Fantastische SAS-Funktionalitäten und wo sie zu finden sind - PROC GROOVY

Dennis Voigt
viadee Unternehmensberatung GmbH
Anton-Bruchhausen-Str. 8
48147 Münster
dennis.voigt@viadee.de

Zusammenfassung

Seit SAS 9.3 hat sich SAS mit der neuen PROC GROOVY Prozedur einer großen neuen bunten Welt geöffnet, mit der sich fehlende Funktionen spielend leicht in SAS BASE integrieren lassen. Durch die geschickte Kombination der Prozedur PROC GROOVY, einigen Open-Source JAVA-Bibliotheken und dem DataStep Java Objekt lassen sich plattformunabhängig Herausforderungen lösen, die sonst schnell als unlösbar ausgeschlossen wurden. In einem Praxisbeispiel wird gezeigt, wie leicht mit reinen SAS BASE Mitteln sowohl unter Windows, als auch auf dem HOST unter z/OS kennwortgeschützte ZIP-Dateien erzeugt werden können und Daten in Microsoft Access Datenbanken exportiert werden können, ohne dabei Zugriff auf Microsoft Office Komponenten in der Umgebung zu haben.

Schlüsselwörter: PROC GROOVY, ZIP, ACCESS, DB, z/OS, UNIX, Windows, JavaObj, Java, JVM

1 Vorwort

Es gibt Herausforderungen, die technisch nicht mit einfachen bzw. im Standard enthaltenen Mitteln in SAS umsetzbar sind. Bei solchen Herausforderungen kann die Prozedur GROOVY Abhilfe schaffen. Im folgenden Dokument werden zwei Anwendungsfälle aus der Praxis anhand von Beispielen erläutert. Der erste Fall beschäftigt sich damit, passwortgeschützte ZIP-Dateien zu erstellen, ohne aus SAS heraus plattformspezifische Befehle auszuführen. Der andere Fall beschäftigt sich mit der Erstellung einer Access-Datei mit reinen SAS Base Mitteln, ohne dabei auf Microsoft-Komponenten zuzugreifen. Der erstellte Code ist plattformunabhängig und kann auf allen Betriebssystemen ausgeführt werden. Um die Anwendungsfälle zu entwickeln und zu testen, wurde SAS mit der Version 9.4 auf z/OS sowie Windows genutzt.

2 Allgemein

Die Prozedur GROOVY ist seit der SAS Version 9.3 Bestandteil von SAS BASE. Beim Aufruf dieser Prozedur wird zur Laufzeit Groovy- bzw. Java-Code kompiliert, der direkt in der Java Virtual Machine (JVM) ausgeführt wird. Die JVM wird unter dem

Userkonto der aufrufenden SAS Session gestartet und besitzt somit die gleichen Zugriffsrechte auf das Dateisystem und der Netzwerkumgebung wie unter SAS. [2]

Es ist darüber hinaus auch möglich den Classpath der laufenden SAS Session individuell zu erweitern, womit sich der Funktionsumfang drastisch erweitert.

Im Folgenden wird ein Aufruf der Prozedur GROOVY abgebildet, der den Classpath der SAS Session durch eine externe JAR-Datei erweitert:

```
FILENAME INDCP "…";

PROC GROOVY;
  ADD PATH=INDCP;

  SUBMIT PARSEONLY;
    import […];

    class KSFE {
      public KSFE() {
        […]
      }
      public void ksfeVoid(){
        […]
      }
    }
  ENDSUBMIT;
QUIT;

DATA _NULL_;
  DECLARE JavaObj j;
  j = _new_ JavaObj ("KSFE");
  j.exceptionDescribe(1);
  j.callvoidMethod("ksfeVoid");
  j.exceptionCheck(e);
  IF e THEN DO;
    j.exceptionsClear();
    PUT "ERROR: Es ist ein Fehler aufgetreten!";
  END;
RUN;
```

Die JAR-Datei wird zunächst durch eine Filename-Anweisung referenziert und danach in PROC GROOVY durch den Befehl „ADD PATH“ erweiternd eingelesen. Die Bestandteile der eingelesenen Datei müssen darüber hinaus durch eine Import-Anweisung der Java-Klasse zugänglich gemacht werden. Im Anschluss wird die Java-Klasse über das SAS DataStep Java Objekt instanziiert und genutzt. Optional wird am Ende des DataSteps geprüft, ob ein Fehler aufgetreten ist. Sollte dies der Fall sein, wird die Meldung „Es ist ein Fehler aufgetreten!“ vom Typ „Error“ im SAS Log ausgegeben.

3 Anwendungsfälle

3.1 Generieren einer passwortgeschützten ZIP-Datei

Seit der SAS Version 9.2 können mittels ODS (Output Delivery System) ZIP-Dateien erzeugt werden, ohne dass plattformspezifische Befehle genutzt werden müssen. Mit der Version 9.4 wurde die Erstellung einer ZIP-Datei erleichtert, indem ZIP-Optionen direkt in der Filename-Anweisung angegeben werden können. Der Anwendungsfall einer passwortgeschützten ZIP-Datei wurde bislang jedoch noch nicht implementiert. Im folgenden Beispiel wird dargestellt, wie man eine Funktion mit PROC GROOVY implementieren kann, um ZIP-Dateien mit Passwortschutz zu erzeugen, ohne dabei plattformspezifische Befehle auszuführen.

Für dieses Beispiel sind die folgenden Open-Source Java-Bibliotheken zwingend erforderlich:

- Zip4j_1.3.2.jar
  [http://www.lingala.net/zip4j/download.php]
  (Erstellen einer ZIP-Datei)

- Commons-io-2.5.jar
  [https://commons.apache.org/proper/commons-io/download_io.cgi]
  (Binäres Bearbeiten einer Datei)

Zunächst wird der Classpath der aktuellen SAS Session um die JAR-Dateien erweitert. Hier wird bereits im Konstruktor eine leere ZIP-Datei erstellt, welche nach und nach durch Dateien angereichert wird. Darüber hinaus werden die zu nutzenden Parameter pro Datei definiert. In diesem Beispiel wird immer eine passwortgeschützte ZIP-Datei erstellt und setzt somit im Konstruktor nur das Übergeben des vollständigen Pfads und des Passworts der ZIP-Datei als Eingabeparameter voraus.

```
FILENAME ZIPPASS "C:\KSFE\JAR\zip4j_1.3.2.jar";
FILENAME COMMONF "C:\KSFE\JAR\commons-io-2.5.jar";

PROC GROOVY;
  ADD PATH=ZIPPASS;
  ADD PATH=COMMONF;

  SUBMIT PARSEONLY;
    import java.io.File;
    import java.io.IOException;
    import org.apache.commons.io.FileUtils;
    import net.lingala.zip4j.core.ZipFile;
    import net.lingala.zip4j.exception.ZipException;
    import net.lingala.zip4j.model.ZipParameters;
    import net.lingala.zip4j.util.Zip4jConstants;
class ZIPPEN {
      private ZipFile zipFile;
      private ZipParameters zipParameters;
      public ZIPPEN(String zipFileName,String zipPassword){
```

```
    try {
      // Initialisieren der ZIP-Datei und -Parameter
      this.zipFile = new ZipFile(zipFileName);
      this.zipParameters = new ZipParameters();
      this.zipParameters
        .setCompressionMethod(
          Zip4jConstants.COMP_DEFLATE
        );
      this.zipParameters
       .setCompressionLevel(
         Zip4jConstants.DEFLATE_LEVEL_NORMAL
       );

      // Einstellen, dass die ZIP-Datei passwortgeschützt wird
      this.zipParameters
        .setEncryptFiles(
          true
        );

      // Standard-Verschlüsselung (ZipCrypto)
      this.zipParameters
        .setEncryptionMethod(
          Zip4jConstants.ENC_METHOD_STANDARD
        );

      // Setzen des Passworts
      this.zipParameters
        .setPassword(
          zipPassword
        );
    }
    catch (ZipException e){
      e.printStackTrace();
    }
  }
```

In der Methode `changeCarriageReturn` wird die Datei binär ausgelesen und bearbeitet. Da für das Beispiel das Windows-Betriebssystem als Zielsystem genutzt wurde, wird bei einem Zeilenumbruch ein „CR - carriage return“ (\r) mit darauffolgendem „LF - line feed“ (\n) benötigt. Die binäre Aufbereitung der Datei ist wichtig, sobald diese auf einem anderen Betriebssystem wie z. B. UNIX erstellt wird (siehe Kapitel 4). Dafür wird über einen regulären Ausdruck geprüft, ob ein LF ohne ein vorangestelltes CR vorhanden ist. Sollte dies zutreffen, wird das LF durch CR LF ersetzt.

```
  public File changeCarriageReturn(String fileInputDir){
    File file = new File(fileInputDir);
    String content = FileUtils.readFileToString(file,"UTF-8");
    content = content.replaceAll("(?<!\\r)\\n", "\r\n");
    FileUtils.write(file,content,"UTF-8");
    return file;
  }
```

In der folgenden Abbildung sind unterschiedliche Betriebssysteme und deren Codierungen für Zeilenumbrüche aufgelistet:

Betriebssystem	Zeichensatz	Abkürzung	Code Hex	Code Dezimal	Escape-Sequenz
Unix, Linux, Android, macOS, AmigaOS, BSD, weitere	ASCII	LF	0A	10	\n
Windows, DOS, OS/2, CP/M, TOS (Atari)		CR LF	0D 0A	13 10	\r\n
Mac OS Classic, Apple II, C64		CR	0D	13	\r
AIX OS & OS/390	EBCDIC	NL	15	21	\025

Abbildung 1: Codierung des Zeilenumbruchs [1]

Durch den Aufruf der Methode `addFileToZip` werden die einzelnen Dateien (z. B. CSV-Dateien) in die bereits erstellte ZIP-Datei geschrieben. Darüber hinaus wird jede hinzugefügte Datei mit den im Konstruktor definierten Parametern versehen.

```
      public void addFileToZip(String fileInputDir){
        // Aufbereiten der Zeilenumbrüche
        File file = changeCarriageReturn(fileInputDir);

        // Hinzufügen der Dateien
        zipFile.addFile(file, this.zipParameters);
      }
    }
  ENDSUBMIT;
QUIT;
```

Die Klasse `ZIPPEN` wird über ein DataStep Java Objekt instanziiert. In diesem Beispiel wird die ZIP-Datei `zipfile.zip` im Pfad `C:\KSFE\OUTPUT\` erstellt und mit dem Passwort `pwKSFE` versehen. Im Anschluss wird die Datei `data.csv` hinzugefügt. Sollten Fehler im Java-Code aufgetreten sein, werden diese über die Funktion `exception-check` ausgelesen und können anhand der zurückgegebenen Variable `e` verarbeitet werden.

```
DATA _NULL_;
  DECLARE JavaObj j;
  j = _new_ JavaObj("ZIPPEN",
                    "C:\KSFE\OUTPUT\zipfile.zip",
                    "pwKSFE");
  j.exceptionDescribe(1);
  j.callvoidMethod("addFileToZip",
                   "C:\KSFE\INPUT\data.csv");
  j.exceptionCheck (e);
  IF e THEN DO;
    j.exceptionClear();
    PUT "ERROR: Ein Fehler ist aufgetreten!";
  END;
RUN;
```

3.2 Generieren einer ACCESS-Datei

Mit der SAS-Komponente SAS/ACCESS können von Haus aus ACCESS-Dateien bzw. -Tabellen erzeugt werden. Doch was macht man, wenn nur SAS BASE zur Verfügung steht oder die Generierung auf dem HOST erfolgen soll? Dafür kann eine Funktion bzw. ein Makro auf Basis von Open-Source Java-Bibliotheken und der Prozedur GROOVY erstellt werden. Das Beispiel bietet die Möglichkeit ACCESS-Dateien ohne vorhandene Microsoft Office Komponenten zu erstellen.

Für dieses Beispiel sind die folgenden Open-Source Java-Bibliotheken zwingend erforderlich:

- Jackcess_2.1.8.jar
  [https://sourceforge.net/projects/jackcess/files/]
  (Zum Erstellen der ACCESS-Datei)

- Commons-lang-2.6.jar
  [https://commons.apache.org/proper/commons-lang/download_lang.cgi]
  (Wird innerhalb der Jackcess-JAR verwendet)

- Commons-logging-1.1.3.jar
  [https://commons.apache.org/proper/commons-logging/download_logging.cgi]
  (Wird innerhalb der Jackcess-JAR verwendet)

Mit der Methode defineTable wird zunächst eine Tabelle erzeugt. Als Eingabeparameter reicht es aus den Namen der Tabelle anzugeben. In dem Beispiel ist das der Name des SAS-DataSets. Die definierte Tabelle wird mit den nachfolgenden Methoden Schritt für Schritt parametrisiert und erweitert.

```
FILENAME JC    "C:\KSFE\JAR\jackcess_2.1.8.jar";
FILENAME CLANG "C:\KSFE\JAR\commons-lang-2.6.jar";
FILENAME CLOG  "C:\KSFE\JAR\commons-logging-1.1.3.jar";

PROC GROOVY;
  ADD PATH=JC;
  ADD PATH=CLANG;
  ADD PATH=CLOG;

  SUBMIT PARSEONLY;

    import java.io.file;
    import com.healthmarketscience.jackcess.*;
    import com.healthmarketscience.jackcess.DatabaseBuilder;

    class EXPORT_ACCESS {
      private TableBuilder myTableBuilder;
      private ArrayList<HashMap<String,String>> myRowList;
      private HashMap<String, String> myHashMap;
      private Database db;
      public EXPORT_ACCESS() {
```

```
    this.myRowList = new ArrayList<HashMap<String, String>>();
    this.myHashMap = new HashMap<String, String>();
  }

  public void defineTable(String tableName) {
    // Initialisieren einer ACCESS-Tabelle mit dem Namen
    // tableName
    this.myTableBuilder = new TableBuilder(tableName);
  }
```

Mit den Methoden addStringItemToRow und addNumericItemToRow werden neue Zeilen mithilfe einer HashMap zellenweise bestückt. Sobald eine Zeile komplett eingelesen wurde, wird die Methode addRowToTable angesprochen, welche die HashMap in die RowList der Tabelle aufnimmt.

```
  // Erstellen einer Tabellenzeile
  public void addRowToTable() {
    this.myRowList.add(this.myHashMap);
    this.myHashMap = new HashMap<String, String>();
  }

  // Hinzufügen des alphanumerischen Werts "item" zur Spalte
  // "name". Sollte der Eingabeparameter einen Missing-Wert
  // beinhalten, wird dieser durch NULL ersetzt.
  public void addStringItemToRow(String name, String item) {
    if (item.equals(".")) item = null;
    this.myHashMap.put(name, item);
  }

  // Hinzufügen des numerischen Werts "item" zur Spalte "name".
  // Dieser Wert wird für die weitere Verarbeitung zum String
  // konvertiert. Sollte der Eingabeparameter einen Missing-Wert
  // beinhalten, wird dieser durch NULL ersetzt.
  public void addNumericItemToRow(String name, double item) {
    String strItem = Double.toString(item);
    if (strItem.equals("NAN")) {
      this.myHashMap.put(name, null);
    } else {
      this.myHashMap.put(name, strItem);
    }
  }
```

Über die Methode `addColumnToTable` werden die einzelnen Spalten sowie deren Spaltentypen in der Tabelle definiert.

```
  public void addColumnToTable(String columnName,
                               String strColumnDataType) {
    DataType columnDataType;
    // Beinhaltet der Eingabeparameter "strColumnDataType" den
    // Wert "TEXT", wird die Spalte als "TEXT" definiert, sonst
    // als "DOUBLE"
```

```
        If (strColumnDataType.equals("TEXT")) {
          columnDataType = DataType.TEXT;
        } else if (strColumnDataType.equals("DOUBLE")) {
          columnDataType = DataType.DOUBLE;
        }

        // Hinzufügen der Spalte an die Tabelle
        This.myTableBuilder.addColumn(
          new ColumnBuilder(columnName,columnDataType)
        );
      }
```

Die beiden Methoden createDB und closeDB werden jeweils am Ende aufgerufen, um die Datenbank mit den entsprechenden Bestandteilen zu bestücken und den aktuellen Zugriff zu beenden.

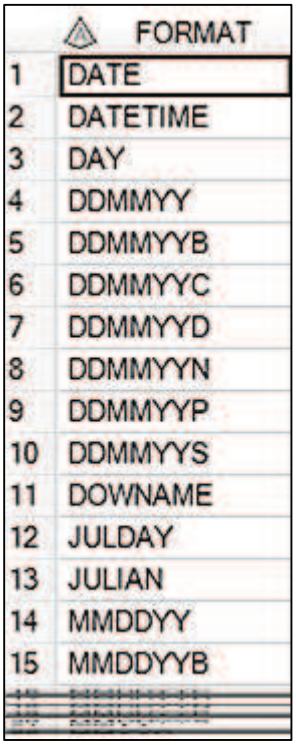
```
      public void createDB(String accessDat) {
        File dbFile = new File(accessDat);
        this.db = DatabaseBuilder.create(Database.FileFormat.V2010,
                                         dbFile);
        Table myTable = this.myTableBuilder.toTable(db);
        for (HashMap<String, String> map : this.myRowList) {
          myTable.addRowFromMap(map);
        }
      }

      public void closeDB() {
        db.close();
      }
    }

  ENDSUBMIT;

QUIT;
```

Damit einige SAS Formate auch richtig in der Tabelle ausgegeben werden, empfiehlt es sich diese als TEXT zu definieren. In diesem Beispiel wird eine Steuertabelle mit Datumsformaten eingelesen (Siehe Abbildung 2), die als TEXT interpretiert werden sollen. Es können auch eigene Formate, Währungen oder ähnliches angegeben werden.



	FORMAT
1	DATE
2	DATETIME
3	DAY
4	DDMMYY
5	DDMMYYB
6	DDMMYYC
7	DDMMYYD
8	DDMMYYN
9	DDMMYYP
10	DDMMYYS
11	DOWNAME
12	JULDAY
13	JULIAN
14	MMDDYY
15	MMDDYYB

Abbildung 2: Steuertabelle für Sonderformate

Um das SAS-DataSet aufzubereiten und dessen Informationen für die Generierung der ACCESS-Datei zu nutzen, wurde das folgende SAS-Makro entwickelt. Am Anfang werden die Variablen und ihre Informationen mit Hilfe der Prozedur CONTENTS ausgelesen. Diese Daten werden dann durch das Einlesen der Steuertabelle für Sonderformate aufbereitet, sodass die aufgelisteten Formate als Text interpretiert werden. Um diese beim Befüllen der ACCESS-Tabelle zu identifizieren, werden sie mit dem Typ 3 gekennzeichnet. Nun müssen die Informationen für den nachfolgenden DataStep mit Makrovariablen zur Verfügung gestellt werden. Dafür werden alle Variablen vom Typ 1 als DOUBLE und alle anderen als TEXT interpretiert. Im letzten DataStep wird die ACCESS-Datei aufgebaut. Anhand der Variablen und Typen werden die Spalten über das DataStep Java-Objekt spaltenweise angelegt. Sobald alle benötigten Spalten in der ACCCESS-Tabelle erzeugt wurden, können diese über eine Schleife befüllt werden. Dabei werden die Werte, die als Sonderformat gekennzeichnet wurden, zum Text mit der richtigen Darstellung konvertiert. Am Ende muss die Tabelle lediglich mit den definierten Bestandteilen bestückt und der Zugriff darauf geschlossen werden.

```
%MACRO EXPORT_TO_ACCESS(LIB=,TAB=,AUSGABE=);

  /* Auslesen der Variableninformationen */
  PROC CONTENTS DATA=&LIB..&TAB.
                OUT=COLUMNS(KEEP=NAME
                                 TYPE
                                 FORMAT
                                 FORMATL
                                 FORMATD
                                 VARNUM)
                NOPRINT
```

```
                NODETAILS;
  RUN;

  DATA COLUMNS_WITH_SPECIALS(DROP=RC);
    SET COLUMNS;

    IF _N_ EQ 1 THEN DO;
      /* Einlesen der Sonderformate aus der Steuertabelle */
      DECLARE HASH HO (DATASET: "STEUER_FORMAT");
      HO.DEFINEKEY("FORMAT");
      HO.DEFINEDONE();
    END;

    RC = HO.FIND();
    IF RC EQ 0 THEN DO;
      /* Sollte das Format der Variable in der Steuertabelle
         vorhanden sein, wird der Typ auf 3 gesetzt */
      TYPE = 3;
    END;
  RUN;

  DATA _NULL_;
    SET COLUMNS_WITH_SPECIALS END=EOF;

    LENGTH FORMAT_GES $100
           TYPE_STR $20;

    IF TYPE EQ 1 THEN DO;
      TYPE_STR = "DOUBLE";
    END;
    ELSE DO;
      TYPE_STR = "TEXT";
    END;

    /* Verketten der Formatinformationen zu einem SAS-Format */
    FORMAT_GES = CATT(FORMAT,COMPRESS(IFC(FORMATL="0","",FORMATL)),
                      ".",COMPRESS(IFC(FORMATD="0","",FORMATD)));

    /* Erstellen der Makrovariablen */
    CALL SYMPUTX(COMPRESS("VAR_NAME_" !! PUT(VARNUM,25.)),
                          UPCASE(NAME),"L");
    CALL SYMPUTX(COMPRESS("VAR_TYPE_NUM_" !! PUT(VARNUM,25.)),
                 TYPE,"L");
    CALL SYMPUTX(COMPRESS("VAR_TYPE_" !! PUT(VARNUM,25.)),
                 COMPRESS(UPCASE(TYPE_STR)),"L");
    CALL SYMPUTX(COMPRESS("VAR_FORMAT_" !! PUT(VARNUM,25.)),
                 COMPRESS(UPCASE(FORMAT_GES)),"L");
    IF EOF THEN DO;
      CALL SYMPUTX("ANZ_VARS",_N_,"L");
    END;
  RUN;
```

```
%LOCAL I;

DATA _NULL_;
  SET &LIB..TAB. END=EOF;

  IF _N_ EQ 1 THEN DO;
    DECLARE JavaObj j ("EXPORT_ACCESS");
    j.EXCEPTIONDESCRIBE(1);
    j.CALLVOIDMETHOD("defineTable","&TAB.");
    j.EXCEPTIONCHECK (e);

    IF e THEN DO;
      j.EXCEPTIONCLEAR();
      PUT "ERROR: Fehler beim Definieren der Tabelle!";
    END;

    /* Anlegen der Spalten/-Eigenschaften */
    %DO I = 1 %DO &ANZ_VARS.;
      j.CALLVOIDMETHOD("addColumnToTable",
                       "&&VAR_NAME_&I..",
                       "&&VAR_TYPE_&I..");
    %END;

    j.EXCEPTIONCHECK (e);

    IF e THEN DO;
      j.EXCEPTIONCLEAR();
      PUT "ERROR: Fehler beim Hinzufügen einer Spalte!";
    END;
  END;

  /* Einlesen der Werte zu den jeweiligen Spalten */
  %DO I = 1 %TO &ANZ_VARS.;
    j.CALLVOIDMETHOD(
      %IF &&VAR_TYPE_&I.. EQ DOUBLE %THEN %DO;
        "addNumericItemToRow"
      %END;
      %ELSE %DO;
        "addStringItemToRow"
      %END;
      , "&&VAR_NAME_&I..",
      %IF &&VAR_TYPE_NUM_&I.. EQ 3 %THEN %DO;
        STRIP(PUT(&&VAR_NAME_&I..,&&VAR_FORMAT_&I..))
      %END;
      %ELSE %DO;
        &&VAR_NAME_&I..
      %END;
    );
  %END;

  j.CALLVOIDMETHOD("addRowToTable");
  j.EXCEPTIONCHECK (e);
```

```
    IF e THEN DO;
      j.EXCEPTIONCLEAR();
      PUT "ERROR: Fehler beim Anfügen einer Zeile!";
    END;

    IF EOF THEN DO;
      j.CALLVOIDMETHOD("createDB","&AUSGABE.");
      j.EXCEPTIONCHECK (e);

      IF e THEN DO;
        j.EXCEPTIONCLEAR();
        PUT "ERROR: Fehler beim Erstellen der Ausgabedatei!";
      END;

      j.CALLVOIDMETHOD("closeDB");
      j.DELETE();
    END;
  RUN;
%MEND;

%EXPORT_TO_ACCESS(LIB=SASHELP,
                  TAB=CARS,
                  AUSGABE="C:\KSFE\OUTPUT\CARS.mdb");
```

4 HOST

Dieses Kapitel befasst sich damit, den beschriebenen Beispiel-Code zum Erstellen einer ZIP-Datei auf den HOST zu migrieren. Da die JVM unter dem Betriebssystem z/OS lediglich das Dateisystem von Unix (hierarchical file system, HFS) unterstützt und nicht das vom HOST (multiple virtual storage, MVS), muss der Programmablauf eines Exports von MVS-Dateien wie folgt entwickelt werden:

1. Die MVS-Eingabedatei wird per OCOPY auf das HFS kopiert
2. Die Eingabedatei wird in einer ZIP-Datei per PROC GROOVY geschrieben
3. Die erstellte ZIP-Datei wird per OGET wieder auf das MVS geschoben
4. Am Ende kann die ZIP-Datei (z. B. über das Windows-Betriebssystem) per FTPS abgezogen werden

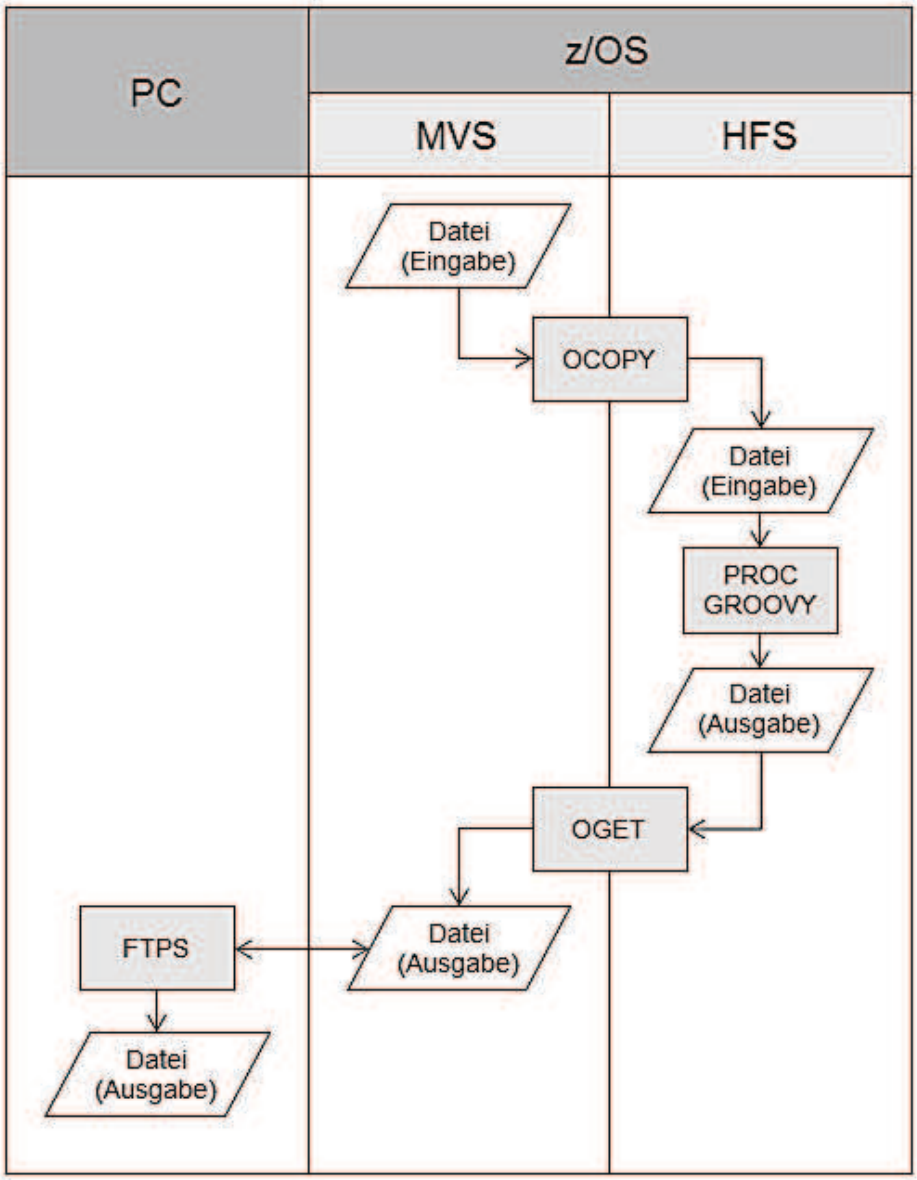


Abbildung 3: Übersicht Programmablauf z/OS

P010 - Löschen der Ausgabe-Zip-Datei

Damit die neue ZIP-Datei ohne Probleme im Schritt P040 per OGET binär verschoben werden kann, wird die Zieldatei im MVS gelöscht.

```
//P010      EXEX  PGM=IEFBR14
//DUMMY     DD DSN=KSFE.OUTPUT.ZIP,
//             DCB=(DSORG=PS,BLKSIZE=27648),
//             SPACE=(TRK,(1,1)),DISP=(MOD,DELETE,DELETE)
```

P020 – Kopieren und Konvertieren der Eingabedateien von MVS nach HFS

Die Eingabedateien müssen im HFS vorhanden sein. Wenn die ZIP-Datei Dateien beinhalten soll, dessen Ursprung das MVS ist, müssen diese per OCOPY auf das HFS kopiert werden. Darüber hinaus sollten Dateien, die im Textformat kopiert werden, von

EBCDIC nach ASCII konvertiert werden. Im nachfolgenden Code wird die MVS-Datei KSFE.INPUT.CSV nach HFS kopiert und mithilfe der Codepage BPXFX311 von EBCDIC (FROM1047) nach ASCII konvertiert.

```
//P020     EXEC  PGM=IKJEFT01
//INMVS    DD DISP=SHR,
//            DSN=KSFE.INPUT.CSV
//OUTHFS   DD PATH='/tmp/ksfe/data/input/data.csv'
//            PATHDISP=(KEEP,DELETE),
//            PATHOPTS=(OCREATE,ORDWR),
//            PATHMODE=(SIRUSR,SIWUSR,SIRGRP,SIROTH),
//            FILEDATA=TEXT
//SYSTSPRT DD SYSOUT=*
//SYSTSIN  DD *
OCOPY INDD(INMVS)          -
      OUTDD(OUTHFS)        -
      TEXT                 -
      PATHOPTS(USE)        -
      CONVERT((BPXFX311)) -
      FROM1047
```

P030 – Generieren der ZIP-Datei auf HFS

Im Schritt P030 kann der SAS Code (siehe Kapitel 3.1) übernommen werden. Hier müssen lediglich alle Windows-Pfade durch Unix-Pfade ersetzt werden. Damit die JVM die Dateien nutzen kann, muss vorher angegeben werden, dass das HFS genutzt werden soll. Dies kann mit dem Einstellen der folgenden Option realisiert werden:

```
OPTIONS FILESYSTEM = HFS;
```

P040 – Binäres kopieren der Ausgabedatei von HFS nach MVS

Per OGET wird die ZIP-Datei binär von HFS nach MVS übertragen. Die Ausgabe KSFE.DATA.OUTPUT befindet sich jetzt als binäre Datei auf dem MVS und kann z. B. per FTPS von einem Windows-Betriebssystem aus abgezogen werden.

```
//P040     EXEC PGM=IKJEFT01
//SYSTSPRT DD SYSOUT=*
//SYSTSIN  DD *
OGET '/tmp/ksfe/data/output/zipfile.zip' 'KSFE.DATA.OUTPUT' BINARY
```

P050 – Löschen der Zwischendateien

Damit das HFS aufgeräumt bleibt, sollten die genutzten Zwischendateien gelöscht werden. Dies kann u. a. per PATHDISP(DELETE) realisiert werden.

```
//P050     EXEC PGM=IKJEFT01
//DELERGS  DD PATH='/tmp/ksfe/data/input/data.csv'
//            PATHDISP=(DELETE)
//DELZIP   DD PATH='/tmp/ksfe/data/output/zipfile.zip'
```

```
//             PATHDISP=(DELETE)
//SYSTSPRT DD SYSOUT=*
//SYSTSIN  DD DUMMY
```

5 Fazit

Die Prozedur GROOVY eröffnet dem Entwickler eine große Bandbreite an neuen Szenarien, die unter SAS nicht bzw. nur bedingt möglich sind. Darüber hinaus sind alle erstellten GROOVY-Funktionen plattformunabhängig und können somit auf jedem Betriebssystem genutzt werden. Mit ein wenig Entwicklungsaufwand und erweiternden Java-Bibliotheken kann ein Unternehmen den Kostenaufwand auf Dauer reduzieren bzw. mehr Umfang anbieten. Bei den Beispielen aus diesem Dokument, handelt es sich um einen geringen Anteil der potenziellen Szenarien. Es können darüber hinaus auch ganze GUI-Anwendungen über SAS gesteuert bzw. parametrisiert werden. Es lohnt sich also diese Prozedur näher kennenzulernen.

Literatur

[1] https://de.wikipedia.org/wiki/Zeilenumbruch (30.01.2018)

[2] S. Reimann: Echt Groovy – Neue Sprachen nutzen mit SAS. Proceedings der 21. KSFE, Shaker-Verlag, Aachen (2017).

[3] Base SAS 9.4 Procedures guide, Sixth Edition SAS, PROC GROOVY. http://documentation.sas.com/api/docsets/proc/9.4/content/proc.pdf?locale=de#namedest=p0pq7kp0misx44n14vvub91rzihh (30.01.2018)