The Progamma Library of Blaise Routines

Dirk Sikkel, Sixtat, The Netherlands

1. Introduction

The Progamma Library of Blaise Routines contains a number of block and procedures that are in a way awkward, given the original philosophy of Blaise. This awkwardness has different causes:

1. *Randomization*. This is a problem because after each change in the data Blaise recalculates all its fields, including those which contain random numbers
2. *Binary data storage of set questions* (multiple response questions). This has not been included in the basic data structure of Blaise
3. *Procedures for marketing research of scientific research*. Such procedures have not been emphasized as Blaise was developed in an official statistics environment.

Part of the 'legacy' of Interuniversitair Expertise Centrum Progamma has been used to deal with these problems. The software that has been developed by Sixtat handles the need for randomization, binary storage and marketing research procedures in the best possible way, although some solutions still may seem artificial to the end user. The source of the Progamma Library is available to Blaise users. They are encouraged to adapt the software to their needs. No support by either IEC Progamma or Sixtat is available. Those who will use the randomization procedures should read section 4, in which two problems are described which have cost the author a lot of time.

2. Contents of the library

The Progamma Library of Blaise Routines contains the following elements.

PROCEDURE SortInd. Sorts an index array of integers.
Typical application: show a sorted list of strings to the respondent.

BLOCK BRandomize. Generates a permutation of the numbers 1, 2, ..., n.
Typical application: randomization of a set of strings, to be used for randomization purposes.

BLOCK BToss. Flip a coin (unbiased)
Typical application: create two equally sized random groups each of which answers a different version of the questionnaire.

BLOCK BBiatoss. Flip a coin (biased)
Typical application: create two unequally sized random groups each of which answers a different version of the questionnaire.

PROCEDURE Decompose. Decomposes a string into an array of substrings.
Typical application: storage of a number of a number of descriptions of response categories in one large string.

BLOCK BRandAns. Generate a question with response categories in random order.
Typical application: presentation of questions of which it is suspected that the response is influenced by the order of the response categories.

BLOCK BRandSet. Generate a random set question (multiple response question) with response categories in random order and write the answers in binary format.
Typical application: presentation of set questions of which it is suspected that the response is influenced by the order of the response categories.

BLOCK BBinSet. Writes the answers to a set question (multiple response question) as binary data to the data file.
Typical applications: (1) when all answers have to be inspected for subsequent calculation or routing this is possible by a simple for-loop; (2) in data analysis the results can be used immediately for e.g. multiple response analyses in SPSS.

PROCEDURE CompStr. Match open answers with known strings.
Typical application: recognition of products and brands in open answers, which can be used for subsequent routing or calculations.

BLOCK BGaborg. Measures purchase intentions for different price levels for a single product.
Typical application: calculation of demand curves of a brand or product without considering the prices of competing brands. This is known as the Gabor-Granger procedure.

BLOCK BBPTO. Measures product preferences for different price levels.
Typical application: calculation of effects of reactions of competitors of a given product.
This is known as the Brand Price Trade Off procedure.

3. Description of the individual routines

In order to use the Programma Library, the source of the questionnaire should contain the following statement

```
INCLUDE "Progamma_Library.inc";
```

This makes available the following types, which are used by the library

```
TIntArr = array[1..30] of integer;
TPermutation = array[1..30] of 1..30;
TOrder = (asc, des);
TCoin = (head, tail);
TAnsArr = array[1..30] of string[80];
TQuestTx = string[1200];
TChar = string[1];
TProdArr = array[1..12] of string[80];
TPricArr = array[1..12] of real;
```

The bounds of the arrays are in a way arbitrary, and may be changed by the user. He should, however, be aware that by doing so he also changes the limitations within the different routines.

PROCEDURE SortInd

Purpose

Sorts an index array of type TPermutation of integer array Arr of type TIntArr

Syntax

```
SortInd(Arr, Ind, n_items, Order)
   Arr: TintArr, array of integers to be sorted
   Ind: TintArr, index array; after sorting Arr[Ind[.]] is an ordered
        sequence of integers
   n_items: integer, number of elements of Arr to be sorted
   Order: Torder, if Order=asc: sort ascending;
                  if Order=des: sort descending
```

Application
SortInd is a method for showing sorted lists of strings, when the order has to be based on answers of the respondent.

BLOCK BRandomize

Purpose
Generates a permutation of the numbers 1, 2, ..., n_items, stored in an array of type TPermutation.

Syntax

```
Randomize(Ind, Inv, n_items).
   Ind: array that contains the permutation
   Inv: array that contains the inverse permutation, such that Inv[Ind[i]]=i
   n_items: number of elements of Ind to be permutated
```

Application
BRandomize can be used for indexing a series of strings in random order. These strings can be used for generating questions with text variables, which appear in random order or for a set of possible responses which appear in random order. The array Ind can be used to determine the answer in the original order. The array Inv contains additional information about the rank order of given texts. When Ind and Inv are defined as fields, this infomation is stored. When they are displayed as auxfields, the information about order in the questionnaire is lost.

Limitation
Maximum size of permutation is 30; this can be increased by changing all numbers 30 to a higher number (also in the type-definition of TPermutation!)

Example

```
DATAMODEL IndInv;
INCLUDE "Progamma_Library.inc";

LOCALS
  i: integer;
  Older: array[1..12] of string[100];

FIELDS
  IndOld, InvOld: TPermutation;
  Randomize: BRandomize;
  QOlder {example of questions of a single type in random order}
     "Please indicate how much you agree with the following statement
     @/@/^Older[IndOld[i]]
     @/@/1 means: completely disagree
     @/5 means: completely agree":
  array[1..3] of 1..5;

RULES
  Older[ 1]:='I feel very responsible for my loved ones';
  Older[ 2]:='I feel myself more responsible for others than when I was younger';
  Older[ 3]:='I am just as care-free as when I was 18';
  Randomize(IndOld, InvOld, 3);
  for i:=1 to 3 do QOlder[i] enddo;
```

BLOCK BToss

Purpose
Generate two equally likely branches in a questionnaire

Syntax

Toss(Nickel)
  Nickel: Tcoin, where Nickel is equally likely head or tail

Application
Toss can be used for the random creation of two equal groups that answer different questions

Example

```
DATAMODEL TossUnBiased;
INCLUDE "Progamma_Library.inc";
FIELDS
  Nickel: TCoin;
  Toss: BToss;
  NickHead "Nickel is head": string[1];
  NickTail "Nickel is tail": string[1];
RULES
  Toss(Nickel);
  if Nickel=head then NickHead else NickTail endif
```

BLOCK BBiaToss

Purpose
Generate two branches in a questionnaire with unequal likelihood

Syntax
```
BiaToss(Nickel,p)
   Nickel: Tcoin, where Nickel is has probability p to be head and probability 1-
p
           to be tail
   p: real number between 0 and 1
```

Application
Toss can be used for the random creation of two unequal groups that answer different questions

Example
```
DATAMODEL TossBiased;
INCLUDE "Progamma_Library.inc";
FIELDS
  Nickel: TCoin;
  BiaToss: BBiaToss;
  NickHead "Nickel is head": string[1];
  NickTail "Nickel is tail": string[1];
RULES
  BiaToss(Nickel);
  if Nickel=head then NickHead else NickTail endif
```

PROCEDURE Decompose

Purpose
Decomposes a string into an array of substrings, based on a user-specified separator.

Syntax
```
Decompose(Ans, AnsArr,c,m)
   Ans: TQuesTxt, the input string of length 1200 (can be modified)
   AnsArr: TAnsArr, the array[1..30] of string[80], the array of substrings
   c: Tchar, the separator between two substrings
   m: integer, the number of substrings (output parameter)
```

Application
In progamma_library Decompose is used for the construction of a variable number response categories out of a single string. Possibly, users may find other applications.

Example
```
Decompose ('French,German,English,Dutch',language,',',m )
```
Which yields
  Language[1]=French,
  Language[2]=German,
  Language[3]=English,
  Language[4]=Dutch
  m=4

BLOCK BRandAns

Purpose
Generate a question with response categories in random order.

Syntax
```
RandAns(QuesTxt, AnsTxt, n)
   QuesTxt: TQuesTxt, the string[1200] that contains the question text.
   AnsTxt: TQuestTxt, the string[1200] that contains the texts of the response
          categories, separated by comma’s
   n: integer, the number of response categories
```

The (original) number of the given answer is written to the field RandAns.Answer.
The separator between the response categories may be changed by the user, by changing the value of separator, e.g.

Separator:=’_’

It may happen that the order of the last one or two response categories has to fixed, e.g. in the case of *don’t know* or *not applicable*. This can be achieved by specifying n be smaller than the actual number of answers. If AnsTx contains m separate texts, then the last m-n answers are not randomized. If n is specified as 0, the number of response categories is determined within BRandAns.

Application
BRandAns is a closed question with possible answers in random order.

Example
```
DATAMODEL RandSing;
INCLUDE "Progamma_Library.inc";

LOCALS
  Color: array[1..7] of string[10];

FIELDS
  Prefer: BRandAns; {single answer, random order}
  ShowPref "The best liked color has (original) number ^Prefer.Answer and is
    ^Color[Prefer.Answer]": TContinue;

RULES
  Color[1]:='Yellow'; Color[2]:='Red'; Color[3]:='Blue'; Color[4]:='Green';
  Color[5]:='Purple'; Color[6]:='Aqua'; Color[7]:='Magenta';
  Prefer('What color do you like best?','Yellow,Red,Blue,Green,Purple,Aqua,
          Magenta',0); {separator is comma (default)}
  ShowPref;
```

Limitations
BRandAns allows for a maximum of 30 response categories; this can easily be changed by the user by

- adding response categories to BRandAns.Questio
- modifying the bounds in the block Brandomize

BLOCK BRandSet

Purpose
Generate a random set question (multiple response question) with response categories in random order.

Syntax
```
RandSet(QuesTxt, AnsTxt, n)
   QuesTxt: TQuesTxt, the string[1200] that contains the question text.
   AnsTxt: TQuestTxt, the string[1200] that contains the texts of the response
          categories, separated by comma’s
   n: integer, the number of response categories
```

The given answers are written to the array [1..30] of 0..1 RandSet.Answer. Thus, they are coded binary, and in the original order as corresponds to AnsTxt.
The separator between the response categories may be changed by the user, by changing the value of separator, e.g.

Separator:=’_’;

It may happen that the order of the last one or two response categories has to fixed, e.g. in the case of *don’t know* or *not applicable*. This can be achieved by specifying n be smaller than the actual number of answers. If AnsTxt contains m separate texts, then the last m-n answers are not randomized. If n is specified as 0, the number of response categories is determined within BRandSet.

Example
```
DATAMODEL RandMult;
INCLUDE "Progamma_Library.inc";

LOCALS
  i: integer;
  Color: array[1..7] of string[10];
  LikeCol: string;

FIELDS
  Like: BRandSet; {multiple answer, binary data, random order}
  ShowLike "The liked colors are ^LikeCol": string[1];

RULES
  Color[1]:='Yellow'; Color[2]:='Red'; Color[3]:='Blue'; Color[4]:='Green';
  Color[5]:='Purple'; Color[6]:='Aqua'; Color[7]:='Magenta';
  Separat:='_';
  Like('Which colors do you
like?','Yellow_Red_Blue_Green_Purple_Aqua_Magenta_none
        of these',7); {separator changed to underscore}
  LikeCol:='';
  for i:=1 to 7 do
    if Like.Answer[i]=1 then LikeCol:= LikeCol + ' '+Color[i] endif
  enddo; {adressing answers in a loop!}
  ShowLike;
```

Limitations
BRandSet allows for 30 response categories. This can easily be changed by the user by

- adding or deleting response categories in BRandSet.Questio
- modifying the bounds in the block BRandomize
- changing the dimension of BrandSet.Answer

BLOCK BBinSet

Purpose
Writes the answers to a set question (multiple response question) as binary data to the data file.

Syntax

```
BinSet(QuesTxt, AnsTxt)
   QuesTxt: TQuesTxt, the string[1200] that contains the question text.
   AnsTxt: TQuestTxt, the string[1200] that contains the texts of the response
          categories, separated by comma's
```

The given answers are written to the array [1..30] of 0..1 BinSet.Answer. Thus, they are coded binary.
The separator between the response categories may be changed by the user, by changing the value of separator, e.g.

separator:='_';

Example

```
DATAMODEL BinMult;
INCLUDE "Progamma_Library.inc";

LOCALS
  i: integer;
  Color: array[1..7] of string[10];
  HateCol: string;

FIELDS
  Hate: BBinSet; {multipe answer, binary data, fixed order}
  ShowHate "The hated colors are ^HateCol": TContinue;

RULES
  Color[1]:='Yellow'; Color[2]:='Red'; Color[3]:='Blue'; Color[4]:='Green';
  Color[5]:='Purple'; Color[6]:='Aqua'; Color[7]:='Magenta';
  Separat:='/';
  Hate('Which colors do you hate?',
    Color[1]+'/'+Color[2]+'/'+Color[3]+'/'+Color[4]+'/'+Color[5]+'/'+Color[6]+'/'
     +Color[7]+'/none of these/do not know');
  HateCol:='';
  for i:=1 to 7 do
    if Hate.Answer[i]=1 then HateCol:= HateCol + ' '+Color[i] endif
  enddo;
  ShowHate;
```

Limitations
BBinSet allows for 30 response categories. This can easily be changed by the user by

- adding or deleting response categories in BBinSet.Questio
- changing the dimension of BBinSet.Answer

PROCEDURE CompStr

Purpose
Match open answers with known strings with the possibility of wildcard and three alternative spellings.

Syntax

```
CompStr(t, t1, t2, t3,equal)
  t: string, input parameter (typically an open answer)
  t1, t2,  t3:  string, input parameters (e.g. product names or brand names)
  equal: integer, output parameter, which is
    1 if t matches t1, t2, or t3
    0 otherwise
```

t2 and t3 may be empty strings (no match)
t1, t2 and t3 may contain up to 5 wildcards, denoted by the underscore (_)
Comparison is carried out case-insentitive (no distinction is made between UPPER CASE and lower case)

Application
A typical application is the matching of an open answer of the respondent with a known product name or brand name, allowing for spelling errors by the respondent.

Example

```
CompStr(t,'Statistic_Ne_erland','Centraal_Bur_Statistiek','',match) yields
   t= Statistics Netherlands              match=1
   t= Centraal Bureau voor de Statistiek  match=1
   t= centraal bureau voor de statistiek  match=1
   t= CBS                                 match=0
   t= Statistiks Netherlands              match=0
```

BLOCK BGaborg

Purpose
BGaborg is a block that contains a standard procedure in marketing research to estimate demand curves of a brand or product without considering the prices of competing brands. This is known as the Gabor-Granger procedure.

Syntax

```
Gaborg(ProdName, BasePric, Increm, n)
   ProdName: string, name of the brand or product
   BasePric: real, the base price of the product (usually the price as marketed)
   Increm: real, steps by which the price is increased and decreased
   n: integer, number of increments and decrements (maximum = 5)
```

ProdName has to be a complete description of the product, e.g. “2.5 kilograms of Dreft Washing Powder”, as the respondent has to judge if the indicated price is a fair price.
The price of the product is set at different levels, and the respondent is asked if he would buy the product for that price. The prices are presented in random order.

The answers are stored in the array Gaborg.purch; purch[1] contains the response (yes or no) on the base price; purch[2] ... [purch[n+1] contain the responses to the decreased prices (BasePric-Increm,

..., BasePric-n*Increm); purch[n+2] ... purch[2n+1] contain the responses to the increased prices (BasePric+Increm, ..., BasePric+n*Increm)

Application
The Gabor Granger procedure is used to determine the price elasticity in a market where no direct reaction from competitors is to be expected. From the data can be estimated what percentage of the consumers buys a product at a given price.

Example
```
DATAMODEL ShowGaborg;
INCLUDE "Progamma_Library.inc";

LOCALS
  Product: string;
  MarketPr: real;
  inc: real;
  n_inc: integer;

FIELDS
  Gaborg: BGaborg;

RULES
  Product:= '2.5 kg of Dreft washing powder';
  MarketPr:= 6.0 ;
  inc:= 0.25; n_inc:=3;
  Gaborg(Product, MarketPr, inc, n_inc);
```

BLOCK BBPTO

Purpose
BBPTO is a block that contains a standard procedure in marketing research to estimate demand curves of a brand or product while considering the prices of competing brands. This is known as the Brand Price Trade Off procedure.

Syntax
```
BPTO(ProdArr, PricArr, Increm, n_prod, n_steps)
     ProdArr: TprodArr, array[1..12] of string[80] with the names of the products
     PricArr: TpricArr, array[1..12] of the base prices
              (usually the prices as marketed)
     Increm: real, the increment, steps by which the prices are increased
     n_prod: integer, number of products
              (maximum 12; it is recommended not to exceed 8)
     n_steps: integer, the number of steps (maximum 20)
```

ProdArr has to contain complete descriptions of the products, e.g. “2.5 kilograms of Dreft Washing Powder”, as the respondent has to judge if the indicated prices are fair prices.
The respondent has to indicate his product preference with prices at base level; subsequently, the price of the preferred product is increased by Increm, and the procedure is repeated n_steps times.

The results are written in the array BPTO.pref; pref[i] is the number of the preferred product in step i. It is recommended to present the products in random order; the most efficient way to do this is outside the block BBPTO, i.e. the array ProdArr already contains the products in random order, see Brandomize.

Application

The BPTO procedure is used to determine the price elasticity in a market where direct reactions from competitors are to be expected. From the data can be estimated onder what conditions what percentage of the consumers buys a product at a given price.

Example

```
DATAMODEL ShowBPTO;
INCLUDE "Progamma_Library.inc";

LOCALS
  inc: real;
  n_inc: integer;

AUXFIELDS
  Product: TProdArr;
  MarketPr: TPricArr;

FIELDS
  BPTO: BBPTO;

RULES
  Product[1]:= 'Blue Band'; Product[2]:='Becel'; Product[3]:='Bona';
  MarketPr[1]:= 1.0; MarketPr[2]:= 0.94; MarketPr[3]:= 1.12;
  inc:= 0.10; n_inc:=8;
  BPTO(Product, MarketPr, inc, 3, n_inc);
ENDMODEL.
```

4. Pitfalls for randomization routines

As noted in section 1, the basic structure of Blaise is not very ‘randomization-friendly’. Therefore, the randomization routines should be used with care. The basic rule is that an instantiation of a block can be used only once. So for two tosses of a coin, two fields of type BToss have to be declared:

```
  Toss1, Toss2: BToss;
  Dime1, Dime2: TCoin;
```

Only then the rules

```
  Toss1(Dime1);
  Toss2(Dime2);
```

yields two independent throws of a coin, whereas

```
  Toss1(Dime1);
  Toss1(Dime2);
```

yields identical outcomes, i.e. Dime2=Dime1.

A second problem to keep in mind is that Blaise evaluates the randomization procedure the first time it is activated on the route and never changes the outcome afterward. This may create a problem for random permutations when the number of integers for permutation is not known beforehand. Both the blocks BRandAns and BrandSet are vulnerable to this problem. The solution to the problem is to make sure that the permutation is not on the route as long as the number of integers to be permuted is unknown. An example of how to solve this problem is shown in the file ShowNestSet.bla.

5. Available files

All routines are contained in the file

Progamma_Library.inc

Apart from the source, this file also contains more technical information about the routines than is given in this document. The routines are illustrated by a number of files with examples:

ShowBPTO
ShowGaborg
ShowIndInv
ShowNestSet
ShowRandAns
ShowSort
ShowStrComp