THE B NEWSLETTER

CWI, Amsterdam

Issue 3, January 1985

IMPORTANT NOTICE

If you want a copy of the Mark 1 *B* Implementation, there is an application form on the back two pages. This applies equally to everyone who has written to us asking for a copy.

CONTENTS

IBM PC Progress

Porting the *B* implementation to the IBM PC is in progress. The two programs constituting the *B* system, the interpreter and the editor, were running within two weeks after we acquired the C compiler. Because of problems with getting these to run in parallel, however, we were obliged to merge the two in one program. That proved to be too big, of course. We have slimmed this integrated *B* system down and it now runs small programs. As soon as the available user space becomes acceptable, this version will be released. We hope to announce this in the next newsletter.

Notes for Contributors

The newsletter is intended to provide information about *B* and to provide a forum for discussions. Therefore, you are encouraged to submit any articles you see fit.

Articles don't have to contain fully thought-out ideas, but may be yet undeveloped thoughts intended to stimulate discussion. The kinds of articles we have in mind are: interesting programs, either written or suggestions; unusual applications; letters, discussions on points of the language, proposed improvements, experience with the language, and so on.

If you are fortunate enough to be connected to the Unix network, then you can submit articles and send mail to *mcvax!timo.* Otherwise, articles and mail should be sent to

B Newsletter
Computer Science Department
CWI
POB 4079
1009 AB Amsterdam
The Netherlands

What is in the name of *B*?

Lambert Meertens

When we started our project in 1975, we were not aware, as far as I can remember, of the existence of a language *B*. We decided to design our language by iteration: design a version, try it out, redesign it, etc., until it is really satisfactory. Since we expected the first versions to be unsatisfactory, but we also wanted to publish the designs for public scrutiny, and even possibly disseminate implementations, we did not want to fix a name for the language (for fear that the final product would get an undeserved bad name through the shortcomings of its predecessors). From [G&M] I quote:

To facilitate discussion, we have called the hypothetical language fulfilling these criteria "*B*".

The subsequent discussion goes on to talk about "B_0", "B_1", $\cdots$. When we had finished the design of B_2 and looked at what we had there, we were so enthusiastic that we decided to present it, even though we had one more — albeit minor — iteration scheduled, as "*B*", being $\lim_{i \to \infty} B_i$. (That is why we always use an italic "B".) We were still loath to fix a definitive name, and decided to postpone that till the final revision. Although at that time we knew that C once had a predecessor B, we were under the — apparently false — impression that it was no longer in actual use. (In response to [G&M] we received some reactions like "Hey, do you know that B was a predecessor of C?", but none of these suggested that B was in actual use.) Since all our publications since then used "*B*" to refer to the language (in particular [MEE]), it would also be confusing to change the name while the version remains the same; anyway, we always point out that this "*B*" has nothing to do with the other one.

Now that we are starting the revision process, it becomes time to decide on a better, definitive, name. Here is a list of all suggestions that have been made up till now:

ABC
ASBIC
Bee
Bottom
HowTo

ABC stands for nothing, but suggests something easy. It has an additional advantage that it is acceptable in many other languages than English. *ASBIC* is an acronym for *A*SBIC *S*pelled *B*ackwards *I*s *C*IBSA. *Bee* is, of course, phonetically the same as *B*. *Bottom*, the weaver, is a character from *A Midsummer-Night's Dream*, with whom *Titania*, the Queen of the Fairies, fell in love at first sight (even though he was looking rather asinine at the time). *HowTo*, finally, is derived from one of the more memorable features of the language, and has a nice ring.

Let us know which name you prefer, and which ones you hate or think otherwise deleterious or unhelpful to the promulgation of the language. If you think you have a better name, it goes without saying that it is mandatory that you respond.

REFERENCES

[G&M] Leo Geurts & Lambert Meertens, Designing a beginners' programming language, in: *New Directions in Algorithmic Languages 1975* (S. A. Schuman, ed.), 1-18, IRIA, Rocquencourt (1976).

[MEE] Lambert Meertens, *Draft Proposal for the* B *Programming Language, Semi- Formal Definition,* Math. Centre, Amsterdam (1981).

A File-Maintenance Program in *B*

Philomena Dunkl
Steven Pemberton

For a while this year Philomena Dunkl, an assistant professor of data-processing at Piedmont Virginia Community College, Charlottesville, Virginia, USA, visited the CWI, and during her stay took the time to learn *B*. Since her background is in commercial computer applications she wrote a 'file-maintenance' program, in other words a program to interactively insert, delete, and change data in a data-base. The version included here has been prepared for publication by Leo Geurts.

The data-base in the program is of employee records, containing names, addresses, birthdates, 'jobcodes' (not further defined), with each record keyed on an identifying code (also not further defined).

The data-structure used is a table, with texts as keys (the identifying codes) and compounds as associates, consisting of eight fields (all texts) for first name, last name, middle initials, street, zipcode (postcode), city, birthdate and jobcode. The program then allows you to add, delete, change, or display an entry in the data-base. It does this by displaying a menu and a prompt:

```
a. ADD NEW RECORD
d. DELETE RECORD
c. CHANGE RECORD
s. SHOW RECORD
q. QUIT PROGRAM

Enter function letter: ?
```

Typing one of these letters then selects the function wanted.

It can be seen that the program consists of one large unit with many refinements, and three other very small units. Of note are the two refinements `record'present` and `affirmative`. This latter for example allows the following:

```
WRITE / "Do you wish to change this record? "
IF affirmative: CHANGE'RECORD
```

and it looks like this:

```
affirmative:
   READ response RAW
   REPORT response in {"y"; "Y"; "yes"; "YES"}
```

Of course, this is only a small demonstration program, and several things would need to be added to make it a proper application package. For instance, there is no way to get an overview of the contents of the data-base, such as a list of all identifying codes used. However, the program as it stands would make a good starting-point for a more complete version.

Steven Pemberton

```
HOW'TO MENU:
   SHARE file
   WRITE / "FILE MAINTENANCE PROGRAM FUNCTIONS" /
   PUT "x" IN function
   WHILE function <> "q":
      DISPLAY'MENU
      SELECT'FUNCTION
DISPLAY'MENU:
   WRITE "a. ADD NEW RECORD" /
   WRITE "d. DELETE RECORD" /
   WRITE "c. CHANGE RECORD" /
   WRITE "s. SHOW RECORD" /
   WRITE "q. QUIT PROGRAM" //
SELECT'FUNCTION:
   ASK'TEXT "function letter" FOR function
   SELECT:
      function = "a":
         REQUEST'ID
         ADD'NEW'RECORD
      function = "d":
         REQUEST'ID
         IF record'present: DELETE'RECORD
      function = "c":
         REQUEST'ID
         IF record'present:
            SHOW'RECORD
            CHANGE'RECORD
      function = "s":
         REQUEST'ID
         IF record'present: SHOW'RECORD
      function = "q":
         WRITE / "*** File Maintenance Program is ending ***" /
         WRITE / "*** Have a nice day! ***" //
      ELSE: WRITE / "Please choose a, d, c, s or q." /
REQUEST'ID: ASK'TEXT "employee identification" FOR id
ADD'NEW'RECORD:
   SELECT:
      id in keys file: EXISTING'RECORD
      ELSE:
         \request record entries
         ASK'TEXT "Last Name" FOR last
         ASK'TEXT "First Name" FOR first
         ASK'TEXT "Middle Initial" FOR mid
         ASK'TEXT "Street Address" FOR addr
         ASK'TEXT "Zipcode" FOR zip
         ASK'TEXT "City" FOR city
         ASK'TEXT "Birthdate (DDMMYYYY)" FOR birth
         ASK'TEXT "Jobcode" FOR job
         STORE'RECORD
         WRITE / "*** Record of employee ", id, " has been added ***" /
EXISTING'RECORD:
   WRITE / "A record for employee ", id, " already exists." /
   SHOW'RECORD
   WRITE / "Do you wish to change this record? "
   IF affirmative: CHANGE'RECORD
```

```
record′present:
   SELECT:
      id in keys file: SUCCEED
      ELSE:
         WRITE / ″*** Record of employee ″, id, ″ cannot be found ***″ //
         FAIL
DELETE′RECORD:
   SHOW′RECORD
   WRITE ″Do you really want this record to be deleted? ″
   IF affirmative:
      DELETE file[id]
      WRITE / ″*** Record of employee ″, id, ″ has been deleted ***″ /
CHANGE′RECORD:
   WRITE ″For the following items enter new data, or depress″ /
   WRITE ″the RETURN key if the item is not to be changed.″ //
   UPDATE′TEXT ″Last Name″ FOR last
   UPDATE′TEXT ″First Name″ FOR first
   UPDATE′TEXT ″Middle Initial″ FOR mid
   UPDATE′TEXT ″Street Address″ FOR addr
   UPDATE′TEXT ″Zipcode″ FOR zip
   UPDATE′TEXT ″City″ FOR city
   UPDATE′TEXT ″Birthdate (DDMMYYYY)″ FOR birth
   UPDATE′TEXT ″Jobcode″ FOR job
   STORE′RECORD
   WRITE / ″*** Record of employee ″, id, ″ has been updated ***″ /
SHOW′RECORD:
   GET′RECORD
   WRITE / ″The record for employee ″, id, ″ contains the following:″ //
   WRITE ″Last Name: ″, last /
   WRITE ″First Name: ″, first /
   WRITE ″Middle Initial: ″, mid /
   WRITE ″Street Address: ″, addr /
   WRITE ″City: ″, city /
   WRITE ″Birthdate: ″, birth /
   WRITE ″Jobcode: ″, job /
STORE′RECORD:
   PUT last, first, mid, addr, zip, city, birth, job IN file[id]
affirmative:
   READ response RAW
   REPORT response in {″y″; ″Y″; ″yes″; ″YES″}
GET′RECORD:
   PUT file[id] IN last, first, mid, addr, zip, city, birth, job

HOW′TO INIT:
   SHARE file
   PUT {} IN file

HOW′TO ASK′TEXT name FOR item:
   WRITE ″Enter ″, name, ″: ″
   READ item RAW

HOW′TO UPDATE′TEXT name FOR item:
   ASK′TEXT name FOR new′data
   IF new′data <> ″″: PUT new′data IN item
```

A Proposal for Matrix/Vector Functions in *B*

Lambert Meertens

Why matrix/vector functions?

As *B* gains use, application packages will become available. Among these will be packages for matrix/vector functions. However, there are reasons to include a collection of elementary matrix/vector functions in the predefined functions of *B*. This was already stated in the Draft Proposal.

One reason is that such functions are of a much more general nature than the capabilities made available by typical application packages. (The same applies to functions on complex numbers. However, it is a simple task to write a package of elementary complex functions in *B*. Matrix functions, on the other hand, are an area full of pitfalls, and are best left to the specialists.)

It should be stressed that this proposal is concerned only with a small collection of elementary functions to be made available in *standard B*. Numerical analysts will need a much larger collection of more specialized tools to ply their trade. Also, this proposal is in no way definitive. All comments are welcomed.

The representation of matrices in *B*

There is a more or less natural *B* data type for modelling matrices, namely the table whose keys are compounds with two fields.

Matrix operations may be defined using an arbitrary algebraic field for the associates. However, the only type in *B* that corresponds to an algebraic field is the type "numeric". In by far the most practical applications, matrices and vectors will deal with real numbers. It seems wise, therefore, to restrict the attention to numbers. A second best are complex numbers. Also, for some functions, such as for taking the transpose M^{T} of a matrix M (interchanging row and column indices), the type of the associates is immaterial, so these functions will accept matrices of any kind.

The indices of matrices and vectors usually form a set, such as $\{1, 2, \cdots, n\}$, or $\{0, 1, \cdots, n-1\}$. But if one looks closer at the definitions of the various operations, it becomes clear that the indices need not form a set of consecutive integers. For example, let X and Y be a $p \times q$ and a $q \times r$ matrix, respectively. The product $Z = X \cdot Y$ is then a $p \times r$ matrix, and the usual definition tells us that an arbitrary entry z_{ik} of Z is given by

$$z_{ik} = \sum_{j=1}^{q} x_{ij} y_{jk}.$$

This definition assumes that the indices are taken from a set $\{1, 2, \cdots\}$. However, for the product $X \cdot Y$ to be defined, the only important thing is that the set of column indices of X is the same as the set of row indices of Y. If we denote that set by Q, then we can reformulate the definition of z_{ik} as

$$z_{ik} = \sum_{j \in Q} x_{ij} y_{jk}.$$

If matrices are represented by tables, the relevant sets of indices can be determined from the values by means of the function `keys`. For example, the following two tables are matrices:

```
{[4, 1]: -1; [4, 8]: 6; [4, 9]:  4;
 [5, 1]:  2; [5, 8]: 4; [5, 9]: -1}
```

and

```
{[1, "buff"]: 1; [1, "fawn"]:  1;
 [8, "buff"]: 0; [8, "fawn"]: -1;
 [9, "buff"]: 0; [9, "fawn"]:  1}.
```

Since their respective sets of column and row indices are the same, `{1; 8; 9}`, their product is well defined:

```
{[4, "buff"]: -1; [4, "fawn"]: -3;
 [5, "buff"]:  2; [5, "fawn"]: -3}.
```

So there is no reason to give up the generality of *B* tables for the indices. An immediate advantage is that we do not have to choose between $\{1, 2, \cdots, n\}$ and $\{0, 1, \cdots, n-1\}$, either of which may be the natural choice in a given application.

Non-rectangular and sparse matrices

In many important applications, matrices are known to have many zero entries. An example are so-called "lower triangular" matrices. (A matrix X is lower triangular if each entry x_{ij} for which $i < j$ is equal to 0.) An example of a lower triangular matrix is

$$\begin{bmatrix} 7 & 0 & 0 \\ 3 & 8 & 0 \\ 1 & 5 & 7 \end{bmatrix}.$$

Multiplying two lower triangular matrices can be done much more efficiently than general matrix multiplication. For large matrices, this can save about a factor of six. The factor is already about 4.5 for 10×10 matrices. For the sake of discussion, let us call such matrices "sparse". (Usually, matrices are only called sparse if almost all entries are 0.)

There is no requirement in B that tables be "rectangular". This gives a natural modelling for sparse matrices: the same type as already chosen for "normal" matrices. "Missing" entries are assumed to be 0. For example, the following would be a 3×3 lower triangular matrix:

```
{[1, 1]: 7;
 [2, 1]: 3; [2, 2]: 8;
 [3, 1]: 1; [3, 2]: 5; [3, 3]: 7}.
```

It might appear at first sight that this representation makes it impossible to determine, e.g., the set Q introduced above. But if we take for Q the intersection of the sets of column indices of X and of row indices of Y (remember that missing entries are taken to be 0), then we can still use the previous definition of matrix multiplication. If Q is empty, the corresponding entry will be missing from the result. The product of, e.g., two lower triangular matrices, will then be lower triangular as well, which saves space and time.

Having separate functions for sparse and dense matrices (which have the same type) is confusing and adds to the complexity. But we can treat full rectangular matrices as a special, albeit frequent, case of the general scheme for sparse matrices. Still, we want the product of two rectangular matrices to be rectangular, and not have missing entries where a zero value is "accidentally" produced. So the product matrix should have a missing entry only if the set Q was empty.

The same ideas apply of course to matrix and vector addition. For example, the sum of the two vectors

```
{[1]: 8; [8]: -1          }
```

and

```
{         [8]:  1; [9]: 3}
```

will be

```
{[1]: 8; [8]:  0; [9]: 3}.
```

These ideas are not new; they were derived from a matrix/vector package for ALGOL 68[MEU78]. But it just so happens that B accommodates them as though it were designed for that purpose.

Which functions should be predefined?

The functions of a usable matrix/vector system must not only contain "algebraic" functions, but also "administrative" functions. An example of an algebraic function is multiplication; an administrative function would be: "extract the j-th column from a matrix".

Algebraic functions. Let us first turn our attention to the algebraic functions. Next to matrices and vectors, a matrix/vector system must also deal with scalars. Scalars are the kind of quantities found as entries in a matrix or vector. So, in our case, they are simply numbers. If we use the term "algebraic value" to stand for "matrix, vector or scalar", then an algebraic function takes one or two algebraic values as operands and returns an algebraic value. Algebraic functions with one operand are taking the inverse and the determinant (only for matrices) and taking the opposite. Matrix transposition can also be considered as an algebraic function, but will be dealt with under the administrative functions. Taking the opposite can be handled as multiplication by -1 and does not need a separate function. The algebraic functions that take two operands are addition, subtraction and multiplication. Addition and subtraction are only meaningful between two algebraic values of the same kind (and not, e.g., between a vector and a matrix), and their meaning then presents no problems. The operations between two scalars (simple numbers) already belong to the predefined functions. For multiplication the issue is more complicated. Let M and N stand for matrices, v and w for vectors, and s for a scalar. Then the products commonly found in mathematical texts have the forms $M\cdot N$, $M\cdot v$, $s\cdot M$ and $s\cdot v$. A more or less obvious meaning can be assigned to $v\cdot w$, namely that of the inner product of v and w, which is by far the most common multiplicative operation between vectors. Sometimes one finds the notation $v^{\mathrm{T}}\cdot w$ for inner product in mathematical texts, and then also $v^{\mathrm{T}}\cdot M$, suggesting a meaning for $v\cdot M$ to us.

The meaning of the various ways of combining two non-scalar operands in a multiplication can be made more precise by reducing each case to the well-understood case of matrix multiplication. We can turn a vector v into a one-column matrix $v\uparrow$ by taking $v\uparrow_{i1} = v_i$ (in which the choice of 1 for the column index is arbitrary). Then the matrix $M\cdot v\uparrow$ is also a one-column matrix, and there is a unique vector w such that $M\cdot v\uparrow = w\uparrow$. $M\cdot v$ is then defined to be that vector w. If we denote the converse operation of "$\uparrow$" by "$\downarrow$", then we can write, more concisely,

$$M\cdot v \;=\; (M\cdot v\uparrow)\downarrow.$$

The inner product of v and w is the single entry in the 1×1 matrix $v\uparrow^{\mathrm{T}}\cdot w\uparrow$. If we extend the meaning of "$\downarrow$" to: turn a one-entry vector into a scalar, then we can write:

$$v\cdot w \;=\; (v\uparrow^{\mathrm{T}}\cdot w\uparrow)\downarrow\downarrow \;=\; (v\uparrow^{\mathrm{T}}\cdot w)\downarrow.$$

Similarly, we have

$$v\cdot M \;=\; (v\uparrow^{\mathrm{T}}\cdot M)^{\mathrm{T}}\downarrow \;=\; M^{\mathrm{T}}\cdot v.$$

A nice property that we have now is $(v\cdot M)\cdot w = v\cdot(M\cdot w)$, since both stand for $v\uparrow^{\mathrm{T}}\cdot M\cdot w\uparrow$, the product of three matrices, and matrix multiplication is associative (i.e., parentheses do not matter). Unfortunately, it is not the case that $(v\cdot w)\cdot M = v\cdot(w\cdot M)$, nor do we have that $(M\cdot v)\cdot w = M\cdot(v\cdot w)$. The problem lies in the different nature of scalar multiplication. If a matrix is multiplied by a scalar, no similar reduction to matrix multiplication is possible at all. Luckily, the syntax of *B* requires the use of parentheses in this case, so the user will be forced to make the intention explicit anyway.

The inverse of a matrix M is denoted by M^{-1}. It is characterized by the property that $M^{-1}\cdot M = M\cdot M^{-1} = I$, in which I is the "identity matrix" with entries 1 on the diagonal, and 0 elsewhere. Here we run into a problem: the notion of "inverse" is ill defined, given our representation for sparse matrices. A matrix may be "singular", meaning it has no inverse. It is in general not easy to see if a matrix is singular. But a matrix with a row or column of all zero entries is certainly singular. How do we know that the matrix does not have such an invisible row or column? We could require that there are as many row as column indices. This is not a very good solution. In particular, it would be possible then that $M\cdot(M^{-1}\cdot v) \neq v$. Luckily, it appears that there is a better way out. I am told by numerical mathematicians that people who compute an inverse matrix M^{-1} almost always do so only because they want to solve an equation $M\cdot v = w$, in which v is the unknown. The solution is then given by $v = M^{-1}\cdot w$. Now, if not only M is given, but also w, the problem is well defined again. The indices of w determine which row indices of M count. This can be seen as follows. If M has some entry m_{ij}, but w has no index i, then a solution of $M\cdot v = w$ is only possible if v has no index j, for otherwise we will find an entry with index i in $M\cdot v$. But if we know that v has no entry v_j, then the elements in the j-th column of M are irrelevant in forming $M\cdot v$, so we may as well disregard the j-th column of M. More precisely, let M' be a copy of M. If there are row indices in M that are not indices of w, the *columns* for which there are entries in those rows are struck from M' (and thereby these rows disappear too). The indices of v will then be the remaining column indices of M', and there must be as many of those as there are indices of w. If any of the indices of w is now missing as row index of M', M' is deemed to be singular. Otherwise, we can try to determine $M'^{-1}\cdot w$ (and, in that process, M' can, of course, still turn out to be singular). This solution requires that, instead of "$^{-1}$", the operation "$^{-1}\cdot$" be a primitive function. This function could be made to work then between two matrices as well. Should a user really need a matrix inverse, it is now easily realized by $M^{-1}\cdot I$. The responsibility for constructing the proper identity matrix I rests now with the user.

A similar problem as for the inverse occurs for the determinant. Any attempt at a reasonable definition of a function "det" must give for a 1×1-matrix M of the form `{[k, k]: v}` the value of $\det(M)$ as `v`. However, if M is, say `{[1, 1]: 3}` and N is `{[2, 2]: 7}`, then $M\cdot N$ equals the empty matrix `{}`, so there is no hope that we could retain the algebraic identity $\det(M\cdot N) = \det(M)\cdot\det(N)$. I do not see a solution here of a comparable simplicity to the one for the inverse. Also, I am under the impression that in actual computations computing the determinant is relatively rare. The conclusion seems to be that we should, maybe, simply not include a predefined function for the determinant.

The various functions have to receive names. For example, it is not possible to use the sign $*$ for matrix multiplication, since $*$ is already in use and overloading functions is not allowed in *B* (it cannot be reconciled with the static type-checking system of *B*). The following is an attempt to name the various operations in a systematic way.

$v+w$	`v vec'plus'vec w`
$M+N$	`m mat'plus'mat n`
$v-w$	`v vec'minus'vec w`
$M-N$	`m mat'minus'mat n`
$s\cdot v$	`s sca'times'vec v`
$s\cdot M$	`s sca'times'mat m`
$v^{\mathrm{T}}\cdot w$	`v vec'times'vec w`
$v^{\mathrm{T}}\cdot M$	`v vec'times'mat m`
$M\cdot v$	`m mat'times'vec v`
$M\cdot N$	`m mat'times'mat n`
$M^{-1}\cdot v$	`m mat'inv'vec v`
$M^{-1}\cdot N$	`m mat'inv'mat n`

A remark can be made here: the two functions vec'plus'vec and mat'plus'mat could be replaced by a single one:

```
YIELD t1 tab'plus'tab t2:
   PUT t1 IN sum
   FOR k IN keys t2:
      SELECT:
         k in keys sum:
            PUT sum[k]+t2[k] IN sum[k]
         ELSE:
            PUT t2[k] IN sum[k]
   RETURN sum
```

The function thus defined is more general; it can be used for pointwise addition of two tables whose keys may be of any type. However, since the keys of vectors may already have an arbitrary type, the function defined above is the same as vec'plus'vec, and mat'plus'mat is superfluous! A similar situation occurs for vec'minus'vec and sca'times'vec. My feeling is that nothing is gained by removing mat'plus'mat, mat'minus'mat and sca'times'mat from the list. The simplification by reducing the number of functions is more apparent than real; it cannot compensate the increase in complexity caused by the loss of a uniform, orthogonal, naming convention.

Administrative functions. In mathematical texts matrix transposition is usually written as a postfix operation, as in M^T. This is not possible in *B*, of course, but we can simply use a monadic function, as in transpose m.

We need functions for selecting a row or column in a matrix. An obvious notation is m row i and m col j. The function row is characterized by:

```
k in keys (m row i)  if and only if
   (i, k) in keys m,  and then
(m row i)[k] = m[i, k],
```

and m col j is of course the same as (transpose m) row j. Other useful functions are those for the selection of the diagonal and codiagonals. However, because of the additional requirements on the two fields selecting matrix entries for these functions, it is unlikely that the user would not have the list of relevant keys readily available, so these functions are easily programmed by the user.

Given the functions row and col, the same is true for many other useful operations. For example, to delete a column of a matrix, the user can write something like:

```
FOR i IN keys(m col j):
   DELETE m[i, j]
```

It is also easy to turn a vector into a matrix of one column. A column of a matrix can then be changed by deleting the old column and adding a matricized version of the new column.

Two more functions that could easily be programmed by the user are probably important enough to include in the predefined collection. They are the projections of the (two-field) keys of a matrix on the first and the second field. The function row'keys can be defined as follows:

```
YIELD row'keys m:
   PUT {} IN rk
   FOR i, j IN keys m:
      IF i not'in rk:
         INSERT i IN rk
   RETURN rk
```

and col'keys is defined similarly. Note that a much more efficient implementation of the function row'keys is possible than the one given here: for a $p \times q$-matrix, $O(p \cdot \log pq)$, rather than $O(pq)$.

REFERENCE

[MEU78] S.G. van der Meulen & M. Veldhorst, TORRIX, A Programming System for Operations on Vectors and Matrices over Arbitrary Fields and of Variable Size, Vol. I, Math. Centre Tract 86, Math. Centre, Amsterdam, 1978.

Speeding Up the *B* Implementation

Guido van Rossum

Introduction

About six months ago, Mark 1 of the *B* implementation was released to the world. This was a portable implementation [M&P], but not very fast (though already much faster than the first prototype implementation). Since then, we have made improvements to many parts of the system, started the detailed design and implementation of a more complete "*B* programming environment" (see Newsletter 1), and started the porting effort to the IBM-PC. While the work on the programming environment and the port to the IBM-PC are far from complete, we have managed to produce a version of the released system which is about four times faster. This article describes some of the "tricks" we used in speeding it up. It is expected that a version incorporating these changes will be officially released soon after the appearance of this Newsletter.

Small Integers

Even though *B* supports values of other types, numbers (and especially integers) are heavily used in most *B* programs. Moreover, the *B* interpreter uses integers a lot for internal purposes.

Number implementation. *B* calls for two types of numbers: exact and approximate. Exact numbers are represented as fractions: a pair of integers, called numerator and denominator. These integers themselves can have any size, and are represented as an array of suitable length of machine integers (usually 16 or 32 bits). It is not too hard to program arithmetic operations for such integers: the same methods as are taught in elementary school for addition, multiplication etc., can be used with the machine integers as "digits" and using the hardware for single-"digit" additions and multiplications [KNU]. Dynamic storage allocation is used to minimize storage requirements while allowing arrays of arbitrary size.

Drawbacks. This representation is very flexible, but does not allow particularly efficient manipulation of small integers: every numeric *B* value is represented as a pointer to a storage area containing the actual value. For instance, to get the value of an integer into a machine register, there is a subroutine which follows the pointer to the integer, performs some checks, extracts the value, and returns it. When lots of integers have to be manipulated (assuming each one fits in a word), this takes much longer than would direct handling of machine words. Also, storage requirements for, say, an array of (small) integers are much larger than in a traditional situation where each integer takes only one word.

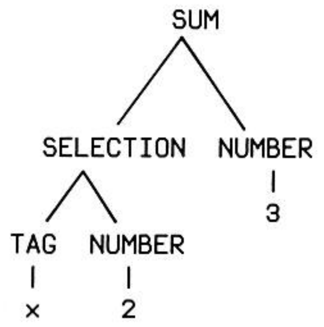
Solution. A scheme similar to that used by most Smalltalk implementations was adopted, see for instance [KRA]. In the pointer values, we introduce a "tag bit", which guides the interpretation of the pointer. If the tag bit is not set, the pointer is indeed a pointer, directed at the value's representation; if it is set, the rest of the pointer's bit pattern is interpreted as an integer value. If a pointer has 32 bits (including the tag bit), we can represent integers up to 31 bits as a pointer with the tag bit set. For larger integers, the tag bit is not set, and the pointer points to the normal representation for arbitrarily-sized integers. (Actually, a limit somewhat smaller than 31 bits has been chosen.)

Of course, the arithmetic routines (and a few others) must be fixed to cope with the new format. Accesses of large numbers are slightly slower, but small integers are manipulated much faster, at the cost of somewhat larger code. This trick alone has about doubled the overall speed of the *B* interpreter, especially because integers are used freely in the representation of *B* units as parse trees to encode the built-in commands and operations, and these of course all fall in the "small" category.

Portability issues. Not all machine architectures have a spare bit in each pointer. However, on machines which require (or encourage) alignment of data on even addresses (or higher powers of two), the low-order bit of a pointer is always zero, so it can be used as the tag bit. The integer value has to be shifted right one position then. Some machines with word addressing, like the CDC Cyber, have more bits in a pointer than can be used for the largest address; here the high-order bit can be used as the tag bit. If the worst comes to the worst, one can always arbitrarily align all values on even addresses, and use the low-order bit as tag bit. This is possible because pointers always point to data that has been allocated on the "heap", so one can change the allocation routine to guarantee that it returns an even address. This causes a word to be left unused sometimes, but the wasted space will probably be less than ten per cent, for not-too-big word sizes.

Changing the interpreter's structure

The *B* interpreter works as follows: a command or unit is first transformed into a "parse tree" (more precisely, an abstract syntax tree), which is then traversed according to the semantic rules. For instance, the expression `x[2]+3` is transformed to something like the following tree:



A traditional way of "executing" such parse trees, which was originally followed, is the following: there exists a routine "evaluate" which, when called with a parse tree as parameter, will return the value to which it evaluates. It determines the type of the root node of the tree (`SUM`, in the example), and then calls itself recursively on the respective subtrees, applies the operator to these results and returns the result. For node types like `TAG` or `NUMBER`, which act as terminal nodes, it retrieves the value directly and returns it (in the case of `TAG` nodes, which refer to targets, the memory is searched for a target of the given name).

Although this mechanism is simple to implement, it is not particularly efficient. It requires one procedure call per visited node in the tree, and uses one "stack frame" per level of recursion, where the recursion depth is equal to the height of the tree. A much more efficient scheme would be to have a linear representation, corresponding to "post-order" traversal of the tree (related to "reversed polish" notation). In the above example the nodes would be visited in the following order: `TAG(x)`, `NUMBER(2)`, `SELECTION`, `NUMBER(3)`, `SUM`. This would need no recursion, but instead we must have an explicit stack to hold intermediate values (as opposed to the implicit stack of call frames used by recursion). Fortunately, stacks are easily created and handled.

For example, a binary operation like `SUM` pops two elements from the top of the stack, and pushes its result, their sum, back onto the stack. After the whole sequence has been executed, the final result can be found on top of the stack (and it is the only element left on the stack). Such a stack takes much less space than the stack used by recursion. Not only does it grow with steps of only the size of a value pointer, while the recursion stack would grow with the size of a stack frame (at least several machine words in size), but also it grows less high: for an expression like `a+b+c+d+e`, parsed as `(((a+b)+c)+d)+e`, the recursion level of the old method is equal to the nesting depth of the tree, while with the new method in this case the stack never grows higher than two entries. (Of course there are "worst case" expressions where both methods have their stacks grow to the same level, but on the average the new method is much better.) The number of procedure calls is greatly reduced, too.

To cut a long story short, the interpreter has been rewritten to use the new scheme. Rather than using a separate store for the linear sequence of operations, we have augmented the parse tree nodes with an extra pointer to the next parse tree in the sequence. In this way, we still have the parse tree available, containing all information about the external appearance of the unit, which is useful for printing or editing it. (Currently, the editor uses yet another representation.) The process that sets up these pointers still uses recursion, but this is done only once, so this does not affect the execution speed. Nodes that show the placement of parentheses, which must be present to allow proper printing or editing, are omitted from the execution sequence, which reduces the number of visited nodes somewhat.

Nodes indicating conditional execution (`IF`, `AND` etc.) and repetition (`FOR`, `SOME` etc.) have two possible successors and so need some extra provisions: these have an extra pointer which points to an alternative next node to continue from if the test fails (or succeeds, in some cases). A traditional stack frame technique (implemented using the same stack as used for intermediate values) is used for calls of other units, refinements and formal parameters to HOW'TO-units (implemented with "thunks"). In the "spirit of *B*", the size of the stack is only limited by the total amount of available memory.

Miscellaneous improvements

Various other changes have been made to improve the interpreter's speed. Most of these involve moving work from the execution phase, where it would be done each time a particular command is executed, to a "preprocessing" phase, where it need be done only once. For instance:

Local targets. Originally, local targets were put in a table, with the target's name as key. In the new system, local targets are put in a fixed-length array (whose size is computed by the preprocessing phase), where accesses are much faster.

Unit calls. When another unit is called, the unit name has to be looked up in a table. In the new system, this is done in the preprocessing phase, and a pointer to the definition of the called unit is planted at the call. A similar thing is done for built-in functions.

Distributed permanent environment. This is not an execution speed improvement, but makes the start-up phase of the *B* system faster: targets in the permanent environment are put on separate files, which are only read in when first needed. The old system puts all permanent targets together on one file, which is read in when the system is started, and written back on exit. This makes the system appear very slow on start-up when there are large tables or lists in the environment, because it has to read all targets, even when you only want to inspect one of them; also, it wastes time when it writes back the values of all targets even when only a few have actually been changed. The new scheme greatly reduces the start-up time, thus making it much more pleasant to use. (It is also a step on the way towards a "virtual memory" system, where infrequently used targets may be written back to disk to make room for others, and makes a more efficient "checkpointing policy" possible, where all changed targets are written back at regular times to safeguard agains crashes and other misfortune.)

REFERENCES

[M&P] Lambert Meertens and Steven Pemberton, *An Implementation of the* B *Programming Language.* CWI, note CS-N8406.

[KNU] D.E. Knuth, *The Art of Computer Programming,* Vol. **2**, Seminumerical Algorithms. Addison-Wesley.

[KRA] Glen Krasner, *Smalltalk-80: Bits of History, Words of Advice.* Addison-Wesley.

New Publications

Only two new publications have appeared since the last *B* newsletter. The publications mentioned in the first two newsletters are all still available, as are the newsletters themselves.

Cursus programmeren voor beginners — Een kennismaking met de programmeertaal B, Deel 1,
Leo Geurts, 85 bladzijden.

This is a translation of *Computer Programming for Beginners — Part 1*, mentioned in the previous newsletter. Dit rapport bevat het eerste deel van een beginnerscursus programmeren, gebaseerd op de nieuwe programeertaal *B*. De meeste elementaire programmeertechnieken en de meeste eigenschappen van *B* komen aan bod. De aandacht gaat vooral uit naar het ontwerpen en schrijven van programma's, niet zo zeer naar de praktische omgang met computers. Veel korte programma's worden als voorbeeld gegeven, of als oefening van de lezer gevraagd.
De tekst vereist geen voorkennis, en is zowel voor cursussen als voor zelfstudie geschikt.
Deel 2 zal dieper ingaan op de taal en op de kunst van het programmeren. Gepubliceerd door het CWI, Notitie CS-N8407, fl. 11.90.

An Implementation of the B *Programming Language,*
Lambert Meertens and Steven Pemberton,
8 pages.

This gives an overview of the implementation of *B* and some of the techniques used in it.
It is a version of the article that appeared in USENIX Washington Conference Proceedings (January 1984), mentioned in the previous newsletter. Published by CWI, note CS-N8406, price Dfl. 3.70.

CWI publications can be ordered from

Publications Department
CWI
POB 4079
1009 AB Amsterdam
The Netherlands

You will be invoiced. The prices quoted exclude postage and packing, and for foreign orders there is an additional charge of Dfl. 6.50 to cover bank charges.

Please, fill out **both** copies below, and **sign** them.

SOFTWARE AGREEMENT

Effective as of .... 198.., Stichting Mathematisch Centrum (SMC), having an office at 413 Kruislaan, 1098 SJ Amsterdam, and

(LICENSEE), having an office at

agree as follows:

SMC grants fee-free to LICENSEE a personal, non-transferable and non-exclusive right to use the computer programs and documentation relating to the Mark 1 implementation of *B* (LICENSED SOFTWARE).

LICENSEE agrees not to sell, lease or otherwise transfer or dispose of the LICENSED SOFTWARE in whole or in part.

SMC makes no warranties, express or implied. SMC shall not be held to any liability with respect to any claim by LICENSEE or a third party on account of, or arising from, the use or the inability to use LICENSED SOFTWARE.

Neither this agreement nor any rights hereunder, in whole or in part, shall be assignable or otherwise transferable.

signed by

For LICENSEE	For Stichting Mathematisch Centrum
Name:	Name: ....
Title:	Title: Director of Software Licensing
Signature and Date:	Signature and Date:

SOFTWARE AGREEMENT

Effective as of .... 198.., Stichting Mathematisch Centrum (SMC), having an office at 413 Kruislaan, 1098 SJ Amsterdam, and

(LICENSEE), having an office at

agree as follows:

SMC grants fee-free to LICENSEE a personal, non-transferable and non-exclusive right to use the computer programs and documentation relating to the Mark 1 implementation of *B* (LICENSED SOFTWARE).

LICENSEE agrees not to sell, lease or otherwise transfer or dispose of the LICENSED SOFTWARE in whole or in part.

SMC makes no warranties, express or implied. SMC shall not be held to any liability with respect to any claim by LICENSEE or a third party on account of, or arising from, the use or the inability to use LICENSED SOFTWARE.

Neither this agreement nor any rights hereunder, in whole or in part, shall be assignable or otherwise transferable.

signed by

For LICENSEE	For Stichting Mathematisch Centrum
Name:	Name: ....
Title:	Title: Director of Software Licensing
Signature and Date:	Signature and Date:

HOW'TO ORDER MARK1:

To order the Mark 1 implementation of *B*, running on UNIX† systems, you should fill out the order form below, and two signed copies of the SOFTWARE AGREEMENT on the next page. Send it to:

B Implementation
Computer Science Department
CWI
POB 4079
1009 AB Amsterdam
The Netherlands

You will then receive:

- a tape with the software (including an installation guide)
- the following documentation:
  - Description of *B*
  - A Users Guide to the *B* System
  - *B* Quick Reference Card
  - Manual Page

Also, one of the two copies of the SOFTWARE AGREEMENT will be returned to you signed.

† UNIX is a Trademark of AT&T Bell Laboratories

ORDER FORM

Please send us the Mark 1 implementation of *B* for UNIX systems for the price of Dfl 100 (US $ 35) (to cover materials, postage and bank charges) for which we will be invoiced.

Name: ....

Title: ....

Firm/Institute: ....

Address: ....

....

Telephone: ....

USENET network address: ....

Check required tape parameters:

density	☐ 800 bpi	☐ 1600 bpi
blocking factor	☐ 1	☐ 20
format	☐ Ansi labelled	☐ Tar

We include **two** copies, both **signed**, of the SOFTWARE AGREEMENT.

Signature and Date: