NEW FEATURES OF BLAISE

Martje Roessingh and Peter Stegehuis
Netherlands Central Bureau of Statistics, Voorburg

1. Introduction

In this paper three new features in Blaise are described: lists, user functions and procedures and coding with trigrams. Lists and user functions are implemented in version 2.4, coding with trigrams will be available in the autumn of 1992. We start in the next section with a simple example and a short introduction to the use of lists. These lists are simple external files that are stored in normal or extended memory. Consulting lists is therefore much faster than consulting external files.

Section three describes a new way of coding. Coding in Blaise can be done step-wise, with an hierarchical classification, or alphabetical by searching a dictionary. In the version of Blaise that is currently under development a more sophisticated algorithm to search the dictionary is implemented in which trigrams (three-letter combinations) are used to build an index for the vocabulary. Coding with trigrams is not included in version 2.41, so you will not find any documentation on this subject in the *Blaise 2.4 Update Manual*. It will be distributed separately.

The subject of the last section is user functions and user procedures. This subject is also treated in the *Update Manual*. Here a small example is presented.

2. Lists

Lists are an alternative to external files: lists are files that are loaded completely into memory, and therefore searching lists will usually be quicker than searching "normal" external files. Lists are stored in conventional and extended memory. They differ in form from external

files: lists are sorted and don't have an index file. The use of lists is similar to the use of external files: you can search and read lists.

There are three different forms of lists: binary, ASCII without separators and ASCII with separators. The records in a list all have the same length and follow each other directly. To make a binary list of an ASCII or a Blaise file is very easy with the program Manipula.

The lists are read into the memory of the computer at the beginning of the program. They are stored in conventional memory or in extended memory, but they cannot be stored in expanded memory. If there is insufficient memory for all lists at the same moment, there will be some shifting and reading at run time. The program tries to keep lists that are used most in memory, but this cannot be garanteed. So reading and shifting may cost a lot of time.

Instead of explaining lists in detail, we will show the use of a list with a simple example.

```
QUESTIONNAIRE Prices;

EXTERNAL
  VAR
    ArtPrice: LIST "ARTPR92.LST" (INTEGER [4], REAL, REAL);

ENDEXTERNAL;

VAR
  LowerBnd, UpperBnd: REAL;

QUEST
  Number: 1..99 (KEY);
  ArtCode "What is the article code": 1..9999;
  ArtPrice "What is the price of the article": 0.00..1000.00;

ROUTE
  Number; ArtCode; ArtPrice;

CHECK
  SEARCHLIST (ArtPrice, ArtCode, ?LowerBnd, ?UpperBnd)
    "The article code does not exist";
  ArtPrice > LowerBnd AND ArtPrice < UpperBnd
    "Price must be between $LowerBnd and $UpperBnd";

ENDQUESTIONNAIRE.
```

A list has to be declared in the external paragraph of the questionnaire. In our example:

```
EXTERNAL
  VAR
    ArtPrice: LIST "ARTPR92.LST" (INTEGER [4], REAL, REAL);

ENDEXTERNAL;
```

ArtPrice is the name that is used later in the questionnaire to refer to the list. On disk the list can be found in the file with the name ARTPR92.LST. This name can also begin with a full path to a directory where the list is stored. INTEGER [4], REAL, REAL means that the records in the list consist of an integer (of 4 bytes) and two reals.

In the example the list is used for two things. Firstly the code of the article (answer to the question Article) must be in the list. So there has to be a record starting with this code. Here this code is an integer. It can be checked this way:

```
CHECK
  SEARCHLIST (ArtPrice, ArtCode)
    "The article code does not exist";
```

SEARCHLIST, like searchfile, is used as a *direct check*: if there is no record beginning with the code an error will occur. In our example we use SEARCHLIST also to read two variables: LowerBnd and UpperBnd, both of type real. To indicate that a variable must be read there is a question mark before this variable. So SEARCHLIST reading the two variables can be formulated like this:

```
  SEARCHLIST (ArtPrice, ArtCode, ?LowerBnd, ?UpperBnd)
```

In another direct check we use those variables to check the price of the article. This is nothing new:

```
  ArtPrice > LowerBnd AND ArtPrice < UpperBnd
    "Price must be between $LowerBnd and $UpperBnd";
```

Besides SEARCHLIST there is READLIST which works analogous to READFILE and READBLAISE. In the *Update Manual* you can find the exact details of the syntax and more examples of the use of lists.

At the CBS lists are in use in a special test version of Blaise since spring 1991. The Department for Foreign Trade Statistics is the most important tester of this feature. It uses lists in several applications, for example in a big questionnaire with only 27 questions but with very complex checks. The questionnaire uses 73 lists, together 1.8 megabytes, and is part of a larger batch system. The batch program starts with some archiving and

two conversions and ends with an integral check on all forms. Those steps together take about 17 minutes for 1300 forms, which is less then one second per form.

Lists are a fast alternative to external files, but there are limitations to their sizes, so it is impossible to substitute a list for every external file. In the *Update Manual* you can read more about the use of lists.

3. Coding with trigrams

Since last spring the coders of the CBS survey of family expenditures are testing a new method of coding in Blaise, which uses so called trigrams: three-letter combinations. This new method resembles an alphabetical search, but the list that appears on the screen consists of descriptions that partially match the searched word. The spelling does not have to be correct, since all descriptions that look like this word will be displayed. An advantage to a hierarchical search is that the coder for example does not have to know whether tomatoes are vegetables or fruit.

In this section we begin with a small example showing some screens made with a very small codelist. Later we explain some of the basics of the algorithm of this method.

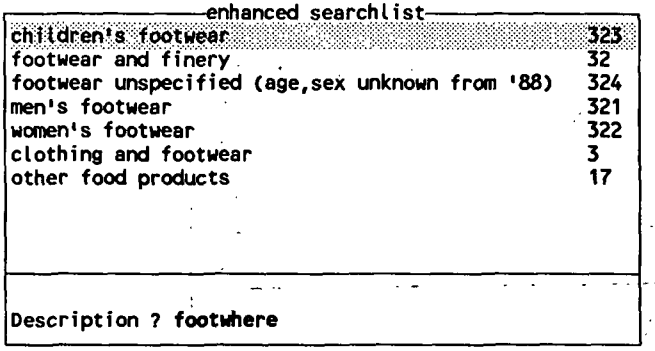
The dictionary of this example contains 100 descriptions of the budget survey and is taken from a publication in English. When we look for "footwear" using the alphabetical search, the screen will look like this:

```
                    alphabetical list
footwear and finery                                    32
footwear unspecified (age,sex unknown from '88)       324
fruit                                                 123
furnishing unspecified                                225
furniture                                             221
furniture, upholstery and linen                        22
game and poultry                                      153
garden and flowers                                    214
gas and electricity                                   241
general body care                                     421

Description ? footwear
```

Using the trigram method we get this screen:

```
                      searchlist
men's footwear                                         321
women's footwear                                       322
children's footwear                                    323
footwear and finery                                     32
clothing and footwear                                    3
footwear unspecified (age,sex unknown from '88)        324

Description ? footwear
```

The alphabetical list only shows the two descriptions that start with "footwear", while the other list consists of every description containing this word. If we don't know how to spell "footwear", but instead look for the word "footwhere", we will get the following screen:



On this screen we find the same descriptions containing "footwear" and also one with "food products". The last one is at the end of the list because it resembles "footwhere" less than the others.

For the coding with trigrams a kind of index is created for all the descriptions in the alphabetical list. The index consists of trigrams with references to the words in which they occur. For example the word "book" is split up in the trigrams " bo", "boo", "ook" and "ok ". So " bo" can be found in the index with a reference to the word "book" among all the references to other descriptions containing the trigram " bo". When a trigram occurs in too many descriptions it will not be placed in the index. In our example this is the case with the trigrams " an", "and" and "nd ", for the word "and" appears in more then 50 descriptions. This threshold can be set by the user when he makes a trigram index.

At the moment a data entry program using trigrams is started, the index will be read into memory (normal or extended). It is also possible to use it from disk but that will slow down the program. When the coder searches a word, it is split in trigrams. The program creates a list of all descriptions with matching trigrams. This list is sorted on the number of matching trigrams. First the descriptions containing at least 50% of the trigrams of the word are shown. If this list is empty the descriptions with 25% or more trigrams are shown. Here again these thresholds can be set

by the user. The displayed list is sorted so the most resembling descriptions come first.

A great advantage of the described method is that no knowledge of the subject is needed to create the index. It is also independent of the language you use. There can be some problems if words consist only of trigrams that are so common that they are not in the index. In that case the coder will have to look for a longer word or a combination of words. There is one general rule for all methods of coding in Blaise: the quality of the coding never better than the quality of your dictionary!

4. User functions and user procedures

In version 2.4 it is possible to use your own Pascal procedures and functions in Blaise data entry programs. User procedures give you complete control over the way you ask questions and plenty of possibilities to influence the layout of the screen. They are meant for those situations where you want to supply your own way of handling a question or a set of questions. For instance, showing a picture on the screen is not possible in Blaise, so if you want the respondent to look at a picture before answering, you can now use your own Pascal procedure.

You can also define your own functions. Like the known functions in Blaise, they have a name, take parameters and return a result. They can then be used in your Blaise program just like the existing standard functions (e.g. SUM, SQRT, JULIAN).

User procedures work on blocks. You have to give the attribute EXTERN to such a block to indicate that your procedure will take over questioning for this block. Your procedure has complete control over the way in which the questions are asked and has the responsibility to return proper answers to the Blaise program. This shows immediately how powerful this new feature is, and how dangerous. You can do many things that weren't possible in Blaise before, positive and negative.

To show the usage of user functions and procedures, a simple example of

a questionnaire with a Pascal unit. The unit contains one function and one procedure:

```
Unit MyOwn;

Interface
Uses DOS, CRT;
Function Age (BirthDat: String): LongInt;
Procedure MyQuest (EntryChar: Char; QuestNr: Word;
                   var Answers: String);
Implementation

Function Age (BirthDat: String): LongInt;
var
 DyBrth, MthBrth, YrBrth, res: Integer;
 DyNow, MthNow, YrNow, DayofWk : Word;
begin
 GetDate (YrNow, MthNow, DyNow, DayofWk);
 VAL (Copy (Birthdat, 1, 2), DyBrth, res);
 VAL (Copy (Birthdat, 4, 2), MthBrth, res);
 VAL (Copy (Birthdat, 7, 4), YrBrth, res);
 if (MthNow < MthBrth) or
    ((MthNow = MthBrth) and (DyNow < DyBrth))
  then Age := YrNow - YrBrth - 1
  else Age := YrNow - YrBrth;
end; {Age}

Procedure MyQuest (EntryChar: Char; QuestNr: Word;
                   Var Answers: String);
var
 Answer1, Answer2: String;
begin
 Answer1 :=Copy (Answers, 1, 20); Answer2 :=Copy (Answers, 21, 40);
 GotoXY (1, 24); ClrEol;
 Write ('What is your name?'); GotoXY (26, 24); Write (Answer1);
 GotoXY (26, 24); Readln (Answer1);
 if Length (Answer1) < 20
 then Answer1 :=Answer1 + '                    ';
 Answer1 := Copy (Answer1, 1, 20);
 GotoXY (26, 24); ClrEol; GotoXY (26, 24); Write (Answer1);
 GotoXY (1, 25); ClrEol;
 Write ('What is your profession? '); GotoXY (26,25); Write (Answer2);
 GotoXY (26, 25); Readln (Answer2);
 Answers := Answer1 + Answer2;
end; {MyQuest}

end.
```

Save this unit as MYOWN.PAS . The questionnaire could be:

```
QUESTIONNAIRE Example;

USES "MyOwn.Pas";
NOTEPAD = 4;

BLOCK MyQuest (EXTERN);
QUEST
  Name   "What is your name?": STRING[20];
  Profes "Please give a short description of
          your profession": STRING[40] (EMPTY);
ROUTE
  Name; Profes
ENDBLOCK; {MyQuest}

QUEST
  SeqNum  "Sequence number of this interview?": 1..10000 (KEY);
  NameProf: MyQuest;
  BrthDat "What is your birth date?": DateType;
  YourAge : 0..120 (PROTECT);
  MarStat "What is your marital status?":
           (Married, Notmar "Not married");
ROUTE
  SeqNum; NameProf; BrthDat; YourAge; MarStat

CHECK
  IF BrthDat = RESPONSE "" THEN
    JULIAN (SysDate) - JULIAN (BrthDat) > 0
    "A birthdate cannot be a future date!";
    YR (BrthDat) >= YR (SYSDATE) - 120
    "A person cannot be that old!";
    COMPUTE BrthDat := STDATE (BrthDat);
    COMPUTE YourAge := AGE (BrthDat);
  ENDIF
ENDQUESTIONNAIRE.
```

To be able to use the unit MYOWN.PAS in the questionnaire, you have to state

```
USES "MyOwn.Pas";
```

in the settings section. Note that the block in the questionnaire has the same name as the procedure in the unit and that it has the attribute EXTERN.

You can now check the syntax of the questionnaire and compile it. During the syntax check the file(s) mentioned after USES must be present (in this example MYOWN.PAS). During compilation either MYOWN.PAS or MYOWN.TPU must be present.

While interviewing, first the key question is asked and then instead of asking the questions in the block the program calls the external procedure to get answers to the questions Name and Profes. The answers to these questions are passed on from the Blaise program to the procedure, concatenated in one string Answers. When first called this string is still empty. In the procedure you can see that the Answers-string is 'unpacked' so you can work with the separate answers.

Then the question "What is your name?" is asked on line 24 of the screen and you can type the answer on this line. The answer is stored in the string Answer1, which is then brought to its proper length (twenty characters). Next the question "What is your profession?" is asked on line 25, and again you are prompted for an answer on the same line. This answer is stored in the string Answer2. At the end of the procedure the strings Answer1 and Answer2 are concatenated to the string Answers, which returns the new answers to the Blaise program.

It is important that the length of Answer1 is exactly twenty characters, because Blaise takes the first twenty characters as the answer to the question Name and the next forty characters of the Answers-string as the answer to the question Profes.

If you want to change the answer to one of the questions in the EXTERN block, you can go back to one of the questions. The program does not automatically call your procedure, since you may just want to page through your questionnaire. As soon as you start editing the answer, by pressing <F2> or a letter-key, your procedure gets called again. On your screen you see the question text as specified in your Blaise questionnaire,

and if you press a key (as if you were editing the answer), your procedure will take over again. Again on line 24 and 25 the two questions are asked, the answers you gave the last time are displayed, and you can change them. The new answers are passed to the Blaise questionnaire again.

The key you pressed to start editing is stored in the variable EntryChar - the first parameter in the heading of the procedure - and can be used in the procedure. For instance, in the Blaise question text you could include: "If you want ..., then press <A>; otherwise press <B>". The second variable in the heading, QuestNr, gives you the possibility to see from which question in the block you 'jumped' to the procedure. If it was the first question in the block (in this example the question Name), then QuestNr gets the value 1, if it was from the second question (here Profes), then QuestNr equals 2. If for instance in our example you start editing the question Profes - the second question in the EXTERN block, then the variable QuestNr can be used to make sure the procedure does not start from the beginning (asking "What is your name?"), but asks only "What is your profession?". In this example the variables EntryChar and QuestNr are not used.

After editing the questions in the EXTERN block, Blaise takes over, storing the given answers in the form. Next the question BrthDat asks for a date of birth. In the check paragraph this date is used to calculate the age of the respondent with the user function Age. This function takes as parameter the answer given to the question BrthDat, and returns the age of the respondent in years, using the system date of the computer for the calculation. The system date is obtained with the standard Pascal function GetDate.

This is obviously a very simple example: you can use the whole screen instead of just two lines, show pictures with the questions, etcetera. If you want to use your own functions or procedures in a questionnaire, you should test it extensively before the survey actually starts. Another small example and some more details about the use of user procedures and user functions can be found in the *Update Manual.*