Listensyntax und PutPutPut
Mach es! Jetzt! Lokal!

Grischa Pfister
iCASUS GmbH
Vangerowstraße 2
69115 Heidelberg
g.pfister@icasus.de

Zusammenfassung

SAS/Base lässt an verschiedenen Stellen in der Syntax Variablenlisten zu. Richtig eingesetzt lässt sich auf diese Art Code schreiben, der weniger Wartungsaufwand notwendig macht. In diesem Beitrag wird deshalb dies Thema genauer beleuchtet.

Die neuen Funktionen DOSUB() und DOSUBL() ermöglichen es, aus dem Data Step heraus direkt SAS/Base Code auszuführen und dann auch die Ergebnisse zuzugreifen. Im Unterschied zu CALL EXECUTE() wird also nicht gesammelt und nach dem Data Step ausgeführt, sondern der Code zur Laufzeit des Data Steps in einem zweiten Thread abgearbeitet. Allerdings handelt es sich hier ausdrücklich nicht um ein Parallelisierungsverfahren, sondern der Data Step wartet auf das Ergebnis der Funktion, bevor er weiterläuft. Auch dies Thema ist es Wert, einmal dargestellt zu werden.

Schlüsselwörter: DOSUB, DOSUBL, PUT, %PUT, Variablenlisten

1 Listensyntax und PutPutPut

SAS/Base lässt an verschiedenen Stellen in der Syntax Variablenlisten zu. Richtig eingesetzt lässt sich auf diese Art Code schreiben, der weniger Wartungsaufwand notwendig macht. Es lohnt sich deshalb, das Thema genauer zu beleuchten.

1.1 Standardlisten und benutzerdefinierte Listen

Es gibt die drei Standard-Listen _ALL_, _CHARACTER_ und _NUMERIC_ die im Data Step in Funktionen und Statements aber auch in der Syntax verschiedener Prozeduren verwendet werden können.Daneben gibt es benutzerdefinierte Listen in der Form VAR1-VAR10 (mit drei weiteren Unterformen, die hier nicht aufgeführt werden) und STR:. Während die erste Variante die Variablen VAR1, VAR2, VAR3, VAR4, VAR5, VAR6, VAR7, VAR8, VAR9 und VAR10 auswählt, steht die zweite Variante für alle Variablen, deren Name mit „STR“ anfängt. Es folgen einige Anwendungsbeispiele.

1.2 Verwendung von _ALL_ in Prozeduren

Die Verwendung der Listensyntax ist nicht auf den Data Step begrenzt, sondern ist auch in vielen Prozeduren möglich. Das erste Beispiel zeigt die Verwendung in Proc Datasets, mit dieser Methode werden für alle Tabellen in SASHELP Metadaten ausgegeben.

```
Proc Datasets lib=sashelp;
  Contents data=_all_ mt=data;
Run;
quit;
```

Das zweite Beispiel zeigt eine typische Aufgabenstellung aus der Praxis: Für eine vorhandene Tabelle fehlt der Formate-Katalog, sie lässt sich nicht öffnen – was tun?
Mit einem einfachen Format-Statement in Proc Print lässt sich die Tabelle doch verwenden, denn das Format-Statement überschreibt temporär die Format-Information für alle Spalten.

```
Proc Print data=Sashelp.Prdsale(obs=5);
Run;

Proc Print data=Sashelp.Prdsale(obs=5);
  Format _all_;
Run;
```

1.3 Listensyntax im Data Step

Im Data Step lässt sich die Listensyntax an verschiedenen Stellen verwenden, Klassiker sind KEEP und DROP, interessant ist aber auch die Zuweisung von Variablen in Arrays.
Im folgenden Beispiel wird die Tabelle SASHELP.CLASS eingelesen, dabei werden alle alphanumerischen Variablen in das Array C gespeichert, die numerischen in das Array N. Es folgt eine Schleife pro Array, in der der Variablenname (vname()) und der Variablenwert(vvalue()) abgefragt und in das LOG ausgegeben werden.
Das Length-Statement ist mit Absicht unterhalb der Zuweisung der Variablen in die Arrays platziert, denn sonst wären NAME und VALUE ebenfalls Elemente des Arrays.

```
Data _Null_;
  Set Sashelp.Cars(obs=1);
  Array c _character_;
  Array n _numeric_;

  Length name $32 value $32767;

  Do i=1 To dim(c);
    name = vname(c(i));
    value = vvalue(c(i));
    Put name= value=;
  End;
  Do i=1 To dim(n);
```

```
    name = vname(n(i));
    value = vvalue(n(i));
    Put name= value=;
  End;
Run;
```

Ein anderes Einsatzfeld von Variablenlisten sind Funktionen, denn viele Funktionen, die eine Liste von Variablen erlauben können auch mit Arrays oder Variablenlisten arbeiten. Hier ein Beispiel für die SUM-Funktion: alle Variablen, deren Name mit „X“ beginnt, werden aufsummiert.

```
Data _null_;
  x1 = 55;
  x2 = 12;
  x3 = 77;
  result = sum(of x:);
  Put x1= x2= x3= result=;
Run;
```

Auch die neuen Konkatenierungsfunktionen CAT*() unterstützen die Listensyntax, allerdings ist Vorsicht bei der Verwendung von _NUMERIC_ geboten. In diesem Fall muss die Variable, die den Wert aufnimmt, vorher als alphanumerisch deklariert sein. Die Konkatenierung von Text-Variablen funktioniert ohne vorherige Deklaration.

```
Data _Null_;
  Set Sashelp.Class(obs=1);
  string = catx(" ",of _character_);
  Put string=;
Run;
```

Bei numerischen Variablen muss nicht zwingend eine Deklaration erfolgen.

```
Data _Null_;
  x1=1;
  x2=2;
  x3=3;
  string = catx(" ",of x:);
  Put string=;
Run;
```

Wird _NUMERIC_ oder _ALL_ verwendet und die Variable ist nicht deklariert, kommt es jedoch zu einem Fehler.

```
Data _Null_;
*  Length string $1024;
  Set Sashelp.Class(obs=1);
  string = catx(" ",of _character_, of _numeric_);
  string = catx(" ",of _all_);
  Put string=;
Run;
```

Das LOG zeigt folgende Fehlermeldung:

```
NOTE: Character values have been converted to numeric values at the
places given by: (Line):(Column).
      43:12   44:12
NOTE: Invalid numeric data, 'Alfred M 14 69 112.5 .' , at line 43
column 12.
NOTE: Invalid numeric data, 'Alfred M 14 69 112.5 .' , at line 44
column 12.
string=.
Name=Alfred Sex=M Age=14 Height=69 Weight=112.5 string=. _ERROR_=1
_N_=1
```

Mit Deklaration funktionieren beide Varianten.

Und noch ein Beispiel für die Verwendung von Listensyntax. Die Call Routinen SORTC() und SORTN() sortieren Text oder Zahlen in aufsteigender Reihenfolge. Auch hier können Listen verwendet werden – allerdings nur für die aufsteigende Sortierung.

```
Data Work.Test;
  Length string $32;
  string = "original";
  x1 = 55;
  x2 = 12;
  x3 = 77;
  Output;
  string = "aufsteigend";
  Call Sortn( of x:);
  Output;
  string = "absteigend";
  Call Sortn( x3,x2,x1);
  Output;
Run;

Proc Print;
run;
```

Diese Funktionaliät wird hin und wieder auch in der Makrosprache benötigt, wenn z.B. ausgewählte Parameter in aufsteigender Reihenfolge verarbeitet werden sollen, das geht auch, allerdings funktioniert in der Makrosprache die Listensyntax nicht.

```
%Macro test(a,b,c);
  %Syscall sortc(a,b,c);
  %Put a=&a b=&b c=&c;
%Mend;

%test(cc,dd,aa);
```

1.4 PUT-Statement und Listensyntax

Weder _CHARACTER_ noch _NUMERIC_ sind als Platzhalter im PUT-Statement erlaubt. Nur _ALL_ ist zulässig, das Ergebnis jedoch nicht wie gewünscht. Es werden alle Variablen – auch die Data Step internen –in der Form VARIABLE=Wert ausgegeben. Mag das für die Debugging-Ausgabe während des Testens noch akzeptabel sein, für die Ausgabe in eine externe Datei ist das nicht schick. Tatsächlich gibt es aber eine Standard-Syntax für die PUT-Anweisung, die die verschiedenen Listen-Varianten unterstützt – und die hat die einfache Form:

```
PUT (VariablenListe) (modifier);
```

Es muss wenigstens ein Modifier angegeben werden, das kann ein Doppelpunkt (der macht nichts), ein „=“ (Ausgabe in der Form VARIABLE=Wert) oder auch ein „/“ (Zeilenumbruch) sein. Es gibt noch eine Reihe weiterer Möglichkeiten, die in der Online-Hilfe beim PUT-Statement dokumentiert sind.
Für das Debuggen im Data Step bieten sich folgende Varianten an:

```
Data _Null_;
  Set Sashelp.Class(obs=1);
  Put (_all_)(=);
Run;
```

oder

```
Data _Null_;
  Set Sashelp.Class(obs=1);
  Put (n:w:)(=);
Run;
```

Für den Export in CSV gibt es eine spezielle Syntax, die mit dem File-Statement zusammen verwendet wird:

```
Data _Null_;
  Set Sashelp.Class;
  File Print dsd;
  Put (_all_)(~);
Run;
```

DSD im File-Statement steht für „delimiter sensitive data“ und sorgt in Kombination mit „~“ im PUT-Statement dafür, dass alle Werte in doppelte Anführungszeichen gesetzt und die Werte durch Kommata getrennt werden – damit ergibt sich eine allgemein gültige Exportroutine von SAS nach CSV.

1.5 %PUT

Auch für das PUT-Statement der Makrospache gibt es eine Erweiterung. Analog zum Data Step, in dem `PUT variable=;` zur Ausgabe des Variablenwertes im LOG

führt, kann in der Makro-Sprache ab Version 9.3 die Syntax `%PUT &=variable;` verwendet werden. Diese kleine aber feine Erweiterung wird zukünftig viel Tipparbeit beim Debuggen von Makros einsparen.

2 MACH ES! JETZT! LOKAL!

Die neuen Funktionen DOSUB() und DOSUBL() ermöglichen es, aus dem Data Step heraus direkt SAS/Base Code auszuführen und dann auch auf die Ergebnisse zuzugreifen. Im Unterschied zu CALL EXECUTE() wird also nicht gesammelt und nach dem Data Step ausgeführt, sondern der Code zur Laufzeit des Data Steps in einem zweiten Thread abgearbeitet. Allerdings handelt es sich hier ausdrücklich nicht um ein Parallelisierungsverfahren, sondern der Data Step wartet auf das Ergebnis der Funktion, bevor er weiterläuft. Während DOSUB() ein Fileref erwartet, kann an DOSUBL() direkt Base-Code übergeben werden.

Ein einfaches Beispiel zeigt, wie die Syntax grundsätzlich funktioniert:

```
Data _Null_;
  code = "Proc Print data=Sashelp.Class; Run;";
  rc = dosubl(code);
Run;
```

Beide Funktionen stehen sowohl in der Makrosprache als auch in benutzerdefinierten Funktionen (PROC FCMP) zur Verfügung – hier lässt sich eine Vielzahl an Anwendungsfällen identifizieren. Es folgen ausgewählte Beispiele ohne Anspruch auf Vollständigkeit.

2.1 Einfache Metadatensteuerung für Programme

Damit lassen sich viele Automatisierungschritte, für die bisher die Makro-Sprache notwendig war, auch im Data Step steuern, z.B. die Verarbeitung mehrerer Tabellen hintereinander. Das Grundprogramm wird in einem Data Step zusammengestellt und mit DOSUBL() aufgerufen, die variablen Teile werden über Metadaten aus einer weiteren Tabelle dazugelesen. Im Beispiel hier wird zunächst eine Tabelle mit den Namen der auszugebenden Tabellen erstellt. Diese Tabelle wird dann als Input für den Programm-Generator verwendet.

```
Data work.tables;
  Length table $41;
  Do table="Sashelp.Class","Sashelp.cars","Sashelp.prdsale";
    Output;
  End;
Run;
```

```
Data _Null_;
  Set work.Tables;
  code = cats("proc print data=",table,"(obs=5);run;");
  rc = dosubl(code);
Run;
```

2.2 Zugriff auf die Ergebnisse aus dem selben Data Step

Da die mit DOSUB[L]() aufgerufenen Programme abgearbeitet werden, bevor der Haupt-Data Step weiter ausgeführt wird, kann anschließend auf die Ergebnisse der Unterroutine zugegriffen werden. Um dies zu illustrieren, wird im folgenden Beispiel zunächst ein PROC SQL-Step zusammengesetzt, der eine Liste der Namen der Schüler aus Sashelp.Class erzeugt und in die Makro-Variable NAMESLIST speichert. Anschließend wird die Anzahl der im SQL verarbeiteten Sätze abgefragt (SQLOBS) und dann in einer Schleife die Namen der Schüler aus der Liste gelesen.

```
%Let namesList =;
Data _Null_;
  Length curName $256 namesList $ 1024;
  * GP SQL-Statement zusammensetzen und abschicken *;
  code = "Select Distinct name Into :namesList Separated By '#' "!!
         "From sashelp.class;";
  rc = dosubl(cats("Proc Sql noprint;",code,"Quit;"));
  * GP Anzahl Obs und Liste der Namen holen *;
  n = input(symget("sqlobs"),best.);
  namesList = symget("namesList");
  * GP Namen ausgeben *;
  Do i=1 To n;
    curName = scan(namesList,i,"#");
    Put (i curName)(=);
  End;
Run;
```

Äußerer und innerer Code können also über Makro-Variablen miteinander interagieren!

2.3 Makrofunktionen, die Base-Code ausführen

Da DOSUB[L]() auch in der Makrosprache verfügbar ist (via SYSFUNC()), können jetzt Makrofunktionen erstellt werden, die Base-Code ausführen und dann trotzdem einen Rückgabewert haben. Das folgende Beispiel greift noch einmal PROC SQL auf und definiert das Makro %DistinctValues(), das als Rückgabewert die distinkte Liste der Ausprägungen einer Variable hat. Parameter sind Tabellen- und Variablenname, es wird ein PROC SQL-Schritt zusammengesetzt und ausgeführt und dann einfach die Liste der Werte zurückgegeben.

```
%Macro distinctValues(table,var);
  %Local code list rc;
  %Let code = Proc SQL noprint%Str(;);
  %Let code = &code Select Distinct &var Into :list Separated By
"#";
  %Let code = &code From &table%Str(;) Quit %Str(;);
  %Let rc = %Sysfunc(dosubl(&code));
  &list
%Mend;
%Put NOTE: Liste:
%distinctValues(sashelp.class(where=(sex="M")),name);

%Put NOTE: Liste: %distinctValues(sashelp.cars,make);
```

2.4 Bekannte Probleme

Der mittels DOSUB[L]() abgesetzte Code läuft in einem eigenen Thread. Vor Ausführung des Codes werden die Makro-Variablen aus der aufrufenden Umgebung in den Thread hinein- und anschließend wieder zurückkopiert (allerdings nicht alle). Das führt (zumindest in SAS Version 9.4M1) zu einem Fehler, wenn schreibgeschützte Makro-Variablen angelegt wurden.

```
%Macro test;
  %Local/ readonly xx = xx;
  Data _null_;
    code = '%Put NOTE: [dosubl()] &=sysindex;';
    rc = dosubl(code);
    Put "NOTE: [thisSession] Sysindex=&sysindex";
  Run;
%Mend;
%Test;
```

Dieser Code führt zu folgender Fehlermeldung:

```
ERROR: The variable XX was declared READONLY and cannot be modified
or re-declared.
NOTE: [dosubl()] SYSINDEX=0
NOTE: [thisSession] Sysindex=5
```

Aktuell lässt sich das nur vermeiden, indem auf die Kombination von DOSUB[L]() mit schreibgeschützten Makro-Variablen verzichtet wird.
Die Variable SYSINDEX dient hier als Beispiel für eine Makro-Variable, die nicht in die Umgebung des zweiten Threads kopiert wird, sie hat dort bei jedem Aufruf den Anfangswert 0.

Wenn per DOSUB[L]() Makros kompiliert werden, muss sichergestellt sein, dass der Katalog WORK.SASMACR schon existiert. Wenn das nicht der Fall ist, wird ein Katalog WORK.SASMAC2 angelegt – der nicht im Suchpfad für Makros enthalten ist.