen nur die untergeordneten Gruppenfelder oder auch einzelnen Elementfelder ansprechen. Das wird in dem Beispiel ja auch so vorgeführt.

Erstellen Sie ein Programm zur Kreisberechnung. Der Anwender soll zur Eingabe des Radius aufgefordert werden. Berechnen Sie mit COMPUTE den Kreisumfang und die Kreisfläche. Geben Sie beide Werte aus. Die Fläche ergibt als ***PI * Radius²***. Der Umfang als ***2 * PI * Radius***. Den Wert von PI erhalten Sie mit der gleichnamigen Funktion. Eine mögliche Lösung finden Sie im Anhang auf der Seite 78 ff.

Hier ist noch ein weiteres Beispiel, was die geraden Zahlen zwischen 0 und 100 auf dem Bildschirm ausgibt. Dazu kommt die Funktion MOD für die Modulo-Berechnung zum Einsatz und CONTINUE wird als Leeranweisung genutzt, wenn eine ungerade Zahl vorliegt.

Erstellen Sie das Beispiel (*GERADEZAHLEN.cbl*), kompilieren Sie es und führen es aus.

```
IDENTIFICATION DIVISION.
  PROGRAM-ID. GERADEZAHLEN.
DATA DIVISION.
  WORKING-STORAGE SECTION.
  01 I PIC 999 VALUE 0.
  01 WERT PIC 999 VALUE 0.
PROCEDURE DIVISION.
  PERFORM WITH TEST AFTER
    VARYING I FROM 1 BY 1
    UNTIL I > 100
    IF FUNCTION MOD( I, 2) <> 0 THEN
     CONTINUE
    ELSE
      DISPLAY I
    END-IF
    END-PERFORM.
  STOP RUN.
  END PROGRAM GERADEZAHLEN.
```

10.3 Umgang mit Strings

Strings dienen zum Darstellen von Texten und der Umgang damit gehört zu den elementarsten Anforderungen bei der Erstellung von Programmen. Allerdings ist der Umgang mit Strings in COBOL schwieriger als es die meisten Umsteiger von modernen Sprachen vermutlich vermuten. Der Grund liegt nicht zuletzt darin, dass COBOL den Speicherbereich, den ein String belegt, nicht so zuverlässig und trivial begrenzt, wie man es bei modernen Sprachen gewohnt ist.

10.3.1 Zeichenketten durchsuchen und Zeichen ersetzen

Eine der wichtigsten Techniken der String-Verarbeitung ist das Ersetzen bestimmter Zeichen in einem Quell-String. Das klappt auch in COBOL, ist aber eben – vermutlich unerwartet – nicht ganz trivial. Das Verb IN-

SPECT dient in COBOL zum Durchsuchen von Zeichenketten nach Suchbegriffen. Darauf basiert auch das Ersetzen von Zeichen. Dieser Abschnitt erklärt diese zentralen Techniken der String-Bearbeitung.

Die Funktion dieses COBOL-Verbs INSPECT besteht im

- Zählen von Zeichen und
- Ersetzen von Zeichen.

10.3.1.1 Zählen von Zeichen

Die Syntax zum Zählen von Zeichen sieht schematisch so aus:

```
INSPECT feld-1 TALLYING feld-2
FOR {ALL | LEADING | CHARACTERS}
{feld-3 | lit-3}
{BEFORE | AFTER} INITIAL {feld-4 | lit-4}
```

Hier ist ein vollständiges Beispiel (*ZEICHENKETTEN.cbl*)

```
IDENTIFICATION DIVISION.
  PROGRAM-ID. STRINGVERARBEITUNG.
DATA DIVISION.
  WORKING-STORAGE SECTION.
  01 WS-CNT1 PIC 9(2) VALUE 0.
  01 WS-CNT2 PIC 9(2) VALUE 0.
  01 WS-CNT3 PIC 9(2) VALUE 0.
  01 WS-CNT4 PIC 9(2) VALUE 0.
  01 WS-STRING PIC X(20) VALUE 'AaBaCDACDAAgFDEAAAFF'.
PROCEDURE DIVISION.
  INSPECT WS-STRING TALLYING WS-CNT1 FOR ALL CHARACTERS.
  DISPLAY "Alle Zeichen : "WS-CNT1.
  INSPECT WS-STRING TALLYING WS-CNT2 FOR ALL 'A'.
  DISPLAY "Anzahl A : "WS-CNT2.
  INSPECT WS-STRING TALLYING WS-CNT3 FOR ALL 'a'.
  DISPLAY "Anzahl a : "WS-CNT3.
  INSPECT WS-STRING TALLYING WS-CNT4 FOR ALL 'AA'.
  DISPLAY "Anzahl AA : "WS-CNT4.
  STOP RUN.
  END PROGRAM STRINGVERARBEITUNG.
```

10.3.1.2 Ersetzen von Zeichen mit REPLACING

Die Syntax zum Ersetzen von Zeichen mit REPLACING sieht schematisch so aus:

```
INSPECT feld-1 REPLACING CHARACTERS
BY {feld-2 | lit-2}
{BEFORE | AFTER} INITIAL {feld-3 | lit-3}
INSPECT feld-1 REPLACING
{ALL | LEADING | FIRST} {feld-2 | lit-2}
BY {feld-3 | lit-3}
{BEFORE | AFTER} INITIAL {feld-4 | lit-4}
```

Das wäre ein schematisches Beispiel zum Ersetzen:

```
INSPECT EING-MENGE REPLACING ALL SPACE BY ZERO
```

AFTER INITIAL '%'

FIRST ',' BY '.'

Ergebnis:

Vorher: >% 123,45 <

Nachher: >%000123.450<

Hier ist wieder ein vollständiges Beispiel (*string2.cbl*):

```
IDENTIFICATION DIVISION.
  PROGRAM-ID. STRINGVERARBEITUNG.
DATA DIVISION.
  WORKING-STORAGE SECTION.
  01 WS-STRING PIC X(20) VALUE 'AaBaCDACDAAgFDEAAAFF'.
PROCEDURE DIVISION.
  DISPLAY "Originaler String : "WS-STRING.
  INSPECT WS-STRING REPLACING ALL 'A' BY 'X'.
  DISPLAY "Neuer String : "WS-STRING.
  INSPECT WS-STRING REPLACING ALL 'XX' BY '__'.
  DISPLAY "String nach weiterer Ersetzung: "WS-STRING.
  STOP RUN.
  END PROGRAM STRINGVERARBEITUNG.
```

10.3.1.3 Ersetzen von Zeichen mit CONVERTING

Man kann INSPECT auch zum Ersetzen nutzen, indem jedes Zeichen einer Zeichenkette in ein anderes Zeichen einer anderen Zeichenkette konvertiert wird. Das soll der Vollständigkeit halber noch erwähnt werden.

Die Syntax sieht schematisch so aus:

INSPECT feld-1 CONVERTING {feld-2 | lit-2}

TO {feld-3 | lit-3}

{BEFORE | AFTER} INITIAL {feld-4 | lit-4}

10.3.2 String-Verkettung in COBOL

Für moderne Programmiersprachen ist die String-Verkettung, obschon ungemein wichtig, ein trivialer Vorgang. In COBOL ist String-Verkettung dagegen ziemlich aufwändig. Das COBOL-Verb STRING spielt in Verbindung mit END-STRING die zentrale Rolle, wobei mit ON OVERFLOW ein Verhalten beim Überschreiben des Wertebereichs der Zielvariablen festgelegt werden kann.

Die Syntax sieht schematisch so aus:

STRING {feld-1|lit-1} . . .

[DELIMITED BY {feld-2|lit-2|SIZE} . . .]

INTO feld-3 [WITH POINTER feld-4]

[ON OVERFLOW anweisung-5]

[NOT ON OVERFLOW anweisung-6]

[END-STRING]

Die Bedeutungen der Begriffe in der Syntax sind folgende:

- Alle Literale sind nicht nummerisch.

- feld-1, lit-1 . . . sind Sendefelder.
- feld-3 ist Empfangsfeld. Es darf nicht druckaufbereitet und nicht mit JUSTIFY definiert sein.
- Bei DELIMITED BY SIZE wird Sendefeld vollständig übertragen.
- Feld-4 ist die Anfangsposition für die Daten im Empfangsfeld. Es muss ganzzahlig und groß genug sein, um die Länge des Empfangsfeldes plus 1 aufzunehmen.
- Bei der Angabe des Pointer-Zusatzes muss der Anfangswert von feld-4 explizit gesetzt werden (1<= feld-4 <= len(feld-3)).
- Ohne Pointer-Angabe ist der implizite Zeiger auf 1 gesetzt.
- Der Wert beträgt des Zeigers beträgt am Ende len(übertragene Zeichen) + 1
- STRING ist beendet, wenn Zielfeld gefüllt oder alle Sendefelder bearbeitet sind.
- Nur der Teil des Empfangsfeldes ist verändert, in den Zeichen übertragen worden sind.
- Ist OVERFLOW angegeben und werden die Grenzen durch feld-3 oder Zeiger überschritten, wird die Bearbeitung beendet und die entsprechende Anweisung ausgeführt.

Sie sehen also, dass es extrem viele Besonderheiten für einen Vorgang zu beachten sind, der in modernen Programmiersprachen kaum noch eine Bemerkung wert ist.

Hier ist ein vollständiges Beispiel (*string3.cbl*):

```
IDENTIFICATION DIVISION.
  PROGRAM-ID. STRINGVERARBEITUNG.
DATA DIVISION.
  WORKING-STORAGE SECTION.
  01 WS-STRING PIC A(45).
  01 WS-STR1 PIC A(27) VALUE 'Cobol ungewohnt kompliziert'.
  01 WS-STR2 PIC A(16) VALUE 'Stringverkettung'.
  01 WS-STR3 PIC A(7) VALUE 'ist in'.
  01 WS-COUNT PIC 99 VALUE 1.
PROCEDURE DIVISION.
  STRING WS-STR2 DELIMITED BY SIZE
    WS-STR3 DELIMITED BY SPACE
    WS-STR1 DELIMITED BY SIZE
    INTO WS-STRING
    WITH POINTER WS-COUNT
    ON OVERFLOW DISPLAY 'OVERFLOW!'
  END-STRING.
  DISPLAY 'WS-STRING : 'WS-STRING.
  DISPLAY 'WS-COUNT : 'WS-COUNT.
  STOP RUN.
  END PROGRAM STRINGVERARBEITUNG.
```

10.3.3 Splitten von Strings in COBOL

Mit dem Verb UNSTRING spaltet man in Kombination mit END-UNSTRING in COBOL Strings an einem vorgegebenen Trennzeichen auf.

Die Syntax sieht schematisch so aus:

```
UNSTRING feld-1
[DELIMITED BY [ALL] {feld-2|lit-2}
```

```
[OR [ALL] {feld-3|lit-3}] ...]
INTO feld-4 [DELIMITER IN feld-5]
[COUNT    IN feld-6]
[WITH POINTER feld-7] [TALLYING IN  feld-8]
[   ON OVERFLOW anweisung-A]
[NOT ON OVERFLOW anweisung-B]
[END-UNSTRING]
```

Die Bedeutungen der Begriffe in der Syntax sind folgende:

- Alle Literale sind nicht nummerisch.
- feld-1 ist das Sendefeld.
- feld-4 ist Empfangsfeld.
- feld-2 bzw. lit-2 sind die Separatoren.
- feld-5 ist das Begrenzerempfangsfeld.
- feld-6 ist das Zählerfeld für die Datenübertragung; es enthält die Anzahl der Zeichen, die innerhalb von feld-1 bis zum Separator gefunden und übertragen wurden.
- feld-7 ist die Anfangsposition für die Datenübertragung im Sendefeld.
- feld-8 ist das Zählerfeld für die Anzahl der Daten empfangenden Felder.
- Die OVERFLOW Bedingung kommt zur Anwendung, wenn alle Daten empfangenden Felder verarbeitet sind und feld-1 enthält noch ungeprüfte Zeichen oder wenn feld-7 die Grenzen von feld-4 überschreitet.

Hier ist ein vollständiges Beispiel (*STRINGSPLITTEN.cbl*):

```
IDENTIFICATION DIVISION.
  PROGRAM-ID. STRINGVERARBEITUNG.
DATA DIVISION.
  WORKING-STORAGE SECTION.
  01 WS-STRING PIC X(100) VALUE
  "Stringverarbeitung in Cobol ist schwer"..
  01 WS-STR1 PIC A(10).
  01 WS-STR2 PIC A(10).
  01 WS-STR3 PIC A(10).
  01 WS-COUNT PIC 99 VALUE 1.
PROCEDURE DIVISION.
  UNSTRING WS-STRING DELIMITED BY SPACE
    INTO WS-STR1, WS-STR2, WS-STR3
  END-UNSTRING.
  DISPLAY 'WS-STR1 : 'WS-STR1.
  DISPLAY 'WS-STR2 : 'WS-STR2.
  DISPLAY 'WS-STR3 : 'WS-STR3.
  STOP RUN.
  END PROGRAM STRINGVERARBEITUNG.
```

10.4 Tabellen/Arrays in COBOL – Grundlagen

Arrays werden in COBOL als Tabellen bezeichnet. Umsteiger als modernen Programmiersprachen müssen hier aufpassen, weil dort diese Gleichsetzung nicht üblich ist. Ein Array bzw. eine Tabelle ist in COBOL nur eine lineare Datenstruktur und eine Sammlung individueller Datenartikel des gleichen Typs. Die Datenartikel einer Tabelle sind intern sortiert und indiziert und liegen im Speicher einfach hintereinander.

10.4.1 Eindimensionale Tabellen

Dieser Abschnitt bietet eine Einführung in den Umgang mit Tabellen. Der hauptsächliche Unterschied zum Anlegen von einfachen Datenvariablen liegt im Schlüsselwort OCCURS. Damit wird die Anzahl der „Wiederholungen" einer Datenstruktur festgelegt. Die bereits erwähnte Besonderheit ist, dass die Felder einer Tabelle im Speicher hintereinander gespeichert werden. So würde ein Array mit 4 Feldern aus jeweils 4 Zahlen einfach 16 Byte hintereinander im Speicher belegen und das Array definiert eine „Metastruktur" zu diesen 16 Byte. Oder anders ausgedrückt – das Array beinhaltet die Information, welche Bytes zu welchem der enthaltenen Felder gehören. Wenn Sie auf diese 16 Byte nicht über das Array, sondern auf einen anderen Weg zugreifen, steht keine Informationen zur Verfügung, wie diese Bytes zusammenhängen. So eine Denkweise mit einer „losgelösten" Metainformation zu der Struktur ist besonders für Programmierer aus objektorientierten Sprachen extrem befremdlich.

Beispiel:

```
01 STAMMSATZ.
05 UMSATZ-TAG OCCURS 366 PIC 9(4)V99.
```

Beispiel Gehaltstabelle

```
01 GEHALTS-TABELLE.
05 GEHALT OCCURS 12.
10 BRUTTO PIC 9(5)V99.
10 ZULAGE PIC 9(5)V99.
10 ABZUG PIC 9(5)V99.
```

• Im Hauptspeicher stehen die Daten direkt hintereinander in der Form:

– *BRUTTO(1),ZULAGE(1),ABZUG(1),BRUTTO(2)* etc.

Beispiel (*array1.cbl*)

```
IDENTIFICATION DIVISION.
  PROGRAM-ID. ARRAY1.
DATA DIVISION.
  WORKING-STORAGE SECTION.
  01 SUCHMUSTER .
    05 WS-STRING1 PIC X(1) OCCURS 20 TIMES.
  01 DATEN.
    05 DATENSATZ OCCURS 5 TIMES.
    10 WS-NAME PIC X(20).
    10 WS-VORNAME PIC X(20).
  01 DATEN2.
    05 DATENSATZ2 OCCURS 5 TIMES.
    10 WS-STRING PIC X(20) OCCURS 10 TIMES.
  01 STAMMSATZ.
    05 JAHR PIC 9(4) OCCURS 20 TIMES  INDEXED BY IND.
PROCEDURE DIVISION.
  MOVE "a" TO WS-STRING1(1).
```

```
  MOVE "b" TO WS-STRING1(2).
  MOVE "c" TO WS-STRING1(3).
  MOVE "d" TO WS-STRING1(4).
  MOVE "e" TO WS-STRING1(5).
  MOVE "z" TO WS-STRING1(20).
  DISPLAY WS-STRING1(1).
  DISPLAY SUCHMUSTER.
  MOVE "Meier" TO WS-STRING(1,1).
  DISPLAY WS-STRING(1,1).
  MOVE "Schmitt" TO WS-NAME(1).
  DISPLAY WS-NAME(1).
  MOVE "Otto" TO WS-NAME(5).
  DISPLAY WS-NAME(5).
  SET IND TO 5.
  STOP RUN.
  END PROGRAM ARRAY1.
```

```
F:\cobolprojekte\kap10>array1
a
abcde                z
Meier
Schmitt
Otto

F:\cobolprojekte\kap10>
```

Abb. 10.2: Gezielt Array-Positionen ansteuern

10.4.2 Mehrdimensionale Tabellen in COBOL

COBOL erlaubt die Arbeit mit mehrdimensionalen Tabellen bzw. Arrays. In diesem Beispiel (*array2.cbl*) sehen Sie, auf was Sie dabei achten müssen.

```
IDENTIFICATION DIVISION.
  PROGRAM-ID. ARRAY2.
DATA DIVISION.
  WORKING-STORAGE SECTION.
  01 WS-TABLE1.
    05 WS-A OCCURS 2 TIMES.
      10 WS-B PIC A(2) VALUE ' X'.
      10 WS-C OCCURS 2 TIMES.
      15 WS-D PIC X(2) VALUE ' Y'.
  01 WS-TABLE2.
    05 WS-E OCCURS 2 TIMES.
      10 WS-F OCCURS 2 TIMES.
      15 WS-G PIC 9(2) VALUE 1.
  01 WS-TABLE3.
    05 WS-H OCCURS 2 TIMES.
      10 WS-I OCCURS 2 TIMES.
      15 WS-J OCCURS 2 TIMES.
      20 WS-K PIC 9(2) VALUE 7.
PROCEDURE DIVISION.
  DISPLAY 'WS-TABLE1 : 'WS-TABLE1.
  DISPLAY 'WS-A(1)   : ' WS-A(1).
  DISPLAY 'WS-C(1,1) : ' WS-C(1,1).
  DISPLAY 'WS-C(1,2) : ' WS-C(1,2).
  DISPLAY 'WS-A(2)   : ' WS-A(2).
```

```
  DISPLAY 'WS-C(2,1) : ' WS-C(2,1).
  DISPLAY 'WS-C(2,2) : ' WS-C(2,2).
  DISPLAY 'WS-TABLE2 : 'WS-TABLE2.
  MOVE 55 TO WS-G(2,2).
  DISPLAY 'WS-TABLE2 : 'WS-TABLE2.
  DISPLAY 'WS-TABLE3 : 'WS-TABLE3.
  MOVE 55 TO WS-K(2,2,2).
  DISPLAY 'WS-TABLE3 : 'WS-TABLE3.
  STOP RUN.
  END PROGRAM ARRAY2.
```

10.4.3 Indexmanipulation bei Tabellen in COBOL

Mit dem Verb SET können Sie in COBOL den Index eines Arrays beeinflussen. Dazu muss dieser aber beim Anlegen mit INDEXED bereits spezifiziert werden. Tabellen können damit (schneller) mit einem (Maschinen-) Index angesprochen werden. So könnte ein Index gesetzt werden:

```
01 STAMMSATZ.
  05 JAHR-TAG OCCURS 366 INDEXED BY IND.
  10 IND PIC 9(4).
```

10.4.4 Einen Schlüssel (Key) festlegen

Tabellen können, wenn sie auf- oder absteigend sortiert sind (!), einen **Schlüssel** enthalten. Dazu dient dann das COBOL-Verb KEY. So könnte ein Schlüssel gesetzt werden:

```
01 PERSONAL-TABELLE.
  05 PERSON OCCURS 100
  ASCENDING KEY IS PERS-NR
  INDEXED BY PERS-IND.
  10 PERS-NRPIC  9(06).
  10 PERS-NAME     PIC  X(20).
```

10.4.4.1 Arbeiten mit dem INDEX

Wenn Sie einen Index besitzen, gibt es eine Reihe an Möglichkeiten zur Modifikation. Das geht mit dem Verb SET. Das ist eine Auswahl:

- SET ind-1 TO ind-2
- SET ind-1 TO var-2
- SET var-1 TO ind-2
- SET ind-1 TO literal
- SET ind-1 UP BY {variable | literal}
- SET ind-1 DOWN BY {variable | literal}

Hier ist ein vollständiges Beispiel (*array3.cbl*):

```
IDENTIFICATION DIVISION.
  PROGRAM-ID. ARRAYS.
DATA DIVISION.
  WORKING-STORAGE SECTION.
  01 WS-TABLE1.
    05 WS-A PIC A(10) VALUE 'Vorgabe' OCCURS 5 TIMES
    INDEXED BY I.
PROCEDURE DIVISION.
  DISPLAY 'WS-TABLE1 : 'WS-TABLE1.
```

```
  MOVE "Das passt so nicht richtig rein" TO WS-TABLE1.
  DISPLAY 'WS-TABLE1 nach Einfuegen: 'WS-TABLE1.
  DISPLAY 'WS-A(1) nach Einfuegen: 'WS-A(1).
  DISPLAY 'WS-A(2) nach Einfuegen: 'WS-A(2).
  DISPLAY 'WS-A(5) nach Einfuegen: 'WS-A(5).
  SET I TO 1.
  DISPLAY 'Steppen durch WS-A nach Einfuegen: 'WS-A(I).
  SET I UP BY 1.
  DISPLAY 'Steppen durch WS-A nach Einfuegen: 'WS-A(I).
  SET I UP BY 2.
  DISPLAY 'Steppen durch WS-A nach Einfuegen: 'WS-A(I).
  STOP RUN.
  END PROGRAM ARRAYS.
```

10.4.5 Suchen in Tabellen in COBOL

Mit dem Verb SEARCH können Sie in Kombination mit END-SEARCH in COBOL Arrays bzw. deren Elemente gezielt durchsuchen. Das wird mit dem Setzen des Index über SET und INDEX BY kombiniert. Das ist die grundsätzliche Syntax:

```
SEARCH struktur [VARYING index]
[AT END    anweisung]
WHEN   bedingung-1 anweisung-1
END-SEARCH
```

Auch hier gibt es – neben der Tatsache, dass *struktur* indiziert sein muss – ein paar Voraussetzungen bzw. Regeln, die zu beachten sind:

- Beginn des Suchvorgangs ab dem Wert von Index
- Nach anweisung-1 wird mit der Anweisung nach SEARCH weiter gearbeitet.
- Ist der Index für *struktur* definiert, wird dieser genommen, sonst der nächste in der Hierarchie.
- Beliebig viele WHEN-Angaben sind erlaubt.
- Wird der Begriff nicht gefunden, greift die AT END-Anweisung.
- Es kann nur eine Dimension durchsucht werden.

Hier ist ein vollständiges Beispiel (*array4.cbl*):

```
IDENTIFICATION DIVISION.
  PROGRAM-ID. ARRAYS.
DATA DIVISION.
  WORKING-STORAGE SECTION.
  01 WS-TABLE1.
    05 WS-A PIC A(10) VALUE 'Vorgabe' OCCURS 5 TIMES
    INDEXED BY I.
  01 WS-SRCH PIC A(5) VALUE 'nicht'.
PROCEDURE DIVISION.
  DISPLAY 'WS-TABLE1 : 'WS-TABLE1.
  MOVE "Das passt so nicht richtig rein" TO WS-TABLE1.
  DISPLAY 'WS-TABLE1 nach Einfuegen: 'WS-TABLE1.
  SET I TO 1.
    SEARCH WS-A
  AT END DISPLAY 'Nicht gefunden'
```

```
WHEN WS-A(I)=WS-SRCH
  DISPLAY 'Gefunden'
END-SEARCH.
MOVE "nicht" TO WS-A(4).
DISPLAY 'Einfuegen von nicht an Stelle 4: 'WS-TABLE1.
SET I TO 1.
SEARCH WS-A
  AT END DISPLAY 'Nicht gefunden'
  WHEN WS-A(I)=WS-SRCH
  DISPLAY 'Gefunden'
END-SEARCH.
STOP RUN.
END PROGRAM ARRAYS.
```

11 Umgang mit Dateien in COBOL

Die effiziente und performante Verwaltung von Dateien ist eine der wichtigsten Aufgaben von COBOL. In diesem Kapitel lernen Sie die wichtigsten Anweisungen dafür kennen.

11.1 Dateihandhabung in COBOL

In COBOL unterscheidet sich der Umgang mit Dateien erheblich von der Art, die man bei modernen Programmiersprachen gewohnt ist. Man differenziert stark nach der Art der Datei und des Zugriffs. Alles in Allem eine sehr „historische" Herangehensweise, die aber vor allen Dingen im Umfeld von Mainframes immer noch sinnvoll, üblich und oft auch notwendig ist. Man betrachtet hier die folgenden Strukturen:

- Feld
- Record
- Physikalischer Record oder Block
- Logischer Datensatz
- Datei

11.2 Dateiorganisation und Zugriffsmethoden – Grundlagen

COBOL unterscheidet zwischen der Dateiorganisation auf dem Datenträger und den Zugriffsmethoden.

11.2.1 Dateiorganisation

- Sequenzielle Dateiorganisation
- Zeilenorientierte Dateiorganisation
- Indexorganisiert
- Relative Dateiorganisation

11.2.2 Zugriffsmethoden

- QSAM – Queued Sequential Access Method ordnet Daten sequenziell.
- VSAM – Virtual Storage Access Method ordnet Daten mit einem Indexschlüssel.

11.2.3 Vor- und Nachteile der Zugriffsverfahren

Nachfolgenden sehen Sie die verschiedenen Vor- und Nachteile der verschiedenen Zugriffsverfahren auf Dateien in COBOL.

11.2.3.1 Relative Dateizugriffe: Vorteile

- Schnellste Zugriffstechnik
- Wenig Overhead
- Kann sequenziell oder direkt gelesen werden

11.2.3.2 Relative Dateizugriffe: Nachteile

- Gefahr von Speicherplatzverschwendung
- Nur ein Schlüssel

- Zwingend ein numerischer Schlüssel
- Art der Datenträger ist eingeschränkt (etwa keine Magnetbänder erlaubt)

11.2.3.3 Indizierte Dateizugriffe: Vorteile

- Mehrere alphanumerische Schlüssel sind möglich
- Nur der Primärschlüssel muss eindeutig sein
- Eine indizierte Datei kann sequenziell oder über jeden Schlüssel gelesen werden

11.2.3.4 Indizierte Dateizugriffe: Nachteile

- Langsamste Form der Dateizugriffe
- Nicht sonderlich speichereffizient
- Eine zwingende Struktur ist notwendig:
    - Daten mit Primärschlüssel
    - Indexdatensätze für alle alternativen Schlüssel
    - Die aktuellen Datensätze sowie weitere Informationen

11.3 File-Access-Modus in COBOL

Für den Zugriff auf Dateien gibt es einen Dateizugriffsmodus, der sich aufgrund der Dateiorganisation und dem Verständnis von Dateien in COBOL ergibt. Als COBOL-Anweisungen verwenden Sie SELECT zur Auswahl einer Datei und OPEN zum Öffnen. Mit ORGANIZATION geben Sie die Dateiorganisation und mit ACCESS MODE Iden Zugriffsmodus an. Mit CLOSE schließen Sie ggfls. eine Datei wieder.

Beispiel mit verschiedenen Zugriffsverfahren – noch ohne konkrete Zugriffe (*dateizugriffsverfahren.cbl*):

```
IDENTIFICATION DIVISION.
  PROGRAM-ID. DATEIZUGRIFF.
ENVIRONMENT DIVISION.
  INPUT-OUTPUT SECTION.
  FILE-CONTROL.
    SELECT file-name1 ASSIGN TO "dat1.dat"
    ORGANIZATION IS SEQUENTIAL
    ACCESS MODE IS SEQUENTIAL.
    SELECT file-name2 ASSIGN TO "dat2.dat"
    ORGANIZATION IS INDEXED
    ACCESS MODE IS SEQUENTIAL
    RECORD KEY IS rec-key1
    ALTERNATE RECORD KEY IS rec-key2.
    SELECT file-name3 ASSIGN TO "dat3.dat"
    ORGANIZATION IS RELATIVE
    ACCESS MODE IS SEQUENTIAL
    RELATIVE KEY IS rec-key1.
    SELECT file-name4 ASSIGN TO "dat4.dat"
    ORGANIZATION IS INDEXED
    ACCESS MODE IS RANDOM
    RECORD KEY IS rec-key1
    ALTERNATE RECORD KEY IS rec-key2.
    SELECT file-name5 ASSIGN TO "dat5.dat"
    ORGANIZATION IS RELATIVE
    ACCESS MODE IS RANDOM
```

```
    RELATIVE KEY IS rec-key1.
    SELECT file-name6 ASSIGN TO "dat6.dat"
    ORGANIZATION IS SEQUENTIAL
    ACCESS MODE IS DYNAMIC
    RECORD KEY IS rec-key1
    ALTERNATE RECORD KEY IS rec-key2.
    SELECT file-name7 ASSIGN TO "dat7.dat"
    ORGANIZATION IS RELATIVE
    ACCESS MODE IS DYNAMIC
    RELATIVE KEY IS rec-key1.
DATA DIVISION.
  FILE SECTION.
  WORKING-STORAGE SECTION.
  01  rec-key1   PIC X(10).
  01  rec-key2   PIC X(10).
PROCEDURE DIVISION.
  STOP RUN.
  END PROGRAM DATEIZUGRIFF.
```

11.4 Grundsätzlicher Dateilesezugriff in COBOL

Mittels der Anweisung READ können Sie aus einer Datei Daten einlesen. Diese muss vorher mit SELECT ausgewählt und mit OPEN INPUT geöffnet werden. Das folgende Beispiel (*datei1.cbl*) zeigt Ihnen anhand eines zeilensequentiellen Zugriffs, wie Sie mit COBOL in bzw. aus einer Datei lesen können.

```
IDENTIFICATION DIVISION.
  PROGRAM-ID. DATEIZUGRIFF.
ENVIRONMENT DIVISION.
  INPUT-OUTPUT SECTION.
  FILE-CONTROL.
  SELECT STUDENT ASSIGN TO 'studenten.txt'
  ORGANIZATION IS LINE SEQUENTIAL.
DATA DIVISION.
  FILE SECTION.
  FD STUDENT.
    01 STUDENT-FILE.
    05 STUDENT-ID PIC 9(5).
    05 NAME PIC A(25).
  WORKING-STORAGE SECTION.
  01 WS-STUDENT.
    05 WS-STUDENT-ID PIC 9(5).
    05 WS-NAME PIC A(25).
  01 WS-EOF PIC A(1).
PROCEDURE DIVISION.
  OPEN INPUT STUDENT.
  PERFORM UNTIL WS-EOF='Y'
    READ STUDENT INTO WS-STUDENT
    AT END MOVE 'Y' TO WS-EOF
    NOT AT END DISPLAY WS-STUDENT
    END-READ
  END-PERFORM.
  CLOSE STUDENT.
  STOP RUN.
  END PROGRAM DATEIZUGRIFF.
```

11.5 Grundsätzlicher Dateischreibzugriff in COBOL

Das Schreiben von Daten in eine Datei geht ziemlich analog wie das Lesen. Nur kommen hier WRITE und OPEN OUTPUT zur Anwendung. Das Beispiel (*datei2.cbl*) zeigt Ihnen anhand eines zeilensequentiellen Zugriffs, wie Sie mit COBOL Daten in eine Datei schreiben können.

```
IDENTIFICATION DIVISION.
  PROGRAM-ID. DATEIZUGRIFF.
ENVIRONMENT DIVISION.
  INPUT-OUTPUT SECTION.
  FILE-CONTROL.
  SELECT STUDENT ASSIGN TO 'studenten.txt'
  ORGANIZATION IS LINE SEQUENTIAL.
DATA DIVISION.
  FILE SECTION.
  FD STUDENT.
  01 STUDENT-FILE.
    05 STUDENT-ID PIC 9(5).
    05 NAME PIC A(25).
  WORKING-STORAGE SECTION.
  01 WS-STUDENT.
    05 WS-STUDENT-ID PIC 9(5).
    05 WS-NAME PIC A(25).
  01 WS-EOF PIC A(1).
PROCEDURE DIVISION.
  OPEN OUTPUT STUDENT.
  MOVE 20006 TO STUDENT-ID.
  MOVE ' Thaler Tim' TO NAME.
  WRITE STUDENT-FILE
  END-WRITE.
  CLOSE STUDENT.
  STOP RUN.
  END PROGRAM DATEIZUGRIFF.
```

11.6 Der File-Status in COBOL

Über den **Dateistatus** kann man verschiedene Dinge steuern, die beim Zugriff auf Dateien auftreten können. Man kann beispielsweise auf Fehler reagieren. Man wird den Dateistatus allgemein so verwenden, dass man nach der SELECT-Anweisung mit FILE STATUS IS arbeitet. Dabei wird eine Variable festgelegt, über die der Dateistatus zur Verfügung gestellt wird und in der WORKING STORAGE SECTION

Etwa so:

```
FILE STATUS IS FILE-CHECK-KEY
...
WORKING-STORAGE SECTION.
...
   05  FILE-CHECK-KEY   PIC X(2).
```

Beim konkreten Öffnen wird man dann der Variablen einen Wert zuweisen. Der Dateistatus besteht in COBOL als 2 Byte.

Das erste Byte ist der Statusschlüssel und der ist am Wichtigsten. Er gibt an, welche Bedingungen allgemein vorliegen. Insbesondere der Wert 0 ist von Bedeutung, denn das bezeichnet eine erfolgreiche Operation und das will man natürlich beim Öffnen haben. Mit einer IF-Anweisung prüft man deshalb fast immer auf 00 oder nicht 00.

```
IF FILE-CHECK-KEY NOT = "00"
```

Von Bedeutung ist noch 1 für das erste Byte. Das bedeutet AT END.

11.7 Read-Write-Zugriff auf eine Datei in COBOL

In vielen Programmen muss man sowohl aus einer Datei lesen als auch in eine Datei schreiben. Anhand eines konkreten Beispiels (*datei3.cbl*) sehen Sie, wie Sie das machen – und auch vor dem ersten Lesezugriff eine bis dahin noch nicht vorhandene Datei anlegen.

```
IDENTIFICATION DIVISION.
  PROGRAM-ID. DATEIZUGRIFF.
ENVIRONMENT DIVISION.
  INPUT-OUTPUT SECTION.
  FILE-CONTROL.
  SELECT ZAEHLER ASSIGN TO 'zaehler.txt'
  FILE STATUS IS FILE-CHECK-KEY
  ORGANIZATION IS LINE SEQUENTIAL.
DATA DIVISION.
  FILE SECTION.
  FD ZAEHLER.
  01 ZAEHLER-FILE.
    05 ZAEHLER-ID PIC 9(5) VALUE 0.
  WORKING-STORAGE SECTION.
  01 WS-ZAEHLER.
    05 WS-ZAEHLER-ID PIC 9(5) VALUE 0.
    05  FILE-CHECK-KEY   PIC X(2).
PROCEDURE DIVISION.
  OPEN INPUT ZAEHLER.
  IF FILE-CHECK-KEY NOT = "00"
  GO TO 0200-INIT
  END-IF.
  READ ZAEHLER INTO WS-ZAEHLER
  END-READ.
  CLOSE ZAEHLER.
  OPEN OUTPUT ZAEHLER.
  ADD 1 TO ZAEHLER-ID.
  DISPLAY ZAEHLER-ID.
  WRITE ZAEHLER-FILE
  END-WRITE.
  CLOSE ZAEHLER.
  STOP RUN.
  0200-INIT.
  OPEN OUTPUT ZAEHLER.
  MOVE '00001' TO ZAEHLER-ID.
  WRITE ZAEHLER-FILE
  END-WRITE.
  CLOSE ZAEHLER.
  DISPLAY "Zaehlerdatei angelegt mit Startwert ", ZAEHLER-ID.
  STOP RUN.
  END PROGRAM DATEIZUGRIFF.
```

Gegen Ende des Buchs soll noch eine anspruchsvollere Aufgabe gestellt werden, welche die Kenntnisse kombiniert, die in dem Kurs vermittelt wurden. Unter anderem benötigen wir dazu Dateizugriffe, den Aufruf von Unterprogrammen, String-Verarbeitung und natürlich die klassischen Syntax-Techniken aus COBOL. Die Aufgabe ist dies:

Erstellen sollen Sie ein Programm, das nur berechtigten Benutzern den Zugang zu bestimmten

Informationen gestattet. Das Programm ist eine Erweiterung der Übung auf Seite 56 und fordert wie dort auch einen Anwender zur Eingabe einer Userid und des Passworts auf. Sofern diese Daten als „legitimiert“ hinterlegt sind, soll ein bestimmter Inhalt angezeigt werden. Andernfalls eine Fehlermeldung, dass der Zugang nicht gestattet ist. Beachten Sie, dass die ACCEPT-Anweisung zur Entgegennahme der Benutzereingaben leider ziemlich unflexibel ist und erfordert, dass die Anzahl der Zeichen bei der Eingabe exakt der Länge des Feldes entsprechen muss, in das die Eingabe erfolgen soll. Sind die Eingabedaten kürzer als der Empfangsbereich, wird der Bereich mit Leerzeichen der entsprechenden Darstellung für den Empfangsbereich aufgefüllt. Deshalb wird in dem Beispiel mit einer festen Anzahl von Zeichen für die Benutzerkennung als auch das Passwort gearbeitet. So eine Einengung kennt man aus modernen Systemen natürlich nicht mehr, aber um das Beispiel nicht zu kompliziert werden zu lassen, soll diese Einschränkung akzeptiert werden.

Die Erweiterung gegenüber der Übung auf Seite 56 sind nun Folgende: Die Zugangsdaten selbst sollen nicht mehr im Quellcode, sondern in einer Textdatei gespeichert werden, die von dem Programm eingelesen werden muss. Die Struktur soll so aufgebaut werden, dass der Benutzername und das Passwort jeweils in einer Zeile notiert und durch ein Trennzeichen (etwa das Zeichen #) separiert werden. Man nennt das dann eine CSV-Datei (Comma-separated values). Dabei werden aus den oben genannten Gründen jeweils genau 4 Zeichen für den Benutzer und das Passwort verwendet.

So könnte die Datei aussehen:

otto#gehe

will#pass

fred#wilm

Das Einlesen der Daten sowie die Verifizierung der Daten sollen jeweils in ein Unterprogramm ausgelagert werden. Dabei muss man String-Verarbeitungstechniken nutzen, um die einzeln eingegebenen Daten des Anwenders und die als ein String gespeicherten Daten in der Datei vergleichen zu können. Eine mögliche Lösung finden Sie im Anhang auf der Seite 78 ff.

11.8 Sortieren von Dateien in COBOL

Mit dem Verb SORT können Sie in COBOL ganz einfach Dateien nach vorgegebenen Kriterien sortieren. Dabei wird aus einer Quelldatei eine sortierte Version als neue Datei erzeugt. Das soll die Originaldatei sein, die unsortiert ist (*studenten.txt*):

20006 Thaler Tim

12306 Teu Tom

00006 Wuest Willi

24501 Rueffel Rudi

43009 Feuerstein Wilma

10006 Feuerstein Fred

61234 Dent Arthur

00100 Blues Jake

Mit dem Code (*datei4.cbl*) wird diese Information sortiert und in die Datei *studenten.new* geschrieben:

```
IDENTIFICATION DIVISION.
  PROGRAM-ID. DATEIZUGRIFF.
ENVIRONMENT DIVISION.
  INPUT-OUTPUT SECTION.
  FILE-CONTROL.
  SELECT STUDENTINPUT ASSIGN TO 'studenten.txt'
  ORGANIZATION IS LINE SEQUENTIAL.
  SELECT STUDENTOUTPUT ASSIGN TO 'studenten.new'
  ORGANIZATION IS LINE SEQUENTIAL.
  SELECT WORK ASSIGN TO WRK.
DATA DIVISION.
  FILE SECTION.
  FD STUDENTINPUT.
  01 INPUT-STUDENT.
    05 STUDENT-ID-I PIC 9(5).
    05 STUDENT-NAME-I PIC A(25).
  FD STUDENTOUTPUT.
  01 OUTPUT-STUDENT.
    05 STUDENT-ID-O PIC 9(5).
    05 STUDENT-NAME-O PIC A(25).
SD WORK.
  01 WORK-STUDENT.
  05 STUDENT-ID-W PIC 9(5).
  05 STUDENT-NAME-W PIC A(25).
PROCEDURE DIVISION.
  SORT WORK ON DESCENDING KEY STUDENT-ID-O.
  USING STUDENTINPUT GIVING STUDENTOUTPUT.
  DISPLAY 'Sortieren OK'.
  STOP RUN.
  END PROGRAM DATEIZUGRIFF.
```

11.9 Verbinden von Dateien in COBOL

Mit dem Verb MERGE verbinden Sie Dateien zu einer neuen Datei. Dabei können Sie Kriterien angeben, wie die Quelldateien zusammengefügt werden. Das sind die Dateien *studenten.txt* (s.o.) und *studenten2.txt*.

61534 Dent Zaphod

93119 Fisch Otto

24502 Roth Felix

60006 Kater Herb

66536 Roth Florian

Mit diesem Code (*datei5.cbl*) werden die beiden Dateien gemerged und das Ergebnis in der Datei *studenten.new* abgelegt:

```
IDENTIFICATION DIVISION.
  PROGRAM-ID. DATEIZUGRIFF.
ENVIRONMENT DIVISION.
  INPUT-OUTPUT SECTION.
  FILE-CONTROL.
  SELECT STUDENTINPUT1 ASSIGN TO 'studenten.txt'
  ORGANIZATION IS LINE SEQUENTIAL.
  SELECT STUDENTINPUT2 ASSIGN TO 'studenten2.txt'
  ORGANIZATION IS LINE SEQUENTIAL.
  SELECT STUDENTOUTPUT ASSIGN TO 'studenten.new'
  ORGANIZATION IS LINE SEQUENTIAL.
```

```
  SELECT WORK ASSIGN TO WRK.
DATA DIVISION.
  FILE SECTION.
  FD STUDENTINPUT1.
  01 INPUT1-STUDENT.
    05 STUDENT-ID-I1 PIC 9(5).
    05 STUDENT-NAME-I1 PIC A(25).
  FD STUDENTINPUT2.
  01 INPUT2-STUDENT.
    05 STUDENT-ID-I2 PIC 9(5).
    05 STUDENT-NAME-I2 PIC A(25).
  FD STUDENTOUTPUT.
  01 OUTPUT-STUDENT.
    05 STUDENT-ID-O PIC 9(5).
    05 STUDENT-NAME-O PIC A(25).
  SD WORK.
  01 WORK-STUDENT.
    05 STUDENT-ID-W PIC 9(5).
    05 STUDENT-NAME-W PIC A(25).
PROCEDURE DIVISION.
  MERGE WORK ON DESCENDING KEY STUDENT-NAME-O.
  USING STUDENTINPUT1, STUDENTINPUT2 GIVING STUDENTOUTPUT.
  DISPLAY 'Zusammenfuegen OK'.
  STOP RUN.
  END PROGRAM DATEIZUGRIFF.
```

12 Anhang

12.1 Lösungen zu Aufgaben

In diesem Abschnitt finden Sie Lösungen zu den Aufgaben, die im Laufe der Unterlagen gestellt und dort nicht direkt behandelt wurden.

Aufgaben in Abschnitt 7.1

Die Aufgabe war, dass Sie alle Stellen sollten, an denen COBOL-Bezeichner klein statt groß geschrieben wurden. Nachfolgend sind sie fett markiert.

```
IDENTIFICATION DIVISION.
PROGRAM-ID. CASESENSITIV.
DATA DIVISION.
WORKING-STORAGE SECTION.
01 WS-NUM1 PIC 9(9) VALUE 3.
01 WS-NUM2 PIC 9(9).
PROCEDURE DIVISION.
  MOVE 15 TO WS-NUM2.
  if WS-NUM1 IS GREATER THAN OR EQUAL TO WS-NUM2 THEN
    display 'WS-NUM1 > WS-NUM2'
  ELSE
    DISPLAY 'WS-NUM1 < WS-NUM2'
  end-IF.
  STOP RUN.
  END PROGRAM CASESENSITIV.
```

Aufgaben in Abschnitt 7.6

Bei dem Programm, das der Aufgabe zugrunde liegt, wurden mehrere Picture-Klauseln verwendet. Sie sollten das Programm ausführen und darauf achten, dass Sie die Zeichen **nicht** (!) wie gefordert eingeben. Also Zahlen statt Buchstaben und umgekehrt. Auch sollten mehr oder weniger Zeichen als notwendig bzw. erlaubt eingegeben werden.

Sie werden erkennen, dass Buchstaben statt Zahlen dazu führen, dass COBOL das als die numerische 0 wertet. Zahlen können jedoch meist an Stelle von Buchstaben angegeben werden. Wenn Sie die Länge eines Feldes überschreiten, werden die überzähligen Stellen abgeschnitten (siehe Abb. 12.1).

Man sollte aber dringend darauf achten, dass die Werte immer mit dem Datentyp einer Variablen übereinstimmen.

```
F:\cobolprojekte\kap7>picture
Geben Sie die erste zweistellige Zahl ein
ab
Geben Sie die zweite dreistellige Zahl ein
def
Geben Sie 5 Buchstaben ein
12345
Geben Sie 5 Buchstaben oder Zahlen ein
123456789
Geben Sie 2 Buchstaben und dann 2 Zahlen ein
12ab
00.00
+000
12345
12345
12ab

F:\cobolprojekte\kap7>
```

Abb. 12.1: Bewusste Fehlangaben

Aufgaben in Abschnitt 8.1

Sie sollten in der Aufgabe ein Programm erstellen, das mittels der besprochenen mathematischen Verben einen Taschenrechner simuliert. Das Programm sollte die Möglichkeit der Addition, Multiplikation, Subtraktion und Division bereitstellen und dazu einen Anwender zur Eingabe zweier Zahlen und der gewünschten Rechenoperation auffordern. Die mathematischen Verben sollten je nach gewünschter Rechenoperation aufgerufen und das Ergebnis ausgegeben werden. Eine mögliche Lösung zeigt das folgende Listing:

```
IDENTIFICATION DIVISION.
  PROGRAM-ID. TASCHENRECHNER.
DATA DIVISION.
  WORKING-STORAGE SECTION.
  01 DATEN.
    05 WS-NUM1 PIC 9(2) VALUE 0.
    05 WS-NUM2 PIC 9(2) VALUE 0.
    05 WS-OP PIC A(1) VALUE NULL.
PROCEDURE DIVISION.
  DISPLAY "Geben Sie die erste Zahl ein".
  ACCEPT WS-NUM1.
  DISPLAY "Geben Sie die zweite Zahl ein".
  ACCEPT WS-NUM2.
  DISPLAY "Geben Sie die Operation ein".
  ACCEPT WS-OP.
  IF WS-OP EQUAL TO "+" THEN
    ADD WS-NUM1 TO WS-NUM2
    END-ADD
  ELSE IF WS-OP EQUAL TO "*" THEN
    MULTIPLY WS-NUM1 BY WS-NUM2
  ELSE IF WS-OP EQUAL TO "/" THEN
    DIVIDE WS-NUM1 INTO WS-NUM2
  ELSE IF WS-OP EQUAL TO "-" THEN
    SUBTRACT WS-NUM1 FROM WS-NUM2
  END-IF.
  DISPLAY WS-NUM2.
  STOP RUN.
  END PROGRAM TASCHENRECHNER.
```

Aufgaben in Abschnitt 8.2

Sie sollten in der Aufgabe ein Programm, das mittels COMPUTE zwei Zahlen addiert, multipliziert, dividiert und subtrahiert. Die Zahlen sollen in zwei Variablen zur Verfügung stehen und sollen die Werte 13 und 5 haben. Der Rest der Division von 13 durch den Wert 5 soll mit Nachkommateil dargestellt werden. Das folgende Listing ist eine Möglichkeit, wie die Aufgabe gelöst werden kann (Abb. 12.2).

```
IDENTIFICATION DIVISION.
  PROGRAM-ID. MATHVERBS.
DATA DIVISION.
WORKING-STORAGE SECTION.
  01 DATEN.
    05 WS-NUM1 PIC 9(2) VALUE 13.
    05 WS-NUM2 PIC 9(2) VALUE 5.
    05 WS-NUM3 PIC 9(2) VALUE 0.
    05 WS-NUM4 PIC 9(2)V9(2) VALUE 0.
PROCEDURE DIVISION.
  COMPUTE WS-NUM3 = WS-NUM1 * WS-NUM2.
  DISPLAY "Multiplikation: ", WS-NUM3.
  COMPUTE WS-NUM3 = WS-NUM1 + WS-NUM2.
  DISPLAY "Addition: ", WS-NUM3.
  COMPUTE WS-NUM3 = WS-NUM1 - WS-NUM2.
  DISPLAY "Subtraktion: ", WS-NUM3.
  COMPUTE WS-NUM3 = WS-NUM1 / WS-NUM2.
  DISPLAY "Ganzzahldivision: ", WS-NUM3.
  COMPUTE WS-NUM4 = WS-NUM1 / WS-NUM2.
  DISPLAY "Division: ", WS-NUM4.
  STOP RUN.
  END PROGRAM MATHVERBS.
```

```
F:\cobolprojekte\anhang>aufgabe8_2
Multiplikation: 65
Addition: 18
Subtraktion: 08
Ganzzahldivision: 02
Division: 02.60

F:\cobolprojekte\anhang>_
```

Abb. 12.2: Mit COMPUTE arbeiten

Aufgaben in Abschnitt 9.1

In dem Listing der Aufgabe wurden Vergleiche gezogen. Diese liefern das:

```
IF WS-NUM1 = WS-NUM2 THEN
```

Der Vergleich liefert TRUE, da die Zahlen gleich sind.

```
IF WS-FELD1 = WS-FELD2 THEN
```

Die Textfelder sind nicht gleich. Ab der Stelle 4 unterscheiden sich die Strings.

```
IF WS-FELD1 > WS-FELD2 THEN
```

Ab dem Zeichen 4 unterscheiden sich die Felder. Das kürzere Feld (FELD1) wird mit einem Leerzeichen aufgefüllt. Das Zeichen an Position 4 im ersten Feld ist f, im zweiten Feld p. Das Zeichen f ist im ANSI-Zeichensatz kleiner als das Zeichen p. Also ist WS-FELD1 nicht größer als WS-FELD2.

```
IF WS-FELD2 = WS-FELD3 THEN
```

Die Textfelder sind nicht gleich. Bereits an Stelle 1 unterscheiden sich die Strings.

```
IF WS-FELD3 > WS-FELD2 THEN
```

Am Zeichen 1 unterscheiden sich die Felder. Das Zeichen an Position 1 im ersten Feld ist r, im zweiten Feld R. Das Zeichen r ist im ANSI-Zeichensatz größer als das Zeichen R. Also ist WS-FELD2 größer als WS-FELD3.

Sie sollten in der Aufgabe ein Programm erstellen, das vom Anwender eine Benutzerkennung und ein Passwort entgegen nimmt. Das Passwort sollte mit einem vorgegebenen Text und das Passwort mit einem anderen Text übereinstimmen. Je nach Eingabe werden unterschiedliche Ausgaben angezeigt. Eine mögliche Lösung wäre das:

```
IDENTIFICATION DIVISION.
  PROGRAM-ID. ENTSCHEIDUNG1.
DATA DIVISION.
  WORKING-STORAGE SECTION.
  01 DATEN.
    05 WS-USERID PIC X(4) VALUES SPACES.
    05 WS-PASSWORT PIC X(4) VALUES SPACES.
PROCEDURE DIVISION.
  DISPLAY "Geben Sie die USERID ein".
  ACCEPT WS-USERID.
  DISPLAY "Geben Sie das PASSWORT ein".
  ACCEPT WS-PASSWORT.
  IF (WS-USERID EQUAL TO "otto" AND WS-PASSWORT EQUAL TO "abcd") THEN
    DISPLAY "Hallo ", WS-USERID
  ELSE
    DISPLAY "Falsche Zugangsdaten "
  END-IF.
  STOP RUN.
  END PROGRAM ENTSCHEIDUNG1.
```

Aufgaben in Abschnitt 10.1

Sie sollten ein Taschenrechnerprogramm mit den Funktionalitäten der Addition, Multiplikation, Subtraktion und Division erstellen. Das Hauptprogramm sollte einen Anwender zur Eingabe zweier Zahlen und der gewünschten Rechenoperation auffordern und die Rechenoperationen in vier Unterprogramme ausgelagert werden, die je nach gewünschter Rechenoperation aufgerufen werden. Dort sollten jeweils die Rechenoperationen durchgeführt und das Ergebnis im Hauptprogramm ausgegeben werden. Eine mögliche Lösung sieht so aus:

Das Hauptprogramm:

```
IDENTIFICATION DIVISION.
  PROGRAM-ID. TASCHENRECHNERHAUPT.
DATA DIVISION.
  WORKING-STORAGE SECTION.
```

```
  01 WS-NUM1 PIC 9(2) VALUE 0.
  01 WS-NUM2 PIC 9(2) VALUE 0.
  01 WS-OP PIC A(1) VALUE NULL.
PROCEDURE DIVISION.
  DISPLAY "Geben Sie die erste Zahl ein".
  ACCEPT WS-NUM1.
  DISPLAY "Geben Sie die zweite Zahl ein".
  ACCEPT WS-NUM2.
  DISPLAY "Geben Sie die Operation ein".
  ACCEPT WS-OP.
  IF WS-OP EQUAL TO "+" THEN
    CALL 'ADDITION' USING WS-NUM1 WS-NUM2
  ELSE IF WS-OP EQUAL TO "*" THEN
    CALL 'MULTIPLIKATION' USING WS-NUM1 WS-NUM2
  ELSE IF WS-OP EQUAL TO "/" THEN
    CALL 'DIVI' USING WS-NUM1 WS-NUM2
  ELSE IF WS-OP EQUAL TO "-" THEN
    CALL 'SUBTRAKTION' USING WS-NUM1 WS-NUM2
  END-IF.
  DISPLAY WS-NUM2.
  STOP RUN.
  END PROGRAM TASCHENRECHNERHAUPT.
```

Das Unterprogramm ADDITION:

```
IDENTIFICATION DIVISION.
  PROGRAM-ID. ADDITION.
DATA DIVISION.
  LINKAGE SECTION.
  01 WS-NUM1 PIC 9(2).
  01 WS-NUM2 PIC 9(2).
PROCEDURE DIVISION USING WS-NUM1 WS-NUM2.
  ADD WS-NUM1 TO WS-NUM2
  END-ADD.
  EXIT PROGRAM.
```

Das Unterprogramm DIVI:

```
IDENTIFICATION DIVISION.
  PROGRAM-ID. DIVI.
DATA DIVISION.
  LINKAGE SECTION.
  01 WS-NUM1 PIC 9(2).
  01 WS-NUM2 PIC 9(2).
PROCEDURE DIVISION USING WS-NUM1 WS-NUM2.
  DIVIDE WS-NUM1 INTO WS-NUM2.
  EXIT PROGRAM.
```

Das Unterprogramm MULTIPLIKATION:

```
IDENTIFICATION DIVISION.
  PROGRAM-ID. MULTIPLIKATION.
DATA DIVISION.
  LINKAGE SECTION.
  01 WS-NUM1 PIC 9(2).
  01 WS-NUM2 PIC 9(2).
PROCEDURE DIVISION USING WS-NUM1 WS-NUM2.
  MULTIPLY WS-NUM1 BY WS-NUM2.
```

```
  EXIT PROGRAM.
```

Das Unterprogramm SUTRAKTION:

```
IDENTIFICATION DIVISION.
  PROGRAM-ID. SUTRAKTION.
DATA DIVISION.
  LINKAGE SECTION.
  01 WS-NUM1 PIC 9(2).
  01 WS-NUM2 PIC 9(2).
PROCEDURE DIVISION USING WS-NUM1 WS-NUM2.
  SUBTRACT WS-NUM1 FROM WS-NUM2.
  EXIT PROGRAM.
```

Aufgaben in Abschnitt 10.2

Sie sollten ein Programm zur Kreisberechnung erstellen. Der Anwender soll zur Eingabe des Radius aufgefordert werden. Eine mögliche Lösung sieht so aus:

```
IDENTIFICATION DIVISION.
  PROGRAM-ID. CIRCLES.
DATA DIVISION.
  WORKING-STORAGE SECTION.
  01   WS-KONSTANTEN.
    05  WS-PIPIC 9V99999 VALUE 0.
  01   WS-KREIS.
    05 WS-FLAECHE PIC 999V99 VALUE ZEROES.
    05 WS-RADIUS PIC 9999V99 VALUE ZEROES.
    05 WS-KREISUMFANG PIC 99999V99 VALUE ZEROES.
PROCEDURE DIVISION.
  MOVE FUNCTION PI() TO WS-PI.
  DISPLAY "Eingabe Radius: ".
  ACCEPT WS-RADIUS.
  COMPUTE WS-KREISUMFANG = 2 * WS-PI *WS-RADIUS.
  DISPLAY "KREISUMFANG ist: ", WS-KREISUMFANG.
  COMPUTE WS-FLAECHE = WS-PI *WS-RADIUS * WS-RADIUS.
  DISPLAY "KREISFLAECHE ist: ", WS-FLAECHE.
  STOP RUN.
  END PROGRAM CIRCLES.
```

```
F:\cobolprojekte\anhang>kreisberechnung
Eingabe Radius:
20
KREISUMFANG ist: 00125.66
KREISFLAECHE ist: 000.00

F:\cobolprojekte\anhang>
```

Abb. 12.3: Kreisberechnung

Aufgaben in Abschnitt 11.7

Sie sollten ein Programm mit einer Zugangskontrolle erstellen. Das Hauptprogramm sollte einen Anwender zur Eingabe einer Userid und des Passworts auffordern. Sofern diese Daten als „legitimiert" hinterlegt sind, sollte ein bestimmter Inhalt angezeigt werden und andernfalls eine Fehlermeldung. Die Zugangsdaten selbst sollen in einer Textdatei gespeichert werden, die von dem Programm eingelesen werden muss. Das Einlesen der Daten sowie die Verifizierung der Daten

sollen jeweils in ein Unterprogramm ausgelagert werden. Eine mögliche Lösung sieht so aus:

Das Hauptprogramm entspricht dem der Übung auf Seite 56. Das Unterprogramm ZUGANG-DATENENTGEGENNAHME kann ebenso von der Übung übernommen werden und den Aufbau der Textdatei mit den Zugangsdaten finden Sie in der Aufgabenstellung.

Neu ist das Unterprogramm ZUGANGVERIFIZIERUNG:

```
IDENTIFICATION DIVISION.
  PROGRAM-ID. ZUGANGVERIFIZIERUNG.
ENVIRONMENT DIVISION.
  INPUT-OUTPUT SECTION.
  FILE-CONTROL.
  SELECT FP1 ASSIGN TO "zugangsdaten.txt"
  ORGANIZATION IS LINE SEQUENTIAL.
DATA DIVISION.
  WORKING-STORAGE SECTION.
  01 WS-DATEIINHALT PIC X(70) VALUE " ".
  01 WS-EOF PIC X VALUE "N".
  01 WS-ANZAHL PIC 99 VALUE 0.
  01 WS-GEFUNDEN PIC 9 VALUE 0.
  01 WS-ZUGANG-TEMP PIC X(9).
  LINKAGE SECTION.
  01 WS-USERID PIC X(4).
  01 WS-PASSWORT PIC X(4).
PROCEDURE DIVISION USING WS-USERID WS-PASSWORT.
  OPEN INPUT FP1.
  PERFORM UNTIL WS-EOF="Y"
  INSPECT WS-DATEIINHALT TALLYING WS-ANZAHL
  FOR ALL "#"
  IF WS-ANZAHL > 0 THEN
    STRING WS-USERID "#" WS-PASSWORT INTO
    WS-ZUGANG-TEMP
    IF WS-DATEIINHALT =  WS-ZUGANG-TEMP
      THEN ADD 1 TO WS-GEFUNDEN
    END-IF
  END-IF
  READ FP1 INTO WS-DATEIINHALT
    AT END MOVE "Y" TO WS-EOF
  END-READ
  END-PERFORM.
  CLOSE FP1.
  IF WS-GEFUNDEN > 0 THEN
    DISPLAY "Willkommen"
  ELSE
    DISPLAY "Falsche Zugangsdaten"
  END-IF.
  EXIT PROGRAM.
```

12.2 Über den Autor

Ralph Steyer ist von Diplom Mathematiker und arbeitet als freiberuflicher Trainer, Autor und Programmierer. Unter http://www.rjs.de finden Sie seine Webseite und unter http://blog.rjs.de seinen Blog. Die beruflichen Schwerpunkte liegen in der Web-Entwicklung sowie Programmierung in Java und .NET. Mit COBOL beschäftigt er sich aber auch schon seit Mitte der 90iger-Jahre. Allerdings überwiegend mit der Umstellung von COBOL-Programmen auf neuere Technologien und der Umschulung von COBOL-Programmieren zu „modernen" Programmiersprachen. Doch durch die unveränderte Bedeutung von COBOL im Banken- und Versicherungsumfeld erweiterte sich in der letzten Zeit nach und nach sein Umgang mit COBOL wieder hin zu aktiver COBOL-Programmierung.

Hier ist noch ein kurzer Abstract der berufliche Laufbahn und Erfahrungen:

- Studium bis 1990 in Frankfurt/Main an der Johann Wolfgang Goethe-Universität.
- Nach dem Studium Programmierer bei einer großen Versicherung in Wiesbaden für versicherungsmathematische PC-Programme.
- Nach knapp 4 Jahren innerbetrieblicher Wechsel in die Konzeption von Großrechnerdatenbanken.
- Seit 1996 Freelancer. Aufteilung der Arbeit in verschiedene Tätigkeitsgebiete - Fachautor, Fachjournalist, EDV-Dozent und Programmierer/Consultant.
- Zahlreiche Buchpublikationen, Videoproduktionen und Onlinetraining im IT-Bereich sowie Fachbeiträge in Computermagazinen.
- Speaker auf verschiedenen IT-Konferenzen.
- Lehrbeauftragter an der Hochschule Rhein-Main in Wiesbaden und der TH Bingen.

12.3 Abbildungsverzeichnis

12.4 Tabellenverzeichnis

Index

A

B

C

D

E

F

G

H

I

K

L

M

N

O

P

Q

R

S

T

U

V

W

Z