Erstellen, Kompilieren und Ausführen eines COBOL-Programms

In dieser Aufgabe wiederholen wir das Anlegen von Datasets (Allocate) sowie das Füllen mit Daten unter Verwendung des ISPF-Editors, Sie lernen kennen, wie man ein COBOL-Programm unter z/OS schreibt, kompiliert und ausführt.

Hinweis: Dieses Tutorial wurde unter Verwendung der Benutzer-ID "PRAK085" erstellt. In allen Dateinamen müssen Sie "PRAK085" durch ihre eigene Benutzer-ID ersetzen.

Aufgabe: ***Arbeiten Sie nachfolgendes Tutorial durch.***

1. Einführung

1.1 Entwicklungsumgebung

In diesem Tutorial werden Sie ihr erstes Hello World Programm auf dem Mainframe entwickeln und ausführen. Wir benutzen hierfür die Programmiersprache Cobol (COmmon Business Oriented Language).

Auf einem Einzelplatzrechner würden Sie vermutlich die gleiche Umgebung sowohl für die Entwicklung als auch für die Ausführung eines neuen Programms benutzen, also z. B. unter Benutzung einer bash oder korn Shell unter Linux, oder der DOS Eingabeaufforderung unter Windows. Auf einem Mainframe kann man ebenfalls Entwicklung und Ausführung beides unter TSO plus ISPF durchführen.

Ein Mainframe wird aber hauptsächlich als Server für die Ausführung von interaktiven- oder Stapelverarbeitungs-Prozessen benutzt. Deshalb werden die Umgebungen für Entwicklung und Ausführung fast immer getrennt. Die Ausführungsumgebungen für interaktive Anwendungen sind z.B. CICS, DB2, IMS Stored Procedures oder WebSphere Java Servlet und Java Enterprise Bean Prozesse. Die Ausführungsumgebung für Stapelverarbeitungsprozesse ist fast immer das Job Entry Subsystem JES.

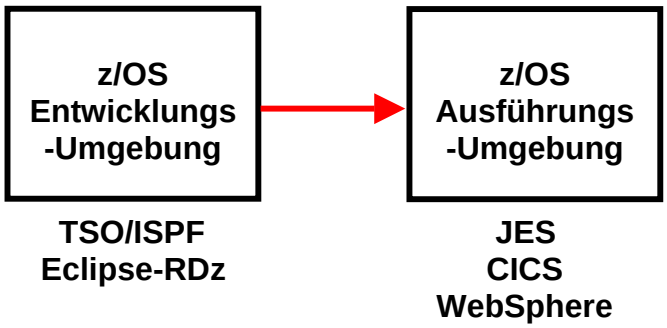


Abb. 1.1 : Entwicklungs- und Ausführungsumgebung

Die Entwicklung und anschließende Ausführung eines neuen Programms besteht damit aus 3 Schritten:

Schritt 1: Sie benutzen die Entwicklungsumgebung um Ihr neues Programm

- **zu editieren,**
- **zu übersetzen (compile),**
- **zu debuggen, und**
- **in einer Library mit anderen von Ihnen entwickelten Programmen abzuspeichern, sowie**
- **um die von Ihrem neuen Anwendungsprogramm benutzten Daten zu erstellen (oder importieren und ebenfalls in einem Data Set abzuspeichern.**

Schritt 2: Danach importieren Sie ihr neues Programm in eine Ausführungsumgebung. Ausführungsumgebungen unter z/OS sind z.B. JES (Job Entry Subsystem), CICS, DB2 Stored Procedures, IMS-DC, IMS Stored Procedures oder WebSphere, oder in seltenen Fällen TSO.

Als letzten Schritt 3 sind Sie dann in der Lage, ihr neues Programm auszuführen, indem Sie z.B. eine neue CICS Transaktion ausführen, ein (JCL oder REXX) Script für einen Stapelverarbeitungsprozess aufrufen, oder ein Programm mittels TSO und ISPF aufrufen.

Als Entwicklungsumgebung für neue Mainframe Anwendung wird heute häufig Eclipse mit der „Rational Developer für System z“ (RDz) Erweiterung eingesetzt. Wir werden dies in einem späteren Tutorial einsetzen. Aus didaktischen Gründen verwenden wir in diesem Tutorial den klassischen Ansatz, bei dem als Entwicklungsumgebung TSO und ISPF benutzt werden.

1.2 Übersetzen (Kompilieren) eines Programms

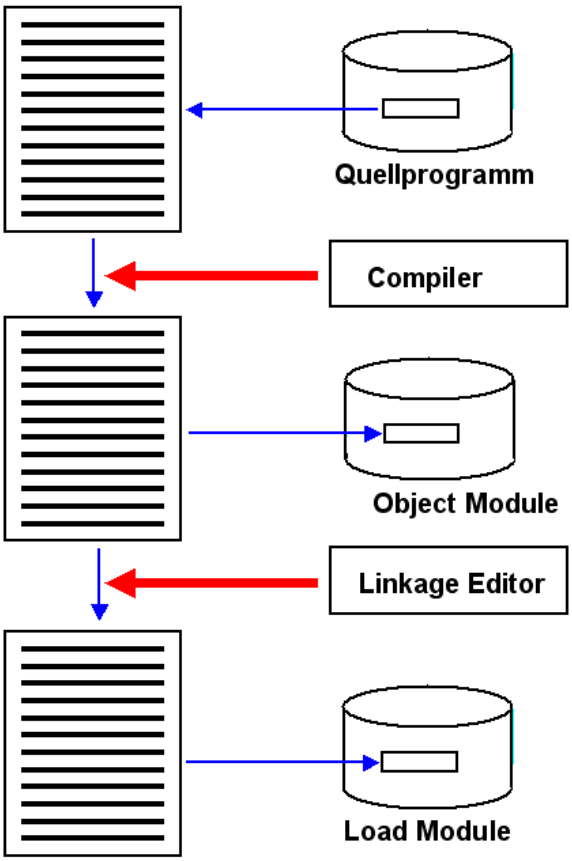


Abb. 1.2 : Übersetzungsvorgang

Ein neues Programm wird als Quellprogramm (Quelltext) mittels eines Editors in einer Programmiersprache wie z.B. Cobol, PLI, C/C++ oder Java erstellt und in einem PDSe Data Set abgespeichert. Ein PDSe Data Set stellt in vielen Fällen eine Programm-Library dar, wobei die Member einzelne Programme speichern.

Ein Compiler übersetzt das Quellprogramm in ein „Object Module“, welches ebenfalls als Member in einem PDSe Data Set gespeichert wird. Das Object Module besteht aus lauter ausführbaren System z Maschinenbefehlen. Im Prinzip könnte das Object Module in den Hauptspeicher geladen werden und dann aufgerufen und ausgeführt werden.

In der Praxis findet das jedoch so gut wie nie statt. Der Grund ist, dass ein größeres Anwendungsprogramm in der Regel aus mehreren Programm-Teilen besteht, die einzeln übersetzt, und als getrennte Object Module in den Membern eines PDSe Data Sets gespeichert werden. Die einzelnen Member werden von einem „Linkage Editor“ zu einem einzigen ausführbaren Programm, dem Load Module zusammengefügt. Eine Aufgabe des Linkage Editors ist es, die (symbolischen) Referenzen der Object Module untereinander aufzulösen und durch Hauptspeicheradressen zu ersetzen.

Weiterhin werden vom Linkage Editor vorgefertigte System Routinen eingebunden. Ein Beispiel hierfür sind die „Access Methoden“ bei der Verwendung eines VSAM Data Sets. Ein anderes Beispiel sind „Language Environment“ (LE) Routinen. Wenn Sie z.B. in einem Cobol, PLI oder C/C++ eine SQRT (Quadratwurzel) Routine aufrufen, wird diese vom Compiler nicht mit übersetzt. Statt dessen bindet der Linkage Editor eine bereits vorhandene Quadratwurzel Routine (die aus lauter Maschinenbefehlen besteht), in das Load Module ein.

Beim Aufruf des neuen Programms wird das Load Module in den Hauptspeicher geladen und dann ausgeführt.

Die Definitionen für all dies werden in einem Script in der JCL (Job Control Language) Sprache zusammengefasst und abgespeichert. Eine make File unter Linux hat eine vergleichbare Funktion.

1.3 Ausführen eines Programms.

Wenn ein Quellprogramms unter TSO/ISPF entwickelt wird, erfolgt die Übersetzung typischerweise mittels eines JES (Job Entry Subsystem) Stapelberarbeitungsprozesses. Der Auftrag hierfür wird mittels eines JCL Scripts erstellt; das Linken mehrer Object Module zu einem Load Modul durch den Linkage Editor erfordert ein weiteres JCL Script. Die Ausführung eines Load Modules ist ein davon getrennter Vorgang. Falls das Load Module in einer CICS, IMS oder WebSphere Ausführungsumgebung gestartet werden soll, ist es erforderlich, das Load Module vorher in dieser Ausführungsumgebung zu installieren.

Die Übersetzung des Quellprogramms und das Linken unter JES ist besonders einfach. Hierzu wird das entsprechende JCL Script mit Hilfe des ISPF Kommandos „submit“ (abgekürzt „sub“) an JES übergeben. JES interpretiert die einzelnen Anweisungen des JCL Scriptes und führt es aus.

Das vorliegende Tutorial würde eigentlich zwei JCL Scripts benötigen, je eines für die Compile und Linkage Edit Schritte. Unser primitives Hello World Programm ermöglicht eine Vereinfachung. Wir erstellen nur ein einziges JCL Script, welches aus der System Library ein weiteres JCL Script IGYWCL aufruft. IGYWCL bewirkt die Schritte

1. **Übersetzen eines Cobol Programms,**
2. **Link Edit**

in einem Schritt. Siehe hierzu Abschnitt 4.3 .

2. Einrichten der Entwicklungsumgebung

In dem hier vorliegenden Tutorial verwenden wir als Entwicklungsumgebung TSO und ISPF. Die Ausführungsumgebung für die Übersetzung ist JES. Spezifisch verwenden wir für die Erstellung des Cobol Quellprogramms den ISPF Editor.

2.1 Anlegen der erforderlichen Data Sets.

Wir benötigen in diesem Tutorial 3 Data Sets:

- **Einen Data Set *PRAK085.TEST.COB* für die Aufnahmen des Quelltextes unseres Cobol Programms**
- **Einen Data Set *PRAK085.TEST.LOAD* für die Aufnahmen Object Codes unseres übersetzten Cobol Programms**
- **Einen Data Set *PRAK085.TEST.CNTL* für die Aufnahmen eines in der JCL (Job Control Language) Sprache geschriebenen Scriptes für die Ablaufsteuerung. Diesen Data Set haben Sie möglicherweise in dem Tutorial 1a bereits angelegt.**

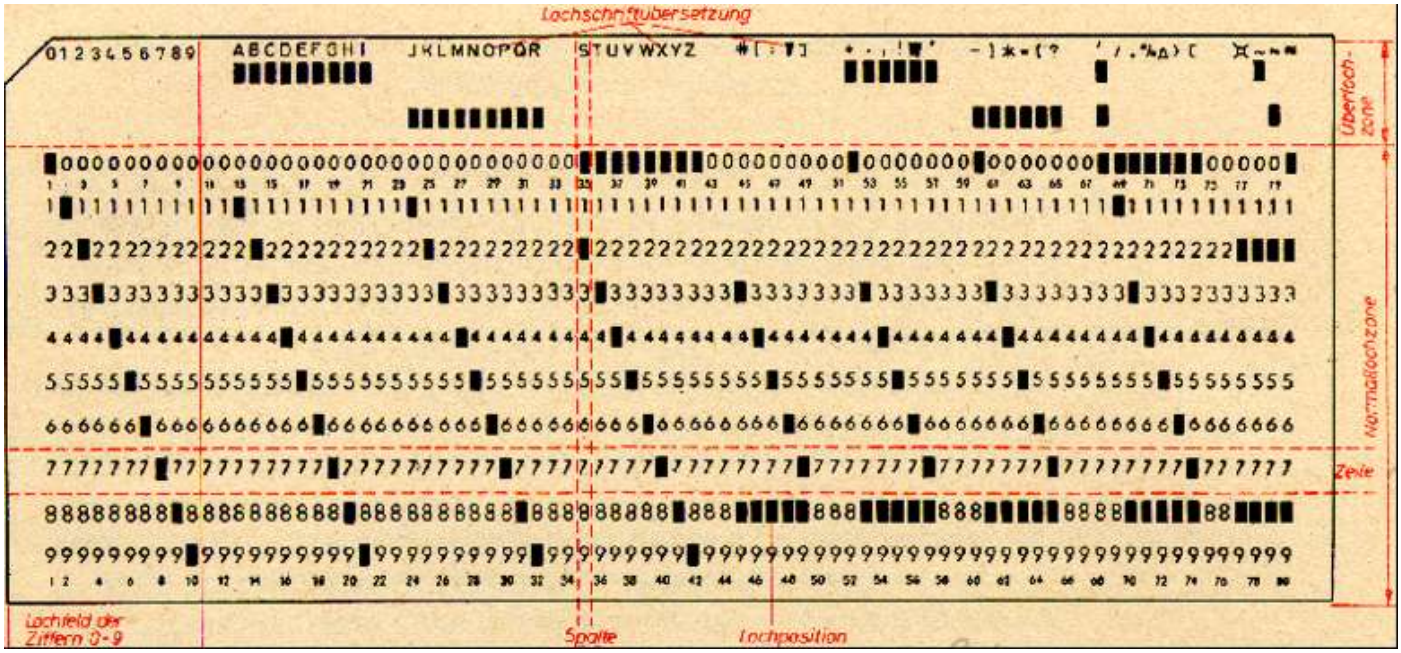
Aufgabe: *Legen Sie zwei Datasets PRAK085.TEST.COB und PRAK085.TEST.CNTL ("PRAK085" durch Ihre Benutzer-ID ersetzen) an. Verwenden Sie die gleichen Parameter wie im Tutorial zur Aufgabe 1.*

Legen Sie den Dataset PRAK085.TEST.LOAD ("PRAK085" durch Ihre Benutzer-ID ersetzen) an, welcher die ausführbare Datei nach dem Kompilieren aufnehmen soll. Verwenden Sie wieder die Ihnen bekannten Parameter, mit einem Unterschied: Statt im Dateiformat (Record format) "Fixed Block" soll dieser Dataset im Dateiformat "Undefined" erstellt werden. Dazu ist an der dafür vorgesehenen Stelle ein "U" als Parameter anzugeben.

Verifizieren Sie, dass diese drei Datasets existieren.

2.2 Spaltenabhängigkeit

Bei der Benutzung des ISPF Editors tritt ein als „Spaltenabhängigkeit bezeichnetes Phänomen auf. Ohne Wissen der historischen Entwicklung ist es unmöglich, dies zu verstehen.



1 Abb. 2.1 IBM Lochkarte

Die Entwicklung der maschinellen Informationsverarbeitung begann mit der Erfindung der Lochkarte durch Herrn Herrmann Hollerith, dem Gründer der Firma IBM. Abb. 2.1 zeigt das Format einer Lochkarte, etwa 19 x 8 cm groß, wie sie 1928 von IBM eingeführt und standardisiert wurde, und bis etwa 1970 in Gebrauch war. Der Vorläufer mit einem etwas anderen Format wurde von Herrmann Hollerith 1890 bei der Volkszählung in den USA und wenig später auch in Deutschland eingesetzt.

Die „IBM Lochkarte“ besteht aus einem Stück biegsamer Karton, in welches Löcher gestanzt werden. Sie hat insgesamt 80 Spalten. In jede Spalte konnte an einer oder an zwei von 12 Stellen jeweils ein Loch gestanzt werden. Die Position konnte in einem Lochkartenleser abgefühlt werden. Damit war es möglich, mittels der sog. BCD Kodierung (Binary Coded Decimal) in jeder Lochkarte 80 alphanumerische Zeichen zu speichern.

Zahlreiche Abbildungen von Lochkarten sind zu finden unter http://www.google.de/search?q=punched+card&hl=de&lr=&as_qdr=all&prmd=imvns&tbm=isch&tbo=u&source=univ&sa=X&ei=wiqlUMu4FobV4QSk0oHICA&ved=0CCoQsAQ&biw=1516&bih=963.

Bis zur Mitte des 20. Jahrhunderts war die Lochkarte das dominierende Medium zur Speicherung von relativ großen digitalen Datenmengen. Ein Lochkartenstapel mit 2000 Karten, etwa 35 cm hoch, hat eine Speicherkapazität von etwa 160 000 Bytes.

Cobol, Fortran und JCL Programme speicherten jeweils ein Statement pro Lochkarte. Hat das Statement eine Länge von < 80 Bytes, bleibt der Rest des Platzes auf der Lochkarte ungenutzt. Ist die Länge > 80 Bytes, benutzt man zwei oder mehr Lochkarten für ein Statement. Die Kennzeichnung von 2 oder mehr Karten pro Statement geschieht durch eine Lochung in einer spezifischen Spalte.

Cobol entstand 1960. Um das Parsing eines Statements zu erleichtern, bestand die Konvention, dass die Label in Spalte 7 und die Befehle in Spalte 12 beginnen müssen. Diese Spaltenabhängigkeit wird auch von der heutigen Version des ISPF Editors und dem dazugehörigen Cobol Compiler vorgeschrieben. Bei anderen Sprachen, z.B. JCL, bestehen ähnliche Einschränkungen. In anderen Cobol Entwicklungsumgebungen und deren Compilern bestehen diese Einschränkungen nicht.

Der heute von Mainframes verwendete Extended Binary Coded Decimal Interchange Code (EBCDIC, ausgesprochen ebsidik) ist eine 1:1 Umsetzung der BCD Lochkartenstanzungen in einen 8 Bit Code.

3. Erstellen des Quelltextes des COBOL-Programms

Unser Hello World Tutorial ist für die vier Programmiersprachen C/C++, Cobol, PL/1 und Assembler verfügbar. In dem hier vorliegenden Tutorial wird mit Cobol Quellcode gearbeitet.

Cobol ist die auf Mainframes am weitesten verbreitete Programmiersprache. Auf anderen Plattformen wird Cobol ebenfalls, aber nicht so häufig wie C/C++ oder Java eingesetzt. Eine signifikante Anzahl neuer Anwendungen wird jedes Jahr in Cobol entwickelt.

Es existieren zahlreiche Cobol Compiler von unterschiedlichen Herstellern. Der wichtigste ist der „z/OS Enterprise Cobol Compiler“, den wir hier benutzen.

Zwei ebenfalls weit verbreitete Cobol Compiler sind:

- **Microfocus Cobol Compiler: http://www.microfocus.com/mcro/cobol/index.aspx**
- **Fujitsu Cobol Compiler: http://www.netcobol.com/products/COBOL-Compilers-and-Development-Tools/overview**

Zwei kostenlose Open Source Cobol Compiler sind:

- **Tiny COBOL: http://tiny-Cobol.sourceforge.net/**
- **OpenCOBOL: http://www.opencobol.org/**

Ein COBOL-Programm ist in Teile (DIVISION), Kapitel (SECTION) und Abschnitte (PARAGRAPH) gegliedert.. Die vier DIVISIONs sind in ihrer festgelegten Reihenfolge:

- **Identification Division mit dem Programmnamen und Kommentaren,**
- **Environment Division, wo Schnittstellen zum Betriebs- und Dateisystem definiert werden,**
- **Data Division mit der Definition der Programmvariablen und Datenstrukturen und**
- **Procedure Division mit dem eigentlichen Code prozeduralen Anweisungen.**

Die ENVIRONMENT und DATA DIVISIONs können unter Umständen ganz entfallen.

Hieraus folgt eine strikte Trennung von Datendeklaration und prozeduralen Anweisungen, durch die sich COBOL auszeichnet. In der Procedure Division kann man nur Variablen benutzen, die vorher in der Data Division deklariert worden sind.

Cobol ist case insensitive. Historisch haben viele Cobol Programme nur Großbuchstaben verwendet. Cobol gilt als eine sehr produktive Sprache für die Entwicklung und anschließende Wartung von Programmen.

Wir benötigen insgesamt 3 Datasets. Der erste Dataset soll den Quellkode des COBOL-Programms aufnehmen, der zweite die ausführbare Datei. Einen dritten Dataset "PRAK085.TEST.CNTL" nimmt das JCL-Script auf und wird zum Kompilieren benutzt.

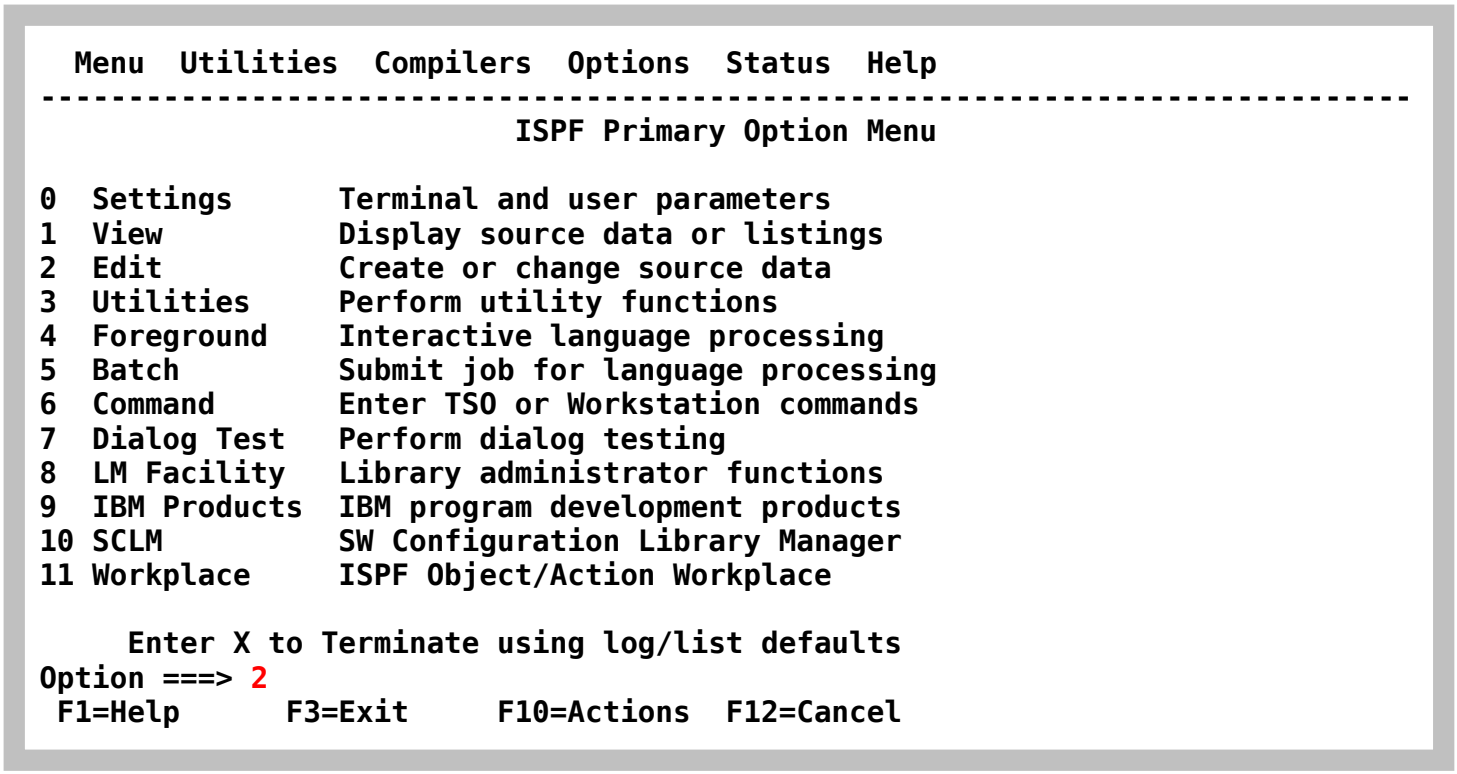


Abb. 3.1: "ISPF Primary Option Bildschirm"

Wir haben bisher die Utilities-Funktion benutzt, um unsere Entwicklungsumgebung anzulegen. Hierzu haben wir drei Partitioned Datasets angelegt. Jetzt wollen wir diesen Speicherplatz benutzen, um ein Programm zu schreiben, zu übersetzen und auszuführen. Dies geschieht mit Hilfe der "Edit"-Funktion. Wie in Abb. 3.1 dargestellt, geben wir eine "2" in die Kommandozeile des "ISPF Primary Option Menu" ein und betätigen die Eingabetaste.

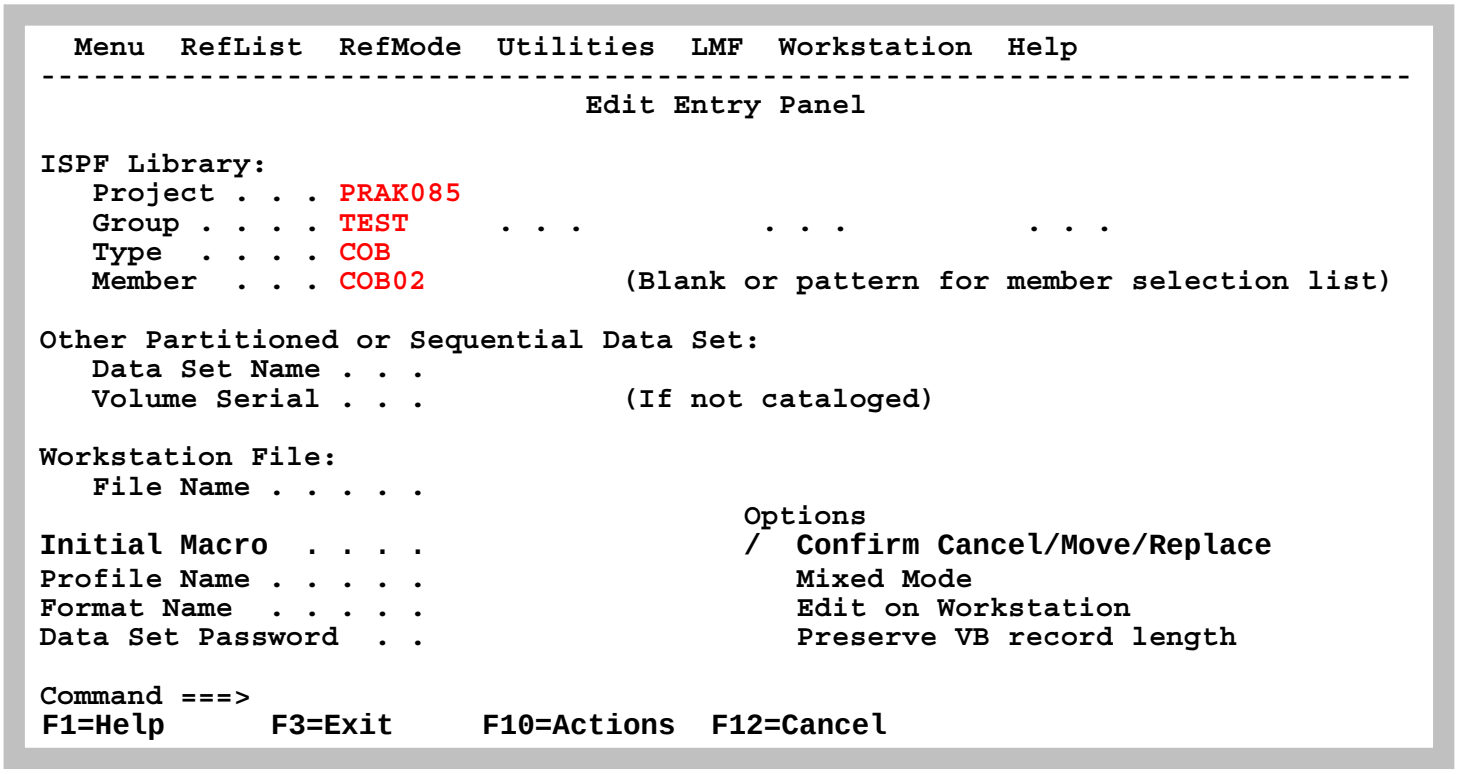


Abb. 3.2 "Edit Entry"-Bildschirm

Wir wollen zuerst das Cobol Quellprogramm mit Hilfe des ISPF-Editors erstellen. Der "Edit Entry"-Bildschirm fordert uns auf, den Namen des zu editierenden Programms einzugeben (s. Abb. 3.2).

Unser Quellprogramm soll als eine (von potentiell mehreren) File in dem für Quellprogramme von uns vorgesehenen Partitioned Dataset PRAK085.TEST.COB gespeichert werden. Files innerhalb eines Partitioned Datasets werden als Members bezeichnet. Zur Unterscheidung brauchen die einzelnen Members einen Namen.

Wir bezeichnen unseren Member als COB02. Der volle Name dieses Members ist PRAK085.TEST.COB.(COB02). Wir geben diese Werte in die dafür vorgesehenen Felder des "Edit Entry"-Bildschirmes ein. Es ist also wie in Abb. 3.2 gezeigt, ihre Benutzer-ID ins Feld "*Project*", "TEST" ins Feld "*Group*", "COB" ins Feld "*Type*" sowie "COB02" ins Feld "*Member*" einzutragen. Anschließend betätigen Sie die Eingabetaste.

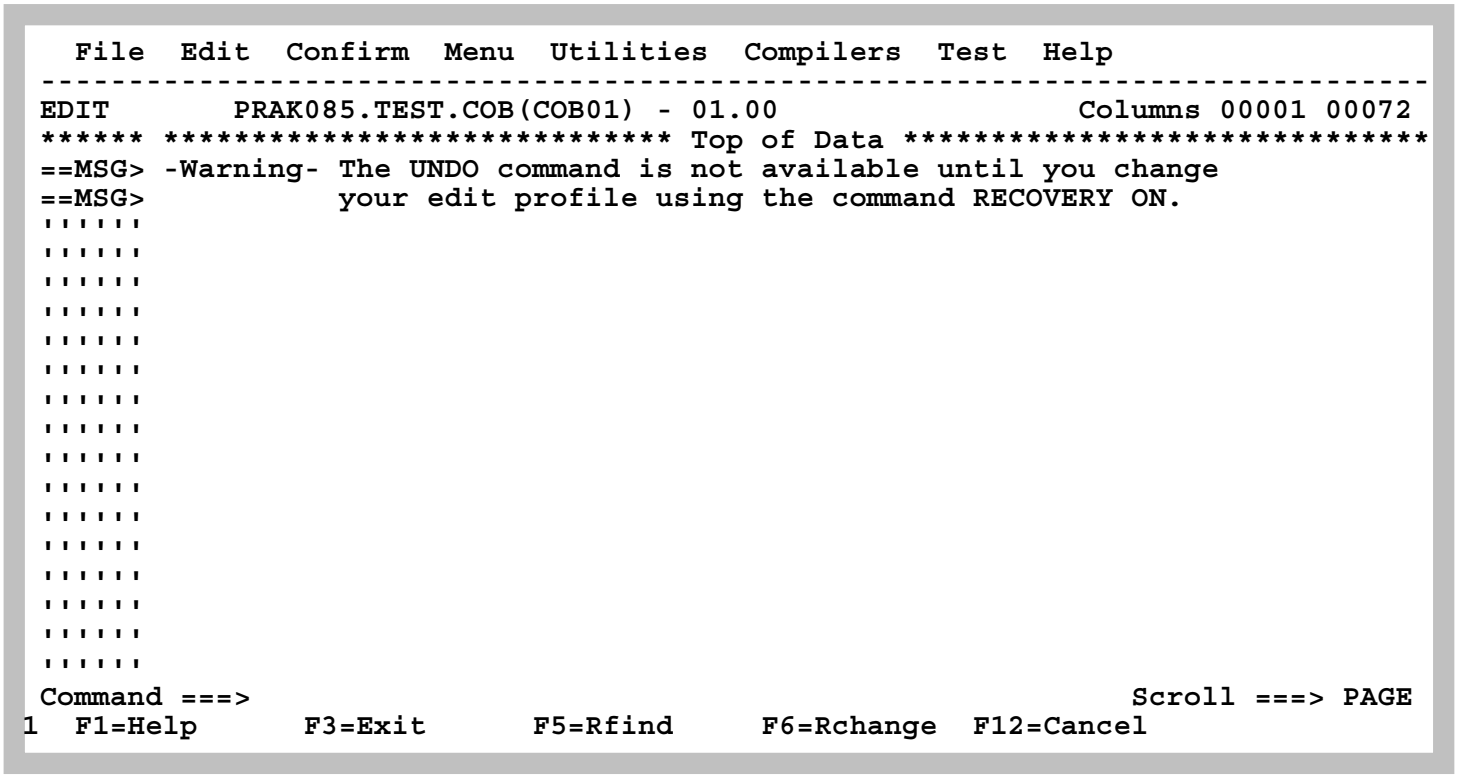


Abb. 3.3: ISPF-Editor

Abb. 3.3 zeigt die Oberfläche des ISPF-Editors. Für die Eingabe des Programmtextes benutzen wir nur die erforderlichen alphanumerischen Tasten. Wir verwenden keine Tasten zur Steuerung außer der DEL-Taste. Wir bewegen den Cursor mit Hilfe der Pfeiltasten.

Zusätzliche ISPF Editor Funktionen sind in dem Tutorial 1.b , ISPF Editor, beschrieben.

An dieser Stelle empfehlen wir Ihnen dringend, aus dem Tutorial 1b, ISPF Editor, die kurzen Abschnitte

- **5.1 Text verbergen,**
- **5.2 Spalten anzeigen**
- **5.3 Hex Darstellung**

sich anzusehen und/oder zu wiederholen-

```
000100         IDENTIFICATION DIVISION.
000200         PROGRAM-ID. COB02.
000300         ENVIRONMENT DIVISION.
000400         INPUT-OUTPUT SECTION.
000410
000500         FILE-CONTROL.
000600             SELECT PRINTOUT
000700                    ASSIGN TO SYSPRINT.
000800         DATA DIVISION.
000900         FILE SECTION.
001000         FD      PRINTOUT
001100                 RECORD CONTAINS 80 CHARACTERS
001200                 RECORDING MODE F
001300                 BLOCK CONTAINS 0 RECORDS
001400                 LABEL RECORDS ARE OMITTED.
001500         01  PRINTREC         PIC X(80).
001600         WORKING-STORAGE SECTION.
001610
001700         LINKAGE SECTION.
001800         PROCEDURE DIVISION.
001900         anfang.
002000             OPEN OUTPUT PRINTOUT.
002100             move 'Hallo Welt, unser erstes TSO-PROGRAMM in COBOL' to prin
002200        -    trec.
002300             WRITE PRINTREC.
002400             CLOSE PRINTOUT.
002500             GOBACK.
002600
```

Minuszeichen beachten, der Name der Variablen ist „printrec“

Abb. 3.4

Jedes Cobol Programm besteht aus 4 „Sections“ : Identtification Division, Environment Division, Data Division und Procedure Division. Die Divisions können wiederum aus mehreren Sections bestehen.

Wir erstellen das in Abb, 3.4 gezeigte Cobol Programm. Zum besseren Verständnis sind die Divisions und Sections farbig markiert.

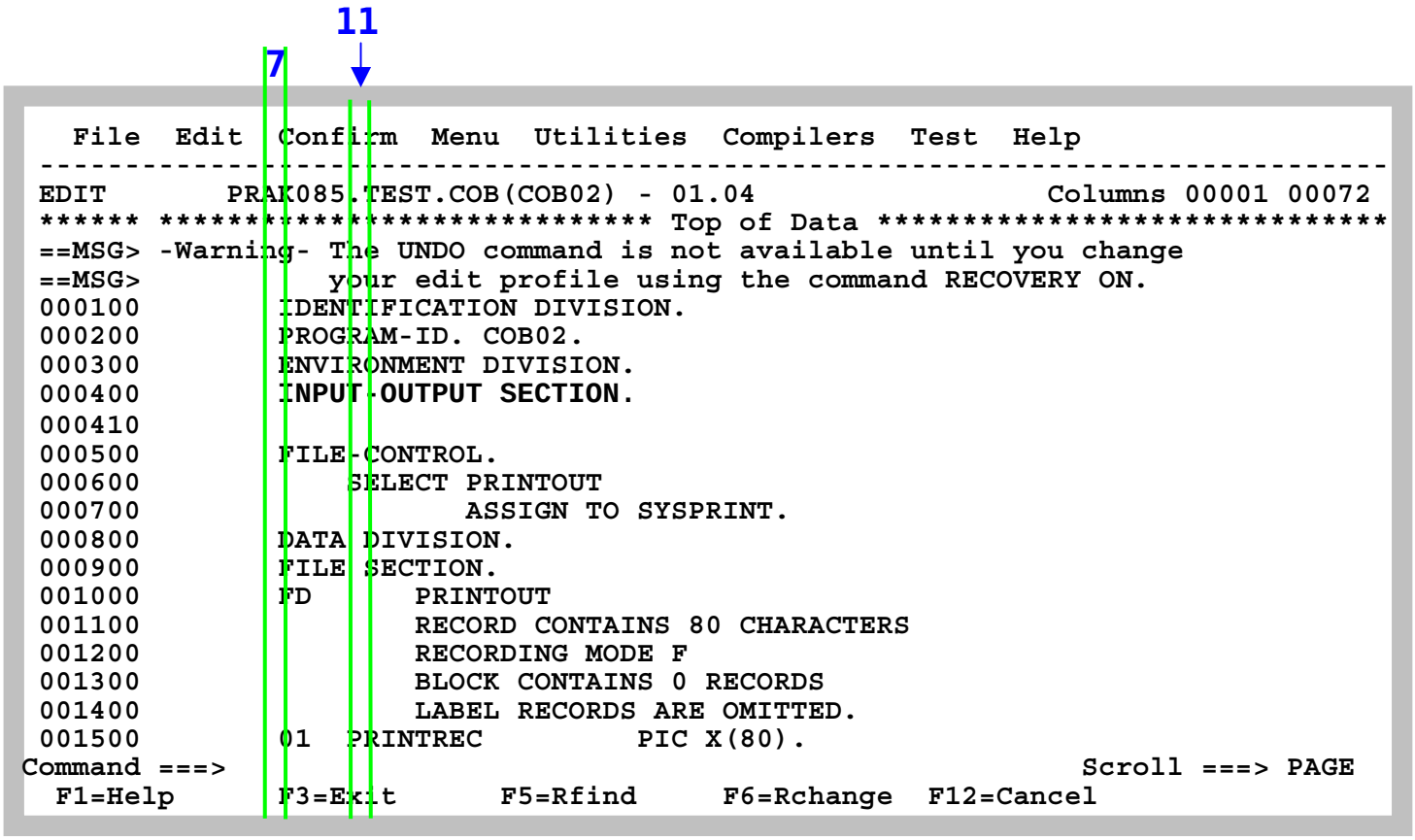


Abb. 3.5: ISPF-Editor mit COBOL-Programm (1/2)

Die Spalten 7 und 12 sind in Abb. 3.5 mit grünen Linien markiert.

Cobol Quellprogramme sind unter ISPF genauso spaltenabhängig wie JCL Scripts. Spezifisch sind:

Spalten	**1-6**	**Numerierung**
Spalte	**7**	**Zeilenkennzeichen**
		blank: Leerzeichen
		- Folgezeile
		*** Kommentar**
		/ Kommentar +Seitenwechsel
Spalte	**8-11**	**Bereich A (Überschriften = Label)**
Spalte	**12-72**	**Bereich B (Befehle)**
Spalte	**73-80**	**frei**

Hier ist vielleicht das COLS Kommando nützlich, siehe Tutorial 1b, Abschnitt 5.2 .

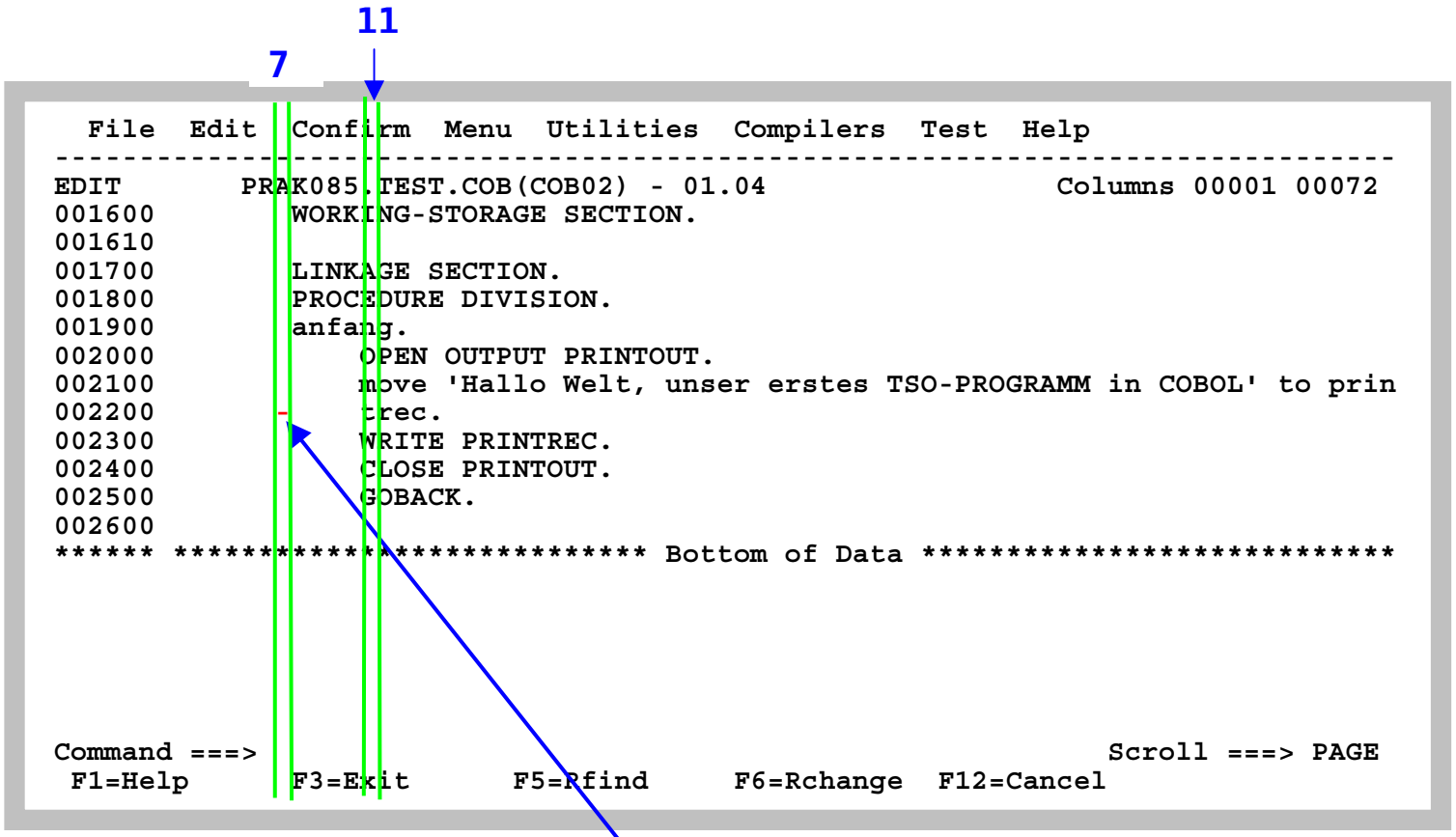


Abb. 3.6: ISPF-Editor mit COBOL-Programm, Fortsetzung (2/2)

Abbildung 3.6 ist die Fortsetzung von Abb. 3.5. Die Spalten 7 und 12 sind in Abbildung 3.6 mit grünen Linien markiert. Beachten sie in Zeile 002200 das Minuszeichen in Spalte 7, welches eine Fortsetzung von Zeile 002100 markiert !!!

Abb. 3.5 und Abb. 3.6 zeigen die beiden Teile des Programms im ISPF Fenster. Durch Betätigen der F3-Taste kehren wir zum vorherigen Bildschirm zurück. Unser Programm wird automatisch abgespeichert (saved).

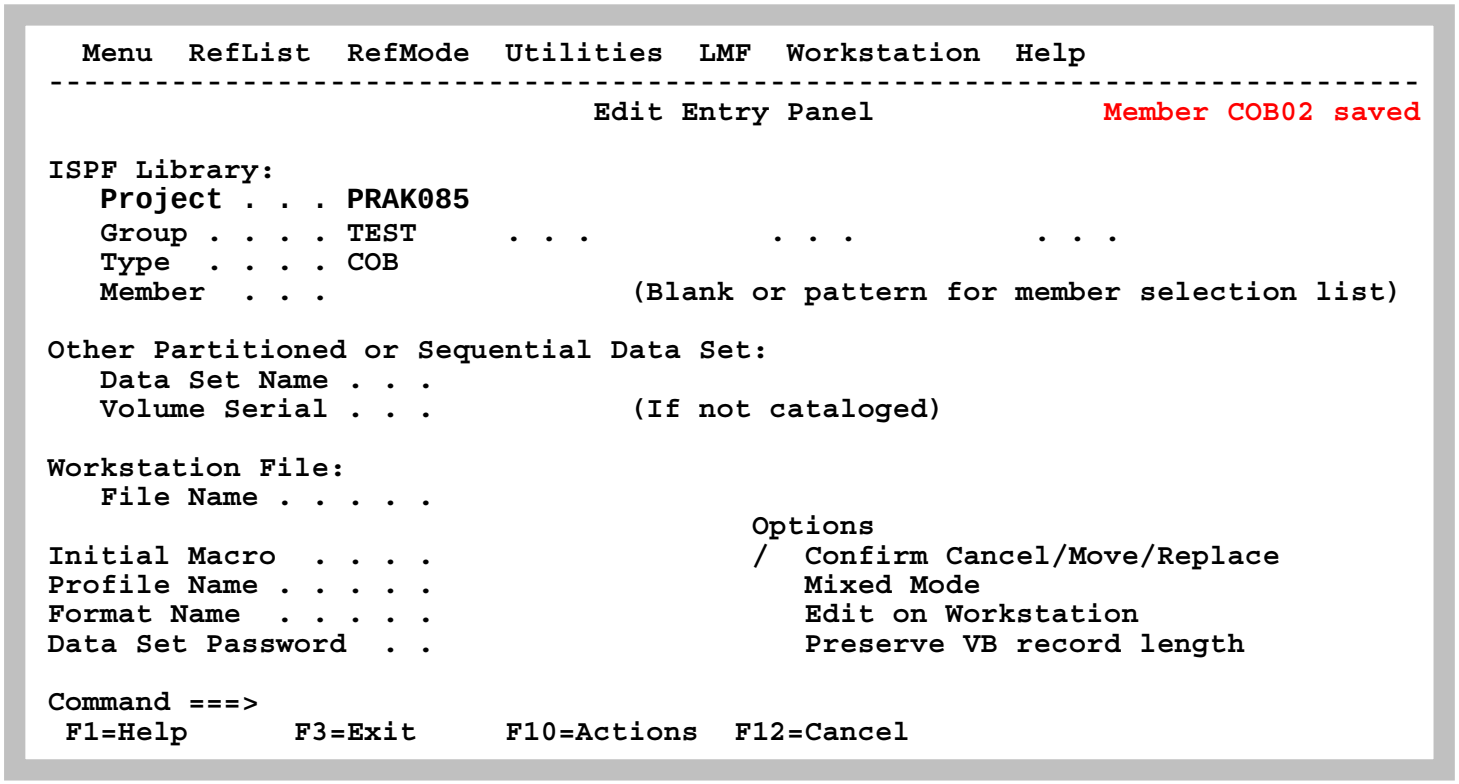


Abb. 3.7: "Edit Entry Panel"-Bildschirm

Rechts oben erscheint die Meldung, dass unser Member abgespeichert wurde (Abb. 3.7).

Durch erneutes Eingeben des Member-Namens sowie durch Bestätigen mit der Eingabetaste lässt sich unser Member nochmals aufrufen und bei Bedarf modifizieren.

Literatur zum Thema Cobol Programmierung:

Tutorial Mainframe
http://www.mainframegurukul.com/tutorials/programming/cobol/cobol-tutorial.html

S/360 Tutprial
http://www.mainframes360.com/2009/08/cobol-tutorial-introduction-writing.html

1 COBOL Programming Course
http://www.csis.ul.ie/cobol/course/Default.htm

4. Erstellen und Ausführung des JCL-Scriptes

4.1 Job Control Language

Scriptsprachen sind Programmiersprachen, deren Ziel es ist, Anweisungsfolgen oder kleinere Anwendungen zu realisieren. Programme, die in Skriptsprachen geschrieben sind, werden auch Skripte oder Scripts genannt. Microsoft verwendet häufig die Bezeichnung Makro. Scripte werden vielfach nicht von einem Compiler in maschinenlesbaren Code übersetzt, sondern zur Laufzeit von einem Interpreter ausgeführt.

Unter z/OS dominieren die beiden Scriptsprachen JCL (Job Control Language) und REXX.

REXX ist eine recht normale Scriptsprache, vergleichbar mit Perl, PHP, Python, Ruby oder Tcl.

JCL ist dagegen – sagen wir – anders. Für viele Neulinge ist es ein Kulturschock.

JCL enstammt dem Lochkartenzeitalter. Es hat eine ungewöhnliche Syntax und ähnliche Spaltenabhängigkeiten wie Cobol unter dem ISPF Editor.

Die positiven Eigenschaften sind: JCL ist sehr mächtig, und ist, nachdem man über die Anfangsschwierigkeiten hinweg ist, sehr leicht erlernbar. Programmierer, die sich einmal mit JCL auseinandergesetzt haben, werden sehr schnell zu gläubigen Bekennern. Für alle von IBM unterstützen Mainframe Sprachen existieren immer zwei Handbücher: User's Guide und Reference Manual. Die JCL User's Guide und Reference Manual sind im Umfang deutlich kürzer als die äquivalenten Dokumente für andere Mainframe Sprachen.

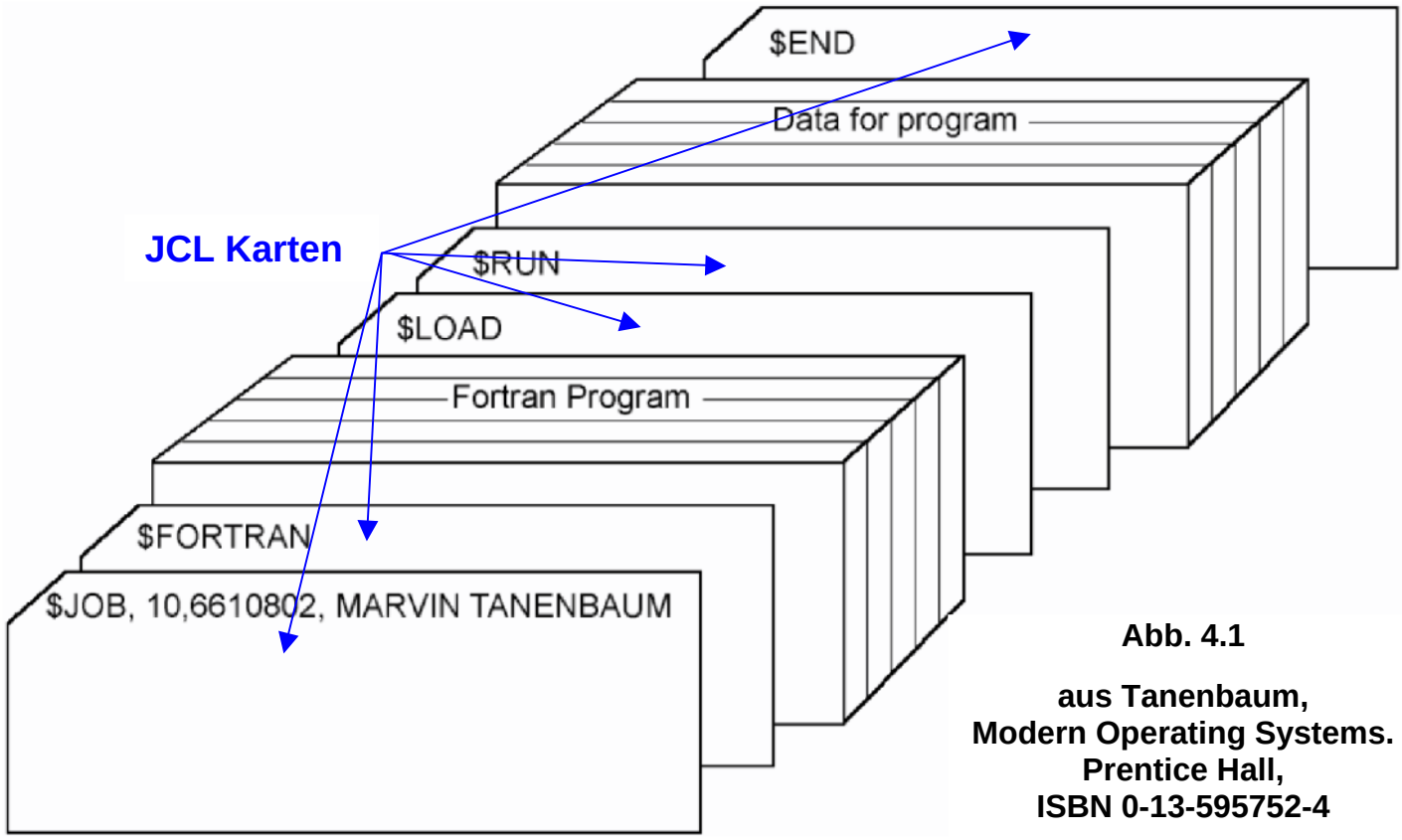


Abb. 4.1

**aus Tanenbaum,
Modern Operating Systems.
Prentice Hall,
ISBN 0-13-595752-4**

Gezeigt ist die Implementierung eines FORTRAN Programms etwa Anno 1960. Es besteht aus einer Reihe von JCL Karten, ein JCL Statement pro Karte. Zwischen den JCL Karten befindet sich das FORTRAN Programm Karten Deck, jeweils eine Karte pro FORTRAN Statement, sowie ein weiteres Karten Deck mit den von dem vom FORTRAN Programm zu verarbeitenden Daten. Das gesamte Kartendeck wurde von dem Lochkartenleser des Rechners in den Hauptspeicher eingelesen und verarbeitet. Die Ausgabe erfolgte typischerweise mittels eines angeschlossenen Druckers.

Später wurde es üblich, das FORTRAN Programm und die Daten auf Magnetband (und noch später auf einem Plattenspeicher) zu speichern, und das Programmdeck und das Datendeck durch zwei Library Call Lochkarten zu ersetzen. Noch später speicherte man dann das JCL Script ebenfalls in einer Library auf dem Plattenspeicher.

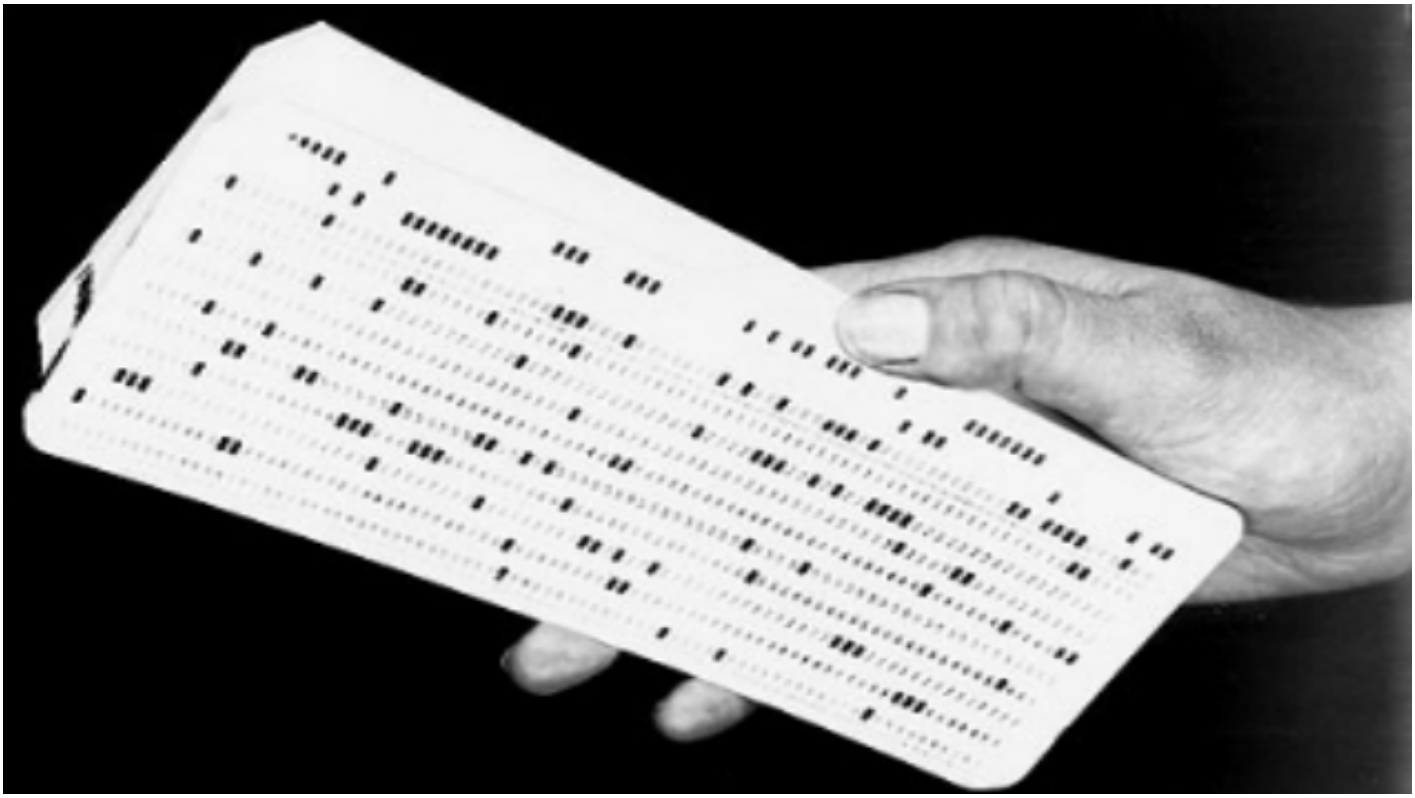


Abb. 4.2 Kartendeck

Hier hält ein Programmierer in der Hand ein Deck mit etwa 100 Lochkarten, die ein Programm enthalten.

4.2 JCL Format

Ein JCL Script besteht aus einzelnen JCL Statements. Ein JCL Statement passte früher auf eine Lochkarte mit 80 Spalten, und an der Länge von 80 Bytes hat sich bis heute nichts geändert.

Wie auch bei anderen Programmiersprachen besteht ein JCL Statement aus genau fünf) Feldern.

/ /	Name	Operation	Parameter	Kommentar

Abb. 4.2
Felder eines JCL Statements

1. **// (zweimal Slash) in Spalte 1 und 2**
2. **Label Feld (Name), bis zu 8 Zeichen lang, beginnt in Spalte 3**
3. **Statement Type (auszuführende Operation)**
4. **Parameter**
5. **Kommentar**

Um die Logik zum Parsen eines Statements zu vereinfachen, führte man einige Konventionen ein:

- **Jedes Feld beginnt immer an einer bestimmten Spalte der Lochkarte. Das vereinfachte damals das Parsen des Statements. Diese Spaltenabhängigkeit existiert auch heute noch !!! Wenn Sie dagegen verstoßen, gibt es eine Fehlermeldung. Warnung: Dies ist mit großem Abstand der häufigste Programmierfehler beim Bearbeiten unserer ersten Mainframe Tutorials.**
- **Jedes JCL Statement beginnt mit den beiden Zeichen / / (forward slashes). Damit war es möglich, die JCL Statements leicht von den Lochkarten des FORTRAN Decks und des Datendecks zu unterscheiden. Die beiden Slashes befinden sich in Spalte 1 und 2.**
- **Das Namensfeld (Label) Feld beginnt in Spalte 3, die maximale Länge ist 8 Zeichen.**
- **Das Operation Feld folgt dem Namensfeld, getrennt durch ein Leerzeichen. (früher musste es in Spalte 12 anfangen, heute nicht mehr erforderlich).**
- **Das Parameter Feld (Operand Feld) folgt dem Operation Feld, getrennt durch ein (oder mehr) Leerzeichen.**
- **Alles was hinter dem Operand Feld folgt, ist ein Kommentar. Zwischen Operand und Kommentar muss sich (mindestens) ein Leerzeichen befinden.**

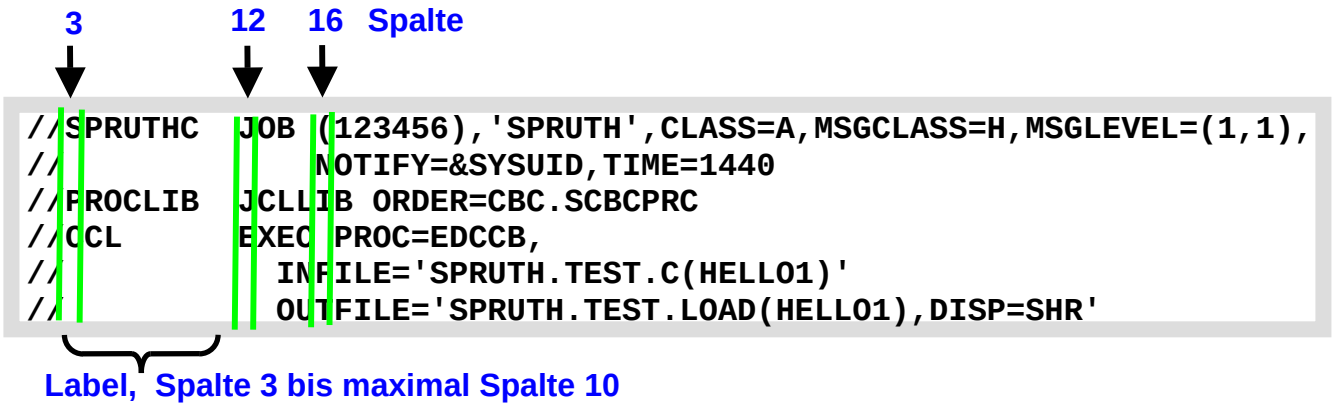


Abb. 4.3 Ein einfaches JCL Script

Unser JCL-Script macht einen reichlich kryptischen Eindruck. JCL-Scripte werden dadurch gekennzeichnet, dass alle Zeilen in Spalten 1 und 2 mit "//" beginnen.

Spalten 3 bis 10 enthalten normalerweise einen Namen (Label), der zwischen 1 und 8 Zeichen lang sein kann.

Beginnend in Spalte 12 wird die Operation spezifiziert. Einige Beispiele:

- **JOB Statement markiert den Anfang eines Jobs**
- **EXEC Statement bezeichnet Prozedur, die ausgeführt werden soll**
- **PROC Statement gibt die Adresse einer Procedur an**
- **DD Statement bezeichnet die zu benutzenden Dateien**

Auf die Operation folgt die Parameterliste in dem Parameterfeld (Operandenfeld). Jeder Parameter ist von dem folgenden Parameter durch ein Komma getrennt. Kein Leerzeichen zwischen den einzelnen Parametern.

Die Spalten 72 – 80 werden von JCL nicht berücksichtigt. Wenn Ihre Parameterliste über Spalte 71 hinausgeht, dann

1. **unterbrechen sie die Parameterliste nach dem letzten vollständigen Parameter, einschließlich des Kommas,**
2. **kodieren Sie / / in den Spalten 1 und 2 der folgenden Zeile**
3. **kodieren Sie ein Leerzeichen in Spalte 3 der folgenden Zeile. Wenn die Spalte 3 alles andere als ein Leerzeichen oder ein Asterik enthält, interpretiert JCL diese Zeile als ein neues JCL Statement.**
4. **Vollenden Sie die Parameter Liste. Die Parameter Liste soll frühestens in Spalte 4 und spätestens in Spalte 16 anfangen.**

An die Parameterliste schließt sich das Kommentarfeld an, getrennt von der Parameterliste durch ein Leerzeichen. Ebenso ist eine Zeile mit den Symbolen //* in Spalten 1, 2 und 3 ein Kommentar.

Achten Sie darauf, dass Ihr JCL Script die richtigen Spalten benutzt !!!

Unser Beispiel in Abb. 4.3 enthält drei Operationen JOB, JCLLIB und EXEC. Die Parameterliste des Job Statements erstreckt sich bis auf die zweite Zeile (Leerzeichen in Spalte 3, Fortsetzung der Parameterliste in Spalte 16. Ebenso hat das EXEC Statement eine Fortsetzung über 2 weitere Zeilen.

Beispiel für einen JCL Befehl:

```
// DCB=(RECFM=FB,LRECL=80,BLKSIZE=400)
```

RECFM, FB, LRECL und BLKSIZE sind Schlüsselwörter der JCL Sprache.

Der Befehl besagt, dass die hiermit angesprochene Datei (bzw. deren Data Control Block, DCB) ein Fixed Block (FB) Record Format (RECFM) hat (alle Datensätze haben die gleiche Länge), dessen Länge (Logical Record Length LRECL) 80 Bytes beträgt, und dass für die Übertragung vom/zum Hauptspeicher jeweils 5 Datensätze zu einem Block von (Blocksize BLKSIZE) 400 Bytes zusammengefasst werden.

4.3 DD Statement

In einem JCL Script definieren Data Definition (DD) Anweisungen die Data Sets, welche ein Programm oder eine Prozedur bei der Ausführung verwendet. Sie müssen ein DD-Statement für jeden Data Set kodieren, der während eines Auftrags Schritt verwendet oder erstellt wird.

Die Reihenfolge der DD-Statements innerhalb eines Auftrags Schrittes ist in der Regel nicht signifikant.

Diese JCL Beispiel veranschaulicht das Format eines DD-Anweisung

```
//PAY DD DSN=HLQ.PAYDS,DISP=NEW VENDOR PAYROLL
```

Das Operations Feld enthält den Eintrag DD (für Data Definition).

Das Namens-Feld enthält einen ein-bis acht-stelligen Namen (hier PAY), als ddname bekannt. Dieser kennzeichnet die DD-Anweisung, so dass andere JCL, Programme, Verfahren, oder das Betriebssystem darauf verweisen können. Der ddname in dieses DD-Anweisung ist PAY.

Das Parameter-Feld enthält zwei Keyword-Parameter:

- **DSN, welches den echten Namen eines Datensatzes identifiziert,**
- **DISP, welches HLQ.PAYDS als einen neuen Datensatz identifiziert, den das System erstellen muss, wenn dieser Job zur Verarbeitung übermittelt wird.**

Das Kommentar-Feld enthält den Eintrag VENDOR PAYROLL.

Ein DD Statement kann einen Data Set sehr umfangreich beschreiben. Es kann die folgenden Angaben enthalten:

- **Der Name, den das Programm verwendet, um den Datensatz beziehen, als ddname bekannt**
- **Der eigentliche Name des Datensatzes und seine Lokation**
- **Physische Eigenschaften des Datensatzes, wie z.B. das Record Format**
- **Die Anfangs-und Endzustand des Datensatzes, als Disposition bezeichnet**

Sie können auch DD Statements benutzen um I/O-Geräte anzufordern, oder um Speicherplatz für neue Datensätze bereitzustellen (allocate).

4.3 Erstellen und Ausführung des JCL-Scriptes

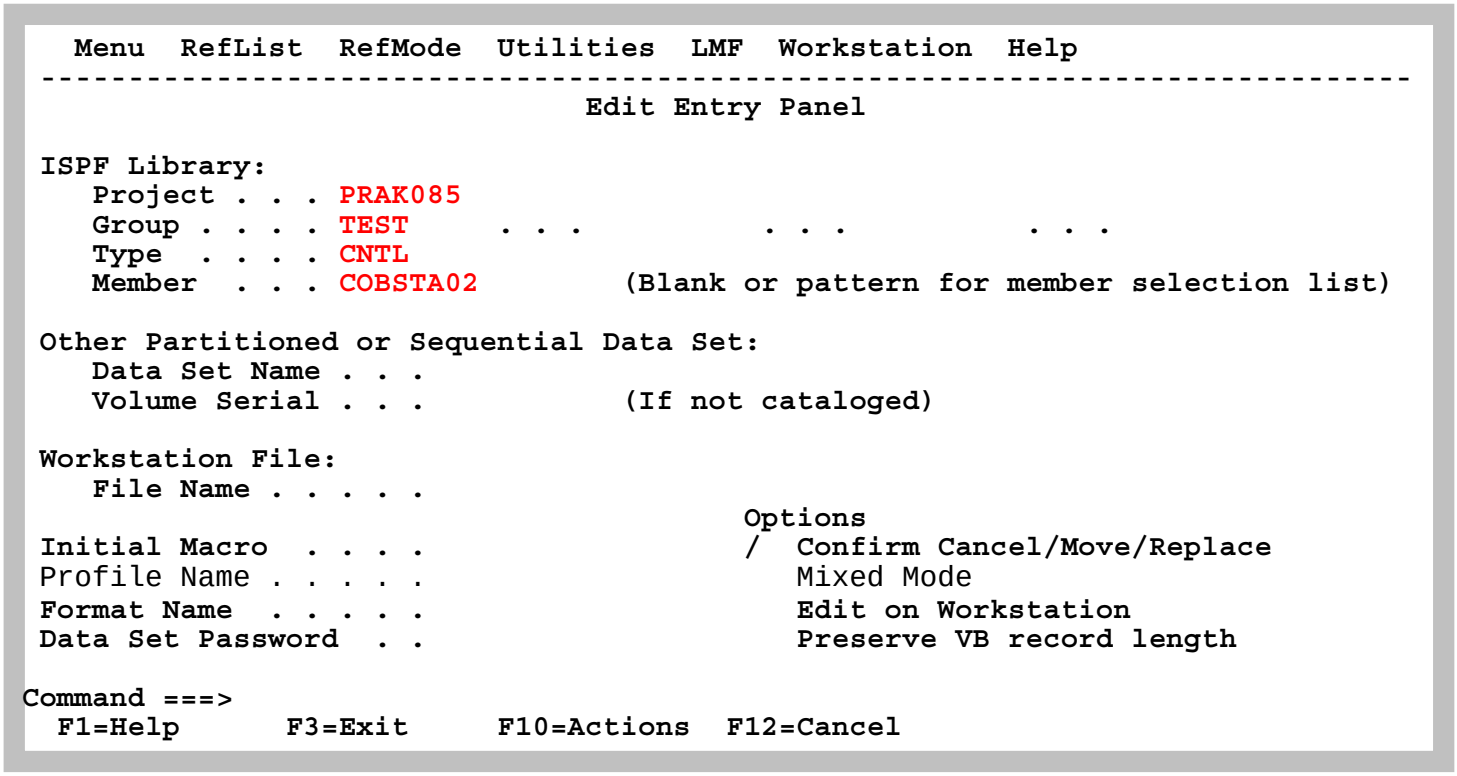


Abb. 4.4: "Edit Entry Panel"-Bildschirm

Wir legen alle "JCL Scripts" als Member in dem von uns dafür vorgesehenen Partitioned Dataset PRAK085.TEST.CNTL ab. Wie bereits erläutert, wollen wir die zwei Schritte Compile sowie Link Edit jedoch mit einem einzigen JCL Script durchführen.

Die hierfür verwendete Scriptsprache ist die "z/OS Job Control Language" (JCL).

Wir geben als Typ "CNTL" sowie als Member "COBSTA02" ein und betätigen die Eingabetaste (s. Abb. 4.4).

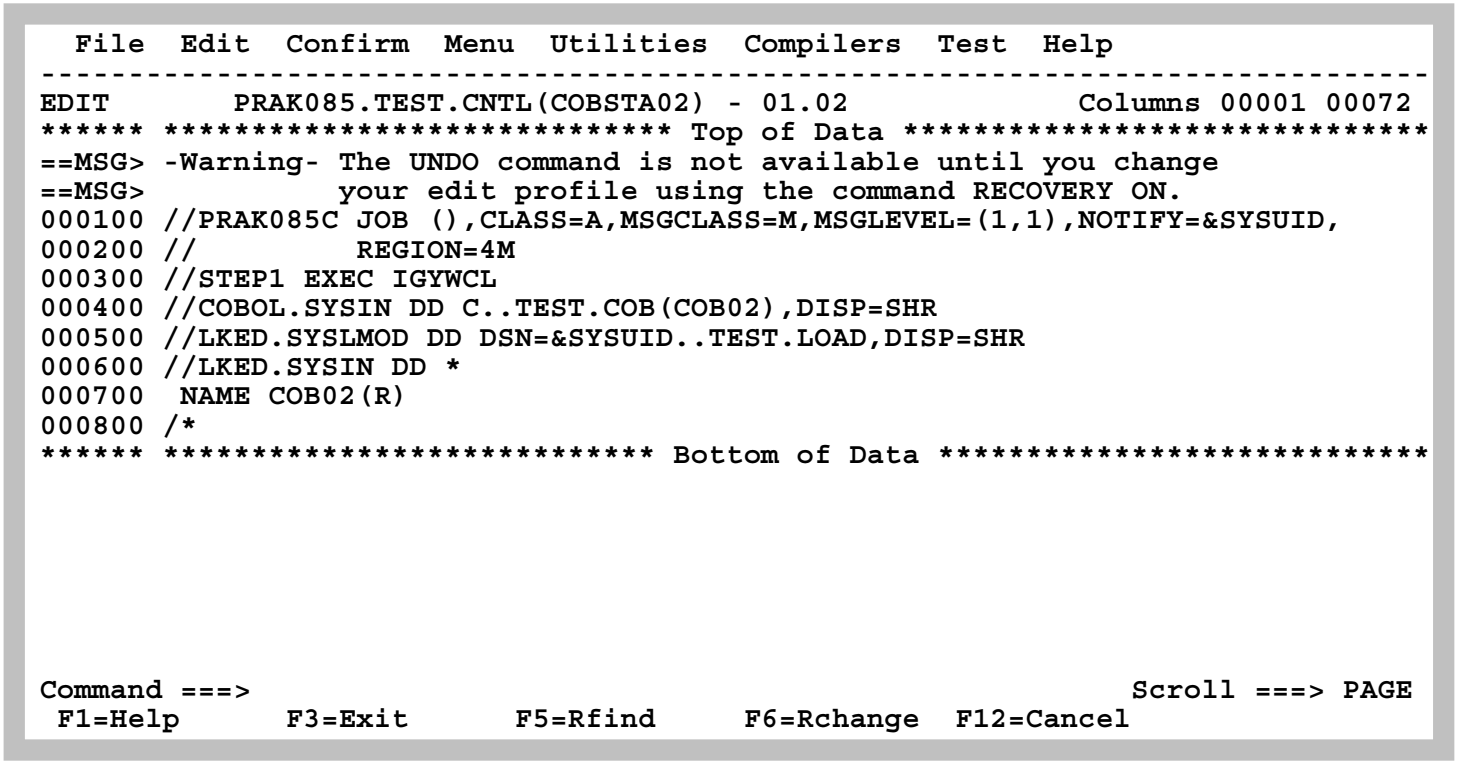


Abb.4.5: JCL-Script
Erstellen Sie das in Abb. 4.5 gezeigte JCL-Script.

Aufgabe: ***Erstellen Sie das in Abb. 4.5 gezeigte JCL-Script.***

Das erste Statement in einem JCL-Script ist immer ein "JOB" Statement (Zeile 000100 in Abb. 4.5). Es enthält eine Reihe von Dispositionsparametern, die von dem "z/OS Job Entry Subsystem" ausgewertet werden. Es ist üblich, als Label für das Job-Statement die TSO-Benutzer-ID (hier "PRAK085") plus einen angehängten Buchstaben (hier C) zu verwenden. Das hierfür vorgesehene Feld hat eine Länge von maximal 8 Zeichen. Aus diesem Grund haben TSO-Benutzer-ID's eine maximale Länge von 7 Zeichen.

Das zweite Statement (Zeile 000300 in Abb. 4.5) unseres Scripts ist ein EXEC Statement. Es enthält die Anweisung, die Prozedur "IGYWCL" abzuarbeiten. " IGYWCL ist ein von z/OS zur Verfügung gestelltes Script (Catalogued Procedure), welches

- **den COBOL-Compiler aufruft,**
- **anschließend den Linkage-Editor aufruft,**
- **den zu übersetzenden Quelltext als Member eines Partitioned Datasets (in unserem Fall) mit dem Namen PRAK085.TEST.COB(COB02) erwartet.**
- **das erstellte Maschinenprogramm (in unserem Fall) unter PRAK085.TEST.LOAD abspeichert.**

Es existiert eine große Anzahl derartiger vorgefertigter Scripte, die zusammen mit z/OS ausgeliefert werden. Der Systemadministrator stellt sie in "JCL Libraries" (JCLLIB) zusammen. z/OS ist ein sehr großes und sehr flexibles System. Es existieren häufig mehrere JCL Libraries. Was, wie und wo ist von einer Installation zur nächsten oft verschieden und wird vom Systemadministrator verwaltet.

Wenn Sie unter z/OS einen Cobol Compiler installieren, gehört sowie eine ganze Reihe weiterer Prozeduren mit zum Lieferumfang. Das Handbuch „Enterprise COBOL for z/OS Programming Guide Version 4 Release 1, SC23-8529-00, December 2007“ enthält in Kapitel 14 auf Seite 249 eine Beschreibung dieser Prozeduren.
Siehe http://www.cedix.de/VorlesMirror/Band3/ProgGuidezOS.pdf

Das JCL Script entnimmt den High Level Qualifier des Cobol Programms PRAK085.TEST.COB(COB02) der Zeile 000100 des JCL Scripts. Der Rest des Namens wird in Zeile 000400 angegeben. Zeile 000500 definiert den File Namen, unter dem das übersetzte Programm abgespeichert wird PRAK085.TEST.LOAD(COB02). Es wird unterstellt, dass der Name des Members unverändert bleibt. IGYWCL benutzt hierfür den Namen SYSLMOD.

```
//IGYWCL PROC  LNGPRFX='IGY.V4R1M0',SYSLBLK=3200,
//             LIBPRFX='CEE',
//             PGMLIB='&&GOSET',GOPGM=GO
//*
//*  CALLER MUST SUPPLY //COBOL.SYSIN DD . . .
//*
//COBOL  EXEC PGM=IGYCRCTL,REGION=2048K
//STEPLIB  DD  DSNAME=&LNGPRFX..SIGYCOMP,                (1)
//             DISP=SHR
//SYSPRINT DD  SYSOUT=*
//SYSLIN   DD  DSNAME=&&LOADSET,UNIT=SYSDA,
//             DISP=(MOD,PASS),SPACE=(TRK,(3,3)),
//             DCB=(BLKSIZE=&SYSLBLK)
//SYSUT1   DD  UNIT=SYSDA,SPACE=(CYL,(1,1))
//SYSUT2   DD  UNIT=SYSDA,SPACE=(CYL,(1,1))
//SYSUT3   DD  UNIT=SYSDA,SPACE=(CYL,(1,1))
//SYSUT4   DD  UNIT=SYSDA,SPACE=(CYL,(1,1))
//SYSUT5   DD  UNIT=SYSDA,SPACE=(CYL,(1,1))              (2)
//SYSUT6   DD  UNIT=SYSDA,SPACE=(CYL,(1,1))
//SYSUT7   DD  UNIT=SYSDA,SPACE=(CYL,(1,1))
//LKED   EXEC PGM=HEWL,COND=(8,LT,COBOL),REGION=1024K
//SYSLIB   DD  DSNAME=&LIBPRFX..SCEELKED,                (3)
//             DISP=SHR
//SYSPRINT DD  SYSOUT=*
//SYSLIN   DD  DSNAME=&&LOADSET,DISP=(OLD,DELETE)
//         DD  DDNAME=SYSIN
//SYSLMOD  DD  DSNAME=&PGMLIB(&GOPGM),
//             SPACE=(TRK,(10,10,1)),
//             UNIT=SYSDA,DISP=(MOD,PASS)
//SYSUT1   DD  UNIT=SYSDA,SPACE=(TRK,(10,10))
```

Abb. 4.6 : Library Procedur `IGYWCL`

Dargestellt ist die Compile and link-edit Prozedur IGYWCL

(1)
STEPLIB can be installation-dependent.

(2)
SYSUT5 is needed only if the LIB option is used.

(3)
SYSLIB can be installation-dependent.

Unter anderem enthält IGYWCL (Abb. 4.6) die Namen COBOL.SYSIN, SYSLMOD und SYSIN, die in unserem JCL Script auftauchen (Abb. 4.5). Der Parameter LKED (linked) besagt, dass eine Verknüpfung besteht.

http://pic.dhe.ibm.com/infocenter/pdthelp/v1r1/index.jsp?topic=%2Fcom.ibm.entcobol.doc_4.1%2FPGandLR%2Fref%2Frpmvs06.htm

Als Alternative stellt z/OS eine IGYWCLG cataloged procedure zur Verfügung. Diese bewirkt COBOL compile, link, and go, also die unmittelbare Ausführung des übersetzten Programms.

Siehe
https://publib.boulder.ibm.com/infocenter/zos/basics/index.jsp?topic=/com.ibm.zos.zappldev/zappldev_116.htm

```
  File  Edit  Confirm  Menu  Utilities  Compilers  Test  Help
--------------------------------------------------------------------------------
EDIT       PRAK085.TEST.CNTL(COBSTA02) - 01.02            Columns 00001 00072
****** ***************************** Top of Data ******************************
==MSG> -Warning- The UNDO command is not available until you change
==MSG>           your edit profile using the command RECOVERY ON.
000100 //PRAK085C JOB (),CLASS=A,MSGCLASS=M,MSGLEVEL=(1,1),NOTIFY=&SYSUID,
000200 //         REGION=4M
000300 //STEP1 EXEC IGYWCL
000400 //COBOL.SYSIN DD DSN=&SYSUID..TEST.COB(COB02),DISP=SHR
000500 //LKED.SYSLMOD DD DSN=&SYSUID..TEST.LOAD,DISP=SHR
000600 //LKED.SYSIN DD *
000700  NAME COB02(R)
000800 /*
****** **************************** Bottom of Data ****************************




Command ===> SUB                                          Scroll ===> PAGE
 F1=Help      F3=Exit      F5=Rfind     F6=Rchange  F12=Cancel
```

Abb. 4.7: Ausführung des JCL-Scriptes

Beachten Sie die beiden DD Statements in Zeile 000400 und 000500.

Unser Compile- und Link-Script kann nun ausgeführt werden. Wir geben, wie in Abb. 4.7 gezeigt, auf der Kommandozeile "SUB" (für Submit) ein und betätigen die Eingabetaste.

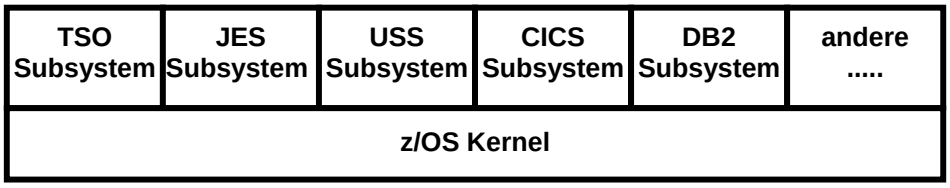


Abb. 4.8 z/OS Subsysteme

Das "Job Entry Subsystem" (JES) des z/OS-Betriebssystems dient dazu, Stapelverarbeitungsaufträge (Jobs) auf die einzelnen CPU's zu verteilen und der Reihe nach abzuarbeiten. Jobs werden dem "JES"-Subsystem in der Form von JCL-Scripten zugeführt, wobei deren erstes JCL-Statement ein JOB-Statement sein muss. PRAK085.TEST.CNTL(COBSTA02*)* ist ein derartiges Script. Das Kommando "SUB" (Submit) bewirkt, dass PRAK085.TEST.CNTL(COBSTA02*)* in die Warteschlange der von JES abzuarbeitenden Aufträge eingereiht wird.

z/OS gestattet es grundsätzlich, Programme entweder interaktiv im Vordergrund (unter TSO) oder als Stapelverarbeitungsprozesse durch JES im Hintergrund abzuarbeiten. Ersteres garantiert bessere Antwortzeiten, letzteres führt zu einem besseren Durchsatz.

Was wir hier machen, ist ein Quellprogramm zu übersetzen. Das kann mehr Zeit in Anspruch nehmen. Es ist deshalb üblich, die Durchführung nicht im Vordergrund durch TSO, sondern im Hintergrund auszuführen. Die Durchführung im Vordergrund hat den Vorteil, dass das übersetzte Programm direkt aufgerufen, und das Ergebnis direkt auf dem Bildschirm wiedergegeben werden kann. Die Durchführung im Hintergrund bewirkt, dass das übersetzte Programm in einem Data Set, nämlich `PRAK085.TEST.LOAD(COB02)`, abgespeichert wird. Es muss dann noch irgendwie, z.B. durch ein TSO Kommando, aufgerufen werden.

```
  File  Edit  Confirm  Menu  Utilities  Compilers  Test  Help
------------------------------------------------------------------------------
EDIT       PRAK085.TEST.CNTL(COBSTA02) - 01.02            Columns 00001 00072
****** ***************************** Top of Data ******************************
==MSG> -Warning- The UNDO command is not available until you change
==MSG>           your edit profile using the command RECOVERY ON.
000100 //PRAK085C JOB (),CLASS=A,MSGCLASS=M,MSGLEVEL=(1,1),NOTIFY=&SYSUID,
000200 //          REGION=4M
000300 //STEP1 EXEC IGYWCL
000400 //COBOL.SYSIN DD DSN=&SYSUID..TEST.COB(COB02),DISP=SHR
000500 //LKED.SYSLMOD DD DSN=&SYSUID..TEST.LOAD,DISP=SHR
000600 //LKED.SYSIN DD *
000700  NAME COB02(R)
000800 /*
****** **************************** Bottom of Data ****************************




IKJ56250I JOB PRAK085C(JOB03979) SUBMITTED
***
```

Abb. 4.9
Meldung "JOB PRAK085C(JOB03979) SUBMITTED"

Der JCL-Kommando-Interpreter überprüft die Syntax des Scripts. Falls er keinen Fehler findet, übergibt (submitted) er den Job zur Abarbeitung an das JES-Subsystem. Die Meldung oberhalb der Kommandozeile besagt, dass dies hier der Fall ist (siehe Abb. 4.9). Der Job erhält die Nummer 03979. Diese Nummer kann z.B. vom Systemadministrator benutzt werden, um den Status der Verarbeitung dieses Jobs abzufragen. Es ist eine gute Idee, wenn Sie sich die Job Nummer (hier 03979) auf einem Stück Papier notieren.

Wir warten einige Sekunden und betätigen anschließend die Eingabetaste. Erscheint keine Meldung, hat JES das JCL-Script noch nicht endgültig abgearbeitet. Wir warten erneut einige Sekunden und Betätigen die Eingabetaste; wir wiederholen dies notfalls mehrfach, bis eine Statusmeldung, so ähnlich wie in Abb. 4.10 dargestellt ist, ausgegeben wird.

http://www.mainframes360.com/2010/02/submitting-job.html oder
http://www.cedix.de/VorlesMirror/Band3/submit.pdf
enthält eine Beschreibung, was beim Submit eines Jobs passiert.

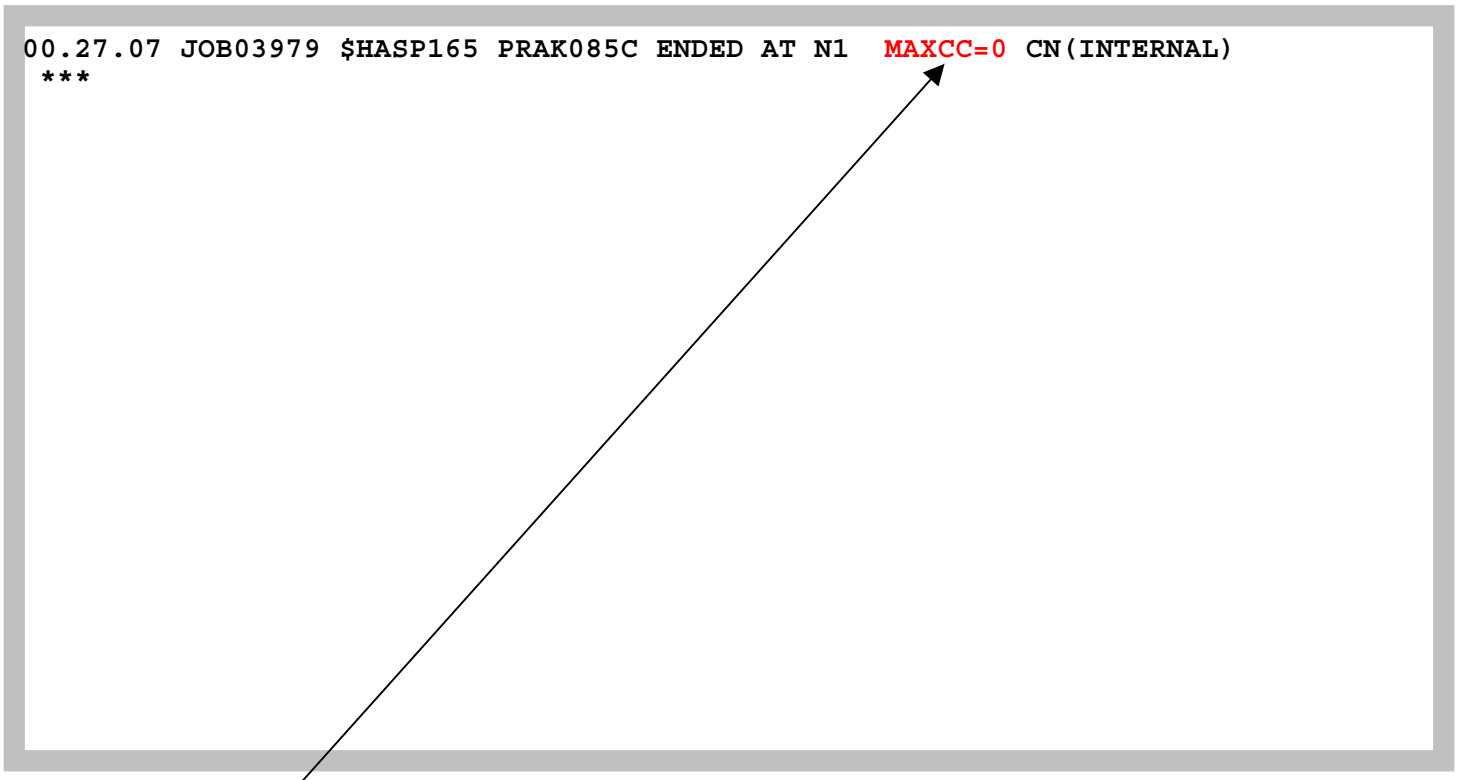


Abb. 4.10: Statusmeldung nach Abarbeitung des JCL-Scriptes

"MAXCC-0" ist eine Erfolgsmeldung: Die Übersetzung ist erfolgreich durchgeführt worden. "MAXCC-4" ist ebenfalls OK, alles andere besagt, dass ein Fehler aufgetreten ist. In diesem Fall hilft SDSF, siehe Abschnitt 6.

Das übersetzte Programm ist nun ausführungsfertig in dem File `PRAK085.TEST.LOAD(COB02)` abgespeichert.

5. Ausführung des COBOL-Programms

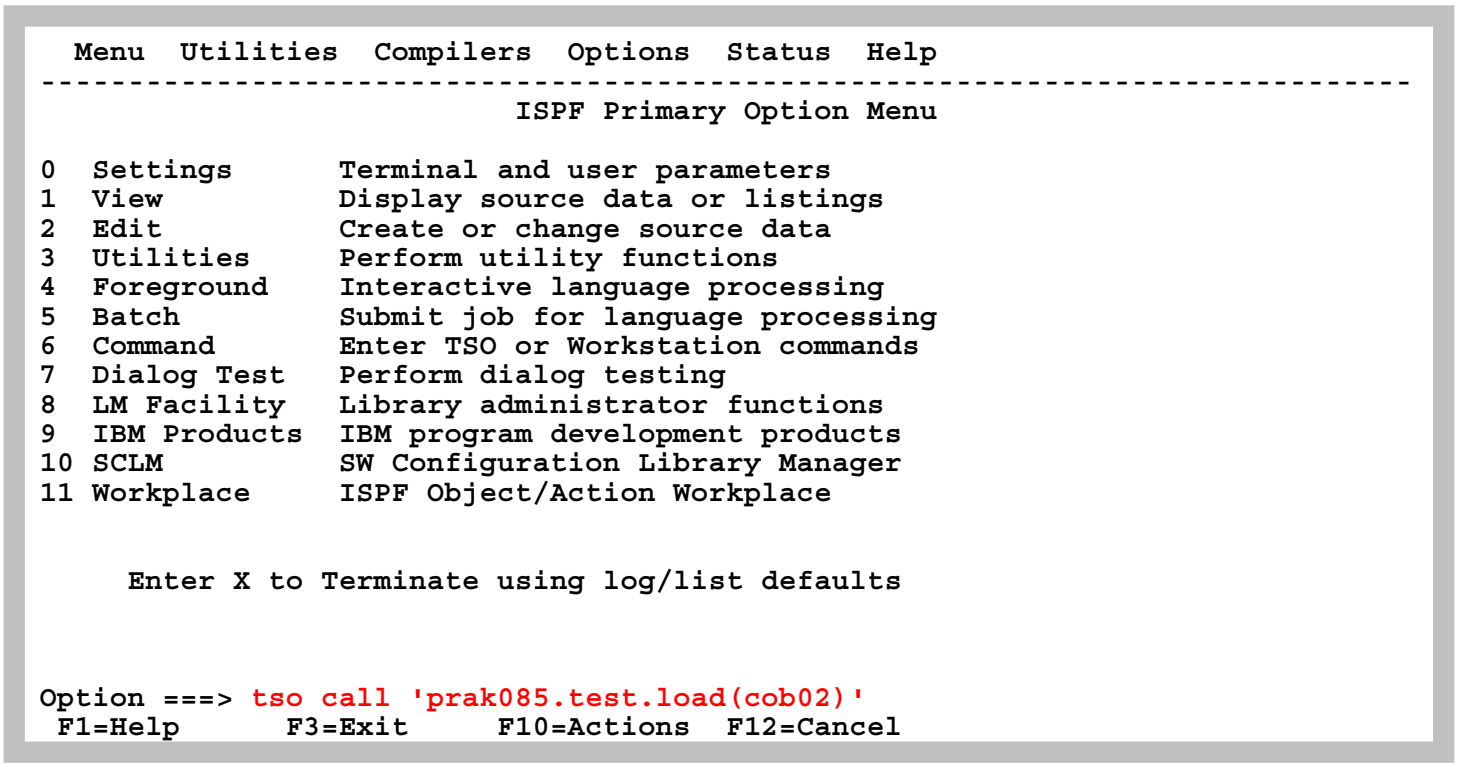


Abb. 5.1: "ISPF Primary Option Menu"-Bildschirm

Wir sind nun soweit, dass unser Programm ausgeführt werden kann. Durch mehrfaches Betätigen der F3-Taste kehren wir in das "ISPF Primary Option Menu" zurück (siehe Abb. 5.1).

Auf der Kommandozeile geben wir den Befehl

tso call 'prak085.test.load(cob02)'

ein und betätigen die Eingabetaste. "prak085.test.load(cob02)" enthält das vom Compiler erzeugte Maschinenprogramm. "call" ist ein TSO-Kommando und ruft ein Programm auf.

Wir sind aber im ISPF-Subsystem und nicht im TSO-Subsystem. "tso call" an Stelle von "call" bewirkt, dass der "call"-Befehl auch innerhalb des ISPF-Subsystems ausgeführt werden kann.

Dieses und die folgenden Tutorials enthalten eine Reihe von Selbst-Test Fragen.

Wir empfehlen, dass Sie die Selbst-Test Fragen bearbeiten. Sie sollen Ihnen die Gewissheit zu geben, dass Sie verstanden haben, was Sie in dem Tutorial gemacht haben. Es kann sein, dass derartige Fragen in der mündlichen Prüfung auftauchen.

Selbst-Test

- **Wurde das Compile und Link unseres Cobol Programms unter dem JES oger dem ISPF oder dem TSO Subsystem durchgeführt ?**
- **Wurde das Ausführen unseres Cobol Programms unter dem JES oger dem ISPF oder dem TSO Subsystem durchgeführt ?**

Wichtiger Hinweis:

Achten Sie darauf, dass Sie bei dem Befehl "tso call 'prakt20.test.load(cob02)'" die richtigen Hochkommas verwenden. Das korrekte Hochkomma steht auf den meisten Tastaturen über dem Zeichen "#".

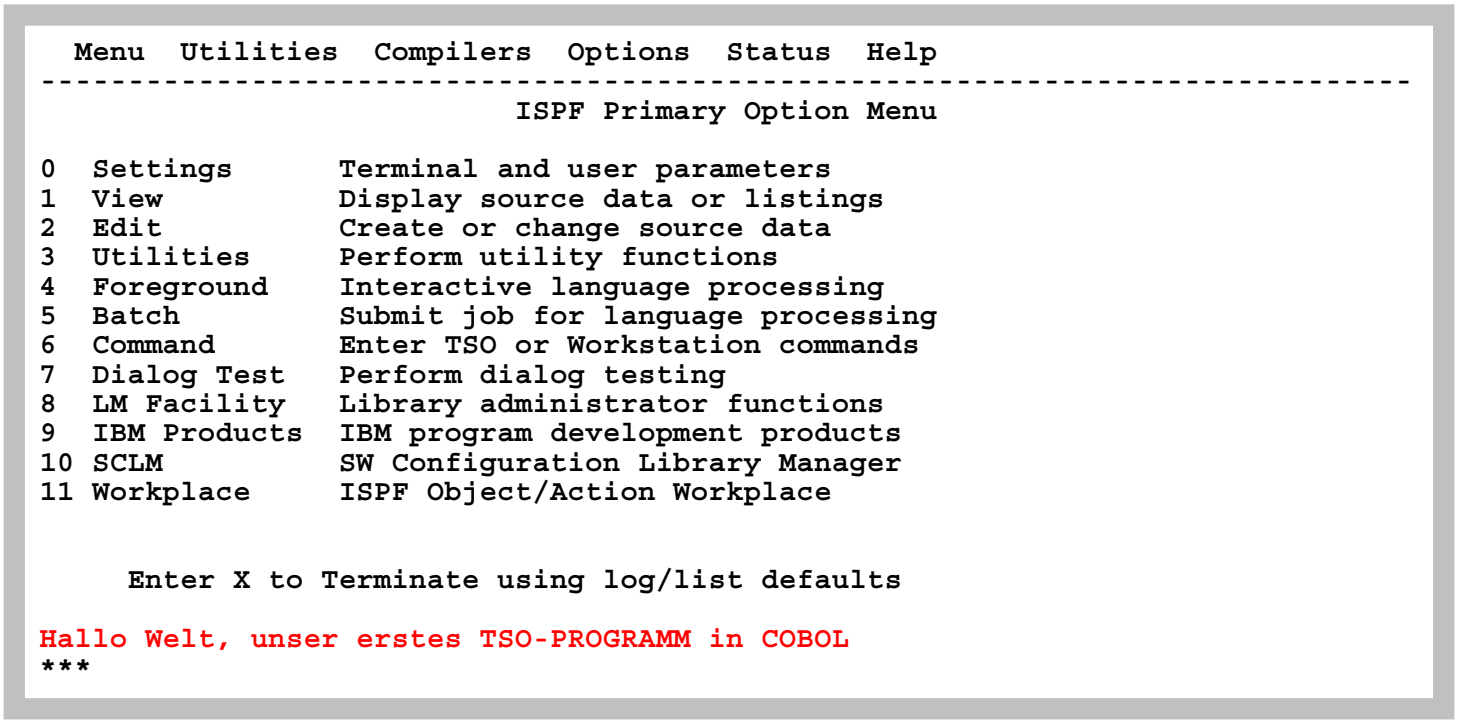


Abb. 5.2: Ausgabe unseres COBOL-Programms

Abb. 5.2 zeigt: Oberhalb der Kommandozeile erscheint die Ausgabe unseres COBOL-Programms. Wir hätten die Ausgabe etwas schöner gestalten können, indem wir `Hallo Welt, unser erstes TSO-PROGRAMM in COBOL` auf einem leeren Bildschirm schön zentriert in der Mitte ausgegeben hätten (fällt Ihnen eine primitive Lösung ein, wie man dies quick und dirty erreichen könnte ?).

Wir nehmen an, Ihnen fallen jetzt viele Möglichkeiten ein, ein aussagefähigeres COBOL-Programm zu schreiben. Sie können ein neues Quellprogramm PRAK085.TEST.COB(COBn) schreiben und hierfür ein neues JCL-Script PRAK085.TEST.CNTL(COBSTAn) erzeugen, was sich von PRAK085.TEST.CNTL(COB02) durch andere INFILE- und OUTFILE-Parameter unterscheidet. Dies resultiert in zusätzlichen Members in unseren drei Partitioned Datasets.

Aufgabe: ***Verfassen Sie ein eigenes funktionsfähiges COBOL-Programm (keine Modifikation des vorgegebenen Hallo-Welt-Programms) und legen Sie den Quellkode in PRAK085.TEST.COB(COBn) ab (n sei eine Ziffer Ihrer Wahl). Das angepasste JCL-Script legen Sie bitte in PRAK085.TEST.CNTL(COBSTAn) ab ("PRAK085" ist bei beiden Datasets durch Ihre Benutzer-ID zu ersetzen). Erstellen Sie je einen Print-Screen von Ihrem ISPF-Fenster mit dem Quellcode Ihres Programms sowie von Ihrem ISPF-Fenster mit der Ausgabe Ihres COBOL-Programms. Erzeugen Sie ebenfalls einen Print-Screen von dem ISPF-Fenster, der das von Ihnen modifizierte JCL-Script enthält. Schicken Sie die drei Print-Screens an Ihren Betreuer.***

Selbst-Test

- **Versuchen Sie doch, einmal ihr eigenes Cobol Programm zu schreiben. Sie finden reichlich Vorlagen hierfür im Internet, z.B. unter http://www.infogoal.com/cbd/cbdprj.htm**

Literatur zum Thema JCL:

Coding JCL Statements
http://www.informatik.uni-leipzig.de/cs/Literature/Textbooks/jcl/CodeJCL.pdf

M.Winkler: „MVS/ESA JCL“. Oldenbourg, 3. Auflage, 1999

z/OS JCL Reference SA22-7597-10 Eleventh Edition, April 2006
http://www.informatik.uni-leipzig.de/cs/Literature/Textbooks/jcl/MvsJclReference.pdf

z/OS JCL Users Guide
http://www.informatik.uni-leipzig.de/cs/Literature/Textbooks/jcl/MvsJclUserGuide.pdf

URLs

http://www.csis.ul.ie/cobol/

http://www.cobol-workshop.de/

http://www.mainframegurukul.com/tutorials/programming/cobol/cobol-tutorial.html

http://www.freeprogrammingresources.com/coboltutr.html

Unix
http://scis.acast.nova.edu/~jclevin/COURSE/COBOL/cobol.html

Enterprise
http://theamericanprogrammer.com/programming/manuals.cobol.shtml